Theoretical Computer Science 450 (2012) 92–108

Contents lists available at SciVerse ScienceDirect

Theoretical Computer Science

journal homepage: www.elsevier.com/locate/tcs

A finite state intersection approach to propositional satisfiability

José M. Castaño *, Rodrigo Castaño

Depto. de Computación, FCEyN, UBA, Argentina

ARTICLE INFO

Keywords:
ALL-SAT
Model counting
FSA intersection
Regular expression compilation
Intersection grammars (FSIG)

ABSTRACT

We use a finite state (FSA) construction approach to address the problem of propositional satisfiability (SAT). We present a very simple translation from formulas in conjunctive normal form (CNF) to regular expressions and use regular expressions to construct an FSA. As a consequence of the FSA construction, we obtain an ALL-SAT solver and model counter. This automata construction can be considered essentially a finite state intersection grammar (FSIG). We also show how an FSIG approach can be encoded. Several variable ordering (state ordering) heuristics are compared in terms of the running time of the FSA and FSIG construction. We also present a strategy for clause ordering (automata composition). Running times of state-of-the-art model counters and BDD based SAT solvers are compared and we show that both the FSA and FSIG approaches obtain an state-of-the-art performance on some hard unsatisfiable benchmarks. It is also shown that clause learning techniques can help improve performance. This work brings up many questions on the possible use of automata and grammar models to address SAT.

1. Introduction

There is a long tradition that analyzed transformations of logic formulas and automata formally [9,15,39,38]. Early studies by Büchi [9], Elgot [15] and Trakhtenbrot analyzed transformations from formulas to automata and vice versa in the context of the relation between FSA and Monadic Second Order logic (MSO). Sometimes referenced as the Büchi–Elgot–Trakhtenbrot Theorem, it was established that FSA and MSO have the same expressive power [38]. This was the basis of the approach that uses Büchi automata to decide satisfiability of modal logic formulas (LTL) [39]. This is also the only reference to an automata based approach to satisfiability found in [7]. Schützenberger, McNaughton, Papert and Kamp established the equivalence between star-free regular expressions, counter-free finite state automata, first-order logic and temporal logic, see [32].

Propositional satisfiability (SAT) solving has many practical applications ranging from artificial intelligence to software verification. Search-based techniques in SAT solving have been enormously successful. State-of-the-art SAT solvers are based on the DPLL (Davis–Putnam–Logemann–Loveland) algorithm, augmented with a number of features. Current research in this area has been mainly dominated by the DPLL algorithm, a search approach. This is explained partly by its success. A few other approaches were also considered; resolution based algorithms (DP), graph based algorithms (BDD) to name the most important. Little or no effort has gone into investigating alternative techniques.

There are applications that require not only a boolean answer but also the number of models for a propositional formula, or to know which are those models (ALL-SAT), or testing for functional equivalence. These tasks are performed using knowledge compilation. In knowledge compilation, a representation in a source language is compiled into a target language in order to perform reasoning tasks in polynomial time. Popular target languages are binary decision diagrams (BDD) and decomposable negation normal form (d-NNF).

* Corresponding author.
E-mail address: jcastano@cs.brandeis.edu (J.M. Castaño).

0304-3975/$ – see front matter
doi:10.1016/j.tcs.2012.04.030

Model counters [18] haven't progressed as much as SAT solvers because SAT heuristics designed to reduce the search space are, in many cases, not applicable, or their effectiveness is heavily reduced. BDD based solving has been an active research topic and there are efficient BDD based SAT solvers and model counters available, since the model count of a formula can be obtained from a BDD encoding. Efficient algorithms for model counting will have a significant impact on many application areas.

The approach presented here is framed into a more general enquiry about propositional satisfiability as a language or automata problem. As far as we know this question has not been pursued. It is difficult to know whether some of the results that are brought to attention in this work were not considered interesting or were not known, even though the existing connections and theoretical results were already present in the literature. From a different perspective, there are a number of works that relate satisfiability to language problems in order to show NP-completeness of the recognition problem for a particular language. Examples of this approach are the proofs that the word and generation problems for Two Level Morphology (TLM) are NP-complete [5]. There are a number of proofs for many other language classes. It should be stressed that the focus in these cases is the NP-completeness property and not an approach to solving SAT.

This work focuses on finite state techniques for SAT solving, an almost totally ignored approach. Approaching SAT as an FSA construction problem offers knowledge compilation capabilities. We can obtain model counting, equivalence testing and ALL-SAT answers from the constructed FSA. An FSA approach offers the advantages of a vast body of research and very simple and thoroughly studied algorithms.

Given the similarities with BDDs [20], it is rather surprising that this approach has not been explored more deeply in the context of propositional satisfiability. It was recently proved that acyclic deterministic finite state automata are equivalent to sequence binary decision diagrams, SDD [14]. A full comparison with BDDs and an understanding of why BDDs were used instead of using a more general machinery as FSA is beyond the scope of this work. We show however that an FSA approach can have competitive results with BDDs on a number of benchmarks.

In order to use FSA in the context of SAT, every valuation satisfying a propositional formula with variables v_i with $i \in [1, n]$ can be represented by a string in e^n, where e is either 1 or 0. If the i-th character of the string is 1 then v_i is *True* in that valuation, otherwise, v_i is *False* (cf. [40] and Theorem 7.3.8 and its corollary in [28][1]). Two crucial aspects are important for this approach to have any reasonable performance: variable ordering (the same as in BDD), and clause ordering, if the formula is in conjunctive normal form (CNF). Consequently satisfying valuations for a formula in propositional logic with n variables can be represented as a subset of L_n, the language of all possible valuations for n variables. This is clearly a regular language, in particular an acyclic regular language, where each word in the language has the same length. Acyclic regular languages have brought a lot of attention in particular for constructing DAWGs (Directed Acyclic Word Graphs). However those approaches are focused on incrementally constructing the automaton, given instances of the strings in the language (like in dictionary construction). In a SAT problem, given an ordering of variables the propositional formula can be interpreted as specifying the language (the set of valuation words), but the problem is finding out that set. Words are not given (as in the dictionary construction), instead 'a grammar', or a description of an automaton is given. A propositional formula can be interpreted as a description of the automaton, in particular as a regular expression which describes the automaton. Therefore, the costly operations to construct the automaton will be intersection and union, see [44]. From a theoretical point of view, this is no improvement, given that automata intersection is P-SPACE complete [17].

We show that for every propositional formula we can construct a regular acyclic automaton that describes the set of satisfying valuations. Given that satisfiability is a problem that is by itself in the realm of FSA, it is therefore expected to be solved by FSM in general, and any other machinery higher in the hierarchy. It also brings into question the power of FSA intersection to capture phenomena that are supposedly beyond context freeness, but that can be solved increasing the size of a grammar/automaton, or by automaton construction. This approach was briefly described in [10].

In the present work we report experiments on a number of benchmarks and show how variable ordering and clause ordering considerably affect the performance. An FSA can be constructed in a competitive time compared to state-of-the-art SAT solvers. We also present an approach to SAT solving as a parsing problem using Finite State Intersection Grammars (FSIGs) [42]. Given FSA intersection is P-Space complete [17] it is no surprise that FSIGs are able to encode SAT problems. It has also been shown that intersection ordering in FSA is NP-complete [8]. It should also be noted that such an approach to parsing is also a natural consequence to considering parsing as a CFG and automata intersection in [25,19]. In [25] it is shown that this approach can be used with formalisms more powerful than CF. As a consequence of the FSIG approach it is shown that parsing as CFG and automata intersection is NP-hard if no restriction is set on the description of the input automata. This is a consequence of building the automata and the complexity of intersection and union operations. It was shown that SAT problems encode unrestricted crossing dependencies (see [31,10]), and therefore the descriptive problem we are looking at is one that is at the heart of many natural problems (e.g. natural languages and biological sequences).

The remainder of this paper is organized as follows. In Section 2 we provide some basic definitions. Section 3 describes the approach presented in [10,11] to construct an automaton that defines the language of possible valuations of a propositional formula. Section 4 describes the variable ordering and clause ordering heuristics. In Section 5 we describe the experiments performed that show the possibilities of an FSA approach to SAT. Section 6 presents the generalization of the FSA approach to the FSIG approach, and the experimental results that show that such a grammar approach can bring significant

[1] Thanks to the anonymous reviewers for pointing out these references.

improvements to the FSA approach to SAT solving. In Section 7 clause learning is addressed as a pre-processing step that improves performance, despite increasing the number of intersections. In Section 8, non-clausal propositional satisfiability experiments are reported. Finally in Section 9 we present conclusions and questions for future work.

