

A HARDWARE/SOFTWARE APPROACH TO ACCELERATE BOOLEAN SATISFIABILITY

Iouliia Skliarova

Department of Electronics and Telecommunications,

Aveiro University, IEETA

3810-193 Aveiro, Portugal

iouliia@ua.pt

António B. Ferrari

ferrari@ieeta.pt

Abstract. *This paper proposes a new algorithm for solving the Boolean satisfiability (SAT) problem. On the basis of this algorithm a software/reconfigurable hardware SAT solver was designed, implemented and compared to a similar realization of the Davis-Putnam-like method. The satisfier suggested uses an application-specific approach, thus an instance-specific hardware compilation is completely avoided.*

1 Introduction

The satisfiability (SAT) problem consists in finding an assignment of values to variables that makes the respective Boolean formula to evaluate to '1'. Usually, the formula is presented in *conjunctive normal form* (CNF). A CNF is a conjunction of a number of clauses, being each clause a disjunction of some literals where a literal corresponds either to a variable or to its negation. For example, the formula $(\bar{x}_1 \vee \bar{x}_3)(x_2)(\bar{x}_2 \vee x_3)(\bar{x}_1 \vee \bar{x}_2)$ contains three variables and four clauses and is satisfied when $x_1=0$, $x_2=1$ and $x_3=1$. On the other hand the formula $(\bar{x}_1)(\bar{x}_2)(x_1 \vee x_2)$ is unsatisfiable.

The SAT problem has a lot of practical applications ranging from logic synthesis, placement, routing and testing of electronic circuits to scheduling, robotics, automatic text processing, etc. A good overview of possible applications can be found in [1]. It should be noted that SAT belongs to the class of NP-complete problems [2]. Thus the design and implementation of efficient algorithms is of great importance in many different areas.

Due to the fact that SAT is an NP-complete problem, the solving time of many instances depends exponentially on the size of the respective formula. As a consequence, it is practically impossible to solve some formulae on conventional computers. Recently various attempts to accelerate SAT solving with the aid of reconfigurable hardware have been performed. It became realistic because in reconfigurable devices we can design functional units optimized for certain types of operations, execute these operations in parallel and customize the memory organization to the required data sizes. Obviously, such implementations do not eliminate the effect of exponential growing of solving time but they do allow delaying this effect, permitting in such a way bigger formulae to be processed [1].

The remainder of this paper is organized as follows. In section 2 the algorithms that were considered and implemented are described in detail. In section 3 a brief overview of the related work in the field is given and the main drawbacks of the existing hardware SAT solvers are pointed out. Then, in section 4 our approach to solve SAT with the aid of reconfigurable hardware is presented. Experimental results on benchmark problems, including performance comparison, are included in section 5. Finally, concluding remarks are given in section 6.

2 Algorithms

We formulate a SAT problem over a ternary matrix \mathbf{T} by setting a correspondence between clauses and variables of a formula, and rows and columns of \mathbf{T} , respectively. If there exist m clauses and n variables then each element t_{ij} , $i=1, \dots, m, j=1, \dots, n$, of the matrix \mathbf{T} is equal to:

- '1' – if clause c_i contains variable x_j ;
- '0' – if clause c_i contains variable x_j with negation;
- '-' (*don't care*) – if clause c_i does not contain variable x_j ;

The problem of satisfying the Boolean formula is equivalent to finding a ternary vector \mathbf{v} , which is orthogonal to each row of the corresponding matrix \mathbf{T} . Two ternary vectors $\mathbf{v}=[v_1 \ v_2 \ \dots \ v_n]$ and $\mathbf{c}=[c_1 \ c_2 \ \dots \ c_n]$ are orthogonal if there exists i , $i=1, \dots, n$, such that either $v_i=0$ and $c_i=1$, or $v_i=1$ and $c_i=0$. If vector \mathbf{v} cannot be found then the formula is unsatisfiable. On the other hand, if vector \mathbf{v} exists then the zeros and ones in it correspond to those variables that must receive values one and zero respectively in order to satisfy the formula.

2.1 Interval-based algorithm

Suppose that each row of matrix \mathbf{T} specifies an interval in the Boolean space of variables x_1, x_2, \dots, x_n . Then the union of all such intervals defines an area in the search space, where a solution cannot certainly be found. The residual area of the search space contains a set of all solutions to the problem. Since each vector specifies an interval in the Boolean space, in the further discussion we will use the terms *vector* and *interval* interchangeably.

