# Eliminating redundancies in SAT search trees

Richard Ostrowski, Bertrand Mazure, Lakhdar Saïs and Éric Grégoire
CRIL - CNRS (FRE-2499) – Université d'Artois
Rue de l'université – SP 18
F-62307 Lens Cedex France
{ostrowski, mazure, sais, gregoire}@cril.univ-artois.fr

## Abstract

*Conflict analysis is a powerful paradigm of backtrack search algorithms, in particular for solving satisfiability problems arising from practical applications. Accordingly, most recent satisfiability solvers implement forms of conflict analysis, at least to some extent. In this paper, a branching criterion initially introduced by Purdom [16] is revisited and extended. Contrary to his author's a priori analysis, it is shown very efficient from a practical point of view in that it allows search trees in SAT solving to be pruned in a significant way while obeying an interesting time and space trade-off. More precisely, we show that redundancies during the search process can be avoided without adding new constraints explicitly. Moreover, the technique can be used not only to prune branches in the search tree, but also to derive implied literals. Extensive experimental results illustrate the feasibility and practical interest of this approach.*

Keywords: SAT, Davis and Putnam's algorithm, Boolean search and satisfiability

## 1 Introduction

SAT, i.e. checking the satisfiability of a Boolean formula in conjunctive normal form, is a canonical NP-complete problem [3] and a basic paradigm in many artificial intelligence domains like automated and non-monotonic reasoning and various computer science domains like VLSI design.

Recent impressive progress in the practical resolution of hard and large SAT instances allows real-world problems that are encoded in propositional clausal normal form (CNF) to be addressed (see e.g. [12, 11, 17, 21, 4, 2, 18, 20, 13]). While there remains a strong competition about building more efficient solvers dedicated to hard random $k$-SAT instances [7], there is also a real surge of interest in implementing powerful systems that solve difficult large real-world SAT problems. Many benchmarks have been proposed and regular competitions (e.g. DIMACS challenge 1993, Beijing 1996, SAT competition 2001-2003) are organized around these specific SAT instances, which are expected to encode structural knowledge, at least to some extent. For such problems, it is widely agreed that conflict analysis (also known as *learning* in the SAT and CSP domains) plays a fundamental role in efficient SAT solvers, in addition to other important features like the *restart* technique [9] and the use of efficient and ad-hoc data structures, like the implementation of the so-called *watched literals* [14]).

Conflict analysis allows both non-chronological backtracking and some relevant information (called *nogoods*) to be extracted and recorded in order to prune the search tree. Various approaches to conflict analysis have been developed in many AI domains, like truth maintenance systems, constraint satisfaction problems (CSP) and automated deduction. In the Boolean satisfiability (or SAT) domain, the first specific results about the integration of such a learning process have been proposed in [19] and [1]; they appear very promising for solving large and hard real-world problems efficiently, which often encode some structural knowledge.

In this paper, a branching criterion initially proposed by Purdom [16] is revisited and applied in an

original and very productive way. Purdom's criterion can be interpreted as a learning technique, at least in the sense that it can be used to avoid future conflicts (in the same search tree). It differs from most known learning techniques in the SAT domain in the fact that the redundant part of the search that is going to be avoided is known before the conflict occurs. As noted by Purdom himself, his criterion plays a role from a theoretical point of view only, since in order to be exploited it requires adding extras constraints obtained from the negation of a CNF sub-formula, which is resource-consuming to obtain. Indeed, in order to derive these additional constraints, a DNF formula is to be transformed into a CNF. This drawback was also noted by Gallo and Urbani [8] : "*Purdom's branching criterion succeeds in reducing the size of the search tree but a price must be paid. In fact, the formula must be transformed into the standard form of set of clauses, which might be quite costly*".

Contrary to this pessimistic a priori remark, it is shown in this paper that Purdom's principle can be exploited and provide an efficient technique. Moreover, a new way to derive implied literals is proposed and extensive experimental results illustrate the efficiency improvements obtained using this approach.

The paper is organized as follows. First, some technical background about SAT is briefly recalled. Then, search redundancy is outlined w.r.t. Purdom's branching criterion. After addressing its main practical *a priori* drawbacks, it is shown how it can be grafted in a powerful way to SAT solvers. Experimental performance is then illustrated on various instances from standard benchmarks libraries[6, 10]. The scope of the technique is then discussed before paths for further promising research are motivated.