2. Definitions

Most of SAT related definitions and notation follow the ones given in [29].
L(A) denotes the language generated by an Automaton or Grammar, A.
A clause is a propositional formula of the form $l_1 \vee \cdots \vee l_n$, where each l_i is a *literal* i.e. a positive or negated propositional variable.
A term is a propositional formula of the form $l_1 \wedge \cdots \wedge l_n$.
Valuations are defined as functions v on a set of variables *Var* and with values in $\{0, 1\}$. Valuations assign a truth value from $\{0, 1\}$ to each propositional variable $p \in Var$. We denote the set of literals (positive or negated variables) determined by the set of variables *Var* by *Lit*. Then if $|Var| = n$, $|Lit| = 2n$. We say that a valuation v *satisfies* a formula ϕ or $v \models \phi$.
Complete set of literals. A complete set of literals is a set $S \subseteq Lit$ such that for every $p \in Var$ exactly one of $p, \neg p$ belongs to S. There is a bijective correspondence between valuations and complete sets of literals. One such mapping associates positive literals with 1 and negative literals with 0. An alternative mapping associates positive literals with 0 and negative literals with 1. It follows that if $|Var| = n$, then there are 2^n complete sets of literals over the set *Var*.
Total ordering. Truth values are ordered, $0 \leq 1$. Given an arbitrary ordering of *Var* there is a total (linear) ordering of valuations, which can be ordered lexicographically or anti-lexicographically. Valuations can be thought of as elements of the Cartesian product $Val = \prod_{p \in Var} \{0, 1\}$. The Cartesian product *Val* can be ordered lexicographically or anti-lexicographically. Ordered elements of *Val* are represented as boolean n-tuples (given $|Var| = n$). Consequently the least element in *Val* is the n-tuple $(0_1, \ldots, 0_i, \ldots, 0_n)$, where $0 \leq i \leq n$, and the maximum element in a valuation is $(1_1, \ldots, 1_i, \ldots, 1_n)$.
Valuations as strings. There is a correspondence between valuations and strings in $\{0, 1\}^n$. Ordered elements in *Val* can be represented as strings in $\{0, 1\}^n$, which can be (anti-)lexicographically ordered. We will say that a word w satisfies a formula ϕ ($w \models \phi$) iff w is the string representation of an element $v \in Val$ and $v \models \phi$.
Regex (anti-)lexicographical order. Regular expressions of the same length allow us to define sets of words that satisfy a formula. These can be ordered (anti-)lexicographically. Anti-lexicographic (colexicographic) order starts from the right. We use anti-lexicographic order in the translation of clauses to regular expressions. Alternatively lexicographic order could be used with a reverse clause ordering. We define anti-lexicographic order for regular expressions of the same length as follows:

Given $a_k, b_k \in \{0, 1, ?\}$, with $<$ a total order such that $? < 0 < 1$:
$[a_1 a_2 \ldots a_n] <_{alex} [b_1 b_2 \ldots b_n] \Leftrightarrow$
$(\exists j\ 0 < j \leq n,\ \forall i\ j < i \leq n,\ (a_i = b_i) \wedge (a_j < b_j))$[2]

Such a clause order will place clauses with lower variables first and those containing higher variables later.
Finite State Intersection Grammar (FSIG). A finite state intersection grammar is the product of the intersection of an acyclic sentence finite state automaton S and a set of finite automata $C_1, \ldots C_n$, usually called constraints: $G = S \cap C_1, \ldots \cap C_n$.

3. Satisfiability as a regular language problem

Barton [5] uses a finite state machine (FSM) to solve propositional SAT in order to show that the descriptive and generative power of PC-Kimmo and Two Level Morphology (TLM) as a grammar device are NP-complete. TLM and PC-Kimmo aimed at the description of morphological properties in a computational linguistics frame. The approach presented in [40] is in the more general framework of constraint satisfaction and introduces a representation of valuations as tuples. However [40] focuses on the construction of the minimized finite state automata (MDFA), an issue that we will ignore. We believe that the prohibitive cost of a direct translation is the reason why such an approach was not further explored. In order to construct the MDFA we assume we have a library that takes as input a regular expression and builds the MDFA. There will be issues of efficiency that will be idiosyncratic and dependant for each implementation of well known algorithms.

We describe how to construct an FSA automaton A for each formula ϕ in CNF,[3] such that the formula is satisfiable iff the language of A is not empty and for every word w in the language of A, $w \models \phi$, i.e. $L(A) = \{w \in \{0, 1\}^n | w \models \phi, n = |V_\phi|\}$. This means that the language of the automaton is the string representation of the set of valuations v such that $v \models \phi$.

The construction is based on the mapping between clauses and the dual terms. It is also based on the direct translation between boolean formulas and regular expressions, given the direct correspondence between $\vee$, $\wedge$, $\neg$ and |, &, ~, respectively and the closure properties of finite state automata.

[2] Note the difference with lexicographic order where the precedence is $i < j$.

[3] The translation can be easily extended to formulas not in CNF. We performed experiments on formulas not in CNF, in edimacs and iscas format, see Section 8.

Table 1
A propositional formula translated into a regular expression.

Formula		Regular expression		String in $\{a, b, \_\}^n$
		$[1\|0]^{10}$	&	
$(\neg v_1 \vee \neg v_3 \vee \neg v_5)$	$\wedge$	~[1 ? 1 ? 1 ? ? ? ? ?]	&	a_a_a_____
$(\neg v_1 \vee v_3 \vee v_6)$	$\wedge$	~[1 ? 0 ? ? 0 ? ? ? ?]	&	a_b__b____
$(v_1 \vee v_4 \vee v_6)$	$\wedge$	~[0 ? ? 0 ? 0 ? ? ? ?]	&	b__b_b____
$(\neg v_3 \vee v_5 \vee v_8)$	$\wedge$	~[? ? 1 ? 0 ? ? 0 ? ?]	&	__a_b__b__
$(\neg v_2 \vee v_5 \vee \neg v_8)$	$\wedge$	~[? 1 ? ? 0 ? ? 1 ? ?]	&	_a__b__a__
$(v_2 \vee v_7 \vee v_9)$	$\wedge$	~[? 0 ? ? ? ? 0 ? 0 ?]	&	_b____b_b_
$(\neg v_2 \vee v_7 \vee v_9)$	$\wedge$	~[? 1 ? ? ? ? 0 ? 0 ?]	&	_a____b_b_
$(v_2 \vee \neg v_7 \vee v_9)$	$\wedge$	~[? 0 ? ? ? ? 1 ? 0 ?]	&	_b____a_b_
$(\neg v_2 \vee \neg v_7 \vee v_9)$	$\wedge$	~[? 1 ? ? ? ? 1 ? 0 ?]	&	_a____a_b_
$(v_2 \vee \neg v_7 \vee \neg v_9)$	$\wedge$	~[? 0 ? ? ? ? 1 ? 1 ?]	&	_b____a_b_
$(\neg v_2 \vee \neg v_7 \vee \neg v_9)$	$\wedge$	~[? 1 ? ? ? ? 1 ? 1 ?]	&	_a____a_a_
$(v_4 \vee \neg v_6 \vee v_{10})$	$\wedge$	~[? ? ? 0 ? 1 ? ? ? 0]	&	___b_a___b
$(\neg v_4 \vee \neg v_9 \vee v_{10})$	$\wedge$	~[? ? ? 1 ? ? ? ? 1 0]	&	___a____ab
$(v_7 \vee \neg v_9 \vee \neg v_{10})$		~[? ? ? ? ? ? 0 ? 1 1]		_____b_aa

Each clause in a CNF formula will be interpreted as a regular expression that describes the automaton representing the set of valuations that satisfy that clause. For instance, a clause such as $v_1 \vee v_2 \vee v_3$, from a formula in CNF, with $|Var| = 10$, will be translated as the regular expression ~[0 0 0 ? ? ? ? ? ? ?]. We use the notation used in XFST (Xerox Finite State Tool [6]) which we used to do the experimentation. This is equivalent to ^'000.....' in languages like Python, awk or perl, with an extended use of the complement operator (^), which is used in these languages as a single character complement. Thus ~[0 0 0 ? ? ? ? ? ? ?] matches any string in $(0|1)^{10}$ that does not start with **000**.[4]

In Table 1 we show how a formula with ten variables is translated into a regular expression (second column) and a string (third column). Each clause and the corresponding regular expression are matched in a line. The first line in the regular expression column specifies the valuation space.

An automaton constructed this way may be used to check which are the strings generated. The translated regular expression is used directly by XFST to compute the automaton. In this case the string generated by the automaton, representing the satisfying valuation of ϕ, will be '1010000110' (or 'ababbbbaab' using the third column representation).

For a formula with m clauses the automaton to be constructed will be equal to the intersection of the corresponding m sub-automata. Therefore the asymptotic complexity of this construction is $O(|Var|^m)$, which might be worse than the exponential boundary $O(2^n)$. This is no surprise given that the automata intersection problem is a PSPACE problem; therefore this does not bring any theoretically interesting result in terms of computational complexity. It is however at the heart of a number of correlated problems, see [23].

This well known fact about the complexity of automata intersection and union [44], provides an explanation as to why, even if this approach is so transparent, it looks like nobody has considered it interesting. A direct implementation will result in an inefficient approach as we will see in the next section. We show that it is possible to address SAT problems using automata construction, and that this approach is capable of reaching a reasonable performance compared to other equivalent approaches, in particular we compare them with BDD and NNF approaches [12], and also it will be more efficient in unsatisfiable hard cases with state-of-the-art DPLL SAT solvers.

The NP-completeness of TLM (transducer and FSA composition in general) generation and recognition is derived from the expressive power of FSA, although the automaton construction is approached in a slightly different way. The transducer automata composition in TLM is made through the translation of the formula, into a string $\{0, 1\}^m$ of all possible valuations in the formula (m is the number of clauses). So the composition has to deal with 2^n possible representations of the formula, where n is the number of variables. This approach does not allow us to use the construction presented by Barton for SAT solving.

From a different perspective, if satisfiability is approached with machinery beyond context-free power, for any control language (in Weir's sense [41]), the control device will have to be at least as powerful as a FSA. Efficiency or complexity issues that are dealt at the FSA construction level will also be present at any other possible control device. Therefore if there is an efficient algorithm to make a control device, then it seems there is an efficient algorithm for FSA construction.

4. Variable and clause ordering heuristics

The approach presented in [11] was tested performing several fixed steps in order to produce the automaton. The steps were:

1. Produce a variable order
2. Produce a clause order

[4] In what follows we use *a* for *1* and *b* for *0*, due to restrictions on XFST use of *0*.

3. Translate the input formula into a regular expression, where the translation essentially produces a regular expression with as many intersections as clauses in the formula.
4. Build the automaton. This step was performed using a FSA manipulation toolbox.

The variable ordering heuristics we decided to test ranged from very simple ones to some very elaborate.

Freq. This is the simplest variable ordering heuristic. We sort variables according to the number of clauses they participate in, placing the most frequent first. In this way frequent variables will correspond to states located at the beginning of the FSA. The increased probability of obtaining a single path or no path at the early states will reduce the size of the constructed automata. If there are ties there is no preference strategy.