The proposed *interval-based algorithm* makes use of the mentioned above property. The algorithm successively analyzes all the rows of matrix \mathbf{T} one by one, constructing in such a way the solution. Initially, a full n -cube specifies the set of possible solutions. In order to find all vectors orthogonal to the first row \mathbf{c}_1 of \mathbf{T} , this row is subtracted from the full n -cube. Next, from the resulting set of intervals S the second row \mathbf{c}_2 is subtracted. The process continues until either the set of intervals S becomes empty (what means that the problem does not have any solution) or all the rows of matrix \mathbf{T} are considered (in this case the current set of intervals S contains all the solutions to the problem). Thus, after the row \mathbf{c}_i , $i=1, \dots, m$, is analyzed, the set S specifies all the vectors that are orthogonal to rows $\mathbf{c}_1, \dots, \mathbf{c}_i$ of matrix \mathbf{T} .

The operation of subtraction is defined as follows. If at some step the set S is composed of vectors $\mathbf{s}_1, \mathbf{s}_2, \dots, \mathbf{s}_k$, then by subtracting from S a row \mathbf{c}_i , $i=1, \dots, m$, we receive:

$$\mathbf{S} - \mathbf{c}_i = \begin{bmatrix} \mathbf{s}_1 - \mathbf{c}_i \\ \mathbf{s}_2 - \mathbf{c}_i \\ \dots \\ \mathbf{s}_k - \mathbf{c}_i \end{bmatrix}$$

The operation of subtraction for two vectors $\mathbf{s}_k=[s_{k1} \ s_{k2} \ \dots \ s_{kn}]$ and $\mathbf{c}_i=[c_{i1} \ c_{i2} \ \dots \ c_{in}]$ is defined as:

$$\begin{bmatrix} s_{k1} - c_{i1} & s_{k2} & \dots & s_{kn} \\ s_{k1} & s_{k2} - c_{i2} & \dots & s_{kn} \\ \dots & \dots & \dots & \dots \\ s_{k1} & s_{k2} & \dots & s_{kn} - c_{in} \end{bmatrix}$$

And finally, the result of subtraction for individual components s_{kl} and c_{il} is equal to:

- '1' - if $s_{kl} = '-'$ and $c_{il} = '0'$;
- '0' - if $s_{kl} = '-'$ and $c_{il} = '1'$;
- \emptyset - in all remaining cases.

It should be noted that if any component of a vector is equal to \emptyset , the vector is discarded.

Let us consider a simple example. Suppose a matrix \mathbf{T} presented in Fig. 1a is given. First of all, the row \mathbf{c}_1 is to be subtracted from the full 3-cube. The resulting set S specifies that the solution should lie in the upper plane of the 3-cube (see Fig. 1b). After subtracting the remaining row \mathbf{c}_2 from the set S we receive the solution depicted in Fig. 1c.

It is important that this algorithm allows to find all solutions to the problem, presenting them in a compact cylindrical form. Finding all or multiple solutions is required in some practical applications [3]. Nevertheless, the algorithm possesses a characteristic drawback. For some problem instances the size of the set S is increasing very fast. That is why special methods that limit the growing rate are applied.

Let us consider these methods. For that we need to define the operation of subsumption. A vector \mathbf{a} subsumes a vector \mathbf{b} if for all $i, i=1, \dots, n$, one of the following expressions is true: $a_i = '-'$ or $a_i = b_i$. Thus, if any vector of the set S subsumes a recently generated interval then this interval can be discarded. On the other hand, if a recently generated interval subsumes some vectors from the set S , then these vectors can be deleted from S . And, finally, if there exists a row in the matrix \mathbf{T} that subsumes an interval in S than this interval can be discarded. It should be noted however that even the application of the proposed methods makes the interval-based algorithm unacceptable for many instances of practical interest.

