Using Boolean Constraint Propagation for Sub-clause Deduction

Sylvain Darras, Gilles Dequen, Laure Brisoux Devendeville, Bertrand Mazure, Richard Ostrowski, Lakhdar Saïs

Using Boolean Constraint Propagation for Sub-clauses Deduction

(full paper)

Sylvain Darras, Gilles Dequen, Laure Devendeville
Laria – Universit de Picardie Jules Vernes
{darras,dequen,devendeville}@laria.univ-picardie.fr

Bertrand Mazure, Richard Ostrowski, Lakhdar Saïs
CRIL CNRS – Universit d'Artois
rue Jean Souvraz SP-18
F-62307 Lens Cedex France
{ostrowski,mazure,sais}@cril.univ-artois.fr

Abstract

Boolean Constraint Propagation (BCP) is recognized as one of the most useful technique for efficient satisfiability checking. In this paper a new extension of the scope of boolean constraint propagation is proposed. It makes an original use of BCP to achieve further reduction of boolean formulas. Considering the implication graph generated by the constraint propagation process as a resolution tree, sub-clauses from the original formula can be deduced. Then, we show how such extension can be grafted to modern SAT solvers where BCP is maintained at each step of the search tree. Preliminary results of "Zchaff" - the state of the art SAT solver - augmented with extended BCP, show the great potential of our approach with respect to certain classes of SAT instances.

Keywords: SAT, Boolean Constraint Propagation, reasoning and search, learning, subsumption.

1 Introduction

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 (e.g. [7, 8, 11, 2]). Many benchmarks have been proposed and regular competitions (e.g. Dimacs'93, Beijing'96, SAT'01-04) are organized around

these specific SAT instances, which are expected to encode structural knowledge at least to some extent. The huge size of the real-world SAT instances currently in the scope of modern SAT solvers such as Zchaff [12], allows to consider practical applications. Such a progress shows us that instances with worst case complexity behavior appears rarely in practice. Indeed, instances from practical applications contain some structures that can benefit to SAT solvers. One of them is that clauses share more common variables comparatively to random instances. For example, the clauses encoding the boolean formula $x_1 \wedge x_2 \rightarrow y_1 \wedge y_2 \wedge \cdots \wedge y_n$ share the two boolean variables x_1 and x_2. In other words, many common sub-clauses results from the elimination of boolean connectives (e.g. $\vee, \wedge, \leftrightarrow$). One can see that after assigning the value $false$ to y_i the other clauses become redundant and can be eliminated by subsumption using the clause $\neg x_1 \vee \neg x_2$.

Our intuition is that when considering large SAT instances at each step of the search process, many clauses become redundant and some sub-clauses might be deduced and then decrease the size of the formula.

Considering large SAT instances at each step of DPL search process, our intution is that some sub clauses can be deduced with the resolution rule and then useless according to the subsumption rule. This idea can be maintened at each node of search tree of the formula and then produce some formulas where the number of clauses cannot be greater than the initial one and where for a given clause

become redondant and then useless according to the subsumption rule. On the other hand, some subclauses can be deduced according to the resolution rule

The main objective of this paper is to show how such structural knowledge can be exploited during search. We focus on sub-clauses deductions that might help reducing the formula to its "real" size and give rise to a constant space complexity approach.

Boolean constraint propagation (or unit propagation), applying in all efficient DPLL implentations [9, 4, 12] can be considered as a restricted form of resolution and as a special case of subsumption rule.

It is recognized as one of the most important paradigm for efficient satisfiability checking. Indeed, most of modern SAT solver are based on the well known Davis-Putnam-Logemann-Loveland (DPLL) procedure [3] where BCP is maintained at each step of the search process. On many SAT instances, a major part of the search space (about 90%) is achieved using BCP. This important role has motivated many works on efficient implementation of BCP (e.g. Zchaff) and on extending its practical use. Among others, many simplification techniques (e.g. [5, 1]), variable ordering heuristics (e.g. [9, 4]), conflict analysis scheme (e.g. [10, 12]), functional dependencies deduction (e.g. [6]) are based on BCP.

In this paper, a new extension of the scope of boolean constraint propagation is proposed. It makes an original use of BCP to achieve further reduction of boolean formulas. More precisely, the constraint graph generated by the boolean constraint propagation process can be mapped to a resolution tree which encodes clauses of the original formula and new resolvent. The set of such possible resolvent can have an exponential size in the worst case w.r.t. the set of clauses encoded in the constraint graph. To avoid such a drawback, the approach proposed in this paper considers only a relevant set of resolvent leading to a polynomial time and constant space complexity approach. Then, we show how such an extension can be grafted to modern SAT solvers.

The paper is organized as follows. After some preliminary definitions, relations between resolution and boolean constraint propagation are discussed. Then a constant space complexity BCP-based approach for sub-clauses deduction is proposed, allowing the formula to be reduced. Its dynamic integration in SAT solvers is presented and some preliminary experimental results

showing the interest of the proposed approach are provided. Finally, promising paths for future research are discussed in the conclusion.