Max and Min. These two heuristics are extensions of the previous heuristic (Freq). In Freq, the order of a variable was not affected by its frequency as a positive or negated literal. **Max** gives preference in the ordering to variables that appear most of the time either negated or positive. **Min** will order first those variables that appear a similar number of times as a positive and negated literal.

Johnson. This heuristic is based on the heuristic for the maximum satisfiability (Max-SAT) problem proposed by Johnson [21]. Johnson's heuristic will iteratively satisfy the most frequent literal. Clauses that contain this literal are removed, and the dual literal is removed from the clauses that contain it. The proposed order is the one in which the variables are set. It is another variation of **Freq** heuristics.

Force. Force is a variable ordering heuristic intended to be used with BDDs and SAT solvers [2]. Force is particularly suitable for problems that possess a structure. This iterative algorithm, like MINCE (Min-cut vertex/variable reordering), tries to get the minimal cut value for variables (vertex or state). The cut value of a variable with index i is the number of clauses that contain variables with indices both $> i + 0.5$ and $\leq i + 0.5$. This also reduces the average clause span.

Anti-Lexicographic Clause Reordering. Anti-lexicographic ordering was used in [10] in order to decide the satisfiability of formulas in CCNF (i.e., where each clause has the full set of variables) in polynomial time $O(n^6)$.

In the translation of CNF formulas to XFST regular expressions the following order was used: $? < a < b$ (a was used instead of 1 and b instead of 0, as we mentioned above). The order of the second and third columns in Table 1 follows the anti-lexicographic ordering. For instance if we have three variables, and each clause has exactly two literals, the anti-lexicographic order in XFST regex will be as follows:

1. [a a ?]
2. [b a ?]
3. [a b ?]

... . . .

27. [? b b].

This heuristic combined with a variable ordering heuristic has the effect of computing first the intersection of clauses with variables that have higher priority order and postpone the computation of intersection in clauses with variables that have less priority. Also due to the anti-lexicographic ordering, clauses with smaller *span* (less difference between smallest and largest variables) will be given priority over clauses with bigger span. A third consequence is that clauses that share variables will be placed together in the ordering. All these facts are exemplified in Table 1, third column, above.

5. Experimental results with a FSA approach

5.1. Tool and setting

In order to test the possibilities of using the FSA construction approach as an ALL-SAT and model counter, we translated CNF encoded formulas into regular expressions as explained above. Then we used XFST to build the automata. XFST has the advantage of having a team with a sound experience of FSA tools. These tools have been developed with other purposes in mind (natural language processing, NLP). There are many open source tools that can be more attractive due to the possibility to modify them to try optimizations or profiling. Given this is a first approach in order to build a proof of concept, we considered that it would be a better choice to use a heavily tested and widely used tool. At the same time, the goal of these experiments was not about the FSA implementation of well known algorithms, but testing on differences in the running time due to variable (state) ordering and clause (sub-automata) intersection, or a more structured FSIG approach. XFST documentation is described extensively in [6]. We used XFST PARC version 2.15.2 available online. Variable ordering heuristics as well as clause reordering were developed in C++ and Python. The applications were attached together with Python and bash scripts. Both variable ordering and clause ordering heuristics were done without prioritizing performance because the goal was to compare as many heuristics as possible. It is worth noting that running time of variable and clause ordering heuristics seem negligible. Of course, if we were considering the strict performance of solvers, all these details have to be taken into account. All these algorithms have limited time and space complexity.

The sequence, as mentioned in the previous section and repeated here, is as follows:

1. Compute a variable order
2. Compute the anti-lexicographic order of clauses
3. Translate into a regular expression

Table 2
uf50 average running time on 1000 instances.

Freq-AL	1.62 s
Force-AL	2.14 s
Johnson-AL	4.20 s

4. Run XFST on the regular expression to build the automaton.
5. The output of XFST shows the properties of the automaton built with the number of solutions.

The running time of each application was observed using the Python time-it module. Each running time corresponds to a single run of the heuristics followed by XFST, except in the case of Force. Since Force has stochastic behavior, we decided to run the heuristic with XFST 5 times for each test case, computing the average running time. Experiments were run on a Linux machine with an Intel Xeon X3430, 2.40 GHz processor with 8 GB of memory.

5.2. Experiments with randomly generated instances

Initially we had run some experiments on a slower machine. It was observed that the direct translation of the formula to a regular expression had a prohibitive processing time, using the XFST regular expression compilation. Those experiments were run on some randomly chosen formulas. For example, processing the translated formula uf50-03.cnf from SATLIB (50 variables/218 clauses), took 864 s by XFST. However, after reordering variables by frequency and then reordering the formula in anti-lexicographic ordering it took only 0.533 s. It is worth mentioning that processing time reflects the size of intermediate automata in the construction. The final size of the constructed automaton is not representative of the complexity of intermediate steps. Reordering the same formula with the well known static variable reordering algorithm, Force [2], did not improve the running time as much. For instance on the same formula uf50-03, after reordering variables with Force, the processing time of XFST took 527 s. If we compare this with the compilation of the same formula by **Ebddres**, an state-of-the-art solver that is based in OBDD (16 s), we obtain a much better performance.

The following table compares two instances of the SATLIB uf50 benchmark with no variable and clause ordering (No Reorder) and Force [2] a static variable ordering algorithm, variable ordering by frequency and anti-lexicographic ordering of clauses (Freq-Anti-Lex) and Ebddress, a BDD based SAT-solver. Ebddress was chosen because it was the latest BDD based SAT-solver and it was developed by a team with long experience in SAT-solving.

Problem	No reorder	Force	Freq-Anti-Lex	Ebddress
uf50-01	1460.12	328.67	2.28 s, 2.5 kb	13.7 s, 376 MB
uf50-03	863.9	527.45	0.5328 s, 19.8 kb	11.9 s, 376 MB

Later, we ran all the heuristics on 1000 formulas of the uf50 SATLIB benchmark. Most of the heuristics exceeded the time limit of 30 s that we had set. The rest had the average timings given in Table 2.

The first part of the heuristic names refers to the variable ordering heuristics described above in Section 4. The second part of the names, AL, refers to anti-lexicographic ordering. In the next subsection, we use NR, denoting no reordering of the original clause ordering. The anti-lexicographic ordering has been shown to be a consistent strategy to limit the explosion of state size in the computation of automata intersection.

5.3. Hard benchmarks experiments

Initially we performed tests using some of the Ebddres benchmarks used in [34].[5] We chose them in order to compare with Ebddres, given it is a BDD based solver. These are hard unsatisfiable problems. Many of these problems have been looked at even with local search solutions [3,1]. **ph** files are instances of the pigeon hole problems. **Chnl** are unsatisfiable instances that model the routing of X wires in N channels [1]. **Urq** files are unsatisfiable randomized instances based on expander graphs [37]. **Fpga** are some satisfiable and unsatisfiable instances from FPGA routing. **Mutcb** instances correspond to the mutilated checker board. Then we added some other classes known to be hard, instances from the Beijing and Hanoi set (**2bit, hanoi**) from SATLIB. We also added some of the BMC-dimacs benchmarks (**barrel, queueinv, longmult**). The details and properties of these benchmarks can be found in Table 14.

The results were very good considering we were just implementing very few heuristics. However, they were rather disparate on some benchmarks.

Table 3 summarizes Min, Max, and Force variable reordering heuristics, with NR (no clause reordering). Null-NR, corresponds to the direct translation of the formula into a regular expression (no variable nor clause reordering). As can

[5] They are available at the Ebddres web page.

Table 3
FSA variable ordering heuristics with no clause reordering.

File	Force-NR	Null-NR	Min-NR	Max-NR
Total sec	16 455	18 384	20 618	21 771
# not solved	24	25	29	33

Table 4
FSA with variable and anti-lexicographic clause reordering heuristics.

File	Force-AL	Min-AL	Max-AL	Null-AL	Freq-AL	Johnson-AL
Total sec	10 847	18 052	19 129	19 710	21 017	25 297
# not solved	17	26	27	28	30	40

Table 5
FSA with Force and AL clause ordering vs other solvers.

Heuristics	Force-AL	sbsat	ebddres	c2d	relsat	sharpsat	clasp
Total time	10 847	18 051	10 764	16 969	21 638	18 321	15 471
Not solved	15	28	17	27	29	29	21
Solving T.	1847	4250	564	2569	4237	920	2871
# solved	36	23	34	24	22	22	30
Average T.	52.77	29.59	16.61	107.04	192.72	41.86	78.31

be seen Force-NR is the best in this set, although Null-NR (plain translation) is pretty close. The good performance of the *NR* class follows from the fact that some of these instances were constructed with some sort of anti-lexicographic ordering. Also in some cases the ordering of the variables seems to be close to the ordering computed by some heuristics.

Table 4 summarizes Freq, Min, Max, and Force, Johnson variable reordering heuristics combined with anti-lexicographic ordering (AL). Null-AL, corresponds to no variable re-ordering. Given the performance on the uf50 benchmark, the heuristics combined with AL were expected to have better performance. Force-AL was the best performing in both classes (NR and AL).

In order to have an approximate comparison with alternative approaches with model counting or ALL-SAT capabilities, we ran experiments with the following solvers: clasp, Ebddres, sbsat, sharpSAT, c2d and relsat. Model counting and ALL-SAT is not relevant for unsatisfiable instances anyway (most of them), given the number of models is zero.[6]

- Clasp obtained the gold medal for SAT/UNSAT crafted problems, in the 2009 SAT competition. It contains many advanced features, and also the capability to work as a model counter or ALL-SAT.
- Ebddres [34] (version 1.0), is a BDD based SAT solver that can generate extended resolution proof traces.
- Sbsat [16] is a state-based, BDD-based satisfiability solver. We used version sbsat-2.7b.
- Relsat is a model counter that was developed some years ago.
- Sharpsat [36], is a #SAT solver that is based on the DPLL algorithm. It is supposed to have a good performance on large structure problems.
- C2d [12] compiles CNF into d-NNF (decomposable negation normal form), a generalization of BDD. SharpSAT was also used to compile CNF into d-NNF[30].