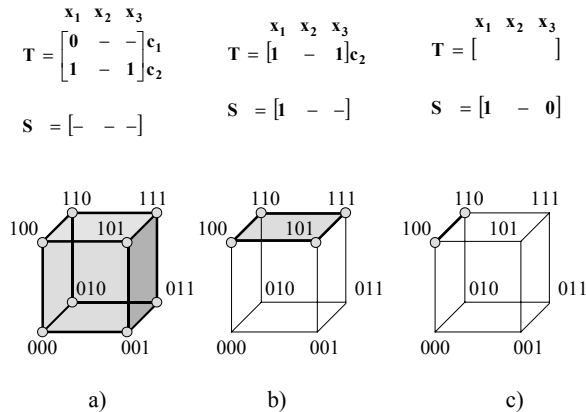


Figure 1: Interval-based algorithm.

2.2 Backtracking search algorithm

This method is based on the well-known Davis-Putnam [4] procedure. The search process is organized by implicitly traversing the space of all possible assignments of values to variables, and is usually represented by a decision tree. The root of the decision tree corresponds to a start point, where all the variables are unassigned. The other nodes represent situations that can be reached during the search process. These nodes are characterized by the respective partial assignments \mathbf{v} . If vector \mathbf{v} is orthogonal to all the rows of \mathbf{T} then the search process is terminated. In the opposite case, the search must proceed either forward (if there are no *conflicts*) or backward (if a *conflict* has appeared). A *conflict* means that the current partial assignment \mathbf{v} cannot satisfy the formula, so it is

meaningless to go deeper in the decision tree because it will not lead us to the solution. The respective conflict appearance conditions will be described later in this section.

In order to pass from one node of the decision tree to the subsequent one it is necessary to make a *decision*. The *decision* consists in choosing one unassigned variable and assigning a value to it (this variable is called a *decision variable*). The next decision variable (and its value) can be selected either statically or dynamically. In the former case all variables are initially pre-ordered using some criteria, while in the later case such a variable and a value are chosen that are more likely to help in satisfying the formula. We have implemented a dynamic selection based on a *maximum occurrence in clauses* heuristics. This heuristics allows to choose a variable that appears in the maximum number of rows of the matrix \mathbf{T} , and to assign to it a value that makes as many rows as possible to become orthogonal to the vector \mathbf{v} . Although this technique is simple, it has a significant effect on the convergence of the search.

After each decision, the current partial assignment may be extended by identifying all direct and transitive implications of the decision variable on other variables. It is accomplished by finding *unit clauses*, i.e. such rows in \mathbf{T} that have only one component k , $k=1, \dots, n$, with value a , being all the other components equal to *don't care*. Obviously, in order to make such rows orthogonal to the vector \mathbf{v} the respective component k of \mathbf{v} must get the value \bar{a} . If there exists a column in the matrix \mathbf{T} that does not contain the value a then this value is assigned to the corresponding component of vector \mathbf{v} . If there exists a column that does not contain zeros nor ones, it corresponds to a *dead variable* and can be deleted from the matrix. The variable is *dead* because it does not influence in any way the satisfiability of the formula.

As soon as some component k , $k=1, \dots, n$, of vector \mathbf{v} receives a value (by means either of decision or implication), all the rows of matrix \mathbf{T} that become orthogonal to \mathbf{v} are deleted together with the column k .

A conflict surges in the following cases:

- There exists a variable that is implied to two opposite values;
- The matrix \mathbf{T} has all the columns deleted but there are still remain some rows.

In both cases it is necessary to backtrack. For that the algorithm erases all the actions performed after the last decision and inverts the value of the current decision variable. If a conflict occurs again, the algorithm recedes to the most-recently assigned variable with unfinished revision and inverts its value. If backtracking beyond the first decision variable is attempted, it means that all possible assignments have been exhausted and there is no solution to the problem.

It should be noted that the computational complexity of the backtracking search algorithm is much lower than that of the interval-based algorithm. But what if we combine these two methods? Actually this approach gave quite promissory results, so let us describe it in detail.

2.3 Hybrid algorithm

Suppose that the maximal size of the set S is limited by s^{max} . Then the interval-based algorithm would only be able to process m' rows of the matrix \mathbf{T} ($m' \leq m$). If s^{max} is a sufficiently small number, then all the required computations will be executed very fast (including the detection of subsumed and subsuming vectors because in case of a big s^{max} this operation turns out to be quite time-consuming). If all the rows of \mathbf{T} get processed (what is actually impossible for practical problems) then the solution is found and the search is stopped. In the opposite case, the elements of the set S form so-called *favorites list*, which is similar to *tabu list* except of having a contrary meaning. A tabu list is usually

composed of some *forbidden* combinations of input variables, while a favorites list contains only *permitted* partial assignments.