## 2 Definitions and notations

SAT consists in checking the satisfiability of a Boolean formula in conjunctive normal form (CNF).

A *literal* is a positive ($l$) or negated propositional variable ($\neg l$). A CNF formula is a set (interpreted as a conjunction) of clauses, where a *clause* is a set (interpreted as a disjunction) of literals. On the contrary, a formula in disjunctive normal form (DNF) is a disjunction of a conjunction of literals. A set of propositional variables (resp. clauses ) occurring in a CNF formula $\Sigma$ is noted $var(\Sigma)$ (resp. $cl(\Sigma)$). An *interpretation* of a Boolean formula is a function $I : var(\Sigma) \rightarrow \{true, false\}$ represented as a set of literals (interpreted as a conjunction) : $I = \{l/I(l) = true\} \cup \{\neg l/I(l) = false\}$. If the interpretation is defined on all the variables appearing in the formula, then the interpretation is said *complete*; otherwise, the interpretation is called *partial*. A set of satisfied (resp. unsatisfied) literals in clause $c$ under a (partial) interpretation $I$ noted $sat(I, c)$ (resp. $unsat(I, c)$) is defined as $I \cap \{l/l \in c\}$ (resp. $I \cap \{\neg l/l \in c\}$). Similarly, the set of unassigned literals in $c$ under an (partial) interpretation is noted $unset(I, c)$ and is defined as $\{l/l \in c \text{ and } l \notin sat(I, c) \cup unsat(I, c)\}$. An interpretation $I$ satisfies a clause $c$ when $|sat(I, c)| \geq 1$. We say that a clause $c$ is *oversatisfied* by $I$ when $|sat(I, c)| \geq 2$. An empty set of clauses (noted $\{\}$) is interpreted as $true$ and an empty clause (noted $()$) is interpreted as $false$.

A *model* is an interpretation that satisfies all the clauses in the formula. Accordingly, SAT consists in finding a model of a CNF formula when such a model does exist or in proving that such a model does not exist.

We call $\Sigma \wedge x$ noted $\Sigma(x)$ a formula obtained from $\Sigma$ by assigning $x$ the truth-value *true*. Formally $\Sigma(x) = \{C | C \in \Sigma, \{x, \neg x\} \cap C = \emptyset\} \cup \{C \backslash \{\neg x\} | C \in \Sigma, \neg x \in C\}$. A size of clause $C_i$ ($|C_i|$) is equal to the number of its literals. A clause of size 2 (resp. 1) is called *binary* (resp. *unary*) clause. The *size* of a CNF formula is given by $|\mathcal{F}| = \sum_{i=1}^{m} |C_i|$. A *unit* (resp. *pure* or *monotone*) literal is a literal which appear in a unary clause (resp. its opposite literal does not appear in the formula). Unit propagation refer to the simplification of the formula by detecting and propagating unit literals only. This process is also known as Boolean Constraint Propagation (BCP).

## 3 Davis and Putnam's like search procedure

The most powerful complete algorithms for SAT still take their roots from the well-known basic Davis and Putnam backtrack search Procedure and the version proposed by Davis, Logemann and Loveland [5], which from now on will be named as DPL. DPL procedure is an enumeration algorithm that generates a binary search tree.

```
function DPL
Input: A CNF formula Σ
Result: TRUE if Σ is SAT, FALSE otherwise
begin
    Σ ← Simplify(Σ)
    if () ∈ Σ then return FALSE
    if Σ = {} then return TRUE
    l ← Decide(Σ)
    if DPL(Σ(l)) then return TRUE
    else return DPL(Σ(¬l))
end

function Simplify
Input: A CNF formula Σ
Result: Σ simplified by BCP
begin
    if ∃ a unit literal l then return Simplify(Σ(l))
    if ∃ a pure literal l then return Simplify(Σ(l))
    return Σ
end
```

**Algorithm 1. DPL algorithm**

The two key points of this algorithm are the simplification process and the branching rule used to select the next variable to be assigned:

1. *Simplify()* implements a Boolean constraint propagation technique. This function help us to get a simpler formula that remains equivalent with respect to satisfiability proving to the original one. Other polynomial-time simplification techniques can be grafted (e.g. monotone literal rule, resolution on binary clauses, etc.).