In Table 5 we compare Force-AL, which showed the best performance among the heuristics we tried, against the above mentioned solvers. As can be seen its overall performance is very good. It is almost tied with ebddres in total time used but Force-AL solved more problems. However the average time used by Force-AL per solved instance is higher than the one used by ebddres and sharpsat.

In Table 6, we present the first four solvers or FSA construction strategy that had the best timings for each subset of problems. It can be seen that, for a number of problems (ph, Urq, chnll, fpga), the first positions are dominated by the FSA construction strategies, Force-AL being the most predominant. However for other subsets, current solvers perform much better.

6 These are the parameters we used to run each solver when we did not use the default values:

- clingo –clasp -n 0 -q (clasp mode, enumerate all models, quite mode)
- NetPlacer -c 6 (affects the output variable order, this is one of the Force executables)
- relsat -♯count -t600 (Count models, time limit)
- sbsat -All 0 -In 0 –max-solutions 0 -t –debug 0 (disable preprocessing options, disable inferences, find all solutions, start a stripped down version of the SMURF solver, disable debug).

Table 6
Best timings for problem subsets.

Problem set	First	Time	Second	Time	Third	Time
ph	Force-NR	844	Force-AL	1847	Max-AL	4060
mutcb	ebddres	7	c2d	264	clasp	844
Urq	ebddres	629	Force-AL	700	Force-NR	825
chnll	Force-AL	33	ebddres	49	Null-NR	999
fpga	Force-AL	908	ebddres	1266	Null-NR	2644
sat-grid	sbsat	3	ebddres	7	Max-AL	7.15
barrel	sbsat	3	relsat	3	clasp	3
queueinv	sbsat	2	relsat	2	clasp	2
hanoi	sbsat	2	clasp	9	relsat	605
2bit	clasp	1213	ebddres	1242	c2d	1242

If we analyze the data from these sets of instances, we can observe significant differences between them. It looks like problems like ph,unsat-fpga cannot be solved with usual features in most solvers, due to a similar distribution of variables in clauses (e.g. each variable occurs the same number of times, see Table 15 in the Appendix).

6. SAT as a Finite State Intersection Grammar (FSIG) recognition problem

There are a number of candidate grammatical formalisms that can be used to recognize the language of satisfiable propositional formulas (L_{SAT}), whose recognition problem has been shown to be NP-complete or NP-hard. Among many of them we can mention LCFRS [31], set automata [27], and FSIG [5]. Using a parser/recognizer for the corresponding formalism (if available), the set of possible valuations is obtained as a result of the parsing process. There is no proof we know of that the languages of FSIGs are NP-hard without use of transducer operations as is done by [5]. Although NP-completeness of the recognition problem in FSIGs follows from properties of automata intersection, the construction we show here is new.

A crucial step to use a parsing algorithm for L_{SAT} seems to be to build an appropriate data structure to *control* the derivation,[7] keeping track of the possible valuations as discussed in [10]. Building an FSA automaton to control the derivation of a CFG back-bone will have the same cost as building the FSA automaton and testing for emptiness. The latter was the approach followed by [11] and discussed above. A Finite State Intersection Grammar (FSIG) [24,42,35,43] has a flat structure: $S \rightarrow cl_1 \cap cl_2 \ldots \cap cl_n$. This can be seen also as an extreme in the approach of parsing as intersection [25]. In a parsing as intersection approach, the input string is generalized into an input automaton. In the formalization presented above there is no parsing in practice, and intersection is performed as established by the input automaton. The approach presented in this section puts some more work to be done by the grammar side allowing us to model in a different way the input automaton. A grammar formalism that encodes the possible valuations in the preterminal nodes can be easily encoded using a grammar-like notation with the following productions:

$$S \rightarrow T_k\ S \mid F_k\ S \mid \epsilon$$

where T_k denotes a *true* assignment and F_k a *false* assignment for the corresponding variable (k). The effect of the grammar-like back-bone is that each derivation or parsing tree encodes a possible valuation for each variable. Consequently for each variable we have two productions:

$$T_k \rightarrow cls_k \cap ct_k \text{ and } F_k \rightarrow cls_k \cap cf_k,$$[8]

where cls_k are the translated regular expression of the clauses that contain variable k as its highest variable and ct_k and cf_k are the constraints imposed to v_k according to the truth value assignment. For example,

- ct_6: [? ? ? ? ? **1** ? ? ? ?] (setting v_6 to 1)
- cf_6: [? ? ? ? ? **0** ? ? ? ?] (setting v_6 to 0)
- cls_6: ~[1 ? 0 ? ? 0 ? ? ? ?] & ~[0 ? ? 0 ? 0 ? ? ? ?]
  (from $(\neg v_1 \vee v_3 \vee v_6) \wedge (v_1 \vee v_4 \vee v_6)$)
- $T_6 \rightarrow ct_6$ & cls_6
- $F_6 \rightarrow cf_6$ & cls_6

We still need to make sure that constraints with possible valuations are checked at every step in the derivation/recognition process (i.e. *control* the derivation).

[7] In the sense of grammars with controlled derivation [13].

[8] This is more or less the approach used in [31] to prove the NP-completeness of LCFRS. The intersection operation is performed using the tuple representation.

In order to do that we are going to use a FSIG; however there are no productions in FSIGs. FSIGs are based in model theory and set theory [42]. A production like backbone is reduced to a FSA which is intersected with the set of constraints. The effect of *control of the derivation* is performed by the automata intersection operation. This is quite similar to the *control of the derivation* that is done by a set automata [27], or that could be done with a Stack Automaton [22]. We simulate the rules backbone with variable definitions used by the FSIG toolbox (XFST). We are simplifying the input to the toolbox in order to apply the preprocessing techniques to the formula independently. Therefore the sentence automaton that recognizes the clauses and performs the corresponding translations to the proper FSA constraints is done as part of the preprocessing process.

A FSIG grammar for a formula with n variables can be represented as follows, where k denotes a given order of *Var*:

$W = \{T_k, F_k\}$ is the set of *terminal* constraints or *unit words*, as defined by the translation above.
$N = \{S_k | k \in [0, n]\}$ is the set of non-terminal variables (or sub-automata).
S_{kh} is the axiom or designated symbol, where kh represents the highest k order.
The set of productions is: $\{S_k \to S_{k-1} \cap T_k, S_k \to S_{k-1} \cap F_k | k \in [1, |W|/2]\}$.

It should be noted that each production is defining a sub-automaton (i.e. a sublanguage).

In these *productions* the *terminal* constraints *control* the derivation. The base constraint $S_0(B_0$ from now on) is $(t|f)^n$, which defines the space of possible valuations. T_k, F_k are the set of constraints corresponding to the translation of a set of clauses cls_k, with the addition of a unit clause with the variable k set to *true* or *false*, respectively as we have seen above. Therefore T_k will produce all valuations that simultaneously satisfy a unit clause v_k and all clauses $c \in C_k$. Conversely, C_k will produce all valuations that simultaneously satisfy a unit clause $\neg v_k$ and all clauses $c \in C_k$. Any clause $c \in C_k$ has v_k or $\neg v_k$ as its literal and v_k as its highest variable.

One of the guidelines to pursue a grammar-like encoding is to use the grammar in order to separate the problem into subproblems, i.e. unrelated sets of clauses. The other was to simulate value assignment and unit resolution. The FSA approach [11] translated each clause into a regular expression and incrementally built the automaton by intersecting the already processed clauses with the next clause. This was equivalent to considering each clause a *terminal* symbol, or base word. In the approach presented here a set of clauses that share a variable in the last position are considered a *word*. As a consequence of processing all clauses together, in the case of anti-lexicographic order, the range of variables being analyzed was fixed on the lower end by the first variable and on the higher end by the highest variable processed so far. This meant that the variable range was always increasing; thus the full intermediate automaton had to be manipulated in each intersection, even when the added clause was unrelated to many of the previous clauses.

Changing the clause order means changing the order of intersections. The order of intersections affects the size of intermediate automata. The FSIG approach presented in this section reduces the impact of clause order because clauses will be grouped in smaller sets according to the grammar representation that is used. By separating the clauses into sets, the automaton can still be built incrementally but only considering a subset of the variables and clauses. This organization of the clauses into sets leads to an incremental construction of the automaton by intersecting the intermediate automata corresponding to each set of clauses. We believe this reduces the size of intermediate automata, because some restrictions imposed by a truth value assignment for a particular variable v_k will be expressed earlier, when the automaton corresponding to the set of clauses C_k is built.

This means that clause order only matters inside a set. The order of intersections between sets will be determined by the grammar. The steps to produce the desired automaton using a FSIG approach are the following.

1. Produce a variable order
2. Produce a clause order.
3. Translate the formula into a grammar formalism (FSIG)
4. Use a compiler (parser) to compute whether the formula is satisfiable or not.

6.1. Example with a grammar

We use XFST as the toolbox for FSIGs. As we mentioned above XFST (FSIGs) does not have a notation for CF grammars, but it is possible to define aliases for sub-automata. The generated input is compiled by XFST into the automaton that accepts the same language as the grammar.

The original formula from Table 1 in Section 3 has 14 clauses and 10 variables. The translation into XFST uses the *define* function for names of automata. It was translated as follows:

First, all the *terminals* are defined as in Table 7 as constraints in XFST (the second column). This is equivalent to initializing the terminals in a table for parsing.

The definitions given in Table 8 were added, where B_0 represents $[t|f]^{10}$.

6.2. NP hardness

In this section we have explained a reduction of SAT to the recognition problem for FSIGs and parsing as intersection of a context-free grammar and an input finite state automata. Both problems derive from well known properties of FSA composition.