After that we apply the backtracking search algorithm described in section 2.2. At each node of the decision tree we verify if the current partial assignment \mathbf{v} is consistent with the favorites list. It is consistent if \mathbf{v} is included in at least one interval \mathbf{s}_k , $k=1, \dots, s^{max}$, i.e. if \mathbf{s}_k subsumes \mathbf{v} . If this is true the search process proceeds as usual. In the opposite case it becomes clear that the current branch of the decision tree will not lead to the solution, so the algorithm should backtrack. As a result, so-called *premature backtracking* is performed. Although this technique is very simple, it has a great influence on the execution time, because it allows to reduce essentially the number of visited nodes in the decision tree.

3 Related work

During the last five years many attempts to accelerate SAT solving with the aid of reconfigurable hardware have been performed [3, 5-10]. The majority of researches implement algorithms derivative from the Davis-Putnam procedure [4], with the exception of the work [7] where a PODEM-based algorithm is realized. It should be noted that practically all authors apply an *instance-specific approach*, i.e. they generate a customized hardware circuit for each Boolean formula to be considered. In this case the total problem solving time is equal to “hardware circuit generation time” + “FPGA configuration time” + “actual execution time”. A direct mapping of a formula to functional components allows to execute some operations in parallel (such as evaluating all the clauses at once), permitting in this way performance to be increased. The main drawback of this approach is that the time of hardware compilation is very significant, and in many cases it is much bigger than the actual execution time, thus downplaying any benefit of the use of fast hardware. That is why some researches [3, 5, 6] propose to employ hardware SAT solvers only for instances with a large volume of input data, to which a long execution time is inherent. More recently, some attempts have been performed [8, 9] that avoid instance-specific placement and routing, reducing in such a way the hardware compilation time.

Another important issue affecting any algorithm implemented on reconfigurable hardware is related to the capacity of the hardware platform. What to do if the formula does not fit into a chosen device? Some authors [3, 5, 6] propose to employ several FPGAs by applying methods of multi-FPGA partitioning. However, as it was pointed out in [3], the resulting circuits require a lot of wiring resources, reducing very significantly FPGA utilization rate. That is why the latest publications [9, 10] suggest to partition the problem between software and reconfigurable hardware.

4 Architecture of software/hardware SAT solver

In order to overcome the main drawback inherent to all existing hardware SAT solvers we propose to apply an *application-specific approach* instead of using an *instance-specific* one. So, the objective is to design such a circuit that can be reused for different problem instances.

The resulting architecture is depicted in Fig. 2. We have implemented two versions of the circuit. The first one executes the backtracking search algorithm presented in section 2.2 and the second one corresponds to the hybrid algorithm described in section 2.3.

The matrix data is maintained in two memory blocks (*Mat_rows* and *Mat_columns* in Fig. 2). As the names indicate, the first block keeps \mathbf{T} , while the second one stores \mathbf{T}^T (a

transpose of \mathbf{T}). Obviously, keeping two blocks of memory requires more hardware resources, however, it allows to access any row and column of \mathbf{T} in one clock cycle. The matrices themselves are not modified during the search process. All possible changes (such as deleting rows and columns) are reflected in the *registers* (see Fig. 2). The current partial assignment \mathbf{v} is also kept in a register. The remaining blocks in Fig. 2 have the following designation. The central *control unit* executes the respective algorithm. The *ALU* is used to perform basic calculations (such as counting the number of ones in a vector, checking two vectors for orthogonality, etc.). The *stack* memory provides support for the backtracking process (i.e. for each node of the decision tree it keeps the corresponding values of the registers).

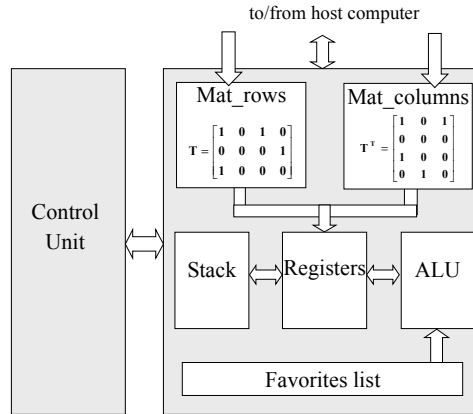


Figure 2: Architecture of hardware SAT solver.