2. *Decide()* selects an unassigned variable as the next branching point. In general, this function relies on the use of elaborate heuristics. The variable is selected using some syntactical knowledge such as the length of clauses, the variables occurrence numbers, etc. For example, a heuristic selecting variables with maximum occurrences in clauses of minimal length (MOM) often proves efficient. More precisely, for each unassigned literal $l$, a score $f(l)$ is defined as follows : $f(l) = \sum_{l \in c} w(c)$ where $w(x) = 2^{-|C|}$. According to this heuristic, the next variable is then chosen according to the maximum value of

$$H(x) = f(x) * f(\neg x) + min(f(x), f(\neg x)).$$

## 4 Search redundancy and Purdom's branching criterion

Purdom's criterion can be stated as follows:

**Definition 1** *Let* $\Sigma$ *be a CNF formula, and* $l \in var(\Sigma)$*,* $\Sigma$ *can be equivalently rewritten as* $(l \vee \alpha) \wedge (\neg l \vee \beta) \wedge \gamma$ *where* $\alpha = \{c \backslash \{l\} / l \in c \text{ and } c \in \Sigma\}$*,* $\beta = \{c \backslash \{\neg l\} / \neg l \in c \text{ and } c \in \Sigma\}$ *and* $\gamma = \{c / \{l, \neg l\} \cap c = \emptyset \text{ and } c \in \Sigma\}$.

Considering the search tree of the classical DPL procedure, $\Sigma(l)$ and $\Sigma(\neg l)$ might have a nonempty intersection w.r.t. truth value assignments. More precisely, assignments satisfying both $\alpha$ and $\beta$ are searched twice, as illustrated in figure 1.

The following property is proposed in order to avoid such search redundancy.

**Property 1 (Purdom[16])** *Let* $\Sigma$ *be a CNF formula,* $l$ *be a branching literal occurring in* $\Sigma$ *such that* $\Sigma = (l \vee \alpha) \wedge (\neg l \vee \beta) \wedge \gamma$*, then* $\Sigma$ *is satisfiable iff* $\Sigma(l)$ *is satisfiable or* $\Sigma(\neg l) \wedge \neg\beta$ *is satisfiable.*

In the property above, $\neg\beta$ is a DNF formula. To derive the corresponding additional constraints, it has to be transformed into an equivalent CNF formula. Thus, a heavy price must be paid w.r.t. time and space complexity as noticed by Gallo and Urbani [8]. For this reason, these authors point out that the property is only to be exploited when $\beta$ is reduced to a simple clause (the literal $\neg l$ occurs only once in $\Sigma$). The negation of such a clause is a set of unit clauses. However, choosing such near-monotone literal as a branching literal might lead to unbalanced sub-trees. On the contrary, most of the efficient branching rules chose the next variable to assign among the most constrained and balanced variables.

**Remark 1** If $l$ is a pure literal in $\Sigma$ then $\beta = \emptyset$. Since an empty set of clauses is interpreted as $true$, $\neg\beta$ is $false$. Consequently, we can derive, the well known monotone literal property : if $l$ is a pure literal occurring in $\Sigma$, then $\Sigma$ is satisfiable iff $\Sigma \wedge l$ is satisfiable.