Table 7
FSIG definition of words or terminals.

Formula		Regular expression	
		define **B0** $[0\|1]^n$	
		define **T5**	define **F5**
		[? ? ? ? **1** ? ? ? ? ?] &	[? ? ? ? **0** ? ? ? ? ?] &
$(\neg v_1 \vee \neg v_3 \vee \neg v_5)$	$\wedge$	~[1 ? 1 ? 1 ? ? ? ? ?];	~[1 ? 1 ? 1 ? ? ? ? ?] ;
		define **T6**	define **F6**
		[? ? ? ? ? **1** ? ? ? ?] &	[? ? ? ? ? **0** ? ? ? ?] &
$(\neg v_1 \vee v_3 \vee v_6)$	$\wedge$	~[1 ? 0 ? ? 0 ? ? ? ?] &	~[1 ? 0 ? ? 0 ? ? ? ?] &
$(v_1 \vee v_4 \vee v_6)$	$\wedge$	~[0 ? ? 0 ? 0 ? ? ? ?] ;	~[0 ? ? 0 ? 0 ? ? ? ?] ;
		define **T8**	define **F8**
		[? ? ? ? ? ? ? **1** ? ?] &	[? ? ? ? ? ? ? **0** ? ?] &
$(\neg v_3 \vee v_5 \vee v_8)$	$\wedge$	~[? ? 1 ? 0 ? ? 0 ? ?] &	~[? ? 1 ? 0 ? ? 0 ? ?] &
$(\neg v_2 \vee v_5 \vee \neg v_8)$	$\wedge$	~[? 1 ? ? 0 ? ? 1 ? ?] ;	~[? 1 ? ? 0 ? ? 1 ? ?];
		define **T9**	define **F9**
		[? ? ? ? ? ? ? ? **1** ?] &	[? ? ? ? ? ? ? ? **0** ?] &
$(v_2 \vee v_7 \vee v_9)$	$\wedge$	~[? 0 ? ? ? ? 0 ? 0 ?] &	[? 0 ? ? ? ? 0 ? 0 ?] &
$(\neg v_2 \vee v_7 \vee v_9)$	$\wedge$	~[? 1 ? ? ? ? 0 ? 0 ?] &	~[? 1 ? ? ? ? 0 ? 0 ?]&
$(v_2 \vee \neg v_7 \vee v_9)$	$\wedge$	~[? 0 ? ? ? ? 1 ? 0 ?] &	~[? 0 ? ? ? ? 1 ? 0 ?]&
$(\neg v_2 \vee \neg v_7 \vee v_9)$	$\wedge$	~[? 1 ? ? ? ? 1 ? 0 ?] &	~[? 1 ? ? ? ? 1 ? 0 ?]&
$(v_2 \vee \neg v_7 \vee \neg v_9)$	$\wedge$	~[? 0 ? ? ? ? 1 ? 1 ?] &	~[? 0 ? ? ? ? 1 ? 1 ?]&
$(\neg v_2 \vee \neg v_7 \vee \neg v_9)$	$\wedge$	~[? 1 ? ? ? ? 1 ? 1 ?] &	~[? 1 ? ? ? ? 1 ? 1 ?]
		define **T10**	define **F10**
		[? ? ? ? ? ? ? ? ? **1**]	[? ? ? ? ? ? ? ? ? **0**]
$(v_4 \vee \neg v_6 \vee v_{10})$	$\wedge$	~[? ? ? 0 ? 1 ? ? ? 0] &	~[? ? ? 0 ? 1 ? ? ? 0] &
$(\neg v_4 \vee \neg v_9 \vee v_{10})$	$\wedge$	~[? ? ? 1 ? ? ? ? 1 0] &	~[? ? ? 1 ? ? ? ? 1 0] &
$(v_7 \vee \neg v_9 \vee \neg v_{10})$		~[? ? ? ? ? ? 0 ? 1 1] ;	~[? ? ? ? ? ? 0 ? 1 1]

Table 8
FSIG definition of productions.

XFST notation	Grammar-like notation
Define S6 (T5 | F5) & B0;	$S_6 \rightarrow B_0\ T_5 \mid B_0\ F_5$
Define S7 (T6 | F6) & S6;	$S_7 \rightarrow S_6\ T_6 \mid S_6\ F_6$
Define S8 (T8 | F8) & S7;	$S_8 \rightarrow S_7\ T_8 \mid S_7\ F_8$
Define S9 (T9 | F9) & S8;	$S_9 \rightarrow S_8\ T_9 \mid S_8\ F_9$
Define S10 (T10 | F10) & S9;	$S_{10} \rightarrow S_9\ T_{10} \mid S_9\ F_{10}$

The translation of each clause into a regular expression can be performed in time polynomial to the size of the original formula, since the size of each regular expression will be proportional to the number of variables in the formula. The grammar back-bone itself has a constant size. The grouping of clauses into sets can also be done in time polynomial to the size of the formula. As a result, the whole translation is finished in time polynomial to the size of the original formula.

Once the input has been built, the original formula will be satisfiable if and only if the language of the resulting FSA is not empty [40,28] (FSIG) and therefore there is a parse tree in the parsing as an intersection approach[9] for the translated input.

6.3. Experimental results with an FSIG approach

As mentioned in the previous section, one of the advantages of the grammar-like approach presented here is the potential reduction in size of intermediate automata. The performance gain obtained in the experiments shows that the size of intermediate automata is indeed reduced. We compared the running time of the FSA approach versus the FSIG approach using the uf50 set from SATLIB. The FSA approach had an average running time of 1.62 s. The average with the FSIG approach reduced to 0.45 s.

We also ran an FSIG implementation on those hard unsatifiable instances detailed in Section 5.3. The complete results can be seen in Table 14. Each column of the table shows the running time of a different solver or model counter. It is interesting to analyze the overall performance and how it compares to that of the other solvers. Table 9 summarizes the total running time, including time spent on unsolved instances, number of instances solved and average running time per solved instance.

[9] The tree representing the generation of the formula.

Table 9
Overall performance comparison.

	FSIG	FSA	ebddres	clasp	c2d	sbsat	sharpsat
Total time	8699	10 246	11 315	14 271	15 769	16 851	17 121
Not solved	12	14	18	20	25	27	27
Solving time	1499	1847	515	2271	769	651	920.99
Solved	37	35	31	29	24	22	22
Average	40.51	52.77	16.61	78.31	32.04	29.59	41.86

Table 10
Effect of clause learning on clauses with 100 variables.

Problem	Lit. limit	Time	# clauses
uf100-01	4	248.78	3976
uf100-01	5	49.57	9091
uuf100-01	5	4.84	9667

Table 11
Iscas 85 benchmark results.

Problem	Var R	Var O	$c2d^r$	$c2d^t$
c499.bench	6.54	2.44	8	9.40
c880.bench	0.36	134.62	46	49.92
c1355.bench	6.55	4.38	18	20.46
c1908.bench	1.24	1.99	81	222
c2670.bench	324.57	OM	350	1018.18
c3540.bench	73.47	107.94	–	–
c5315.bench	678.58	OM	–	–
c7552.bench	OM	OM	384	>2700

The number of instances solved by the FSIG has increased with respect to the original FSA approach, and the average running time has also decreased. We used the same parameters for variable ordering (Force) and clause ordering (anti-lexicographic).

7. Clause learning

Clause learning [33] was introduced as an improvement to the DPLL algorithm. Clause learning has been shown to be equivalent to unrestricted resolution and there have been variations to make it more efficient (e.g. [4]). We present a clause learning scheme intended to be used as a preprocess of a CNF encoded formula. The restrictions we impose on resolution are: generating deductions in a particular order and discarding resolvents that exceed a number of literals.

This alternative and simple limitations to unrestricted resolution allow an implementation that results in little overhead and a series of possible customizations.

The algorithm we use to implement this kind of *clause learning* uses two loops based on the anti-lexicographic order of clauses and assumes there is a convenient order of variables (we used Freq as described in Section 4). It also uses sets of clauses grouped by the highest order variable in each clause, as we used in the FSIG section to define *terminals*. It compares clauses that share some literal and its dual. The corresponding resolvents are added to the appropriate set if they satisfy the literal limit and they are not tautological. For instance, if we have the clauses $v_1 \vee v_3 \vee v_5$ and $\neg v_1 \vee \neg v_2 \vee v_6$, we can add the clause $\neg v_2 \vee v_3 \vee v_5 \vee v_6$ to the set of clauses that have variable v_6.

The first loop starts from the first (lowest and most frequent) variable to the last (and least frequent) variable. All clauses that share the lowest variable in the clause will be compared (e.g. previous resolution example), adding resolvent clauses to the appropriate set. This loop has the effect of adding clauses to the most sparse space of the formula (right part of Fig. 1).

The second loop starts with the last (highest and least frequent) variable and performs the same operation. Deduced clauses in the previous loop will be available. Deduced clauses will be stored in the corresponding set, and given we started from the last variable, the deduced clauses in this loop will also be available. The effect of this second loop is to fill with learnt clauses the initial space of the formula (left part of Figs. 1 and 2).

The effect of clause learning can be appreciated in Figs. 1 and 2. In Fig. 1 a formula with 20 variables is represented in anti-lexicographic order. Each vertical line represents a clause with a literal in each position. The leftmost corner contains the shortest span in the most frequent variables. This is the starting point of the first loop. The rightmost part contains the largest span (from variables 1–20). This is the starting point of the second loop. Fig. 2 shows the *space* of variables 1–5 that is now *filled* with new learnt clauses. These new clauses restrict the space of possible solutions in that range of variables.

Table 12
Variable ordering heuristics combined with anti-lexicographic ordering.

