ELSEVIER

Available online at www.sciencedirect.com

SCIENCE DIRECT®

Discrete Applied Mathematics 130 (2003) 251–276

DISCRETE APPLIED MATHEMATICS

www.elsevier.com/locate/dam

# Equivalent literal propagation in the DLL procedure

Chu-Min Li

*LaRIA, Fac. de Mat. et d'Informatique, Université de Picardie Jules Verne, 5 rue du Moulin, Neuf, 80000 Amiens, France*

Received 8 June 2000; received in revised form 22 July 2001; accepted 3 March 2002

**Abstract**

We propose a simple data structure to represent all equivalent literals such as $l_1 \leftrightarrow l_2$ in a CNF formula $\mathscr{F}$, and implement a special look-ahead technique, called *equivalency reasoning*, to propagate these equivalent literals in $\mathscr{F}$ in order to get other equivalent literals and to simplify $\mathscr{F}$. Equivalent literal propagation remedies the ineffectiveness of unit propagation on equivalent literals and makes easy many SAT problems containing both usual CNF clauses and the so-called equivalency clauses (Ex-OR or modulo 2 arithmetics). Our approach is also compared with general CSP look-back techniques on these problems.


*Keywords:* Davis–Logemann–Loveland procedure; Equivalency reasoning; SAT

## 1. Introduction

Consider a propositional formula $\mathscr{F}$ in conjunctive normal form (CNF) containing $m$ clauses on a set of Boolean variables $\{x_1, x_2, \ldots, x_n\}$. The *satisfiability* (*SAT*) *problem* deals with testing whether clauses in $\mathscr{F}$ can all be satisfied by some consistent assignment of truth values (1 or 0) to variables. Solving a formula $\mathscr{F}$ generally means to find a satisfying assignment if $\mathscr{F}$ is satisfiable or to conclude that $\mathscr{F}$ is unsatisfiable. SAT is fundamental in many fields of computer science, electrical engineering, and mathematics. In fact, it was the first NP-Complete problem [5]. The Davis–Logemann–Loveland procedure (DLL) [8] is one of the best complete method to solve SAT, which is outlined in Fig. 1.

*E-mail address:* cli@laria.u-picardie.fr (C.-M. Li).

0166-218X/03/$ - see front matter © 2003 Elsevier B.V. All rights reserved.
PII: S0166-218X(02)00407-9

```
procedure DLL(F)
Begin
  if F is empty then return "satisfiable";
  F:=UnitPropagation(F);
  if F contains an empty clause then return "unsatisfiable";
  /* branching rule */
  select a variable x in F according to a heuristic H,
  if the call DLL(F∪{x}) returns "satisfiable" then
  return "satisfiable"
  else return the result of calling DLL(F∪{x̄});
End

procedure UnitPropagation(F)
Begin
  while there is no empty clause and a unit clause l exists
  in F, make l true, and simplify F (remove all clauses
  containing l and delete l̄ from all remaining clauses);
  return F;
End
```

Fig. 1. The DLL procedure.

DLL performs a backtracking depth-first search in a binary tree. Its basic strategy is to simplify $\mathscr{F}$ using unit propagation. When $\mathscr{F}$ contains no unit clauses, a branching rule is applied to add a unit clause so that unit propagation becomes possible. However, two sub-formulas usually should be separately solved in this case, making DLL exponential in general. So in order to speed up DLL, one should simplify $\mathscr{F}$ using more reasoning and avoid branching whenever possible.

Modern DLL procedures such as *Satz* [14] generally use a reasoning based on unit propagation to deduce implied literals in order to simplify $\mathscr{F}$ before branching. For example, if $\mathscr{F}$ contains no unit clause but two binary clauses $x_1 \vee x_2$ and $x_1 \vee \bar{x}_2$, unit propagation in $\mathscr{F} \cup \{\bar{x}_1\}$ leads to a conflict. So, $x_1$ is an implied literal and could be used to simplify $\mathscr{F}$.

Unfortunately, the reasoning based on unit propagation is not effective in all cases. In the above example, if $\mathscr{F}$ contains two slightly different binary clauses $\bar{x}_1 \vee x_2$ and $x_1 \vee \bar{x}_2$ instead of $x_1 \vee x_2$ and $x_1 \vee \bar{x}_2$, $x_1$ is not an implied literal any more so that $\mathscr{F}$ cannot be simplified in the same way.

The ineffectiveness of unit propagation for clauses such as $\bar{x}_1 \vee x_2$ and $x_1 \vee \bar{x}_2$ makes many SAT problems difficult for the DLL procedure. In particular, the DLL procedure is extremely inefficient in handling the so-called equivalency clauses (Ex-OR or modulo 2 arithmetics).

Note that the two clauses $\bar{x}_1 \vee x_2$ and $x_1 \vee \bar{x}_2$ mean $x_1 \leftrightarrow x_2$. If $\mathscr{F}$ contains these two clauses, all occurrences of $x_1$ (resp. $\bar{x}_1$) can be substituted by $x_2$ (resp. $\bar{x}_2$), so that $\mathscr{F}$, having one variable less, can be simplified further. For example, if $\bar{x}_1 \vee x_3$ and $x_2 \vee \bar{x}_3$ are clauses in $\mathscr{F}$, the substitution changes the first clause into $\bar{x}_2 \vee x_3$, so $x_2 \leftrightarrow x_3$, then all occurrences of $x_2$ (resp. $\bar{x}_2$) can be substituted by $x_3$ (resp. $\bar{x}_3$). Therefore, an equivalency relation can be propagated in order to simplify $\mathscr{F}$.

In this paper, we propose an approach for efficiently propagating an equivalency relation such as $x_1 \leftrightarrow x_2$ in $\mathscr{F}$, in order to deduce other equivalences and to

simplify $\mathscr{F}$ without branching. The approach makes easy many problems containing both equivalency clauses (called EQ part) and other CNF clauses (called CNF part) which were hard for a DLL procedure.

The paper is organized as follows. In Section 2, we propose a simple equivalent literal representation allowing fast equivalent literal propagation and efficient backtracking management. In Section 3, we discuss equivalency clauses and define equivalency reasoning to propagate equivalent literals. In Section 4, we present the implementation of equivalency reasoning in *Satz*. Section 5 illustrates equivalency reasoning and compares it with conflict-driven learning on two well-known classes of hard SAT problems. Experimental results are reported in Section 6. We discuss related work and conclude in Sections 7 and 8, respectively.

## 2. Equivalency relation representation

We use special equivalency relation representation to achieve efficient equivalent literal substitution, which is heavily used in our approach. Our equivalency relation representation is based on the following considerations.

In the implementation of a DLL procedure, the two operations below are essential:

`clause-to-literal`: get all literals of a given clause;

`literal-to-clause`: get all clauses containing a given literal.

When a clause becomes unitary, a DLL procedure should use the `clause-to-literal` operation to satisfy the (only) literal of the clause. When a literal is satisfied (a literal is satisfied if the DLL procedure branches on it or if it is the only literal of a unit clause), the DLL procedure should use `literal-to-clause` operation to satisfy all clauses containing the literal and to shorten all clauses containing its complement (the complement should be deleted from these clauses).

Let $l$ (with or without index) be a literal. If $l_1 \leftrightarrow l_2$, all occurrences of $l_1$ in $\mathscr{F}$ should be substituted by $l_2$. The substitution means that (i) all clauses containing $l_1$ should be modified to contain $l_2$ instead, and (ii) the data structure allowing to find all clauses containing $l_1$ should be integrated into the data structure allowing to find all clauses containing $l_2$, so that DLL can easily find all clauses containing $l_1$ and $l_2$ from $l_2$ after the substitution of $l_1$ by $l_2$.

However, if $l_1 \leftrightarrow l_2$ is true only after branching on some literal (the literal is assumed to be true), the substituted occurrences of $l_1$ or the changed clauses should be resumed when backtracking to the literal (the literal is assumed to be false), i.e. the changes in $\mathscr{F}$ due to $l_1 \leftrightarrow l_2$ should be undone. The substitution and resumption operations could be very time consuming when there are many backtrackings and many equivalent literals. For this reason, we do not physically modify any data structure storing $\mathscr{F}$ but introduce the notion of equivalent literal class in a DLL procedure to logically accomplish the same operations.

Roughly speaking, every literal in $\mathscr{F}$ belongs to one and only one class. All literals in a class are equivalent. An arbitrary literal in the class is chosen to represent the

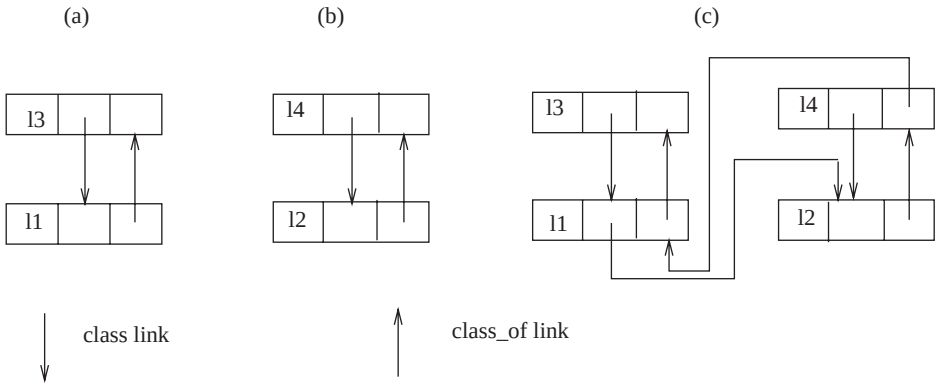


Fig. 2. (a) and (b): two literal classes, (c): the class after the uniting operation.

class. Then all clauses in $\mathscr{F}$ are considered and treated as a logical *or* of their literal classes. If $l_1$ becomes equivalent to $l_2$ after branching, the class of $l_1$ and the class of $l_2$ are united. If the new class is represented by $l_2$, all occurrences of $l_1$ in $\mathscr{F}$ are logically substituted by $l_2$.