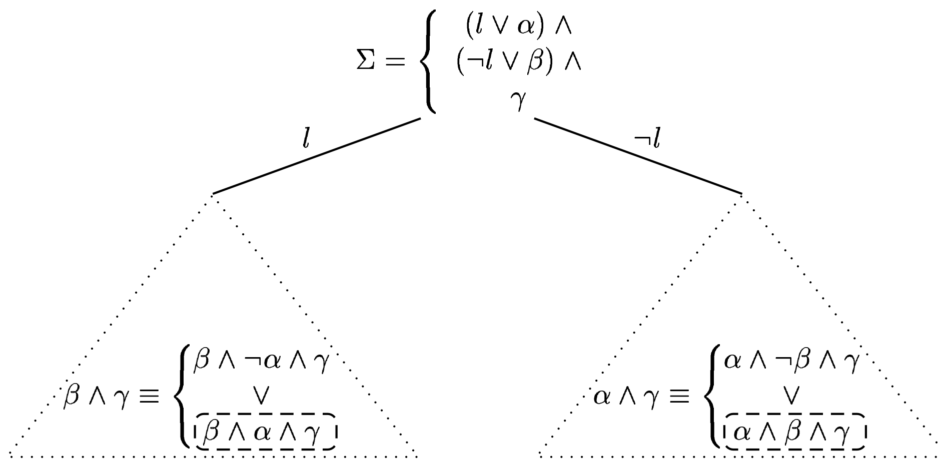

**Figure 1. Redundancy search in classical branching**

**Example 1** Let $\Sigma$ be a set of clauses :

$$\begin{array}{rcl}\Sigma & = & (a \vee b \vee c)\wedge \\ & & (a \vee d \vee \neg f) \\ & & (\neg a \vee d \vee \neg e)\wedge \\ & & (\neg a \vee f \vee g)\wedge \\ & & (\neg a \vee \neg h)\wedge \\ & & (\neg d \vee f)\end{array}$$

$\Sigma$ can be rewritten in an equivalent way as : $\Sigma = (a \vee \alpha) \wedge (\neg a \vee \beta) \wedge \gamma$, where $\alpha = (b \vee c) \wedge (d \vee \neg f)$, $\beta = (d \vee \neg e) \wedge (f \vee g) \wedge (\neg h)$ and $\gamma = (\neg d \vee f)$. The set of additional constraints are given by $\neg\beta = (\neg d \vee \neg f \vee h) \wedge (\neg d \vee \neg g \vee h) \wedge (e \vee \neg f \vee h) \wedge (e \vee \neg g \vee h)$.

The example above clearly illustrates the main drawback of Purdom's criterion, preventing its (direct) practical use.

## 5 Purdom's branching criterion revisited

In this section, it is shown how Purdom's criterion can actually be efficiently exploited in DPL. Let us consider the following example where some interesting information is deduced from a special kind of boolean function that appears in many classes of real world SAT instances.

**Example 2** Let $\Sigma$ be a set of clauses :

$$\begin{array}{rcl}\Sigma & = & (y \vee \neg x_1 \vee \neg x_2 \ldots \neg x_i \ldots \neg x_n)\wedge \\ & & (\neg y \vee x_1)\wedge \\ & & \ldots \\ & & (\neg y \vee x_i)\wedge \\ & & \ldots \\ & & (\neg y \vee x_n)\end{array}$$

One can see that $\Sigma$ encodes a Boolean function of the form: $y = \wedge(x_1, \ldots, x_i, \ldots, x_n)$. $y$ is called the output (dependent) variable and $x_i, 1 \leq i \leq n$ are the input (independent) variables.

From Property 1, the following two cases ca be derived:

- $x_i$ as a branching literal: $\Sigma$ is satisfiable iff $\Sigma(x_i)$ is satisfiable or $\Sigma(\neg x_i) \wedge (y \wedge x_1 \wedge \ldots x_{i-1} \wedge x_{i+1} \wedge \ldots x_n)$. Interestingly, a set of unit literals is derived.

- $y$ as branching literal: $\Sigma$ is satisfiable iff $\Sigma(y)$ is satisfiable or $\Sigma(\neg y) \wedge (\neg x_1 \vee \ldots \neg x_{i-1} \vee \neg x_{i+1} \vee \ldots \neg x_n)$. This last clause is redundant and belongs to $\Sigma(\neg y)$.

As shown in Example 2, some interesting information can be deduced when the branching process is done on independent variables. Truth value of the dependent variables are fixed when independent ones are assigned [15].