File	Min AL	Force-AL	Johnson-AL	Null-AL	Freq-AL	Max-AL
ph07.cnf	1.031	1.036	1.046	1.04	1.04899	1.074
ph08.cnf	1.031	1.042	1.036	4.08	1.026	1.047
ph09.cnf	3.03	1.046	3.069	261.06	3.028	3.036
ph10.cnf	13.04	5.06	20.16	*	12.08	13.07
ph11.cnf	68.05	8.27	83.234	*	67.043	39.05
ph12.cnf	335.05	14.09	362.31	*	146.06	68.05
ph13.cnf	*	23.11	*	*	332.05	335.07
ph14.cnf	*	42.34	*	*	*	*
ph15.cnf	*	77.37	*	*	*	*
ph16.cnf	*	151.21	*	*	*	*
ph17.cnf	*	322.05	*	*	*	*
ph18.cnf	*	*	*	*	*	*
mutcb8	1.039	1.048	*	1.03	345.31	33.03
mutcb9	3.03	3.05	*	3.03	*	517.00
mutcb10	7.046	6.059	*	6.05	*	*
mutcb11	24.09	22.075	*	20.05	*	*
mutcb12	78.08	70.08	*	65.05	*	*
mutcb13	314.09	269.89	*	241.06	*	*
mutcb14	*	*	*	*	*	*
Urq3_5.cnf	2.05	6.06	*	395.06	18.07	2.05
Urq4_5.cnf	242.06	94.07	*	*	269.09	242.06
Urq5_5.cnf	*	*	*	*	*	*
chnl10_11.cnf	69.07	2.09	*	69.07	69.08	60.07
chnl10_12.cnf	70.07	3.10	*	70.07	69.09	70.07
chnl10_13.cnf	70.09	3.11	*	71.09	70.10	70.09
chnl11_12.cnf	337.08	4.11	*	335.08	332.11	338.08
chnl11_13.cnf	336.09	5.12	*	337.08	333.09	337.09
chnl11_20.cnf	347.21	15.27	*	345.18	343.28	346.21
fpga11_15_unsat	*	12.16	*	*	*	*
fpga11_20_unsat	*	20.48	*	*	*	*
fpga12_11_sat	*	173.08	*	*	*	*
fpga12_12_sat	*	101.89	*	*	*	*
fpga13_9_sat	*	*	*	*	*	*
sat-grid-pbl-0010	72.03	1.06	*	1.03	250.05	1.05
sat-grid-pbl-0015	*	47.05	*	46.05	*	1.04
sat-grid-pbl-0020	*	*	*	*	*	5.06
barrel2	1.81	1.03	1.98	2.93	1.86	1.85
barrel3	4.15	268.08	8.22	310.12	9.42	10.16
barrel4	50.14	*	186.46	*	332.33	432.17
queueinv2	2.08	3.05	30.07	1.51	12.06	2.06
queueinv4	*	*	*	245.08	*	*
longmult0	*		*	*	*	*
longmult1	*		*	*	*	*
hanoi4	*	63.38	*	78.27	*	*
hanoi5	*	*	*	*	*	*
2bitcomp_5	*	4.04	*	*	*	*
2bitmax_6	*	*	*	*	*	*
2bitadd_10/11/12	*	*	*	*	*	*

Fig. 1. Translated formula uf20-010.cnf from SATLIB without clause learning.

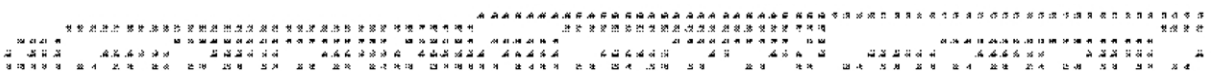

Fig. 2. Translated deduced clauses for formula uf20-010.cnf variables 1–5.

Table 10 shows some experiments using clause learning with different limits in the number of literals, where lit. limit is the maximum number of literals allowed in a learnt clause, and # clauses are the total number of clauses after the clause learning process finished. One of the conclusions we can obtain is that it is not the number of intersections to be performed,

Table 13
Variable ordering heuristics without anti-lexicographic ordering.

File	Min-NR	Force-NR	Johnson-NR	Null-NR	Max-NR
ph07.cnf	1.028	1.043	1.031	1.01	1.030
ph08.cnf	1.032	1.041	2.064	1.016	1.032
ph09.cnf	3.036	5.061	66.19	2.02	3.036
ph10.cnf	9.042	8.073	204.23	119.032	9.04
ph11.cnf	38.052	12.091	*	363.04	39.05
ph12.cnf	146.05	18.114	*	*	146
ph13.cnf	485.06	27.337	*	*	484
ph14.cnf	*	39.370	*	*	*
ph15.cnf	*	56.034	*	*	*
ph16.cnf	*	81.038	*	*	*
ph17.cnf	*	118.283	*	*	*
ph18.cnf	*	181.937	*	*	*
ph19.cnf	*	294.20	*	*	*
mutcb8	1.038	1.057	*	1.01899	33.03
mutcb9	3.044	3.054	*	3.02	517.00
mutcb10	9.045	21.063	*	6.02	*
mutcb11	40.09	66.072	*	20.04	*
mutcb12	148.09	236.279	*	61.05	*
mutcb13	577.08	*	*	223.03	*
mutcb14	*	*	*	592.08	*
Urq3_5.cnf	7.049	8.06	452.89	294.02	7.048
Urq4_5.cnf	*	217.87	*	*	*
Urq5_5.cnf	*	*	*	*	*
chnl10_11.cnf	47.07	206.47	59.12	30.03	47.09
chnl10_12.cnf	64.07	559.09	80.15	41.03	63.07
chnl10_13.cnf	84.08	*	106.19	55.04	84.09
chnl11_12.cnf	181.08	*	214.21	118.04	181.10
chnl11_13.cnf	236.0989	*	200.21	155.05	236.09
chnl11_20.cnf	*	*	*	*	*
fpga11_15_unsat	*	*	*	*	*
fpga11_20_unsat	*	*	*	*	*
fpga12_11_sat	*	*	*	424.03	*
fpga12_12_sat	*	*	*	420.06	*
fpga13_9_sat	*	*	*	*	*
sat-grid-pbl-0010	534.49	1.047	*	1.02	2.03
sat-grid-pbl-0015	*	93.054	*	93.00	114.08
sat-grid-pbl-0020	*	*	*	*	*
barrel2	3.05	1.037	4.51	2.10	3.25
barrel3	*	71. 08	*	156.11	*
barrel4	*	*	*	*	*
queueinv2	*	2.051	*	2.066	*
queueinv4	*	*	*	200.05	*
longmult0	*	*	*	*	
longmult1	*	*	*	*	
hanoi4	*	*	*	*	*
hanoi5	*	*	*	*	*
2bitcomp_5	*	323.78	*	*	*
2bitmax_6	*	*	*	*	*
2bitadd_10	*	*	*	*	*
2bitadd_11	*	*	*	*	*
2bitadd_12	*	*	*	*	*

but how the intersections are made, that has a significant effect on running time. Variables were ordered by Freq heuristic and clauses by anti-lexicographic order.

Additional experimentation with the FSIG approach described in Section 6 with a more limited clause learning (only the second loop and literal limit 4) resulted in a sharp improvement using uf100-01, with a resulting time of 24 s (versus 248 above). Using the hard benchmarks, there is little improvement, due to clause learning having little effect (this is one of the reasons why these benchmarks are hard), although there are two additional files that can be processed within the time limit (Urq5.5 with an average processing time of 100 s, with a wide range variation from 5 to 265 s, barrel3 decreased processing time from 96 to 46 s and barrel4 finished processing in 420 s). There were no significant changes in the other instances.

8. Non-clausal problems

One of the advantages of using a translation of propositional formulas into regular expressions, to compute an automaton, is that it is not necessary to restrict the input to clausal formulas. Therefore we can translate non-clausal formulas into

Table 14
Grammar and FSA approach vs other solvers, time limit reached at 600 s. Indicated by ∗.

File	Grammar	Force-AL	sbsat	ebddres	c2d	relsat	SharpSAT	clasp
ph07	5.040	1.042	1.00	1.00	1.00	1.00	1.00	1.00
ph08	5.036	1.046	1.00	1.00	6.00	2.00	1.00	1.00
ph09	5.038	5.06	0.99	1.00	85.00	18.00	5.00	6.00
ph10	5.044	8.27	1.00	2.00	*	231.00	50.00	48.00
ph11	6.002	14.09	2.00	5.00	*	599.00	*	400.00
ph12	7.000	23.11	30.00	12.00	*	599.00	*	*
ph13	9.001	42.34	432.00	39.00	*	*	*	*
ph14	13.001	77.37	*	*	*	*	*	*
ph15	20.000	151.21	*	*	*	*	*	*
ph16	35.010	322.05	*	*	*	*	*	*
ph17	66.402	*	*	*	*	*	*	*
mutcb8	5.034	1.048	1.00	1.0	2.00	1.00	1.00	1.00
mutcb9	7.002	3.05	1.00	1.0	2.00	2.00	1.00	1.00
mutcb10	9.052	6.059	3.00	1.0	3.00	12.00	5.00	1.00
mutcb11	21.001	22.075	*	1.0	5.00	*	98.99	5.00
mutcb12	51.003	70.08	*	1.0	13.00	*	441.00	32.00
mutcb13	154.817	269.89	*	1.0	42.00	*	*	204.00
mutcb14	388.692	*	*	2.0	197.00	*	*	*
Urq3_5.cnf	5.033	6.06	*	1.0	23.00	*	*	79.00
Urq4_5.cnf	15.003	94.07	*	28.0	234.00	599.99	*	*
Urq5_5.cnf	*	*	*	*	*	*	*	*
chnl10_11.cnf	8.002	2.09	*	2.0	*	*	49.00	32.00
chnl10_12.cnf	9.014	3.10	*	3.0	*	599.99	53.00	36.00
chnl10_13.cnf	10.010	3.11	*	3.0	*	599.99	55.00	42.00
chnl11_12.cnf	10.000	4.11	*	6.0	*	*	*	319.00
chnl11_13.cnf	12.002	5.12	*	7.0	*	*	*	469.00
chnl11_20.cnf	35.003	15.27	*	28.0	*	*	*	566.00
fpga11_15_unsat	17.003	12.16	*	*	*	*	*	*
fpga11_20_unsat	36.003	20.48	*	*	*	*	*	*
fpga12_11_sat	41.002	173.08	*	14.99 r	*	*	*	*
fpga12_12_sat	22.015	101.89	*	18.00 r	*	*	*	*
fpga13_9_sat	290.826	*	*	33.00 r	*	*	*	*
sat-grid-pbl-0010	5.041	1.06	1.00	1.00	2.00	*	1.00	1.00
sat-grid-pbl-0015	57.002	47.05	1.00	1.00	6.00	*	*	*
sat-grid-pbl-0020	*	*	1.00	5.00	84	599.99	*	*
barrel2	5.031	1.03	1.00	1.00	1.00	1.00	1.00	1.00
barrel3	96.027	268.08	1.00	*	3.00	1.00	1.00	1.00
barrel4	*	*	1.00	*	4.00	1.00	1.00	1.00
queueinv2	6.005	3.05	1.00	1.01	2.00	1.00	1.00	1.00
queueinv4	*	*	1.00	293.00	3.00	1.00	1.00	1.00
longmult0	*	*	*	*	*	*	*	*
longmult1	*	*	*	*	*	*	*	*
hanoi4	*	63.38	1.00	*	9.00	5.00	2.00	1
hanoi5	*	*	1.00	*	*	*	*	8
2bitcomp_5	6.003	4.04	7.00	*	2.00	2.00	1.00	1
2bitmax_6	*	*	*	*	12.00	42.00	2.00	2
2bitadd_10	*	*	161.00	*	28.00	322.00	149.00	10
2bitadd_11	*	*	*	*	*	*	*	*
2bitadd_12	*	*	*	*	*	*	*	*