When backtracking, the new class is simply split to resume the two old classes. Since $l_2$ is no more representative of $l_1$, the clauses containing $l_1$ are automatically resumed.

Every class has a complementary class. Thus, if literals $l_1$ and $l_3$ are in the same class, then $\bar{l}_1$ and $\bar{l}_3$ are in the complementary class. A class is united or split always at the same time as its complementary class. If $l$ is chosen to represent a class, then $\bar{l}$ is necessarily chosen to represent the complementary class.

We say that two literal classes are *distinct* if their intersection is empty and they are not complementary.

We use the data structure illustrated in Fig. 2 to represent literal classes and to achieve the uniting and splitting operations on them. Note that a literal represents its class if and only if its `class` link is empty (nil).

The uniting operation saves the added `class` and `class_of` links in a stack to be deleted by a splitting operation when backtracking. Note that the `class_of` link is not simply the inverse link of the `class` link. The `class` link allows to quickly get the class of a literal without necessarily going through all literals of the class, but one needs to get all literals of a class by the `class_of` link. The two functions *get_literal_class*(*a_literal*) and *get_literals_of_a_class*(*a_class*) defined in Fig. 3 based on the two links allow to achieve these considerations.

We can easily extend the `clause-to-literal` and `literal-to-clause` operations to literal classes. Given a clause, we get all its literals using the `clause-to-literal` operation and all its literal classes using the `get_literal_class` function. Given a literal, we get all corresponding clauses as follows. We first get the class of the literal using the `get_literal_class` function and then search for all literals of the class

```
function get_literal_class(a_literal)
Begin
    while a_literal.class ≠ nil do a_literal:=a_literal.class;
    return a_literal;
End

function get_literals_of_a_class(a_class)
Begin
    S:={a_class};
    while a_class.class_of ≠ nil do
    begin
        a_class:=a_class.class_of
        S:=S ∪ {a_class}
    end
    return S;
End
```

Fig. 3. From a literal to its class and vice versa.

using the `get_literals_of_a_class` function, from each of which we get the clauses using the `literal-to-clause` operation.

In the sequel, we also denote a literal class by its representative which is an arbitrary literal in the class.

Our approach for representing equivalent classes can be considered as an adaptation of the classical "Union/find" algorithm (see [6]) in a backtracking search. In particular, while the classical "union/find" algorithm essentially deals with the finding of a class, we also need to find all literals of a class.

Furthermore, we should efficiently maintain equivalent classes with numerous branchings and backtrackings. For this purpose, the splitting operation should be performed in exactly the inverse order as the uniting operation, i.e. after deleting the most recent `class` and `class_of` links, we should obtain the two old classes, see Fig. 2. The path compression heuristic as described in [6] is not used for three reasons: (i) a naive path compression would destroy existing class links so that the splitting operation could not easily give the two old classes, (ii) a sophisticated path compression would complicate the program but would not necessarily give good results, since every uniting or splitting implies a path compression, and (iii) the execution profile of our program shows that the numerous calls to the `get_literal_class` function without path compression only take negligible time.

## 3. Equivalency clauses and equivalency reasoning

Our main purpose is to complete unit propagation and to propagate equivalent literals in $\mathscr{F}$ to simplify $\mathscr{F}$ without branching. We apply this idea on SAT problems involving a special structure: equivalency clauses. An equivalency clause of length $k$ can be written as

$$l_1 \not\leftrightarrow l_2 \not\leftrightarrow \cdots \not\leftrightarrow l_k, \tag{1}$$

where the Ex-OR operator $\not\leftrightarrow$ is commutative and associative. The equivalency clause is equivalent to $2^{k-1}$ CNF clauses. For example, a binary equivalency clause is equivalent to two CNF clauses: $l_1 \vee l_2$ and $\bar{l}_1 \vee \bar{l}_2$, and a ternary equivalency clause is equivalent to four CNF clauses: $\bar{l}_1 \vee \bar{l}_2 \vee \bar{l}_3$, $\bar{l}_1 \vee l_2 \vee l_3$, $l_1 \vee \bar{l}_2 \vee l_3$, and $l_1 \vee l_2 \vee \bar{l}_3$.

An equivalency clause is satisfied if there are an odd number of satisfied literals, which is equivalent to the modulo 2 arithmetic equation:

$$\sum_{1 \leqslant i \leqslant k} l_i = 1 \bmod 2.$$

It is shown in the Reed–Muller algebra or in the Ex-OR logic that a complete Boolean algebra can be developed in terms of the Ex-OR and AND operators. When designing combinatorial logic circuits, a significant proportion of logic functions can be represented with fewer terms if Ex-OR gates are used. For a comprehensive presentation of Ex-OR logic and its simplification, see [30].

An equivalency clause can be negated with the following property:

$$\begin{aligned} \neg(l_1 \not\leftrightarrow l_2 \not\leftrightarrow \cdots \not\leftrightarrow l_k) &\equiv \bar{l}_1 \not\leftrightarrow l_2 \not\leftrightarrow \cdots \not\leftrightarrow l_k \\ &\equiv l_1 \not\leftrightarrow \bar{l}_2 \not\leftrightarrow \cdots \not\leftrightarrow l_k \\ &\cdots \\ &\equiv l_1 \not\leftrightarrow l_2 \not\leftrightarrow \cdots \not\leftrightarrow \bar{l}_k. \end{aligned} \tag{2}$$

Using the relation $x \not\leftrightarrow y \equiv \bar{x} \leftrightarrow y$, all operators $\not\leftrightarrow$ can be replaced by $\leftrightarrow$, and vice versa. Note that if $k$ is odd, we have $l_1 \not\leftrightarrow l_2 \not\leftrightarrow \cdots \not\leftrightarrow l_k \equiv l_1 \leftrightarrow l_2 \leftrightarrow \cdots \leftrightarrow l_k$.

Since all equivalency clauses of length $> 3$ can be simply transformed into ternary equivalency clauses by adding new variables, we only consider binary ($k = 2$) and ternary ($k = 3$) equivalency clauses in this paper. We use the $\leftrightarrow$ operator instead of $\not\leftrightarrow$ for our convenience and define five inference rules on them.

$$l_1,\ l_1 \leftrightarrow l_2 \leftrightarrow l_3 \vdash l_2 \leftrightarrow l_3, \tag{3}$$

$$l_1 \leftrightarrow l_1 \leftrightarrow l_2 \vdash l_2, \tag{4}$$

$$l_1 \leftrightarrow l_2 \leftrightarrow l_3,\ l_1 \leftrightarrow l_2 \leftrightarrow l_4 \vdash l_3 \leftrightarrow l_4, \tag{5}$$

$$l_1 \rightarrow (l_3 \leftrightarrow l_4), \bar{l}_1 \rightarrow (l_3 \leftrightarrow l_4) \vdash l_3 \leftrightarrow l_4, \tag{6}$$

$$l_1 \rightarrow (l_3 \leftrightarrow l_4), \bar{l}_1 \rightarrow (\bar{l}_3 \leftrightarrow l_4) \vdash l_1 \leftrightarrow l_3 \leftrightarrow l_4. \tag{7}$$

All these rules can be realized by a constant number of resolution steps after writing the equivalency clauses in CNF form. Rule 3 is realized in two resolution steps, rule 4 in one resolution step, rule 5 in six resolution steps (see Table 1 for the first four steps), rule 6 in two resolution steps, and rule 7 is simply two different ways to write the same thing.

We call the application of these rules in a formula $\mathscr{F}$ *equivalency reasoning*, which, when working on equivalency clauses, is more compact than resolution and avoids intermediate resolvents.

Table 1
Two ternary equivalency clauses to which rule 5 is applicable

| Clause C1 | | | | | Clause C2 | | | | | Resolvent of C1 and C2 | | | | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| $x_1$ | $\vee$ | $x_2$ | $\vee$ | $x_3$ | $\bar{x}_1$ | $\vee$ | $x_2$ | $\vee$ | $\bar{x}_4$ | $x_2$ | $\vee$ | $x_3$ | $\vee$ | $\bar{x}_4$ |
| $\bar{x}_1$ | $\vee$ | $\bar{x}_2$ | $\vee$ | $x_3$ | $x_1$ | $\vee$ | $\bar{x}_2$ | $\vee$ | $\bar{x}_4$ | $\bar{x}_2$ | $\vee$ | $x_3$ | $\vee$ | $\bar{x}_4$ |
| $\bar{x}_1$ | $\vee$ | $x_2$ | $\vee$ | $\bar{x}_3$ | $x_1$ | $\vee$ | $x_2$ | $\vee$ | $x_4$ | $x_2$ | $\vee$ | $\bar{x}_3$ | $\vee$ | $x_4$ |
| $x_1$ | $\vee$ | $\bar{x}_2$ | $\vee$ | $\bar{x}_3$ | $\bar{x}_1$ | $\vee$ | $\bar{x}_2$ | $\vee$ | $x_4$ | $\bar{x}_2$ | $\vee$ | $\bar{x}_3$ | $\vee$ | $x_4$ |

The purpose of equivalency reasoning is to deduce all possible unit equivalency clauses (rule 4), binary equivalency clauses (rule 3, 5, and 6), and ternary clauses (rule 7). The deduced clauses are in turn used in the subsequent reasoning. For complexity reasons, we obviously do not include all possible rules to deduce equivalency clauses of length $\leqslant 3$. For example, the following rule:

$$l_1 \rightarrow (l_2 \leftrightarrow l_3 \leftrightarrow l_4), \bar{l}_1 \rightarrow (l_2 \leftrightarrow l_3 \leftrightarrow l_4) \vdash l_2 \leftrightarrow l_3 \leftrightarrow l_4$$

is not considered, since its application is neither effective nor efficient in our approach.