2 Definitions and preliminaries

A *CNF formula* Σ is a set (interpreted as a conjunction) of *clauses*, where a clause is a set (interpreted as a disjunction) of *literals*. A literal is a positive or negative propositional variable. We note $var(\Sigma)$ (resp. $lit(\Sigma)$) the set of variables (resp. literals) occurring in Σ. A *unit clause* is a clause containing a single literal called *unit literal*. A binary clause contains two literals associated with two distinct variables. A clause containing literals of distinct variables is called fundamental. It is called tautological when it contains two opposite literals. The size of the formula Σ is given by $|\Sigma| = \sum_{c \in \Sigma} |c|$, where $|c|$ is the number of literals in c.

In the following, we use formula (resp. variable) instead of CNF formula (resp. propositional variable). In addition to the set-based notations, we define the negation of a set A of literals as the set $\bar{A}$ of the corresponding opposite literals. We note $A_\vee$ (respectively $A_\wedge$) the disjunction (resp. conjunction) of all literals of A.

An *interpretation* of a formula Σ is an assignment of truth values $\{true, false\}$ to its variables. It is called partial interpretation if only a subset of variables of $var(\Sigma)$ are assigned. A *model* of a formula is an interpretation that satisfies the formula. Accordingly, SAT consists in finding a model of a formula when such a model exists or in proving that such a model does not exist.

A formula ψ is a logical consequence of ϕ (noted $\phi \vDash \psi$) iff any model of ϕ is also a model of ψ.

Let c_1 and c_2 be two clauses of Σ. i) *resolution rule*: If there exists a literal l (called a pivot of the resolvent) s.t. $l \in c_1$ and $\neg l \in c_2$, then a *resolvent* on l of c_1 and c_2 can be defined as $res(l, c_1, c_2) = c_1 \backslash \{l\} \cup c_2 \backslash \{\neg l\}$. ii) *subsumption rule*: When $c_1 \subseteq c_2$ (i.e. c_1 is a sub-clause of c_2), then c_1 subsumes c_2. Resolution (resp. subsumption) rules leads to a new formula $\Sigma \cup res(l, c_1, c_2)$ (resp. $\Sigma \backslash c_2$) equivalent to Σ with respect to SAT. A resolvent $r = res(l, c_1, c_2)$ is called a subsuming resolvent iff $\exists c \in \Sigma$ s.t. r subsumes c. Repeatedly applying resolution rule leads to resolution proof system that can prove unsatisfiability of a formula.

For a formula Σ and a literal $l \in lit(\Sigma)$, we define $\Sigma(l) = \{c | c \in \Sigma, \{l, \neg l\} \cap c = \emptyset\} \cup \{c \backslash \{\neg l\} | c \in \Sigma, \neg l \in c\}$ as the result of setting l to the value $true$. For simplicity, we note $\Sigma(l_1)(l_2)\ldots(l_n)$ as $\Sigma(l_1, l_2, \ldots, l_n)$.

Boolean Constraint Propagation refers to the iterative process of setting all unit literals the value $true$ until encountering an empty clause or no unit clause remains in the formula. $\Sigma_{bcp}(l)$ is the formula obtained by BCP on $\Sigma(l)$. The set of unit literals propagated by the application of BCP on $\Sigma(l)$ is noted $UPL(\Sigma, l)$. This notation can be extented to a set of literals L: $UPL(\Sigma, L) = \bigcup_{l \in L} UPL(\Sigma, l)$.

BCP can be seen as a restricted form of resolution. At each step a subsuming resolvent $res(l, \{l\}, c_2) = c_2 \backslash \{\neg l\}$ is produced. Last but not least, BCP is an important component of the well known DPLL procedure.

DPLL procedure performs a backtrack depth-first search through a binary tree. After a simplification step using BCP and setting pure literal (those whose negation does not appear in the formula) to true, a *decision variable* is chosen and recursively set to true respectively to false the two associated *decision literals*.

3 Exploiting BCP for sub-clause deduction

In this section, we show how BCP can be further extended, allowing sub-clause of the formula to be deduced. We first introduce the implication graph generated by BCP and its two possible translations to a resolution tree.

3.1 Boolean constraint propagation and Implication graph

An *Implication Graph* (IG) is a directed acyclic graph that captures the boolean propagation process. A constraint graph defined below is generated according to a given formula and a set of decision literals.

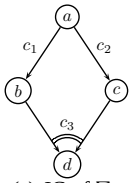
Definition 1 (Implication graph) *Let Σ be a formula and I a set of decision literals. An implication graph associated to Σ and I is a labelled directed acyclic graph $\mathcal{G}_{ig}(\Sigma, I) = (\mathcal{V}, \mathcal{E})$ where :*

1. *$\mathcal{V} = \{v | \eta(v) \in I \cup UPL(\Sigma, I)\}$ a set of vertices ; where $\eta(v)$ is the literal labelling the vertex v. Literals labelling a set of vertices is denoted $\eta(\mathcal{V}) = \{\eta(v) | v \in \mathcal{V}\}$.*
2. *$\mathcal{E} = \{\langle v_j, v_i\rangle | \eta(v_i) = l_i, \eta(v_j) = l_j, \exists c = \{\neg l_1, \ldots, \neg l_{i-1}, l_i, \neg l_{i+1}, \ldots, \neg l_n\} \in \Sigma$, $c \cap \eta(\mathcal{V}) = \{l_i\}$, $c \cap \overline{\eta(\mathcal{V})} = \{\neg l_1, \ldots, \neg l_{i-1}, \neg l_{i+1}, \ldots, \neg l_n\}$ and $j \in \{1, \ldots, i-1, i+1, \ldots, n\}\}$. Each directed edge $\langle v_i, v_j\rangle$ is labelled with a clause c, i.e. label($\langle v_i, v_j\rangle$)$= c$.*