The maximum dimension of memory block *Mat_rows* is 256x256. Since each matrix element is encoded by two bits, the circuit can process ternary matrices with dimensions $\leq 256 \times 128$. For the hybrid algorithm a favorites list is maintained (see Fig. 2), which is based on RAM and has the maximum capacity of 64 entries.

The implementation is based on an ADM-XRC PCI board [11] containing one XCV812E Virtex Extended Memory FPGA [12]. It should be underlined that this FPGA is very well suited to the proposed architecture because it includes a lot of additional RAM blocks (corresponding to the capacity of more than 1 million bits) that we can use in order to store the matrices and the favorites list. The first circuit runs at 30.5 MHz and the second one at 32.8 MHz.

The second important problem pointed out in section 3 is the available capacity of FPGA. The satisfier depicted in Fig. 2 supports only *limited* matrix dimensions, while practical problems require to process matrices with much bigger dimensions. Because of that we suggest to partition the problem between software and reconfigurable hardware. However our partitioning is different from that proposed in [9]. We assign to hardware only sub-problems that appear at various levels in the decision tree and satisfy the constraints imposed by the circuit (such as the maximum allowed matrix dimensions). The sub-problems that do not respect the constraints are initially processed in a software application implementing the same algorithm. As a result, the upper part of the decision tree is handled in software, while the bottom part is treated in hardware. Each time the FPGA is activated, it must receive from software the corresponding matrix data (and the favorites list contents in case of the hybrid algorithm). If the FPGA finds the solution, it is dispatched to the host computer and the search process is stopped. Otherwise, the control is just returned to software, which continues to traverse the decision tree eventually reaching some other node where the respective matrix dimension fall within the constraints.

5 Experimental results

In order to estimate the effectiveness of the proposed approach, a number of experiments on benchmark problems from DIMACS [13] have been conducted. Tables 1 and 2 contain the results of solving the *Pigeon hole* problem with the aid of two architectures. The first architecture implements a backtracking search algorithm both in software and in hardware, and the second one is based on the hybrid algorithm. We will refer to these architectures as *s/h_B* and *s/h_H* respectively.

The first columns in both tables contain the problem instance name; the second ones keep the initial matrix dimensions. The right-hand columns in Tables 1 and 2 indicate how many times the FPGA was activated for each problem instance. It can be seen that the instances *hole6* and *hole7* are entirely solved by hardware, while the remaining problems need to be initially processed in software.

In our case the total time required to solve a problem is equal to:

$$t_{\text{total}} = t_{\text{config}} + t_{\text{sw}} + t_{\text{hw}}$$

t_{config} denotes the FPGA configuration time. It should be noted that when a series of instances is to be solved, t_{config} might be omitted because the FPGA is configured only once, and after that can be reused for any instance. t_{sw} is the time of solving a part of the problem (the upper part of the decision tree) in software. t_{hw} includes the time required for both solving a part of the problem (the bottom part of the decision tree) in the reconfigurable hardware SAT solver and all the communications between the host computer and the FPGA. The software part for all the experiments was executed on an AMD Athlon/1GHz/256MB running Windows2000 and the hardware part was performed in an ADM-XRC board attached to the host computer via the PCI bus.

We did a comparison of our results with GRASP [14], which is one of the most efficient software satisfiers. The GRASP was executed on the same platform with the options *+dB +S500* for *hole6-hole9* instances and *+dB +S5000 +T5000* for *hole10* instance. The speedup resulting from our approach is given by $t_{\text{GRASP}}/t_{\text{total}}$. As can be seen from Tables 1 and 2, the architecture *s/h_H* gives better results for big-dimension problems, while for small-dimension problems the performance of both architectures is similar.

Table 1: Experimental results for the solver *s/h_B*.

Instance	m * n	t _{GRASP} (s)	t _{config} (s)	t _{sw} (s)	t _{hw} (s)	t _{total} (s)	Speedup	Matrix transfers
<i>Hole6</i>	133 * 42	0.13	0.37	0.0050	0.0138	0.3888	0.33	1
<i>Hole7</i>	204 * 56	3.004		0.0064	0.0203	0.3967	7.57	1
<i>Hole8</i>	297 * 72	38.995		0.0948	0.3002	0.7649	50.98	14
<i>Hole9</i>	415 * 90	421.43		0.9145	2.6479	3.9325	107.17	126
<i>Hole10</i>	561 * 110	3430.5		9.3003	26.529	36.199	94.77	1260

Table 2: Experimental results for the solver *s/h_H*.