In practice, complementary literals in equivalency clauses are rewritten using property 2 to put forward identical literals. For example, one rewrites $l_1 \leftrightarrow \bar{l}_1 \leftrightarrow l_2$ in its equivalent form $l_1 \leftrightarrow l_1 \leftrightarrow \bar{l}_2$ before applying rule 4 to obtain $\bar{l}_2$. One also rewrites $l_1$, $\bar{l}_1 \leftrightarrow l_2 \leftrightarrow l_3$ into $l_1$, $l_1 \leftrightarrow \bar{l}_2 \leftrightarrow l_3$ to apply rule 3.

Note that an application of rule 4 to a ternary equivalency clause results in a unit propagation and the satisfaction of the clause, and an application of rule 5 makes one of the two involved equivalency clauses redundant and removed. So we have

**Proposition 1.** *Let $n$ be the number of distinct literal classes in $\mathscr{F}$ and $x$ be any literal class. Let $k$ be the number of ternary equivalency clauses containing $x$. If none of rules* 4 *and* 5 *is applicable in $\mathscr{F}$, then $k < n/2$.*

**Proof.** If $k \geqslant n/2$, then there are at least $n$ literal classes different from $x$ in these ternary clauses. One class occurs more than once. Either there is a clause such as $x \leftrightarrow l \leftrightarrow l$ to which rule 4 is applicable, or two clauses such as $x \leftrightarrow l \leftrightarrow l_1$ and $x \leftrightarrow l \leftrightarrow l_2$ to which rule 5 is applicable. □

## 4. The Implementation of equivalency reasoning

### 4.1. Preprocessing

Given an input $n$ variable and $m$ clause formula $\mathscr{F}$ in CNF form, we search for and add resolvents of length $\leqslant 3$ in $\mathscr{F}$, which implies among other things unit propagations and applications of rules 3, 4 and 5. We illustrate ternary resolvent searching in Fig. 4, where every ternary clause is clearly visited three times. The procedure is linear if a limited number of resolvents are added. Binary resolvent searching is similar.

```
Procedure ternary_resolvent_searching(F)
Begin
  For every variable x in F do
  Begin
      /* mark every element of the first set */
      for every ternary clause c containing x do
          mark the two literals of c different from x by c;
      /* collect all marked elements in the second set */
      for every ternary clause d containing x̄ do
          if a literal l of d is marked by a clause c,
          then write c and d as x ∨ l ∨ l1 and x̄ ∨ l ∨ l2
               and add the resolvent l ∨ l1 ∨ l2 into F;
      /* unmark all elements of the first set */
      unmark all marked literals;
  End.
End;
```

Fig. 4. Ternary resolvent searching based on linear time intersection of two sets.

Rule 3 is applied by unit propagation. Rules 4 and 5 are applied by adding resolvents. An equivalency clause in CNF form to which rule 4 is applicable is simply $l_1 \vee l_2$ and $\bar{l}_1 \vee l_2$. Since the preprocessing adds the resolvent $l_2$ into $\mathscr{F}$, rule 4 is applied.

The first two columns of Table 1 show two ternary equivalency clauses in CNF form to which rule 5 is applicable. A ternary resolvent is obtained from the first two CNF clauses in every line by annihilating $x_1$ and is given in the third column. Then 2 binary resolvents $x_3 \vee \bar{x}_4$ and $\bar{x}_3 \vee x_4$ are obtained from the four ternary resolvents by annihilating $x_2$, giving a binary equivalency clause. One of the two original ternary equivalency clauses as well as the four ternary resolvents become redundant and are removed. This is exactly the application of rule 5 (by resolution).

Obviously, a ternary equivalency clause is also considered and treated as a list of three distinct literal classes. A link is set from variables to ternary equivalency clauses so that all ternary equivalency clauses in which a given variable occurs can easily be obtained.

Fig. 5 sketches the preprocessing algorithm. Note that the preprocessing does not add the binary resolvent for two clauses such as $l_1 \vee l_2$ and $\bar{l}_1 \vee l_3$, nor the ternary resolvent for two clauses such as $l_1 \vee l_2$ and $\bar{l}_1 \vee l_3 \vee l_4$. If the preprocessing procedure adds no or few resolvents (this is often the case when preprocessing a hard instance), it can be executed in $O(n+m)$ time.

Now there is no more any unit clause and binary equivalency clause in $\mathscr{F}$. Rules 3, 4 and are completely covered by the preprocessing. Nevertheless, rules 6 and 7 are not covered by the preprocessing. Further equivalency reasoning is made possible by branching.

After the preprocessing, we always denote by $n$ and $m$ the number of distinct literal classes and the total number of CNF and ternary equivalency clauses in $\mathscr{F}$.

### 4.2. Literal equivalence driven reasoning

Apart from the preprocessing at the root, equivalency reasoning is always made possible by branching. So we implement the five inference rules defined in Section 3

```
procedure Preprocess(F)
Begin
for each literal l make l the representative of the class {l};
Repeat
   /* Rule 3 is applied by unit propagation */
   F:=UnitPropagation(F);
   let l1, l2, l3 and l4 be distinct class representatives;
   if l1 ∨ l̄2 ∈ F and l̄1 ∨ l2 ∈ F then
   begin
       unite the class of l1 and the class of l2;
       remove l1 ∨ l̄2 and l̄1 ∨ l2 from F;
   end

   /* Rule 4 is applied by the next line */
   if l1 ∨ l2 ∈ F and l̄1 ∨ l2 ∈ F then F := F ∪ {l2};

   /* Rule 5 is applied by the next two lines */
   if l1 ∨ l2 ∨ l3 ∈ F and l̄1 ∨ l2 ∨ l4 ∈ F then F := F ∪ {l2 ∨ l3 ∨ l4};
   if l1 ∨ l2 ∨ l3 ∈ F and l̄1 ∨ l2 ∨ l3 ∈ F then F := F ∪ {l2 ∨ l3};

   if l1 ∨ l2 ∈ F and l̄1 ∨ l2 ∨ l3 ∈ F then F := F ∪ {l2 ∨ l3};
   let C1 and C2 be two clauses,
   if C1 ⊂ C2 then F:=F-{C2}
until F contains an empty clause or no change happens in F;
if F contains an empty clause, then return ''unsatisfiable'';

/* inspecting all equivalency clauses */
Repeat
   let x1, x2, x3 be variables of class representatives,
   if x1 ∨ x2 ∨ x3 ∈ F and x̄1 ∨ x̄2 ∨ x3 ∈ F and x̄1 ∨ x2 ∨ x̄3 ∈ F
   and x1 ∨ x̄2 ∨ x̄3 ∈ F then remove the four clauses from F
   and add x1 ↔ x2 ↔ x3 into F;
   if x̄1 ∨ x̄2 ∨ x̄3 ∈ F and x̄1 ∨ x2 ∨ x3 ∈ F and x1 ∨ x̄2 ∨ x3 ∈ F
   and x1 ∨ x2 ∨ x̄3 ∈ F then remove the four clauses from F
   and add ¬(x1 ↔ x2 ↔ x3) into F
until no change happens in F;
return F;
End
```

Fig. 5. The preprocess subprocedure.

into a highly optimized DLL procedure called *Satz* [14]. Equivalency reasoning enhanced *Satz* is called *EqSatz* and is sketched in Fig. 6. Note that the unit propagation procedure is extended to work on equivalent literal classes as described in Section 2.

Rule 3 is integrated into unit propagation. After a branching and a unit propagation, the subprocedure called Set_Equivalences collects all equivalent literals produced by the unit propagation. Then *Equivalency_Reasoning* procedure sketched in Fig. 7 is executed from $\mathcal{S}$ and applies rules 4 and 5 to propagate these literal equivalences.

For every new literal equivalence $l_1 \leftrightarrow l_2$, Equivalency_Reasoning procedure searches for all ternary equivalency clauses containing both a literal in the class of $l_1$ (or $\bar{l}_1$) and a literal in the class of $l_2$ (or $\bar{l}_2$) to apply rule 4. Then the procedure compares all equivalency clauses (C1) containing a literal in the class of $l_1$ (or $\bar{l}_1$) with those (C2) containing a literal in the class of $l_2$ (or $\bar{l}_2$) to apply rule 5. Property 2 is used

```
Procedure EqSatz(F)
Begin
  if F is empty then return "satisfiable";
  F:=UnitPropagation(F)); S:={};
  S=Set_Equivalences(F, S); F:=Equivalency_Reasoning(F, S);
  if F contains an empty clause then return
  "unsatisfiable";

  /* branching rule */
  examine equivalent literal classes in F
  and apply rules 6 and 7 to add equivalency clauses;
  select a literal class x in F;
  if EqSatz(F ∪ {x}) returns "satisfiable" then return
  "satisfiable", otherwise return EqSatz(F ∪ {x̄})
End;

procedure Set_Equivalences(F, S)
Begin
  for all l1 ↔ l2 ∈ F or (l1 ∨ l̄2 ∈ F and l̄1 ∨ l2 ∈ F) do
    S := S ∪ {l1 ↔ l2};
  return S;
End;
```

Fig. 6. The DLL procedure *EqSatz*.

```
Procedure Equivalency_Reasoning(F, S)
Begin
repeat
  take (and remove) a binary equivalency clause l1 ↔ l2 from S;
  mark all ternary equivalency clauses containing a literal
  in the class of l1 or l̄1, for every marked ternary
  equivalency clause c, mark also its two variables that
  are not in the class of l1 or l̄1 by c;
  /* Rule 4 */
  for every ternary equivalency clause c containing a literal
  in the class of l2 or l̄2, and c is marked do
  begin
     rewrite it as l1 ↔ l2 ↔ l3 using property 2;
     F:=UnitPropagation(F ∪ {l3}); S:=Set_Equivalences(F, S);
  end;
  /* Rule 5 */
  for every ternary equivalency clause d containing
  a literal in the class of l2 or l̄2, and d is not marked do
  begin
     if one of the three variables in d is marked by
     a ternary equivalency clause c,
     then rewrite c and d as l1 ↔ l ↔ l3 and l2 ↔ l ↔ l4;
     if l3 = ¬l4 then add an empty clause into F
     else S := S ∪ {l3 ↔ l4};
  end;
  unite the class of l1 (l̄1) and the class of l2 (l̄2);
  unmark all marked ternary equivalency clauses and variables
until an empty clause is produced or S is empty;
End.
```

Fig. 7. The procedure equivalency_reasoning.

to rewrite equivalency clauses to put forward identical literals to apply the inference rules. Here we essentially compute the intersection of two sets in linear time: rule 5 is applied to C1 and C2 sharing a common literal $l$ neither in the class of $l_1$ ($\bar{l}_1$) nor in the class of $l_2$ ($\bar{l}_2$), which is similar to the ternary resolvent searching illustrated in Fig. 4.