Let us now present how Purdom's branching criterion can be used in Davis and Putnam's like-procedures.

```
function DPL_R
Input: A cnf formula Σ, a stack s of sets of clauses
Result: TRUE if Σ is SAT, FALSE otherwise
begin
    Σ ← Simplify_R(Σ,s)
    if () ∈ Σ then return FALSE
    if Σ = {} then return TRUE
    l ← Decide(Σ) /* Σ = (l ∨ α) ∧ (¬l ∨ β) ∧ γ */
    if DPL_R(Σ(l),s) then return TRUE
    else
        push(s,β)
        if DPL_R(Σ(¬l),s) then return TRUE
        pop(s)
        return FALSE
    endif
end

function Simplify_R
Input: A cnf formula Σ, a stack s of sets of clauses
Result: Σ simplified by BCP and property 1
begin
    if ∃ unit literal l then return Simplify_R(Σ(l),s)
    if ∃ pure literal l then return Simplify_R(Σ(l),s)
    foreach β ∈ s do
        if ∀c ∈ β, c is satisfied then return {()}
        if Exactly one clause c of β is not satisfied
        then
            Σ ← Σ ⋃_{i=1}^{n} {l_i} s.t. l_i ∈ c ∩ unset(c)
            return Simplify_R(Σ,s)
        endif
    endforeach
    return Σ
end
```

**Algorithm 2. DPL_R algorithm**

First of all, when a new improvement to the DPL-like procedures or backtrack search is proposed, it should concern at least one of the following issues:

- simplification: how to simplify the formula?
- branching rule: how to select the next variable assign?
- conflict analysis: what should be done in case of conflict?

The integration of Property 1 in DPL improves the first and the third points.

From Property 1, we can see that the set of clauses obtained from $\neg\beta$ are added when $\Sigma(l)$ is unsatisfiable (i.e. when we backtrack to the decision literal $l$, and we start the search on the right branch). This is clearly shown in Algorithm 2 were clauses $\beta$ are pushed in the stack when starting the search on the right branch and the stack is updated when backtrack occurs. Thanks to the duality between SAT and UNSAT, we do not need to add $\neg\beta$, we just push the pointers to the clauses where $\neg l$ occurs in the stack. Indeed, when $\beta$ is satisfied (i.e. the clauses containing $\neg l$ are oversatisfied) the formula $\neg\beta$ is unsatisfied. So we just need to check if such clauses are oversatisfied by maintaining for each clause the number of satisfied literals. In other words the constraints $\neg\beta$ are considered without explicitly adding them to the database of formulas.

Another important feature is how to deduce literals from such constraints without adding them explicitly. Obviously, when all the clauses $c \in \beta$ are satisfied except one clause $c$, then the negation of its unassigned literals are implied.

Consequently, the function Simplify() is modified. At each step, we check whether there exists an oversatisfied set of clauses under the current assignments in the stack. If the answer is positive, then we backtrack; otherwise if we find a set of clauses that is satisfied except one clause $c$, then all literals $l \in unset(I, c)$ are implied and propagated. Fore more details on the implementation, see Algorithm 2.

## 6 Experimental results

In this section, experimental results about the application of Algorithm 2 to many classes of problems are presented. Let us stress that our goal in conducting these tests was simply to check the actual feasibility of our approach. We use a very simple DPL-like procedure without other important improvements that can be found in the currently most efficient Solver (such as the restart technique, other classical learning schemes, watched literals, etc.).

A comparison is conducted between a basic DPL procedure with and without the use of Property 1 on various classes of instances from the DIMACS and Sat competition library [6, 10]. These benchmark con-