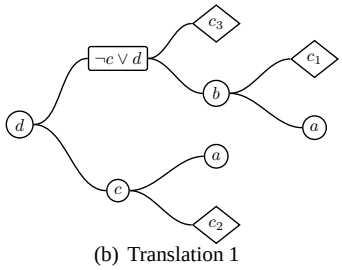
In the definition of IG each node corresponds to a variable assignment. The set of literals associated to the predecessors of a vertex v ($pred(v)$) corresponds to its antecedent assignments). Directed edges from all $v' \in pred(v)$ to v are all labelled with the same clause c (noted $cl(v)$). The clause c and $\eta(pred(v))$ give us the reason of its implication. Let us note that in general a literal l can be implied thanks to different clauses and literals (i.e. reasons). When all the reasons are recorded, IG is called complete; otherwise it is called incomplete. In case of complete IG, $pred(v)$ is a superset where each element corresponds to a particular reason. For clarity of the presentation, in this paper we only consider incomplete implication graph.

For

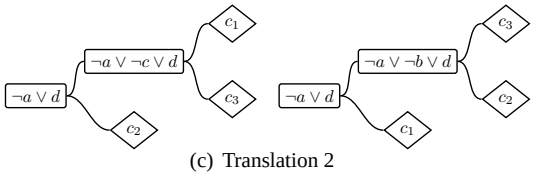
an implication graph $\mathcal{G}_{ig}(\Sigma, I)$, vertices corresponding to decision literals I have no incoming edge and are called source vertices ($sources(\mathcal{G}_{ig})$). Vertices with no outgoing



(a) IG of Σ



(b) Translation 1



(c) Translation 2

Figure 1: From the implication graph to resolution tree

edge are called sink vertices ($sink(\mathcal{G}_{ig})$). When a conflict occurs, $\mathcal{G}_{ig}$ contains two sink vertices labelled with two opposite literals.

Let us mention that implication graphs have been widely exploited by modern SAT solvers to learn from conflicts [12], to achieve nogood recording or to operate non chronological backtracking [10]. In such context, each decision literal l is labelled with a *decision level* α corresponding to the level where l is assigned.

α is an integer corresponding to the number of decision variables assigned from the root to the current node.

At each step, literals propagated by BCP at a given level receive the same labelling.

We give in the following a new translation of IG to a special form of resolution tree.

3.2 Implication graph and resolution tree

In this section we show that an implication graph can be translated in two different ways to a resolution tree (RT).

The definition below gives a general description of a resolution tree.

Definition 2 (Resolution Tree) *A resolution tree associated to a formula Σ is a directed acyclic graph $\mathcal{G}_{rt} = (\mathcal{N}, \mathcal{E})$ such that :*

- *each node n is labelled by a clause c (i.e. $\eta(n) = c$).*

- *clauses labelling internal nodes are obtained by resolution from clauses labelling its two child nodes. Arcs between an internal node and its two child nodes represent such resolution operation.*
- *leaf nodes are labelled with clauses of* Σ

As mentioned in section 2, BCP can be seen as restricted form of resolution. At each step subsuming resolvent is produced between a unit clause and an other clause of the formula. This first obvious translation (not described in this paper) gives rise to resolution tree that describes precisely such a restricted form of resolution.

Algorithm 1, describes the second translation of an implication graph to a resolution tree where internal nodes are labelled with new resolvent. This new translation gives us a new picture on BCP, in that it leads to powerful resolution-based technique.

Algorithm 1 IG2RT_2(**in** $\mathcal{G}_{ig} = (\mathcal{V}, \mathcal{A})$: IG, **out** $\mathcal{G}_{rt} = (\mathcal{N}, \mathcal{E})$: RT)

1: let $\mathcal{N} = \emptyset, \mathcal{E} = \emptyset$
2: **for** each $v \in sink(\mathcal{G}_{ig})$ **do**
3: let $pred(v) = \{v_1, v_2, \ldots, v_k\}$ s.t. $\eta(v_i) = l_i$,$\eta(v) = l$
4: let $r = (\bigcup_{1 \leq i \leq k} \{\neg l_i\}) \cup \{l\}$
5: $\mathcal{N} = \mathcal{N} \cup \{w\}$ s.t. $\eta(w) = r$
6: $s = pred(v)$
7: $\mathcal{G}_{rt} = translate\_2(v, w, s, r, \mathcal{G}_{ig})$

Algorithm 2 translate_2(**in** v, w: vertex, s: set of vertices, r : resolvent, $\mathcal{G}_{ig} = (\mathcal{V}, \mathcal{A})$: IG, **out** $\mathcal{G}_{rt} = (\mathcal{N}, \mathcal{E})$: RT)