regular expressions, using the mapping $\vee$, $\wedge$, $\neg$ to |, &, ~ respectively. This can be done in a straightforward way using XFST. We used Iscas 85 Benchmark files, which have the following syntax in this order, and one or more lines of each:

INPUT (VAR)
OUTPUT (VAR)
DEFVAR = OP(VARLIST)

where OP are for instance NOT, AND, NAND, NOR, OR and XOR. VARLIST is a list of one or more (according to the operator arity) INPUT or DEFVARs variables.

The translation using XFST was as follows (see Fig. 3). Start with the first line of the type VAR = OP(VARLIST), and replace it by *define gVAR*. The function *define* in XFST defines an automaton, and *gVAR* is the variable used by XFST to refer to this automaton. In the body of the automaton definition we replace the operator by the corresponding regular expression operator mapping (i.e.~, | or &). If there is an input variable in VARLIST, replace it by the regular expression [? ? ...1... ? ?]

Table 15
Relevant data of the benchmarks.

File	#Var	#CL	# Freq	Range Freq	Ratio Var/#Range	+w	-w	+-w
ph07	56	204	1	[8, 8]	7	196	8	
ph08	72	297	1	[9, 9]	8	288	9	
ph09	90	415	1	[10, 10]	9	405	10	
ph10	110	561	1	[11, 11]	10	550	11	
ph11	132	738	1	[12, 12]	11	726	12	
ph12	156	949	1	[13, 13]	12	936	13	
ph13	182	1 197	1	[14, 14]	13	1183	14	
ph14	210	1 485	1	[15, 15]	14	1470	15	
ph15	240	1 816	1	[16, 16]	15	1800	16	
ph16	272	2 193	1	[17, 17]	16	2176	17	
ph17	306	2 619	1	[18, 18]	17	2601	18	
mutcb8	121	344	4	[5, 8]	15.12	282	62	
mutcb9	155	451	4	[5, 8]	19.37	372	79	
mutcb10	193	572	4	[5, 8]	24.12	474	98	
mutcb11	235	707	4	[5, 8]	29.37	588	119	
mutcb12	281	856	4	[5, 8]	35.12	714	142	
mutcb13	331	1 019	4	[5, 8]	41.37	852	167	
mutcb14	385	1 196	4	[5, 8]	48.12	1002	194	
Urq3_5	46	470	15	[10, 128]	0.35	7	9	454
Urq4_5	74	694	15	[8, 128]	0.57	17	15	662
Urq5_5	121	1 210	17	[6, 128]	0.94	23	19	1168
chnl10_11	220	1 122	1	[11, 11]	20	1100	22	
chnl10_12	240	1 344	1	[12, 12]	20	1322	24	
chnl10_13	260	1 586	1	[13, 13]	20	1560	26	
chnl11_12	264	1 476	1	[12, 12]	22	1452	24	
chnl11_13	286	1 742	1	[13, 13]	22	1716	26	
chnl11_20	440	4 220	1	[20, 20]	22	4180	40	
fpga10_9_sat	135	549	2	[10, 13]	10.38			
fpga11_15_unsat	330	2 340	1	[15, 15]	22	2310	30	
fpga11_20_unsat	440	4 220	1	[20, 20]	22	4180	40	
fpga12_11_sat	198	968	2	[12, 16]	12.37	957	143	
fpga12_12_sat	216	1 128	2	[13, 17]	12.7	1116	156	
fpga13_9_sat	176	759	3	[10, 14, 15]	11.73	750	126	
sat-grid-pbl-0010	110	191	6	[3, 8]	13.75	2	10	179
sat-grid-pbl-0015	420	781	6	[3, 8]	52.5	2	15	419
sat-grid-pbl-0020	930	1 771	6	[3, 8]	116.25	2	18	759
barrel2	50	159	6	[4, 20]	2.5	12	17	130
barrel3	275	942	8	[3, 29]	9.48	172	10	788
barrel4	578	2 035	8	[3, 66]	8.75	160	177	1698
queueinv2	116	399	21	[3, 52]	2.23	39	28	332
queueinv4	256	955	31	[3, 86]	2.97	98	81	776
longmult0	437	1 206	14	[0, 232]	1.88	43	102	1061
longmult1	791	2 335	17	[0, 234]	3.38	65	203	2067
hanoi4	718	4 934	16	[4, 39]	18.41	2780	44	2100
hanoi5	1931	14 468	17	[4, 45]	42.91	8164	104	6200
2bitcomp_5	125	310	3	[6, 9, 16]	7.81	30	30	250
2bitmax_6	252	766	3	[6, 16, 17]	14.82	60	46	660
2bitadd_10	590	1 422	3	[6, 16, 25]	23.6	360	62	1100
2bitadd_11	649	1 562	3	[6, 16, 25]	25.96	286	66	1210

or if it is negated by [? ? ...0... ? ?].[10] There are as many positions available, as input variables in the formula. In some cases, to the definition of the variable we add an intersection with $[0|1]^n$, where n is the number of input variables. This might seem redundant but it is necessary to restrict undesired strings. Last the OUT variables are defined. If there is only one OUT variable, the formula is satisfiable iff there is an assignment of values for the input variables that satisfies the conditions of the OUT variable (and all the variables under it). If there are more OUT variables, it allows us to test the properties of each OUT variable separately. This can be seen as a bottom-up computation of the different expressions that compose the formula. The structural definition of the formula is respected. It can be observed in the results presented in the following table that the ordering of variables has an impact on the performance of the XFST machinery. The reordering of variables was made using a simple strategy according to the most frequent variable across the subformulas. Table 11 shows the results of using XFST with and without variable reordering. Reordering of variables is referenced by Var R(enaming), and Var O(riginal) with no variable mapping. We compared XFST with c2d.[11] In Table 11 c2d^t stands for the timing we obtained in our experiments

[10] We repeat that we used *a* for *1* and *b* for 0.

[11] We used the same parameters for c2d (-in memory -dt_method 0 -dt_count 25 -count) reported on the c2d web page, but have some differences in timing.

```
# c17
# 5 inputs
# 2 outputs
# 0 inverter
# 6 gates ( 6 NANDs )

INPUT(1)
INPUT(2)
INPUT(3)
INPUT(6)
INPUT(7)

OUTPUT(22)
OUTPUT(23)

10 = NAND(1, 3)               define g10 ~[a ? a ? ?];
11 = NAND(3, 6)               define g11 ~[? ? a a ?];
16 = NAND(2, 11)              define g16 ~([? a ? ? ?] & g11);
19 = NAND(11, 7)              define g19 ~([? ? ? ? a] & g11);
22 = NAND(10, 16)             define g22 ~( g10 & g16);
23 = NAND(16, 19)             define g23 ~( g16 & g19);
                              define OUT g22 & g23;
```

Fig. 3. Iscas85 c17.bench non-clausal formula (left), translated to XFST (right).

and c2d^r stands for the compilation time reported at the web page (see also [12]). OM means it has been reached the memory limit available at XFST.

The results however are not directly comparable, because the XFST implementation, as is apparent from the description, did not use a CNF input; however c2d was run on the CNF translation of the iscas format.

9. Conclusions and future work

SAT solving and model counting is now present in many practical applications but these are NP- and #NP-complete problems. This paper makes a number of contributions related to the evaluation of an FSA based SAT approach with ALL-SAT and model counting capabilities.

As a first step, and most importantly, it is shown that the FSA approach is very competitive versus the traditional DPLL approach in some hard problems (problems where most of the features added to the basic DPLL algorithm don't help). This should not be interpreted as saying that the FSA approach is better than the DPLL or BDD/NNF approaches. There are decades of research and experience that cannot be surpassed with this initial proposal.

The experiments show variable and clause order have an enormous impact on performance. Variable ordering and clause ordering (automata intersection) are problems known to be NP-complete. Force heuristic generated far better variable orderings in problems with a particular structure. That was not the case in random class problems (uf50), where Freq is better than Force. The anti-lexicographic ordering proved to be a consistent strategy, still with room for improvement.