| instance | $\#V$ | $\#C$ | SAT | *DPL* | | *DPL_R* | | | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | #Nodes | time | #Nodes | #Cuts | #Implied | time |
| 2bitcomp_5 | 125 | 310 | SAT | 75 | 0.00 | 75 | 0 | 15 | 0.00 |
| 2bitmax_6 | 252 | 766 | SAT | 2226 | 0.07 | 2194 | 60 | 0 | 0.07 |
| qg0-7 | 686 | 6816 | SAT | 3747681 | 1370 | 3030547 | 388947 | 339487 | 1217 |
| ii32b4 | 381 | 6918 | SAT | 1620 | 0.63 | 1560 | 35 | 2 | 0.65 |
| ii32d2 | 404 | 5153 | SAT | 68217 | 10.34 | 67338 | 261 | 94 | 11.41 |
| aim-100-1_6-no-1 | 100 | 160 | UNSAT | 13873294 | 87.38 | 5521257 | 226738 | 1934381 | 43.71 |
| aim-100-1_6-yes1-1 | 100 | 160 | SAT | 20 | 0.00 | 17 | 2 | 2 | 0.00 |
| aim-100-2_0-no-1 | 100 | 200 | UNSAT | 4475735 | 32.16 | 2047429 | 99946 | 796458 | 19.01 |
| aim-100-2_0-yes1-1 | 100 | 200 | SAT | 38965 | 0.33 | 16803 | 532 | 5021 | 0.17 |
| aim-200-1_6-yes1-1 | 200 | 320 | SAT | 25 | 0.00 | 23 | 0 | 6 | 0.00 |
| aim-200-2_0-yes1-1 | 200 | 400 | SAT | 1309265 | 23.78 | 547623 | 14212 | 106007 | 11.41 |
| aim-50-1_6-no-3 | 50 | 80 | UNSAT | 17611 | 0.06 | 4693 | 594 | 1895 | 0.02 |
| aim-50-1_6-yes1-3 | 50 | 80 | SAT | 54 | 0.00 | 51 | 2 | 7 | 0.00 |
| aim-50-2_0-no-1 | 50 | 100 | UNSAT | 3791 | 0.02 | 2598 | 196 | 700 | 0.01 |
| aim-50-2_0-yes1-1 | 50 | 100 | SAT | 358 | 0.00 | 212 | 14 | 49 | 0.00 |
| c499_gr_rcs_w5 | 1560 | 15777 | UNSAT | 239 | 0.25 | 239 | 0 | 8 | 0.25 |
| c499_gr_rcs_w6 | 1872 | 18870 | SAT | 5070 | 1.53 | 4445 | 51 | 673 | 1.62 |
| example2_gr_rcs_w5 | 2220 | 23144 | UNSAT | 19565 | 19.75 | 17411 | 2045 | 1491 | 19.80 |
| example2_gr_rcs_w6 | 2664 | 27684 | SAT | 787 | 0.13 | 787 | 0 | 1 | 0.14 |
| dlx1_c | 295 | 1592 | UNSAT | 36952631 | 662.53 | 24874151 | 1153 | 252691 | 563.77 |
| 2dlx_cc_mc_ex_bp_f2_bug011 | 4824 | 48160 | SAT | 3565 | 13.81 | 3565 | 0 | 0 | 23.06 |
| 2dlx_cc_mc_ex_bp_f2_bug012 | 6598 | 72227 | SAT | 20645 | 15.76 | 10751 | 26 | 0 | 13.07 |
| 2dlx_cc_mc_ex_bp_f2_bug014 | 6295 | 67003 | SAT | 273144 | 151.28 | 273144 | 0 | 0 | 247.92 |
| grid_05_05 | 48 | 81 | UNSAT | 1012 | 0.01 | 242 | 83 | 20 | 0.00 |
| grid_05_10 | 98 | 171 | UNSAT | 69884 | 1.25 | 10172 | 5223 | 585 | 0.20 |
| grid_10_10 | 198 | 361 | UNSAT | 42035094 | 1713.35 | 654006 | 284255 | 14674 | 27.94 |

**Table 1. DPL vs DPL_R**

sist of various SAT instances: problems from real-life applications and academic ones. In Table 1, a significant sample of our extensive experimentations is given, showing the obtained performance improvement. First of all, for all tested instances the size of the search tree (number of nodes) is less or equal to those obtained with basic DPL. Secondly, when the search is not or less redundant, little overhead can be seen with respect to CPU times.

On many classes , impressive improvement are obtained either in time and in the number of nodes. See for example, ...

Interestingly, on hard and large instances, we obtain very significant improvements in comparison to basic DPL procedure. However, the time limit[1] used in the experimentation is not sufficient to solve some of the instances. For example, `2bitadd_10` instance ($590$ variables and $1422$ clauses) which does not appear in Table 1, we can see a significant difference between the performance of DPL and DPL_R, in both Cpu times ($26112$ sec. vs $16230$ sec.) and number of nodes ($850.3$ million of nodes vs $433.1$ million of nodes).

[1]Time limit is fixed to 100 Cpu time minutes

## 7 Conclusions and future works

In this paper search redundancy is investigated thanks to a simple and efficient integration of Purdom branching criterion in basic DPL procedure. Our approach is shown to be very efficient from a practical point of view in that it allows search trees in SAT solving to be pruned in a significant way while obeying an interesting time and space trade-off. Particularly, we show that redundancies during the search process can be avoided without adding new constraints explicitly. Moreover, the technique can be used not only to prune branches in the search tree, but also to derive implied literals.