1: **If** $pred(v) \neq \emptyset$ **then**
2: let $s = \{v_1, v_2, \ldots, v_k\}$ s.t. $\eta(v_i) = l_i, \eta(v) = l$
3: **for** $i = k$ downto 1 **do**
4: let $p_i \in pred(v_i)$ and $c_i = \eta(\langle p_i, v_i \rangle)$
5: $\mathcal{N} = \mathcal{N} \cup \{f_i\}$ s.t. $\eta(f_i) = c_i$
6: let $temp_s = s$ and $s = (s \backslash \{v_i\}) \cup pred(v_i)$
7: let $p = \{p | p = \eta(p_i)$ s.t. $p_i \in pred(v_i)\}$
8: let $temp_r = r$ and $r = (r \backslash \{\neg l_i\}) \cup \bar{p}$
9: $\mathcal{N} = \mathcal{N} \cup \{r_i\}$ s.t. $\eta(r_i) = r$
10: $\mathcal{E} = \mathcal{E} \cup \{\langle f_i, r_i \rangle, \langle w, r_i \rangle\}$
11: $\mathcal{G}_{rt} = translate\_2(p_i, r_i, s, r, \mathcal{G}_{ig})$
12: $s = temp_s, r = temp_r$

Example 1 *Let* $\Sigma = (\neg a \vee b) \wedge (\neg a \vee c) \wedge (\neg b \vee \neg c \vee d)$. *We note* c_i, *with* $1 \leq i \leq 3$ *a clause number i of* Σ. *Figure 1(b) and Figure 1(c) show the first (respectively second) translation of the implication graph (see figure 1(a)) to a resolution tree.*

Remark 1 *We note that, in the resolution tree shown in Figure 1(b) (resp. Figure 1(c)) the internal nodes are labelled with sub-clauses (resp. new clauses). As the traversal of the implication graph starts on the sink vertex with label* d, *we can note that all the new clauses (Figure 1(c)) contain such a literal. In algorithm 1, if we consider all the vertices of the implication graph (line 2), then we obtain a set of different resolution trees (i.e. forest).*

In algorithm 1, the translation starts on sink vertex v of IG (line 2), a new node w labelled with the clause r made of $\eta(v)$ and $\overline{\eta(pred(v))}$ the negation of the literals labelling its predecessors is added to RT. Then, algorithm 2 is call.

At each step, we consider the current vertex v (resp. w) of IG (resp. RT), a set of vertices s that contains the current predecessor to process next, and the current resolvent r. Note that at each step, $\eta(s) = r \backslash \{l\}$ where $l = \eta(v)$ (algorithm 1, line 3). For each vertex v_i of the current set s, two new nodes f_i (resp. r_i) labelled with $cl(v_i)$ (resp. $r = res(l_i, cl(v_i), \eta(w))$) and two new directed edges $\{\langle f_i, r_i\rangle, \langle w, r_i\rangle\}$ are added to RT (lines 4-10), then the process is repeated recursively.

In line 12 (algorithm 2), it is important to note that the current set s and the current resolvent r are updated to their original contents. Then, at each iteration (line 3) we consider the same set s and resolvent r. For a given vertex v, such updating is done in order to consider all the combination of its possible ancestors (i.e. a vertex a is an ancestor of v iff there exists a path from a to v). Consequently, algorithm 1 has an exponential worst case complexity behavior and can lead to an exponential number of new clauses. To avoid such a drawback, we introduce in the sequel a new constant space complexity approach that considers only a subset of such resolvent.

3.3 A constant space & polynomial time complexity approach

In this section, a polynomial time and constant space complexity approach is presented. First we consider only a subset of pertinent implications of the resolution tree (second translation). More precisely, an implication is considered if it subsumes - either directly or by resolution - clauses of the original formula Σ. Such a restriction leads to a constant space complexity approach (i.e. the size of the formula decreases). Second, the resolution tree is not explicitly built.

Let us first introduce the following definitions and properties.

Definition 3 *Let Σ be a formula, $l \in lit(\Sigma)$ and $\mathcal{G}_{rt} = (\mathcal{N}, \mathcal{E})$ a RT obtained from $\mathcal{G}_{ig}(\Sigma, l)$. A clause c is l-sub-inferred from Σ (noted $\Sigma_l \vDash^* c$) if $\exists c' \in \Sigma$ such that one of the following condition is satisfied,*

1. $c \in \eta(\mathcal{N})$ *and* $c \subset c'$
2. $\exists c'' \in \eta(\mathcal{N})$ *s.t.* $c = res(p, c', c'') \subset c'$ *where* $p \in c'$ *and* $\neg p \in c''$

Proposition 1 *Let Σ be a formula and $l \in lit(\Sigma)$. If $\Sigma_l \vDash^* c$ then $\Sigma \vDash c$*

Proof 1 *By construction of $\mathcal{G}_{rt} = (\mathcal{N}, \mathcal{E})$ from $\mathcal{G}_{ig}(\Sigma, l)$ all nodes of $\mathcal{N}$ are labelled with resolvent obtained from clauses of Σ. Consequently, $\forall d \in \eta(\mathcal{N})$, we have $\Sigma \vDash d$. Moreover, in the definition of $\Sigma_l \vDash^* c$, we distinguish two cases. In the first case, $c \in \eta(\mathcal{N})$, then $\Sigma \vDash c$. In the second case, $c = res(p, c', c'')$ where $c' \in \Sigma$ and $c'' \in \eta(\mathcal{N})$, then $\Sigma \vDash c$.*