Finally, the class of $l_1$ (resp. $\bar{l}_1$) and the class of $l_2$ (resp. $\bar{l}_2$) are united. Since the equivalency $l_1 \leftrightarrow l_2$ is true only after a branching, the new class has to be split when backtracking. The time spent for uniting and splitting operations is negligible in our approach. Note that if $\mathscr{S}$ contains $l_1 \leftrightarrow l_2$, $l_2 \leftrightarrow l_3, \ldots$, the procedure is actually optimized in the obvious way to treat all these equivalences at the same time.

A naive comparison of equivalency clauses to apply rule 5 would have complexity $O(k_1 * k_2)$, where $k_1$ (resp. $k_2$) is the number of equivalency clauses containing one of literals in the class of $l_1$ (resp. $l_2$) or $\bar{l}_1$ (resp. $\bar{l}_2$). The comparison executed in Equivalency_Reasoning has linear complexity $O(k_1 + k_2)$. Every application of rule 5 makes one ternary equivalency clause redundant which is then removed.

Note that $k_1 < n/2$ and $k_2 < n/2$ by Proposition 1. So the complexity of the comparison is bounded by $O(n)$. Set_Equivalences procedure works on binary CNF and equivalency clauses and can be executed in the worst case in $O(n+m)$ time.

If Equivalency_Reasoning procedure does not deduce anything from the equivalent literals collected by Set_Equivalences, i.e. neither rule 4 nor rule 5 is applicable, a ternary equivalency clause is visited at most three times, trying to apply rule 4 or rule 5 (one time for every variable in the clause). In this case, the complexity of Equivalency_Reasoning procedure is bounded by $O(n+m)$.

If $O(n)$ new equivalent literals are deduced by the procedure, its complexity is bounded by $O(n(n+m))$. This is really the most favorable case in our approach, since the input formula will easily be solved after deducing $O(n)$ new equivalent literals.

### 4.3. Deducing implications to apply rules 6 and 7

Rules 6 and 7 are naturally integrated in the branching rule of *Satz*. Given a free class $x$, *Satz* examines $x$ by, respectively, adding two unit clauses $x$ and $\bar{x}$ into $\mathscr{F}$ and makes two experimental unit propagations to see the impact of branching on $x$. Following this line, *EqSatz* performs an experimental equivalency reasoning after each experimental unit propagation to search for implications such as $x \rightarrow (l_1 \leftrightarrow l_2)$ or $\bar{x} \rightarrow (l_1 \leftrightarrow l_2)$, which enable the application of rules 6 and 7 to add new equivalency clauses into $\mathscr{F}$.

Like *Satz*, *EqSatz* tries to branch on the variable allowing to maximize the reduction of search space by taking equivalency reasoning into account and uses three functions to estimate the reduction of search space.

The first function is $nb\_fixed\_vars(\mathscr{F}_1, \mathscr{F}_2)$ giving the number of variables of $\mathscr{F}_2$ that are not in $\mathscr{F}_1$, and the second is $nb\_eq\_pairs(\mathscr{F}_1, \mathscr{F}_2)$ giving the number of equivalent literal pairs in $\mathscr{F}_1$ that are not equivalent in $\mathscr{F}_2$. The third function $nb\_binary\_clauses(\mathscr{F}_1, \mathscr{F}_2)$ ($nb\_bin\_cls(\mathscr{F}_1, \mathscr{F}_2)$ in short) is defined roughly to be the number of binary clauses in $\mathscr{F}_1$ that are not in $\mathscr{F}_2$. For more details about the

```
let F' and F'' be two copies of F
For each distinct literal class x do
Begin
  F' := UnitPropagation(F' ∪ {x});
  F'' := UnitPropagation(F'' ∪ {x̄});
  S:={}; S:=Set_Equivalences(F', S);
  F':=Equivalency_Reasoning(F', S);
  S:={}; S:=Set_Equivalences(F'', S);
  F'':=Equivalency_Reasoning(F'', S);
  if both F' and F'' contain an empty clause
  then return "F is unsatisfiable"
  else if F' contains an empty clause then x := 0, F := F''
  else if F'' contains an empty clause then x := 1, F := F'
  else
  begin
    S:={};
    /* Rule 6 */
    for all l3 ↔ l4 ∈ F' ∧ F'' do S := S ∪ {l3 ↔ l4};
    /* Rule 7 */
    for all l3 ↔ l4 ∈ F' and l̄3 ↔ l4 ∈ F'' do
       /* compare x ↔ l3 ↔ l4 with existing clauses
       to apply rule 5 */
       if there is a l5 such that l5 ↔ l3 ↔ l4 ∈ F
       then S := S ∪ {x ↔ l5}
       else if there is a l5 such that x ↔ l3 ↔ l5 ∈ F
            then S := S ∪ {l4 ↔ l5}
            else if there is a l5 such that x ↔ l4 ↔ l5 ∈ F
                 then S := S ∪ {l3 ↔ l5}
                 else F:=F ∪ {x ↔ l3 ↔ l4};
    F:=Equivalency_Reasoning(F, S);
    if F contains an empty clause
    then return "F is unsatisfiable";
    let w(l) denote the weight of the class of l,
    w(x) := nb_fixed_vars(F', F) + nb_eq_pairs(F', F) + nb_bin_cls(F', F)/2;
    w(x̄) := nb_fixed_vars(F'', F) + nb_eq_pairs(F'', F) + nb_bin_cls(F'', F)/2;
  end
End;
For each x do H(x) := w(x̄) * w(x) * 1024 + w(x̄) + w(x);
Branching on x such that H(x) is the greatest.
```

Fig. 8. The branching rule of *EqSatz*.

motivation of the three functions, see [15] in which we also gave a preliminary presentation of equivalency reasoning. Fig. 8 shows the branching rule of `EqSatz`.

All new literal equivalences such as $l_3 \leftrightarrow l_4$ which belong to both $\mathscr{F}'$ and $\mathscr{F}''$ are added into $\mathscr{F}$ and propagated. Every time we have new literal equivalences $l_3 \leftrightarrow l_4 \in \mathscr{F}'$ and $\bar{l}_3 \leftrightarrow l_4 \in \mathscr{F}''$, a ternary equivalency clause $x \leftrightarrow l_3 \leftrightarrow l_4$ is added into $\mathscr{F}$ and compared with existing ternary equivalency clauses to apply rule 5.

So the branching rule covers all possible applications of rules 3, 4 and 5. However, it does not cover all applications of rules 6 and 7. In fact, the branching rule entirely relies on experimental unit propagation to verify the implications necessary to apply the two rules, but unit propagation may miss some implications. Computing the transitive closure of the implications may remedy the situation but it is costly if it is repeated at every node of a search tree. Furthermore, an equivalency clause added by rule 6

or 7 when examining $y$ may enable an implication $x \rightarrow (l_1 \leftrightarrow l_2)$ for a variable $x$ examined before $y$, but the branching rule does not re-examine $x$ to obtain it for efficiency reasons.

The total number of CNF and ternary equivalency clauses in $\mathscr{F}$ is bounded by $O(n^2 + m)$ by Proposition 1. The branching rule examines each class $x$ using two unit propagations (complexity bounded by $O(n + m)$) and two equivalency reasonings (complexity bounded by $O(n(n^2 + m))$ if $\mathscr{F}$ contains $O(n^2)$ equivalency clauses). The application of rules 6 and 7 roughly corresponds to another equivalency reasoning. So the complexity of the branching rule is bounded by $O(n^2(n^2 + m))$.

## 5. Equivalency reasoning and conflict-driven learning

Conflict-driven learning is a look-back reasoning in a backtracking search, which consists in analyzing conflicts encountered during the search and in adding clauses to the existing clause database to prevent the solver from meeting the same conflicts.

We use two well-known examples to illustrate equivalency reasoning and compare it with conflict-driven reasoning.

### 5.1. Dubois formulas in DIMACS suite

We illustrate the application of rule 5 and the equivalent literal substitution by solving dubois*.cnf formulas in DIMACS [1] suite in linear time without branching. These formulas consist of $n$ variables and $2n/3$ ternary equivalency clauses. Table 2 shows an example for $n = 12$, where the first two columns of each line contain two equivalency clauses and the last column shows the deduced equivalent literals.

Applying rule 5 to the two equivalency clauses C1 and C2 in the first line we obtain $x_3 \leftrightarrow x_4$ shown in the third column. After logically substituting $x_3$ by $x_4$ in the second line by uniting the class of $x_3$ and the class of $x_4$ and applying again rule 5 we obtain $x_2 \leftrightarrow x_5$. Similarly, in the third line, we obtain $x_1 \leftrightarrow x_6$. However, by applying rule 5 to the last line we have $x_1 \leftrightarrow \bar{x}_6$, a contradiction.

All Dubois formulas in DIMACS suite can be proved unsatisfiable in linear time by deducing $n/3$ pairs of equivalent literals in this way. However, these formulas are hard for a DLL procedure such as Satz, because every branching reduces at most two ternary equivalency clauses (in the sense of rule 3).

In fact, in a Dubois formula, all variables have two occurrences in ternary equivalency clauses (two ternary occurrences). At a branching point, any free variable has 0, 1 or 2 binary occurrences. Consequently, it has 2, 1 or 0 ternary occurrences, respectively.