We also introduce a number of ideas that have shown very promising experimental results, such as a more elaborate FSIG approach to SAT, a clause-learning implementation as a preprocessing scheme and an extension of the main FSA approach to deal with formulas in non-clausal representations.

We believe that the most added value of these results are the questions they bring up front. It would be interesting to assess whether some of the knowledge built upon the DPLL and BDD tradition and experience can be used in an FSA approach. The formal languages community would be more suitable in finding the optimal way of constructing an MDFA for this very specific class of languages or pointing out other convenient grammar formalisms. Another important question that should be elucidated is whether an approach to a fixed large window of k variables and m clauses can solve practical problems in a different way.[12]

Acknowledgments

This research was funded by ANPCyT PICT-00263-2008. Many thanks also to the anonymous reviewers who provided detailed and helpful comments to improve this version.

Appendix

The following pages contain the complete data of the summarized tables (Tables 12–15) presented in Section 5. Table 15 presents relevant data of the instances from these benchmarks. #Var denotes the number of variables, #CL denotes the

[12] See [35,26] for intersection of a limited number of automata.

number of clauses. # Freq denotes the different number of occurrences for each variable. It is interesting to note that there are a number of instances that have the same frequency for all variables (#Freq is 1), or a limited variation of frequencies per variable (e.g. mutcb has 4 different frequencies, which range from 5 to 8. Range Freq denotes the range of different frequencies, between a minimum of the number of occurrences of a variable and the maximum frequency in the range. +w denotes the number of clauses that have only positive literals. -w denotes the number of clauses that have only negative literals and +-w denote the number of clauses that have both positive and negative literals. It is interesting to note that some of these hard benchmarks have only positive and negative literals.

References

[1] F. Aloul, I. Lynce, S. Prestwich, Symmetry breaking in local search for unsatisfiability, in: 7th International Workshop on Symmetry and Constraint Satisfaction Problems, Providence, RI, 2007.
[2] F.A. Aloul, I.L. Markov, K.A. Sakallah, FORCE: a fast and easy-to-implement variable-ordering heuristic, in: ACM Great Lakes Symposium on VLSI, ACM, 2003, pp. 116–119.
[3] F.A. Aloul, A. Ramani, I.L. Markov, K.A. Sakallah, Solving difficult SAT instances in the presence of symmetry, in: DAC, ACM, 2002, pp. 731–736.
[4] G. Audemard, G. Katsirelos, L. Simon, A restriction of extended resolution for clause learning sat solvers, in: 24nd Conference on Artificial Intelligence, AAAI'10, July 2010.
[5] G.E. Barton, Computational complexity in two-level morphology, in: Proc. of the 24th ACL, New York, 1986, pp. 53–59.
[6] K. Beesley, L. Karttunen, Finite State Morphology, CSLI Publications, 2003.
[7] A. Biere, M. Heule, H. van Maaren, T. Walsh (Eds.), Handbook of Satisfiability, IOS Press, 2009.
[8] G. Bonfante, J. Le Roux, Intersection optimization is NP-complete, in: Sixth International Workshop on Finite-State Methods and Natural Language Processing, FSMNLP 2007, Postdam Germany, 2007.
[9] J.R. Büchi, Weak second-order arithmetic and finite automata, Zeit. Math. Logik. Grund. Math. (1960) 66–92.
[10] J. Castaño, Two views on crossing dependencies, language, biology and satisfiability, in: 1st International Work-Conference on Linguistics, Biology and Computer Science: Interplays, IOS Press, 2011.
[11] José M. Castaño, Rodrigo Castaño, Variable and clause ordering in an fsa approach to propositional satisfiability, in: Béatrice Bouchou-Markhoff, Pascal Caron, Jean-Marc Champarnaud, Denis Maurel (Eds.), CIAA, in: Lecture Notes in Computer Science, vol. 6807, Springer, 2011, pp. 76–87.
[12] A. Darwiche, New advances in compiling CNF into decomposable negation normal form, in: ECAI, 2004, pp. 328–332.
[13] J. Dassow, G. Păun, A. Salomaa, Grammars with controlled derivations, in: G. Rozenberg, A. Salomaa (Eds.), Handbook of Formal Languages, vol. 2, Springer, Berlin, 1997.
[14] Shuhei Denzumi, Ryo Yoshinaka, Hiroki Arimura, Shin ichi Minato, Notes on sequence binary decision diagrams: relationship to acyclic automata and complexities of binary set operations, in: Jan Holub, Jan Žďárek (Eds), Proceedings of the Prague Stringology Conference 2011, Czech Technical University in Prague, Czech Republic, 2011, pp. 147–161.
[15] C.C. Elgot, Decision problems of automata design and related arithmetics, Trans. Amer. Math. Soc. (1961).
[16] J.V. Franco, M. Kouril, J.S. Schlipf, J. Ward, S. Weaver, M. Dransfield, W.M. Vanfleet, SBSAT: a state-based, BDD-based satisfiability solver, in: E. Giunchiglia, A. Tacchella (Eds.), SAT, in: Lecture Notes in Computer Science, vol. 2919, Springer, 2003, pp. 398–410.
[17] M.R. Garey, D.S. Johnson, Computers and Intractability: A Guide to the Theory of NP-Completeness, W. H. Freeman & Co., New York, NY, USA, 1979.
[18] C.P. Gomes, A. Sabharwal, B. Selman, Model Counting, in: Handbook of Satisfiability, in: Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009, pp. 633–654.
[19] Dick Grune, Ceriel J.H. Jacobs, Parsing Techniques — A Practical Guide, Springer, 2010.
[20] T. Hadzic, E.R. Hansen, B. O'Sullivan, On Automata, MDDs and BDDs in Constraint Satisfaction, 2008.
[21] P. Hansen, B. Jaumard, Algorithms for the maximum satisfiability problem, Computing 44 (April) (1990) 279–303.
[22] John E. Hopcroft, Jeffrey D. Ullman, Introduction to Automata Theory, Languages, and Computation, Adison-Wesley Publishing Company, Reading, Massachusetts, USA, 1979.
[23] George Karakostas, Richard J. Lipton, Anastasios Viglas, On the complexity of intersecting finite state automata and NL versus NP, Theoret. Comput. Sci. 302 (1–3) (2003) 257–274.
[24] Kimmo Koskenniemi, Finite-state parsing and disambiguation, in: COLING, 1990, pp. 229–232.
[25] Bernard Lang, Recognition can be harder than parsing, Comput. Intell. 10 (1992) 486–494.
[26] K. Lange, P. Rossmanith, The emptiness problem for intersections of regular languages, in: I. Havel, V. Koubek (Eds.), Mathematical Foundations of Computer Science 1992, in: Lecture Notes in Computer Science, vol. 629, Springer, Berlin, Heidelberg, 1992, pp. 346–354.
[27] Klaus-Jörn Lange, Klaus Reinhardt, Set automata, in: Combinatorics, Complexity and Logic; Proceedings of the DMTCS'96, Springer, 1996, pp. 321–329.
[28] H.R. Lewis, C.H. Papadimitriou, Elements of the Theory of Computation, 2nd ed., Prentice Hall PTR, Upper Saddle River, NJ, USA, 1997.
[29] V.W. Marek, Introduction to Mathematics of Satisfiability, Chapman and Hall/CRC, 2010.
[30] C. Muise, J.C. Beck, S. McIlraith, Fast d-DNNF Compilation with sharpSAT, 2010.
[31] G. Satta, Recognition of linear context-free rewriting systems, in: ACL, 1992, pp. 89–95.
[32] I. Schiering, W. Thomas, Counter-free automata, first-order logic and star-free expressions, in: Developments in Language Theory II, Magdeburg, Germany, 1995, pp. 166–175.
[33] J.P. Marques Silva, K.A. Sakallah, Grasp—a new search algorithm for satisfiability, in: Proceedings of the International Conference on Computer-Aided Design, 1996, pp. 220–227.
[34] C. Sinz, A. Biere, Extended resolution proofs for conjoining BDDs, in: Proc. of the 1st Intl. Computer Science Symp. in Russia, CSR 2006, St. Petersburg, Russia, June 2006, pp. 600–611.
[35] P. Tapanainen, Applying a Finite-State Intersection Grammar, in: E. Roche, Y. Schabes (Eds.), Finite-State Language Processing, MIT Press, Cambridge, 1997, pp. 311–327.
[36] M. Thurley, sharpSAT - Counting Models with Advanced Component Caching and Implicit BCP, in: A. Biere, C.P. Gomes (Eds.), SAT, in: Lecture Notes in Computer Science, vol. 4121, Springer, 2006, pp. 424–429.
[37] A. Urquhart, Hard examples for resolution, J. ACM 34 (1) (1987) 209–219.
[38] M. Vardi, Logic and Automata: A Match Made in Heaven, in: J. Baeten, J. Lenstra, J. Parrow, G. Woeginger (Eds.), Automata, Languages and Programming, in: LNCS, vol. 2719, Springer, 2003, pp. 64–65.
[39] M.Y. Vardi, P. Wolper, Automata-Theoretic techniques for modal logics of programs, J. Comput. Syst. Sci. 32 (April) (1986) 183–221.
[40] N.R. Vempaty, Solving constraint satisfaction problems using finite state automata, in: AAAI, 1992, pp. 453–458.
[41] D. Weir, A geometric hierarchy beyond context-free languages, Theoret. Comput. Sci. 104 (2) (1992) 235–261.
[42] A. Yli-Jyrä, Contributions to the theory of finite-state based linguistic grammars, Ph.D. Thesis, Electronic Publications at the University of Helsinki, 2005.
[43] Anssi Yli-jyr, Schematic finite-state intersection parsing, in: Short Papers Presented at the 10th Nordic Conference of Computational Linguistics, NODALIDA-95, 1995.
[44] S. Yu, Q. Zhuang, K. Salomaa, The state complexities of some basic operations on regular languages, Theoret. Comput. Sci. 125 (2) (1994) 315–328.