Proposition 2 *Let Σ be a formula and $l \in lit(\Sigma)$. $\Sigma_l \vDash^* c$ can be computed in $O(|\Sigma| \times |var(\Sigma)|)$.*

Proof 2 *Let us give a proof sketch on the complexity of such computation (for more details see algorithm 3). To compute $\Sigma_l \vDash^* c$, first BCP is processed on $\Sigma \wedge l$ and $\mathcal{G}_{ig} = (\mathcal{V}, \mathcal{E})$ is computed. Such computation is achieved in linear time. In the second step, we try to find a clause $c' \in \Sigma$ such that $c \subset c'$. To achieve that, only clauses containing literals from $UPL(\Sigma, l)$ are considered; otherwise such a clause can not be l-sub-inferred. Now let c' be such a clause. To achieve l-sub-inference, for each literal $p \in c' \cap \eta(\mathcal{V})$ a traversal of $\mathcal{G}_{ig}$ is realized starting on the vertex $v \in \mathcal{V}$ labelled by p. A first clause r made of p and $\overline{\eta(pred(v))}$ is computed. At this step $r \in \Sigma$. The next step is to process iteratively the vertex $w \in pred(v)$ by generating a new resolvent $r = res(\eta(w), r, cl(w))$. According to the definition 3, r and c' are checked, then three cases are distinguished. If $r \subset c'$ (direct subsumption) or there exists a subsuming resolvent between r and c' then c' is reduced and search continues on the deduced sub-clause to get smaller one. If the two first cases do not apply, the search process is continued on the predecessor of one literal of r which does not appear in c'. Consequently, only one traversal of $\mathcal{G}_{ig}$ is needed which can be done in $O(n + m)$ where $n = |\mathcal{V}|$ and $m = |\mathcal{E}|$. As each clause of Σ is considered, and for each clause the number of traversal of $\mathcal{G}_{ig}$ is bounded by the length of the clauses, the worst case complexity of the global computation process is $|\Sigma| \times O(n+m) = O(|\Sigma| \times |var(\Sigma)|)$.*

Based on the proposition 2, we propose a polynomial time approach able to infer sub-clauses during the search.

4 Inferring sub clauses during search

In this section, we present a practical approach to deduce sub-clauses from a given formula Σ and an implication graph. As the well-known look-ahead [5] local treatment, the sub-clauses deduction can be very helpful for branching selection heuristic of any DPLL-like techniques, for detecting local inconsistencies and for reducing the size of the search tree. Let us describe the following algorithm $GetSubclause$ (Algorithm 3) that can be used to simplify a given formula Σ thanks to the implication graph $\mathcal{G}_{ig}$. We assume that y is one of the literals from the set of literals assigned at the current decision level α (this set is noted $\eta(\mathcal{V}_\alpha)$). Let A be a subset of literals from $\eta(\mathcal{V})$ such that $A_\wedge \rightarrow y$.

Algorithm 3 $GetSubclause$(**in** $\mathcal{G}, A, y, c, \alpha$))

If $\exists x_r \in \bar{A} \cup \{y\} | \neg x_r \in c$ **and** $\forall x \in (\bar{A} \cup \{y\}) - \{x_r\}, x \in c$ **then**
  $\Sigma \leftarrow \Sigma - \{c\} \cup \{c - \{x_r\}\}$
  If $pred(x_r) \neq \emptyset$ **then**
    $GetSubclause(\mathcal{G}, A - \{x_r\} \cup pred(x_r), y, c, \alpha)$

If $\forall x \in \bar{A} \cup \{y\}, x \in c$ **then**
  $\Sigma \leftarrow \Sigma - \{c\} \cup \{\bar{A_\vee} \vee y\}$
  $x \leftarrow choice(A)$
  If $pred(x) \neq \emptyset$ **then**
    $GetSubclause(\mathcal{G}, A - \{x\} \cup pred(x), y, c, \alpha)$
  else
    Choose $x \in A | x \notin c$ **and** $\neg x \notin c$
    If $pred(x) \neq \emptyset$ **then**
      $GetSubclause(\mathcal{G}, A - \{x\} \cup pred(x), y, c, \alpha)$

(a) $\mathcal{G}_{ig}(\Sigma_1, \{d\})$ (b) $\mathcal{G}_{ig}(\Sigma_2, \{z, d\})$

Figure 2: IG of formulas Σ_1 (example 2) and Σ_2 modified in section 4.1. The source is the literal d