If the branching variable $x$ has one binary occurrence, e.g., $x \leftrightarrow l_2$, we have a chain, e.g., $x \leftrightarrow l_2$, $l_2 \leftrightarrow l_3, \ldots, l_{j-1} \leftrightarrow l_j$, where $j \geqslant 2$. Unit propagation after the branching stops at $l_j$ and reduces the eventual ternary occurrence of $l_j$ together with the ternary

[1] ftp://dimacs.rutgers.edu/pub/challenge/sat.

Table 2
Solving a Dubois formula in DIMACS suite by equivalency reasoning

| Clause C1 | Clause C2 | Applying rule 5 to C1 and C2 |
|---|---|---|
| $x_3 \leftrightarrow x_{11} \leftrightarrow x_{12}$ | $x_4 \leftrightarrow x_{11} \leftrightarrow x_{12}$ | $x_3 \leftrightarrow x_4$ |
| $x_2 \leftrightarrow x_{10} \leftrightarrow x_3$ | $x_5 \leftrightarrow x_{10} \leftrightarrow x_4$ | $x_2 \leftrightarrow x_5$ |
| $x_1 \leftrightarrow x_9 \leftrightarrow x_2$ | $x_6 \leftrightarrow x_9 \leftrightarrow x_5$ | $x_1 \leftrightarrow x_6$ |
| $x_7 \leftrightarrow x_8 \leftrightarrow x_1$ | $x_7 \leftrightarrow x_8 \leftrightarrow \bar{x}_6$ | $x_1 \leftrightarrow \bar{x}_6$ |

occurrence of $x$. Note that the variables between $x$ and $l_j$ cannot have any ternary occurrence since they have two binary occurrences.

If the branching variable $x$ has two binary occurrences, e.g., $x \leftrightarrow y$, and $x \leftrightarrow z$, unit propagation after the branching follows two chains. If the two chains form a cycle, no ternary equivalency clause is reduced. Otherwise at most one ternary equivalency clause is reduced at the end of each chain.

So if a DLL procedure should reduce all ternary equivalency clauses before reaching a conflict, $2^{n/3}$ branchings are needed to solve a Dubois formula, which is often the case for Satz, whose complexity is thus $O(2^{n/3})$.

Learning can be also used to solve all the Dubois formulas in linear time. For example, when solving the above formula in Table 2, after branching successively on $x_3$, $x_{11}$, $x_2$ and $x_9$ and assigning 0 to these branching variables, conflict-driven learning as presented in [1] gives the clause $\bar{x}_1 \vee \bar{x}_6$ among several resolvents. Backtracking on $x_9$, learning gives $x_1 \vee x_6$, so $x_1 \not\leftrightarrow x_6$, backtracking on $x_2$, learning gives $x_2 \not\leftrightarrow x_5$, and so on. The reader may find it helpful to really do the learning by hand.

So learning deduces the same "crucial" equivalent literals as equivalency reasoning. However, two reasons make it less efficient than equivalency reasoning on Dubois formulas. First, conflict-driven learning relies on branching to reach a conflict, while branching is not necessary for equivalency reasoning; second and more important, learning should record and manage many intermediate resolvents while equivalency reasoning directly deduces the required equivalent literals.

### 5.2. Pretolani formulas in DIMACS suite

We illustrate the application of rule 7 combined with rule 5 to solve Pretolani formulas in linear time even without branching.

A Pretolani formula is constructed from a graph. A graph is a finite set of vertices together with a finite set of edges joining pairs of these vertices. Loops or multiple edges are not allowed here. Let $G$ be a graph. One labels each edge of $G$ with a distinct boolean variable, then assigns a charge $charge(a)$ of 0 or 1 to each vertex $a$. The total charge $Charge(G)$ of $G$ is the sum modulo 2 of the charges assigned to the vertices of $G$. An equivalency clause using Ex-OR operator is then defined for each vertex mentioning exactly all variables attached to the vertex. The equivalency clause for vertex $a$ is negated iff $charge(a) = 0$.

Figs. 9 and 10 show a complete graph K4 and its associated equivalency clauses.

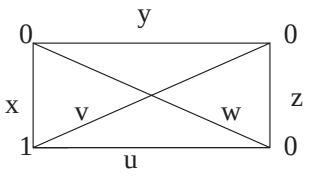


Fig. 9. A K4 graph.

$$\neg(x \not\leftrightarrow y \not\leftrightarrow w)$$
$$\neg(y \not\leftrightarrow z \not\leftrightarrow v)$$
$$\neg(z \not\leftrightarrow u \not\leftrightarrow w)$$
$$x \not\leftrightarrow u \not\leftrightarrow v$$

Fig. 10. 3-SAT generated from the K4 in Fig. 9.

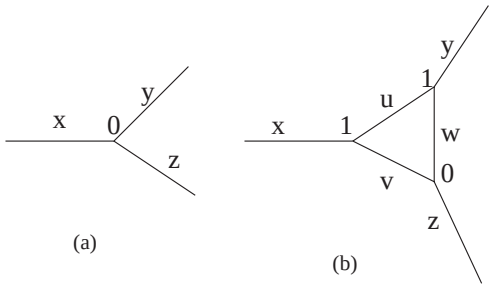


Fig. 11. Expand a vertex to a triangle (from (a) to (b)) or shrink a triangle to a vertex (from (b) to (a)).

The conjunction of all equivalency clauses is unsatisfiable iff $charge(G) = 1$ [27]. The graph-based propositional formulas were originally defined by Tseitin [26] to study the complexity of resolution.

One can expand a K4 graph as follows. Replace a vertex by a triangle and join the three vertices of the triangle by the three edges to the replaced vertex, then pick another vertex in the obtained graph and replace it by another triangle, and so on. A Pretolani formula of $n$ variables is defined on a graph of $2n/3$ vertices and $n$ edges expanded from a K4 in this way. All formulas are unsatisfiable.

Consider the last expanded triangle as (b) in Fig. 11, the three vertices give three equivalency clauses $x \leftrightarrow u \leftrightarrow v$, $y \leftrightarrow u \leftrightarrow w$ and $\neg(z \leftrightarrow v \leftrightarrow w)$ (recall that $x \leftrightarrow u \leftrightarrow v \equiv x \not\leftrightarrow u \not\leftrightarrow v$). One assumes $x$ to be true. As consequence, rule 3 applied to the first equivalency clause gives $u \leftrightarrow v$. Then one substitutes $u$ by $v$ in the second equivalency clause before applying rule 5 to obtain $y \leftrightarrow \bar{z}$. So $x \rightarrow (y \leftrightarrow \bar{z})$. Similarly $\bar{x} \rightarrow (y \leftrightarrow z)$. Applying rule 7, one obtains $\neg(x \leftrightarrow y \leftrightarrow z)$, corresponding to the vertex in (a). Note that the parity of the sum of the three vertex parities in (b) is equal to the parity of the vertex in (a).

Since a triangle can be shrunk to a vertex in constant time using equivalency reasoning, all Pretolani formulas can be proven unsatisfiable in linear time by successively shrinking all triangles.

If a DLL procedure encounters the triangle as (b) in Fig. 11 before reaching the first conflict, it can use learning to deduce the four CNF clauses equivalent to $\neg(x \leftrightarrow y \leftrightarrow z)$, with many resolution steps and intermediate resolvents, which shrinks the triangle. Other triangles can be shrunk when backtracking.

Therefore, a DLL procedure with conflict-driven learning can also solve Pretolani formulas in linear time in the best case. Once again the difficulty for learning is the management of numerous learned resolvents and the recognition of crucial resolvents if one does not want to record all resolvents. In practice it is rather difficult for a DLL procedure with learning to solve Pretolani formulas in linear time. See Table 4 for some experimental results on these formulas of the state-of-the-art DLL procedures with learning.

### 5.3. Discussion

The application of rules 5 and 7 illustrated in this section is typical in our approach. The effectiveness of equivalency reasoning heavily relies on these two rules when solving hard SAT problems containing equivalency clauses. It can be noticed that the application of rule 7 is in turn based on rule 5.

Rule 3 is simply a part of unit propagation. The two other rules 4 and 6 are less frequently applicable when solving a hard SAT problem. The application of rule 6 also relies on rule 5.

In our experimentation, equivalency reasoning always spends more than 80% of its time trying to apply rule 5, while the time spent for other rules is negligible, except for rule 7 to add new ternary equivalency clauses into the existing clause database.

According to Proposition 1, although equivalency reasoning adds new ternary equivalency clauses using rule 5, it guarantees that the total number of ternary equivalency clauses in $\mathscr{F}$ is always less than $n^2/2$ after applying rules 4 and 5.

## 6. Experimental evaluation

We evaluated *EqSatz* on a set of benchmark instances containing equivalency clauses. All experiments were conducted on a Macintosh G3 300 MHz with 96 Mb memory under Linux. The run time is expressed in seconds.

We also evaluated *EqSatz*−, which is *EqSatz* without rule 7, to show the impact of rule 7. As can be seen, although rule 5 is essential for equivalency reasoning, rule 7 is also important for the performance of *EqSatz* on hard problems such as DIMACS Pretolani formulas, DIMACS parity problem and Urquhart problem.

Although specialized for instances containing equivalency clauses, *EqSatz* is also one of the best solvers for other instances. In fact, if the input formula contains no equivalency clause, the call to Equivalency_Reasoning in *EqSatz* immediately returns and the overhead of *EqSatz* compared with Satz is essentially the call to Set_Equivalences

and is not important. For a general comparison of *EqSatz* with publicly available state-of-the-art complete SAT solvers, see satex [24] web site,[2] where it is shown that of 1303 structured SAT instances *EqSatz* solves 1237 instances on a 400 MHz Pentium II PC under Linux (the time limit is set to 10 000 s to solve an instance), compared with 1280 solved instances for Chaff (version zchaff) [22], 1260 for Relsat (version relsat-200) [1], 1241 for Sato [31], 1237 for Satz. Other tested solvers solve fewer instances than *EqSatz*.