Instance	m * n	t _{GRASP} (s)	t _{config} (s)	t _{sw} (s)	t _{hw} (s)	t _{total} (s)	Speedup	Matrix transfers
<i>Hole6</i>	133 * 42	0.13	0.36	0.0349	0.0074	0.4022	0.32	1
<i>Hole7</i>	204 * 56	3.004		0.0617	0.0113	0.4331	6.94	1
<i>Hole8</i>	297 * 72	38.995		0.2129	0.1443	0.7172	54.37	12
<i>Hole9</i>	415 * 90	421.43		1.3318	1.4476	3.1394	134.24	124
<i>Hole10</i>	561 * 110	3430.5		12.192	13.003	25.555	134.24	1246

6 Conclusion

The paper presents two architectures of SAT solver based on a rational collaboration between software and reconfigurable hardware. The first architecture implements a version of the Davis-Putnam algorithm, while the second one realizes the proposed hybrid algorithm. The suggested technique allows to overcome some important problems inherent to the majority of existing hardware satisfiers, and as a result, enables to achieve a significant speedup compared to purely software solutions.

An interesting possibility, which is the objective of future work, is to implement a non-chronological backtracking algorithm in hardware (the backtracking is called non-chronological if the algorithm is able to recede various levels in the decision tree by identifying and skipping those of its branches that cannot lead to a solution).

Acknowledgment

This work was sponsored by the grant FCT-PRAXIS XXI/BD/21353/99.

References

- [1] Gu, J., et al.: Algorithms for the Satisfiability (SAT) Problem: A Survey. DIMACS Volume Series on Discrete Mathematics and Theoretical Computer Science: The Satisfiability (SAT) Problem. American Mathematical Society, 1996.
- [2] Garey, M., R., Johnson, D., S.: Computers and Intractability: A Guide to the Theory of NP-Completeness. W.H.Freeman and Company, San Francisco, 1979.
- [3] Suyama T., et al.: Solving Satisfiability Problems Using Reconfigurable Computing. IEEE Transactions on VLSI Systems, vol. 9, no. 1, 2001, pp. 109-116.
- [4] Davis, M., Putnam, H.: A Computing Procedure for Quantification Theory. Journal of ACM, vol. 7, 1960, pp. 201-215.
- [5] Zhong P., et al.: Solving Boolean satisfiability with dynamic hardware configurations. In: R. W. Hartenstein, R. W., Keevallik, A. (Ed.): Field-Programmable Logic: From FPGAs to Computing Paradigm. Springer-Verlag. Berlin, Aug./Sept. 1998, pp. 326-335.
- [6] Platzner, M.: Reconfigurable accelerators for combinatorial problems. IEEE Computer, April 2000, pp. 58-60.
- [7] Abramovici, M., de Sousa, J., T.: A SAT solver using reconfigurable hardware and virtual logic. Journal of Automated Reasoning, vol. 24, nos. 1-2, February 2000, pp. 5-36.
- [8] Boyd, M., Larrabee, T.: ELVIS – a scalable, loadable custom programmable logic device for solving Boolean satisfiability problems. Proceedings 8th IEEE Int. Symp. on Field-Programmable Custom Computing Machines (FCCM), 2000.
- [9] De Sousa, J., et al.: A configware/software approach to SAT solving. Proceedings 9th IEEE Int. Symp. on Field-Programmable Custom Computing Machines, FCCM, May 2001.
- [10] Skliarova, I., Ferrari, A., B.: A SAT Solver Using Software and Reconfigurable Hardware. Proceedings of the Design, Automation and Test in Europe Conference (DATE), March 2002, p. 1094.
- [11] <http://www.alphadata.co.uk>
- [12] Xilinx. The programmable Logic Data Book. Xilinx, San Jose, 2000.
- [13] <http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/benchm.html>
- [14] Silva, J., M., Sakallah, K., A.: GRASP: a search algorithm for propositional satisfiability. IEEE Transactions on Computers, vol. 48, n^o. 5, May 1999, pp. 506-521.