Considering a clause c of Σ, the function $GetSubclause$ finds, if it exists, subclauses subsuming c (see line 7 of the algorithm 3) and new clauses whose generating resolvent clause with c subsumes c (see line 1 of the algorithm 3). When a direct subsumption is found, a clause c is subsumed directly by the implication $A_\wedge \rightarrow y$, so c contains y and all literals from $\bar{A}$. Note that in this case, we cannot produce any subsuming resolvent of c. However, we can expect finding a subset $B = \{x_1, ..., x_k\} \subset A$ such that $x_1 \wedge ... \wedge x_k \rightarrow y$. Thus, if it exists $x_1 \wedge ... \wedge x_k \rightarrow y$, then $\forall z | (z \in A, z \notin \{x_1, ..., x_k\}), \exists x_{i_1}, ..., x_{i_l} \subset B | x_{i_1} \wedge ... \wedge x_{i_l} \rightarrow z$. That's to say if one finds a shorter subsumption $B_\wedge \rightarrow y$ than $A_\wedge \rightarrow y$ then all literals appearing in A but not in B are implied by a subset of literals appearing in $A \cap B$. To find a shorter set B, we have to look into specific predecessors of literals of A. Since the literals appearing in $A \backslash B$ have precedessors in B, they have been assigned after some literals of B. To increase the probability to find at this step such an implication, the $choice$ function returns the variable of A that has been assigned last. Otherwise, the clause c will be subsumed in the same way when processing of literals of $A \backslash B$. Indeed, let z be a variable in this subset. When, during the computation of z, $B_z = \{z_1, ..., z_p\}$ will be found such that $(B_z)_\wedge \rightarrow z$, the clause $c_z = \neg z_1 \vee ... \vee \neg z_p \vee z$ will be deduced. While $z_1, ..., z_p, z \in A$, the clause deduced from $A_\wedge \rightarrow y$ contains literals $\neg z_1, ..., \neg z_p, \neg z$. The resolvent between c_z and the clause equivalent to $(A_\wedge \rightarrow z)$ allows to delete literal $\neg z$ from clause c. So is for all literals of $A \backslash B$. That's why c will be subsumed by the clause equivalent to $(B_\wedge \rightarrow y)$ whatever predecessor is chosen during the computation of y.

Finally, searching for all the subsumptions from the graph $\mathcal{G}_{ig}$ consists in: $\forall y \in \eta(\mathcal{V}_\alpha), \forall c \in \Sigma | y \in c, GetSubclause(\mathcal{G}, pred(y), y, c, \alpha)$.

Example 2 *To illustrate further, let us consider the Implication Graph of the Figure 2(a) obtained from Σ_1 by assigning d to $true$ on the formula*

$$\Sigma_1 = \left\{ \begin{array}{cc} c_1 : \neg d \vee x_1 & c_4 : \neg x_1 \vee \neg x_2 \vee x_4 \\ c_2 : \neg d \vee x_2 & c_5 : \neg x_3 \vee \neg x_4 \vee x_5 \\ c_3 : \neg x_1 \vee x_3 & c_6 : \neg d \vee \neg x_3 \vee x_5 \vee x_6 \\ & c_7 : \neg x_1 \vee x_2 \vee x_5 \end{array} \right\}$$

Trying to deduce a sub-clause of c_6 and considering implication of x_5 through the Implication Graph of the Figure 2(a), we will first consider $x_3 \wedge x_4 \rightarrow x_5$ which is

equivalent to the clause $\neg x_3 \vee \neg x_4 \vee x_5$. One can see that the variable x_3 belongs to the current implication and to the clause c_6. As x_4 does not belong to c_6, any implication containing this literal does not subsume c_6.

Then, we have to find other literals belonging to this subsumption from the predecessors of x_4: x_1 and x_2. Following our principle, x_1 and x_2 are not literals of c_6 and d is predecessor of both of them. Then, we can deduce $d \wedge x_3 \rightarrow x_5$ and the corresponding clause $c'_6 : \neg d \vee \neg x_3 \vee x_5$ subsumes c_6. Following the process on c'_6 upon the current set $A = \{x_3, d\}$, the function $choice$ chooses from A the last assigned variable: x_3. Through $\mathcal{G}_{ig}(\Sigma_1, \{d\})$, x_1 and d are sucessively visited and the implication $d \rightarrow x_5$ is deduced. The corresponding sub-clause $c''_6 : \neg d \vee x_5$ directly subsumes c_6. These two subsumptions from c_6 to c'_6 and then from c'_6 to c''_6 has been possible thanks to the function $choice$ which chose x_5. Indeed, if this function chose the variable d, the computation would have stopped without other deduction since d has no antecedent. Subsumption from c'_6 to c''_6 would have been found later, when calling $GetSubclause(G_{ig}, pred(x_3), x_3, c, \alpha)$. It will (trivialy) show $x_1 \rightarrow x_3$, and then $d \rightarrow x_3$, equivalent to the clause $\neg d \vee x_3$, whose resolvent with c'_6 is $c''_6 : \neg d \vee x_5$.

Applying this technique on the clause c_7 and from the same $\mathcal{G}_{ig}(\Sigma_1, \{d\})$, we can deduce the implication $x_1 \wedge x_2 \rightarrow x_5$. The resolvent between the corresponding clause of this implication and c_7 is $c_r : \neg x_1 \vee x_5$, which subsumes c_7. Finally, this implication graph allows to reduce c_6 by two literals and the clause c_7 by one literal. Such reductions can lead to further unit propagation that improve the search process.