### 6.1. Solving the challenge DIMACS 32-bit parity problem

*EqSatz* was originally motivated by the challenge DIMACS 32-bit parity problem formulated by Selman et al. [23] at IJCAI'97. To the best of our knowledge, *EqSatz* is the *only* procedure able to solve all the 10 par32* instances in reasonable time. It is shown in satex that *EqSatz* is also the fastest solver to solve the 10 par16* instances.

DIMACS parity problem is the SAT-encoding of the minimal disagreement parity problems contributed by Crawford and Kearns [7]. Informally the problem is the following: given a set of sample input bit vectors and a set of sample parities, find the bits of the input vectors on which the parities were computed. The problem contains a lot of equivalency clauses because of parity computing.

Selman et al. formulated the challenge and commented: "given the amount of effort that has been spent on the problem, any algorithm solving it will have to do something significantly different from current methods" [23]. We believe that equivalency reasoning is significantly different from other approaches to make a DLL procedure able to answer the challenge.

Table 3 shows the performance of *EqSatz* and Satz on the challenge problem. It also gives the corresponding number of backtrackings (*t_size*) which is the half of the number of recursive calls to *EqSatz* plus 1. As in the next tables, #cls and #eq_cls, respectively, denote the total number of clauses in the input CNF formula and the number of ternary equivalency clauses after the preprocessing, a ternary equivalency clause being counted as 4 clauses in #cls. As can be seen, all instances contain a large EQ part. For every solved formula, we also give the number (#g_eq) of ternary equivalency clauses generated using rule 7 at the root. #g_eq does not include ternary equivalency clauses generated below the root.

For each $k \in \{8, 16, 32\}$ there are 10 instances divided into two groups, par*k*-*i*-c.cnf and par*k*-*i*.cnf, for $1 \leqslant i \leqslant 5$. For $k \leqslant 16$ we give the average of a group for each item. When $k = 32$, the instances are too large to be solved by *Satz*. Without rule 7, *EqSatz* only solves two 32-bit parity instances. Note that for par32* instances, the number of generated ternary equivalency clauses at the root is slightly larger than the number of static ternary equivalency clauses.

### 6.2. Other experimental results

We use four other separate benchmarks involving equivalency clauses in the literature to evaluate the impact of equivalent literal propagation in a DLL procedure and to

[2] http://www.lri.fr/∼simon/satex.

Table 3
Run time (in seconds) and search tree size (t_size) of *EqSatz* and *Satz* on DIMACS challenge parity problem

| Instance | #var | #clause | #eq_cls | #g_eq | *EqSatz−* | *Satz* | | *EqSatz* | |
|---|---|---|---|---|---|---|---|---|---|
| | | | | | Time | Time | *t*_size | Time | *t*_size |
| par8-*i*-c | 75.8 | 277.2 | 46.8 | 0 | 0 | 0 | 1 | 0 | 0 |
| par8-*i* | 350 | 1120.4 | 46.8 | 33.4 | 0 | 0 | 2 | 0 | 0 |
| par16-*i*-c | 333 | 1328 | 256 | 265 | 0.5 | 2.7 | 1358 | 0.3 | 1.2 |
| par16-*i* | 1015 | 3342 | 256 | 272.4 | 0.5 | 9.6 | 924 | 0.2 | 1 |
| par32-1-c | 1315 | 5254 | 1097 | 1314 | — | — | — | 1133 | 3672 |
| par32-2-c | 1303 | 5206 | 1085 | 1336 | 3006 | — | — | 50 | 209 |
| par32-3-c | 1325 | 5294 | 1107 | 1185 | — | — | — | 3972 | 15123 |
| par32-4-c | 1333 | 5326 | 1115 | 1295 | — | — | — | 793 | 1488 |
| par32-5-c | 1339 | 5350 | 1121 | 1414 | — | — | — | 9265 | 38348 |
| par32-1 | 3176 | 10277 | 1097 | 1325 | — | — | — | 989 | 3089 |
| par32-2 | 3176 | 10253 | 1085 | 1346 | 6736 | — | — | 241 | 651 |
| par32-3 | 3176 | 10297 | 1107 | 1201 | — | — | — | 8899 | 23827 |
| par32-4 | 3176 | 10313 | 1115 | 1332 | — | — | — | 827 | 2885 |
| par32-5 | 3176 | 10325 | 1121 | 1455 | — | — | — | 11855 | 35133 |

compare the performance of *EqSatz* with five state-of-the-art DLL procedures and Heerhugo [11] on the same instances: Pretolani problem in DIMACS suite, Massacci's data encryption standard (DES) problem,[3] Biere et al.'s bounded model checking (BMC) problems,[4] and Urquhart hard examples for resolution.

The five state-of-the-art DLLs compared are Sato (version 3.2) [31], Grasp (version 1998, command line: sat-grasp +B2147483647 +C2147483647 +T2147483647 +S2147483647 +g20 +rt4 +dDLIS +V0 input-formula) [18], Relsat (version 1.1.2, command line: relsat 4 input-formula) [1], Chaff (Chaff2) [22] and Satz (version 214). Heerhugo is a breath-first solver using a very strong general-purpose reasoning adapted from Stalmarck's method [25].

Sato, Grasp, Relsat and Chaff use both look-ahead techniques, such as unit propagation and variable ordering heuristics for branching, and look-back techniques such as intelligent backtracking and learning, while Satz uniquely uses look-ahead techniques. So the comparison between *EqSatz* and Satz in the experimentation illustrates the impact of equivalent literal propagation and the comparison of *EqSatz* with Sato, Grasp, Chaff and Relsat might be considered as a comparison between look-back techniques and equivalent literal propagation on the instances involving equivalency clauses. Note that Chaff is a very recent and very efficient solver with a careful engineering of look-back and re-start techniques.

We use default options for these solvers unless otherwise specified. The time limit is set to 7200 s, except for Urquhart examples for which the time limit is 360 000 s (100 h). When a solver is stopped before the time limit because of memory shortage, its time is marked by "?".

[3] available from http://www.uni-koblenz.de/∼massacci.
[4] available from http://www.cs.cmu.edu/∼modelcheck.

Table 4
Run time on DIMACS pret* problem, each column (except the first) corresponds to an instance. The second column corresponds to instance pret300_40, the third to pret450_40, etc.

| #vars | 300 | 450 | 600 | 750 | 1500 | 3000 | 6000 | 9000 |
|---|---|---|---|---|---|---|---|---|
| #eq_cls | 200 | 300 | 400 | 500 | 1000 | 2000 | 4000 | 6000 |
| #g_eq | 93 | 138 | 183 | 230 | 462 | 925 | 1851 | 2776 |
| Grasp | 9 | 31 | 82 | 166 | 1742 | > 7200 | — | — |
| Sato | 28 | 37 | 102 | > 7200 | — | — | — | — |
| Relsat | 1 | 2 | 4 | 10 | 72 | 696 | > 7200 | — |
| Chaff | ? | 0 | 0 | 1 | 4 | 18 | ? | 169 |
| Heerhugo | 25 | 84 | 200 | 389 | 3618 | > 7200 | — | — |
| Satz | > 7200 | — | — | — | — | — | — | — |
| *EqSatz−* | 7 | 293 | > 7200 | — | — | — | — | — |
| *EqSatz* | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |

### 6.2.1. Performance on DIMACS pret* problem

*EqSatz* solves Pretolani formulas in empirically linear time, all the instances being solved within two backtrackings, though equivalency reasoning does not cover all applications of rules 6 and 7. The number of the ternary equivalency clauses generated at the root is roughly proportional to the size of the problem. Table 4 shows the performance of the 6 DLL procedures and Heerhugo. Note that though Sato, Grasp and Relsat are substantially faster than Satz on these instances, it seems that they still have an exponential behavior.

From the performance of *EqSatz−*, it can be seen that rule 7 is essential to solve Pretolani formulas.

### 6.2.2. Performance on DES instances

DES instances are contributed by Massacci [20]. These are SAT-encoding of cryptographic key search problem and contain few equivalency clauses from three rounds. All instances are satisfiable. We only report on three round instances here. The original instances involve a huge number of variables with no clauses. So we compact them by making unit resolution and pure literal elimination and renaming the variables to be contiguous. Table 5 displays the performance of the six DLLs and Heerhugo on the instances after simplification. *EqSatz* is the third fastest procedure to solve these instances and is substantially faster than Satz, illustrating the impact of equivalency reasoning to solve these instances even when there are very few equivalency clauses.

Rule 7 is not important for DES instances.

### 6.2.3. Performance on BMC instances

BMC problems are contributed by Biere et al. [2] and arise from (bounded) model checking. All instances are unsatisfiable. We select the most difficult barrel* and queueinvar* instances and the representative half longmult* instances.

Table 6 displays the performance of the seven solvers on barrel* instances. *EqSatz* is the fastest solver for these instances containing a large EQ part, and finds the inconsistency by equivalency reasoning without branching.