The results obtained in this paper, open interesting directions of research. We have particularly shown, that search redundancy of DPL like techniques might be avoided in a very simple way. We plan to investigate its integration in the most efficient SAT solvers such as Zchaff [14]. So harder and larger instances might

be addressed using recent advances in SAT. An other way to improve performance is to design specialized branching rules, in order achieve important cuts in the search tree. The example 2 show clearly, how chosen branching literals are important to deduce interesting informations. Finally, a good theoretical direction is to investigate the relationships between search redundancy and nogoods recording.

## 8 Acknowledgments

This work has been supported in part by the *IUT* of Lens and the *Région Nord/Pas-de-Calais* under the TACT-TIC Programme.

## References

[1] R. J. Bayardo Jr. and R. C. Schrag. Using csp look-back techniques to solve real-world sat instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 203–208, Providence (Rhode Island, USA), July 1997.

[2] A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using sat procedures instead of bdds. In *Design Automaton Conference, DAC'99*, 1999.

[3] S. A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, pages 151–158, New York (USA), 1971.

[4] J. Crawford and A. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Twelfth National Conference on Artificial Intelligence, AAAI'94*, 1994.

[5] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *Journal of the Association for Computing Machinery*, 5:394–397, 1962.

[6] Dimacs. Second Challenge on Satisfiability Testing organized by the Center for Discrete Mathematics and Computer Science of Rutgers University, 1993. `http://dimacs.rutgers.edu/Challenges/`.

[7] O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3–sat formulae. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, volume 1, pages 248–253, Seattle, Washington (USA), Aug. 4–10 2001.

[8] G. G. Gallo and G. Urbani. Algorithms for testing the satisfiability of propositional formulae. *Journal of Logic Programming*, 7(1):45–61, July 1989.

[9] C. P. Gomes, B. Selman, and H. Kautz. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence, (AAAI'98)*, pages 431–437, 1998.

[10] H. H. Hoos and T. Stützle. The satisfiability library (satlib). http://www.intellektik.informatik.tu-darmstadt.de/SATLIB/.

[11] H. Kautz and B. Selman. Planning as satisfiability. In *European Conference on Artificial Intelligence (ECAI'92)*, pages 359–363, Vienna, Austria, 1992.

[12] T. Larrabee. Efficient generation of test patterns unsing boolean satisfiability. *IEEE Transaction on CAD*, 11:4–15, 1992.

[13] F. Massacci and L. Marraro. Logical cryptanalysis as a sat-problem: Encoding and analysis of the u.s. data encryption standard. *Journal of Automated Reasoning*, 2000.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proceedings of the 38th Design Automation Conference (DAC'01)*, 2001.

[15] R. Ostrowski, E. Gregoire, B. Mazure, and L. Sais. Recovering and exploiting structural knowledge from cnf formulas. In *Proc. of the Eighth International Conference on Principles and Practice of Constraint Programming (CP'2002)*, pages 185–199, Ithaca (N.Y.), September 2002. LNCS 2470, Springer Verlag.

[16] P. W. Purdom. Solving satisfiability with less searching. *IEEE transactions on pattern analysis ans machine intelligence*, PAMI-6(4):510–513, July 1984.

[17] R. Reiter and A. Mackworth. A logical framework for depiction and image interpretation. *Artificial Intelligence*, 43(2):125–155, 1989.

[18] O. Shtrichman. Tuning SAT checkers for bounded model checking. In *Computer Aided Verification*, pages 480–494, 2000.

[19] J. P. M. Silva and K. A. Sakallah. Grasp – a new search algorithm for satisfiability. In *IEEE/ACM International Conference on Computer-Aided Design*, November 1996.

[20] J. P. M. Silva and K. A. Sakallah. Boolean satisfiability in electronic design automation. In *in Proceedings of the IEEE/ACM Design Automation Conference, DAC'00*, June 2000.

[21] H. Zhang and J. Hsiang. Solving open quasigroup problems by propositional reasoning. In *International Computer Symposium*, Hsinchu, Taiwan, 1994.