Indeed, starting from original formula Σ_1, assigning x_5 to $false$ will not produce unit clause. However, with $c_6 : \neg d \vee x_5$ and $c_7 : \neg x_1 \vee x_5$, assigning x_5 to false implies $d = false$ and $x_1 = false$.

4.1 Local subsumption

Considering dynamic use of sub-clauses inference DPLL-like technique, the algorithm 3 previously described finds sub-clauses available only in the whole solving tree. As the number of such sub-clauses is restricted, the algorithm 3 can be improved to find subsumptions available only in part of the solving tree delimited by a decision level. Let β be this decision level. These subsumptions will be deleted when a backtrack occurs at or before decision-level β.

As previous version, considering a clause c of Σ and the set of literals $\eta(\mathcal{V}_\alpha)$ assigned at current decision level α, the function $GetSubClauseLevel$ tries to deduce subsumptions from c (see line 1 and 7 of the algorithm 4) available as long as all literals from $\bar{A}$, not belonging to c and whose decision level is different from α, keep their truth value. When a literal l from A has been assigned at a lower decision level than α and does not belong to c it can be ignored from A while it is assigned (i.e. backtrack on l is not yet occurred). $A \backslash \{l\}$ can be used to produce sub-clause of c. To illustrate this new method, let us consider the Implication Graph in the Figure 2(b), obtained from the formula Σ_2 when assigning d to $true$ at decision-level α and assigning z to $true$ at decision-level $\beta < \alpha$. Σ_2 is obtained from Σ_1, provided in example 2, by substituting the clause c_4 for the clause $c'_4 : \neg x_1 \vee \neg x_2 \vee \neg z \vee x_4$. From this graph, the implication $x_1 \wedge x_2 \wedge z \rightarrow x_5$, can be deduced. The corresponding clause is $c' : \neg x_1 \vee \neg x_2 \vee \neg z \vee x_5$. The resolvent between c' and c_7 is $c_r : \neg x_1 \vee \neg z \vee x_5$ which does not subsume c_7 be-

Algorithm 4 $GetSubClauseLevel$(**in** $\mathcal{G}, A, y, c, \alpha$))

if $\exists x_r \in \bar{A} \cup \{y\} | \neg x_r \in c$ **and** $\forall x \in ((\bar{A} \cup \{y\}) - \{x_r\}) \cap \eta(\mathcal{V}_\alpha), x \in c$ **then**
  $\Sigma \leftarrow (\Sigma - \{c\} \cup \{c - \{x_r\}\})_{dl > max_{x \notin c, x \in A \cap (\eta(\mathcal{V} - \mathcal{V}_\alpha))}(dl_x)}$
  if $pred(x_r) \neq \emptyset$ **then**
    $GetSubClauseLevel(\mathcal{G}, A - \{x_r\} \cup pred(x_r), y, c, \alpha)$
  end if
else
  if $\forall x \in (\bar{A} \cup \{y\}) \cap \eta(\mathcal{V}_\alpha), x \in c$ **then**
    $\Sigma \leftarrow (\Sigma - \{c\} \cup \{(\bar{A} \cap c)_\vee \vee y\})_{dl > max_{x \notin c, x \in A \cap \eta(\mathcal{V} - \mathcal{V}_\alpha)}(dl_x)}$
    $x \leftarrow choice(A)$
    if $pred(x) \neq \emptyset$ **then**
      $GetSubClauseLevel(\mathcal{G}, A - \{x\} \cup pred(x), y, c, \alpha)$
    end if
  else
    Choose $x \in A | x \notin c$ **and** $\neg x \notin c$
    if $pred(x) \neq \emptyset$ **then**
      $GetSubClauseLevel(\mathcal{G}, A - \{x\} \cup pred(x), y, c, \alpha)$
    end if
  end if
end if

cause z does not appear in this clause. However, since z has been assigned at a lower decision-level, c'_4 can be considered as only composed of $\neg x_1 \vee \neg x_2 \vee x_4$ while z keeps its current value. Thus, the resolvent $c_r : \neg x_1 \vee x_5$ subsumes c_7 until a backtrack occurs at decision level lower or equal to β. The implication $d \wedge z \rightarrow x_5$ can also be deduced and corresponds to the clause $c'' : \neg d \vee \neg z \vee x_5$. In the same way, c'' will subsume c_6 only if we consider that z will keep its current value, so that c'' is equivalent to $\neg d \vee x_5$ (for decision-level greater than β). So c_6 is directly subsumed. This implication graph allows to reduce c_6 by two literals and the clause c_7 by one literal, for decision-levels greater than β.