Table 5
Running time on DES instances. The name of each instance is preceded by "cnf-r3-"

| Instance | b1-k1.1 | b1-k1.2 | b2-k1.1 | b2-k1.2 | b3-k1.1 | b3-k1.2 | b4-k1.1 | b4-k1.2 |
|---|---|---|---|---|---|---|---|---|
| #var | 1461 | 1450 | 2855 | 2880 | 4255 | 4418 | 5679 | 5721 |
| #cls | 8966 | 8891 | 17857 | 17960 | 26778 | 27503 | 35817 | 35963 |
| #eq_cls | 48 | 48 | 96 | 96 | 144 | 144 | 192 | 192 |
| #g_eq | 0 | 0 | 0 | 0 | 0 | 0 | 5 | 0 |
| Sato | 871 | > 7200 | 3 | > 7200 | > 7200 | > 7200 | > 7200 | > 7200 |
| Grasp | 5446 | 3991 | 50 | 15 | 49 | 54 | 21 | 33 |
| Relsat | 1080 | 454 | 18 | 22 | 37 | 44 | 45 | 48 |
| Chaff | 9 | 8 | 0 | 0 | 0 | 1 | 3 | 1 |
| Heerhugo | 1917 | 1808 | 96 | 136 | 90 | 235 | 82 | 152 |
| Satz | > 7200 | > 7200 | 946 | 1468 | 32 | 101 | 357 | > 7200 |
| *EqSatz*− | 959 | 1014 | 1223 | 614 | 10 | 11 | 16 | 17 |
| *EqSatz* | 995 | 1023 | 1276 | 629 | 11 | 11 | 17 | 18 |

Table 6
Running time on BMC barrel* instances

| Instance | barrel5 | barrel6 | barrel7 | barrel8 | barrel9 |
|---|---|---|---|---|---|
| #var | 1407 | 2306 | 3523 | 5106 | 8903 |
| #cls | 5383 | 8931 | 13765 | 20083 | 36606 |
| #eq_cls | 870 | 1476 | 2310 | 3408 | 6408 |
| #g_eq | 30 | 60 | 0 | 0 | 0 |
| Sato | 25 | 281 | 530 | 726 | > 7200 |
| Grasp | 2312 | ? | ? | ? | ? |
| Relsat | 264 | 4428 | > 7200 | > 7200 | > 7200 |
| Chaff | 7 | 25 | 29 | 90 | 892 |
| Heerhugo | 4 | 10 | 24 | 49 | ? |
| Satz | 293 | 2461 | 57 | 5 | > 7200 |
| *EqSatz*− | 0 | 1 | 1 | 2 | 6 |
| *EqSatz* | 0 | 1 | 2 | 3 | 7 |

Table 7 displays the performance of the seven solvers on longmult* instances. *EqSatz* is the third fastest solver on these instances. *EqSatz*' is *EqSatz* with a slightly modified branching rule. Refering to Fig. 8, we remove the two functions nb_fixed_vars and nb_eq_pairs from the weight of a literal, i.e. the weight of a literal in *EqSatz*' is defined to be the number of newly generated binary clauses if the literal is satisfied. With the simplified branching rule, *EqSatz*' becomes the fastest solver for longmult* instances.

Table 8 shows the performance of the seven solvers on queueinvar* instances. *EqSatz* is one of the fastest solver for these instances and compete with Chaff.

*EqSatz* and *EqSatz*− have the same performance on BMC instances, meaning that rule 7 is not important for these instances.

Table 7
Running time on BMC longmult* instances. Each problem name mult* should be preceded by "long"

| Instance | mult1 | mult3 | mult5 | mult7 | mult9 | mult11 | mult13 | mult15 |
|---|---|---|---|---|---|---|---|---|
| #var | 791 | 1555 | 2397 | 3319 | 4321 | 5403 | 6565 | 7807 |
| #cls | 2335 | 4767 | 7431 | 10335 | 13479 | 16863 | 20487 | 24351 |
| #eq_cls | 29 | 87 | 145 | 203 | 261 | 319 | 377 | 435 |
| #g_eq | 0 | 0 | 0 | 0 | 0 | 0 | 0 | 0 |
| Sato | 0 | 41 | 181 | 348 | 733 | 1110 | 1916 | 1646 |
| Grasp | 0 | 1 | 40 | 4013 | ? | ? | ? | ? |
| Relsat | 0 | 1 | 27 | 3402 | > 7200 | > 7200 | > 7200 | > 7200 |
| Chaff | 0 | 0 | 2 | 79 | 993 | 1656 | 1573 | 1435 |
| Heerhugo | 1 | 5 | 5747 | > 7200 | > 7200 | > 7200 | > 7200 | 345 |
| Satz | 0 | 0 | 11 | 331 | 1948 | 4371 | 6965 | > 7200 |
| *EqSatz−* | 0 | 1 | 13 | 291 | 1957 | 3113 | 3732 | 4870 |
| *EqSatz* | 0 | 1 | 13 | 274 | 1681 | 3050 | 3662 | 4867 |
| *EqSatz′* | 0 | 0 | 2 | 60 | 305 | 636 | 937 | 1029 |

Table 8
Running time on BMC queueinvar* instances. Each instance name should be preceded by "queue"

| Instance | invar10 | invar12 | invar14 | invar16 | invar18 | invar20 |
|---|---|---|---|---|---|---|
| #var | 886 | 1112 | 1370 | 1168 | 2081 | 2435 |
| #cls | 5622 | 7335 | 9313 | 6496 | 17368 | 29671 |
| #eq_cls | 51 | 53 | 55 | 75 | 70 | 72 |
| #g_eq | 6 | 6 | 6 | 10 | 5 | 5 |
| Sato | 15 | 97 | 576 | 1398 | > 7200 | > 7200 |
| Grasp | 20 | 47 | 83 | 107 | 472 | > 7200 |
| Relsat | 20 | 31 | 162 | 93 | 257 | 834 |
| Chaff | 0 | 1 | 2 | 2 | 12 | 23 |
| Heerhugo | 709 | 2868 | > 7200 | > 7200 | > 7200 | > 7200 |
| Satz | 12 | 54 | 250 | 1017 | 8 | 13 |
| *EqSatz−* | 3 | 6 | 10 | 9 | 9 | 13 |
| *EqSatz* | 3 | 5 | 10 | 9 | 9 | 14 |

### 6.2.4. Performance on Urquhart's hard examples for resolution

Urquhart [27] used a particular family of bipartite graphs with $m^2$ vertices in each side. The graphs were first defined by Marqulis [19]. Urquhart further linked the $m^2$ vertices in each side by a cycle and constructed his examples from the obtained graph. We refer the reader to Section 5.2 for a brief description of the construction.

As suggested by Urquhart, we generate with the help of Cantarell and Jurkowiak a bipartite graph for $m = 2$ and $m = 3$, respectively, using the five permutations specified by Gabber and Galil [10, p. 365]. Then we generate 3-SAT formulas in two fashions:

(1) while there is an equivalency clause of length $> 3$ such as $l_1 \leftrightarrow l_2 \leftrightarrow \cdots \leftrightarrow l_k$, introduce a new variable $y$ and replace the clause with two equivalency clauses $y \leftrightarrow l_1 \leftrightarrow l_2$ and $y \leftrightarrow l_3 \leftrightarrow \cdots \leftrightarrow l_k$;

Table 9
Running time on Urquhart's hard examples for resolution

| Instance | Urquhart2-1 | Urquhart2-2 | Urquhart3-1 | Urquhart3-2 |
|---|---|---|---|---|
| #var | 36 | 60 | 99 | 153 |
| #cls | 96 | 160 | 264 | 408 |
| #eq_cls | 24 | 40 | 66 | 102 |
| #g_eq | 0 | 0 | 0 | 0 |
| Sato | 0 | 2 | ? | ? |
| Grasp | ? | ? | ? | ? |
| Relsat | 0 | 28 | $> 360000$ | $> 360000$ |
| Chaff | 0 | 192 | ? | ? |
| Heerhugo | 40 | 1876 | $> 360000$ | $> 360000$ |
| Satz | 0 | 138 | $> 360000$ | $> 360000$ |
| *EqSatz−* | 0 | 2 | 195985 | $> 360000$ |
| *EqSatz* | 0 | 2 | 144849 | $> 360000$ |

(2) before generating the equivalency clauses, replace each vertex of degree $k > 3$ by a cycle of $k$ vertices and join each vertex of the cycle by the $k$ edges of the replaced vertex. This transformation is due to Kirkpatrick [12].

In Table 9, Urquhart2-1 ($m = 2$) and Urquhart3-1 ($m = 3$) are generated in the first fashion while Urquhart2-2 and Urquhart3-2 are generated in the second fashion. These are very small instances, but they are surprisingly hard for the tested SAT solvers. *EqSatz* is the best solver for these instances and is the only solver solving Urquhart3-1 in roughly 40 h.

Note that rule 7 is important to solve Urquhart formulas.

## 7. Discussion and related work

Equivalency clauses constitute a major obstacle to the DLL procedure. For example, while the empirical complexity of Satz on hard random 3-SAT instances appears to be $O(2^{n/21})$, its complexity on DIMACS pret* problem is $O(2^{n/3})$. Urquhart [27] has shown his examples have exponential complexity for any DLL procedure (including *EqSatz*) or even general resolution.

The instances uniquely composed of equivalency clauses are intrinsically easy. Urquhart showed that his examples have a refutation of length $O(n^4)$ in a standard axiomatic system for propositional calculus [27]. Given a set of equivalency clauses, Warners and Van Maaren [29] also proposed an approach solving them in polynomial time. They select an equivalency clause of length $k$, $x_1 \leftrightarrow l_2 \leftrightarrow \cdots \leftrightarrow l_k$, write it as $x_1 \equiv l_2 \leftrightarrow \cdots \leftrightarrow l_k$, and substitute in all other equivalency clauses the occurrence of $(\bar{x}_1)\ x_1$ by $(\neg)l_2 \leftrightarrow \cdots \leftrightarrow l_k$, increasing in general the length of these clauses (by $k-2$ in the worst case). $x_1$ is called *dependent variable*.

For a formula $\mathscr{F}$ having no CNF part such as Urquhart examples, the selected equivalency clause can be easily satisfied since its dependent variable does not occur

elsewhere after the above substitution, so that it is removed from $\mathscr{F}$. By repeatedly removing equivalency clauses, $\mathscr{F}$ is solved in polynomial time.

Warners and Van Maaren's simplification procedure also allows to solve formulas having a small CNF part such as the five DIMACS 32-bit parity instances par32-*i*-c in which more than 80% of clauses are equivalency clauses. Their approach is in this case used as a preprocessing step for an extended DLL procedure. Note that repeating the simplification in every tree node would generate longer and longer equivalency clauses and would be very costly.

Using the original Davis–Putnam (DP) procedure [9] (with variable elimination rule instead of branching rule) and ZBDD [21] to represent clauses, Chatalic and Simon [4] also solve Urquhart's examples in polynomial time.

Instances containing both EQ part and CNF part are generally hard to solve. Massacci [20] noticed in his SAT-encoding of cryptographic key search problem that the problem becomes hard for current AI techniques as soon as equivalency clauses begin to appear. Equivalency reasoning proves very useful for these instances.

Various reasonings are proposed in the literature. Apart from the very strong reasoning implemented in Heerhugo inspired from Stalmarck's method, Brafman [3] proposed a simplifier efficiently implementing well-known 2-SAT techniques and a novel use of transitive reduction to reduce formula size. Marques Silva [16] proposed algebraic simplification techniques to simplify CNF formulas. These reasonings are proposed to preprocess SAT formulas to be solved by another SAT solver. Efficiently incorporating them into every tree node of a backtracking search awaits future work.

Besides conflict-driven learning, Marques-Silva and Glass [17] integrated recursive learning into Grasp to "learn" clauses at a tree node without encountering a conflict. Equivalency reasoning also learns clauses without encountering a conflict, but focuses on equivalency clauses and uses special inference rules.

*EqSatz* relies on unit propagation to discover initial equivalent literals after each split to start equivalency reasoning. However, it does not cover all 2-SAT reasoning. In particular, it does not include a recognition of implication cycles, such as the one in 2CL [28]. Le Berre [13] proposed another implementation of the equivalency reasoning using simple and double unit propagations, which, while missing some equivalent literals deduced by *EqSatz*, may recover some other equivalent literals *EqSatz* is not deducing, by recognizing implication cycles. Le Berre's approach is currently limited in the preprocessing step. The integration of his implementation in *EqSatz* and its efficient incorporation at every tree node of *EqSatz* require further research.

Compared with related work, the originality of our approach is that it aims at special structural properties of CNF formulas and is simple to be efficiently repeated at *every* node of a search tree.

There are three reasons for the efficiency of equivalency reasoning in a node of a search tree: (i) for every new equivalence $l_1 \leftrightarrow l_2$, the application of rules 4 and 5 has linear complexity; (ii) the more there are new equivalent literals, the more equivalency reasoning is important, because more search space is cut; (iii) equivalency reasoning never generates and never deals with equivalency clauses of length $> 3$, and the generated binary and ternary equivalency clauses are in turn used in subsequent reasoning.

## 8. Conclusions and future work

Unit propagation fails to exploit equivalent literals such as $l_1 \leftrightarrow l_2$ without branching. The ineffectiveness of unit propagation makes many SAT problems difficult for the DLL procedure. In particular, DLL performs very poorly in handling equivalency clauses, a common structure in the SAT-encoding of many hard real-world problems. In order to remedy the ineffectiveness of unit propagation, we proposed an equivalent literal propagation in this paper.

We used a simple data structure to represent all equivalent literals into equivalent classes, allowing efficient backtracking management. Based on the data structure, we have implemented equivalency reasoning to propagate these equivalent literals among all ternary equivalency clauses.

We have implemented a subprocedure called Equivalency_Reasoning in *EqSatz* which applies rules 4 and 5 in linear time for two equivalent literals $l_1 \leftrightarrow l_2$ to fix some literals and to deduce new equivalent literals. Using Equivalency_Reasoning subprocedure, we have integrated rules 6 and 7 into the branching rule of *EqSatz* to deduce new binary and ternary equivalency clauses when examining free variables.

Our approach makes DLL able to efficiently solve one of the 10 challenge problems in propositional reasoning formulated by Selman et al., and many other real-world problems.

In the future, it would be interesting to identify inference rules for effective equivalent literal propagation in structures other than equivalency clauses. Furthermore, combining equivalent literal propagation with other pruning techniques such as conflict driven learning is promising to make DLL solve more SAT problems, since conflict-driven learning allows to extract and memorize information other than equivalent literals.

## Acknowledgements

We thank Daniel Le Berre for informing us about BMC problems, anonymous referees for their comments which helped to improve this paper. We also thank Alexander Zimmermann, Pierre Berezig and Paul W. Purdom for their help to improve the English of this paper.

## References

[1] R.J. Bayardo Jr., R.C. Schrag, Using CSP look-back techniques to solve real-world SAT instances, in: Proceedings of AAAI-97, Providence, Rhode Island, July 1997.

[2] A. Biere, A. Cimatti, E. Clarke, Y. Zhu, Symbolic model checking without BDDs, in: R. Cleaveland (Ed.), Proceedings of Tools and Algorithms for the Analysis and Construction of Systems (TACAS'99), Lecture notes in Computer Science, Vol. 1579, Springer, Berlin, 1999.

[3] R.I. Brafman, A simplifier for propositional formulas with many binary clauses. in: Proceedings of IJCAI-01, Seattle, USA, 2001.

[4] P. Chatalic, L. Simon, Zres: the old Davis–Putnam procedure meets ZBDDs, in: D. McAllester (Ed.), 17th International Conference on Automated Deduction (CADE'17), Lecture Notes in Artificial Intelligence (LNAI), Vol. 1831, June 2000, pp. 449–454.

[5] S.A. Cook, The complexity of theorem proving procedures, in: Third ACM Symposium on Theory of Computing, Ohio, 1971, pp. 151–158.
[6] T.H. Cormen, C.E. Leiserson, R.L. Rivest, Introduction to Algorithms, MIT Press, Cambridge, MA, 1990, pp. 440–461.
[7] J.M. Crawford, M.J. Kearns, R.E. Schapire, The minimal disagreement parity problem as a hard satisfiability problem, Draft version 1995.
[8] M. Davis, G. Logemann, D. Loveland, A machine program for theorem proving, Common. ACM 5 (1962) 394–397.
[9] M. Davis, H. Putnam, A computing procedure for quantification theory, J. ACM 7 (3) (1960) 201–215.
[10] O. Gabber, Z. Galil, Explicit constructions of linear size superconcentrators, in: Proceedings of the 20th Annual Symposium Foundations of Computing Science, IEEE, New York, 1979, pp. 364–370.
[11] J.F. Groote, J.P. Warners, The propositional formula checker HeerHugo, J. Autom. Reason. 24 (2001) 101–125.
[12] D.G. Kirkpatrick, Topics in the complexity of combinatorial algorithms, Ph.D. dissertation, Technical Report No 74, Department of Computer Science, University of Toronto, Toronto, Canada, December 1974.
[13] D. Le Berre, Exploiting the real power of unit propagation lookahead, in: H. Kautz, B. Selman (Eds.), Proceedings of Workshop on Theory and Applications of Satisfiability Testing (SAT'2001), Boston, USA, june 2001, Official Proceedings of Workshop in ENDM: Electronics Notes in Discrete Mathematics.
[14] C.M. Li, Anbulagan, heuristics based on unit propagation for satisfiability problems, in: Proceedings of IJCAI-97, IBN 1-55860-480-4, Nagoya, Japan, August 1997, pp. 366–371.
[15] C.M. Li, Integrating equivalency reasoning into David–Putnam procedure, in: Proceedings of AAAI-2000, Austin, Tx, USA, 2000.
[16] J.P. Marques-Silva, Algebraic simplification techniques for propositional satisfiability, in: Proceedings of the Sixth international conference on Principles and Practice of Constraint Programming (CP'2000), September 2000.
[17] J.P. Marques-Silva, T. Glass, Combinational equivalence checking using satisfiability and recursive learning, in: Proceedings of the IEEE/ACM Design, Automation and Test in Europe Conference (DATE), 1999.
[18] J.P. Marques-Silva, K.A. Sakallah, Conflict analysis in search algorithms for propositional satisfiability, in: Proceedings of the International Conference on Tools with Artificial Intelligence, November 1996.
[19] G.A. Marqulis, Explicit construction of concentrators, Problems Inform Transmission 9 (1973) 325–332.
[20] F. Massacci, Using Walk-SAT and Rel-SAT for cryptographic key search, in: Proceedings of IJCAI-99, pp. 290–295.
[21] S. Minato, Zero-suppressed BDDs for set manipulation in combinatorial problems, in: Proceedings of the 30th ACM/IEEE Design Automation Conference, 1993.
[22] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik, Chaff: engineering an Efficient SAT solver, in: Proceedings of the 38th Design Automation Conference (DAC'01), June 2001.
[23] B. Selman, H. Kautz, D. McAllester, Ten challenges in propositional reasoning and search, in: Proceedings of IJCAI-97, ISBN 1-55860-480-4, Nagoya, Japan, August 1997.
[24] L. Simon, P. Chatalic, SatEx: a web-based framework for SAT experimentation, in: H. Kautz, B. Selman (Eds.), Proceedings of the Workshop on Theory and Applications of Satisfiability Testing (SAT'2001), Boston, june 2001, Official Proceedings of Workshop in ENDM: Electronics Notes in Discrete Mathematics.
[25] G. Stalmarck, A system for determining propositional logic theorems by applying values and rules to triplets that are generated from a formula, Technical Report, European Patent N 0403 454 (1995), US Patent N 5 276 897, Swedish Patent N 467 076 (1989), 1989.
[26] G.S. Tseitin, On the complexity of derivation in propositional calculus, in: A.D. Slisenko (Ed.), Studies in Constructive Mathematics and Mathematical Logic, Part 2. Consultant Bureau, New York, London, 1968,
pp. 115–125.
[27] A. Urquhart, Hard examples for resolution, J. ACM 34 (1) (1987) 209–219.

[28] A. Van Gelder, Y.K. Tsuji, Satisfiability testing with more reasoning and less guessing, in: D.S. Johnson, M. Trick (Eds.), Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge, American Mathematical Society, Providence, RI, 1996.
[29] J.P. Warners, H. Van Maaren, A two phase algorithm for solving a class of hard satisfiability problems, Operations Res. Lett. 23 (1998) 81–88.
[30] D.N. Warren-Smith, Digital logic systems, Electronics World Magazine, February 1999 Edition, pp. 113–118 (also see http://users.senet.com.au/∼dwsmith/concept1.htm).
[31] H. Zhang, An efficient propositional prover, in: Proceedings of the International Conference on Automated Deduction, July 1997, pp. 272–275.