5 Preliminary comparative experimental results

Our sub-clause detection approach can be used as soon as BCP exists and then can be applied at each node of the DPLL search-tree. Let us recall this technique can be applied either if BCP leads to a conflict or not. In this section, we provide some preliminary comparative experimental results showing the impact which such an approach can have on a panel of formulae resulting from either industrial and structured problems. The goal of this preliminary experimentation is to measure the influence of our technique on the number of nodes developped in the DPLL search-tree when one exhaustively deduces the sub-clauses of a given formula. The exhaustive application of the sub-clause deduction at each node of the search-tree has a no inconsiderable increase of time consuming as a result. Within a practical framework, our sub-clause deduction approach should be both empirically and heuristically limited. For these experimentations, we only provide the size of the search-tree in terms of number of nodes and that independently of the computation time of our treatment. During the pretreatment, we apply our sub-clause detection approach trying to produce all the subsuming sub-clauses so that there is no possibility to deduce shortened clause from the initial formula. A dynamic implementation of this technique, like mentioned in previous section, applying at each node of the search-tree is our future work. Table 1 shows comparative

Instance	S/U	Zchaff nodes	Pretreatment+Zchaff nodes	subs	var fixed
barrel6	U	31 866	24 766	1207	342
barrel7	U	66 789	62 054	1 600	455
SAT.dat.k90	S	N/A	3 684 949	5 680	6 373
logistics.b	S	3 810	422	859	405
logistics.c	S	9 577	1 282	1 434	557
abp4-1-k31-unsat. shuffled-as.sat03-403	U	N/A	0	106	2 712
abp1-1-k31-unsat. shuffled-as.sat03-402	U	N/A	0	178	2 878
2bitadd_10	U	60 605	60 605	0	0
2bitadd_11	S	7 870	7870	0	0
longmult6	U	5 833	5 949	442	690
longmult7	U	26 942	19 457	496	730
longmult9	U	273 182	273 836	631	798
longmult10	U	711 397	562 517	689	826
longmult12	U	1 164 158	906 626	824	870
longmult13	U	1 567 022	997 981	956	892
longmult15	U	625 534	286 858	1 269	1 057
bf0432-007	U	864	295	427	363
bf2670-001	U	64	0	212	290
flat200-10	S	14 202	14 202	0	0
flat200-100	S	1 157	1 157	0	0

Table 1: Preliminary comparative results

results on selected benchmarks in terms of number of decisions between standalone state-of-the-art solver Zchaff[1] and our sub-clause deduction approach helping Zchaff as a pretreatment. All the results have been computed on AMD Athlon 2000+ with 512Mo RAM under Linux/OS. Table 1 provides also the number of deduced subsumptions and the number of fixed variables. Note that in column "S/U", "S" means "Satisfiable" and "U" means "Unsatisfiable". Finally, Zchaff has a timeout of 7200 seconds. For the class of unsatisfiable instances longmult, we can note for Zchaff applied on pretreated benchmarks by the sub-clauses deduction approach a gain greater than 50% in comparison with Zchaff without the sub-clause deduction pretreatement. For some of them like abp*, our pretreatment proves the unsatisfiability before Zchaff runs. Note that for these formulas, Zchaff was not able to response in less than 7200 seconds, and comparatively, the pretreatment is less than one minute time computing. However, no subsumption are found while processing families flat* and 2bitadd* although any BCP exists.

[1] Zchaff version 2004.11.15

6 Conclusion and future work

In this paper a new extension of the scope of boolean constraint propagation is presented. We have described two possible ways to translate the BCP implication graph to a resolution tree. The second translation, gives us a new picture on BCP, usually considered as limited form of resolution. Indeed, many interesting and new resolvent can be generated using the BCP implication graph leading to a powerful resolution-based technique. To make such extension practicable, we have shown that when a subset of such resolvent (those that achieve a sub-clause deduction) are considered, we obtain a polynomial time approach that can be grafted to DPLL-like techniques. Clearly, our preliminary experimental are encouraging. On some classes of instances, a substantial reduction on the number of nodes has been obtained. To substantiate our claim on the usefulness of the proposed approach, further experimental validation are needed. Using different criteria, we also plan to investigate in a systematic way the pertinence of a given resolvent with respect to its practical potential.

References

[1] F. Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *AAAI*, 2002.

[2] A. Biere, E. Clarke, R. Raimi, and Y Zhu. Verifying safety properties of a PowerPC microprocessor using symbolic model checking without BDDs. *Lecture Notes in Computer Science*, 1633:60–72, 1999.

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

[4] G. Dequen and O. Dubois. kcnfs: An efficient solver for random k-SAT formulae. In *International Conference on Theory and Applications of Satisfiability Testing (SAT), Selected Revised Papers, LNCS*, volume 6, pages pp 486–501, may 2003.

[5] O. Dubois, P. André, Y. Boufkhad, and J. Carlier. Sat versus unsat. In *Second DIMACS Challenge*, pages 415–436, 1996.

[6] E. Grégoire, B. Mazure, R. Ostrowski, and L. Sais. Automatic extraction of functional dependencies. In *SAT'04*, May 2004.

[7] H. Kautz and B. Selman. Planning as satisfiability. In *ECAI'92*, pages 359–363, Vienna, Austria, 1992.

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

[9] C. M. Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *IJCAI'97*, pages 366–371, Nagoya (Japan), August 1997.

[10] J.P.M. Silva and K.A. Sakallah. Grasp - a new search algorithm for satisfiability. In *CAD'96*, 1996.

[11] J.P.M. Silva and K.A. Sakallah. Boolean satisfiability in electronic design automation. In *DAC'00*, June 2000.

[12] L. Zhang, C. Madigan, M. Moskewicz, and S. Malik. Efficient conflict driven learning in a boolean satisfiability solver. In *ICCAD'01*, pages 279–285, San Jose, CA (USA), November 2001.