Proceedings of the Seventh Annual Symposium on Combinatorial Search (SoCS 2014)

Handling All Unit Propagation Reasons in Branch and Bound Max-SAT Solvers

André Abramé and Djamal Habet
Aix Marseille Université, CNRS, ENSAM, Université de Toulon,
LSIS UMR 7296, 13397, Marseille, France.
emails: {andre.abrame,djamal.habet}@lsis.org

Abstract

Unit propagation (UP) based method are widely used in Branch and Bound (BnB) Max-SAT solvers for detecting disjoint inconsistent subsets (IS) during the lower bound (LB) estimation. UP consists in assigning to true (propagating) all the literals which appear in unit clauses. The existing implementations of UP only consider the first unit clause causing the assignment of each variable, thus the propagations must be done and undone chronologically to ensure that all the unit clauses are properly exploited. Max-SAT BnB solvers transforms the formulas to ensure IS disjointness. These transformations remove clauses from the formula thus propagations are frequently undone. Since the propagations are undone in chronological order, many useless unassignments and reassignments are performed. We propose in this paper a new unit propagation scheme which considers all the unit clauses causing the assignment of the variables by UP. This new scheme allows to undo propagations in a non-chronological way and thus it reduces the number of redundant propagation steps made by BnB solvers. We also show how the information available with this new scheme can be used to influence the characteristics of the IS built by BnB solvers. We propose a heuristic which aims at reducing their size, and thus improving the quality of the LB estimation. We have implemented the new propagation scheme as well as the IS building heuristic in our solver MSSOLVER. We present and discuss the results of the experimental study we have performed.

Introduction

The Max-SAT problem consists in finding, for a given CNF formula, a Boolean assignment of the variables of this problem which maximizes (minimizes) the number of satisfied (falsified) clauses. This NP-hard problem (Papadimitriou 1994) is the optimization version of the SAT problem. It has a wide range of applications since many problems can be expressed as Max-SAT instances in both theoretical (Max-Clique, Max-Cut, etc.) and real-life (routing (Xu, Rutenbar, and Sakallah 2002), bioinformatics (Strickland, Barnes, and Sokol 2005), etc.) domains. In the weighted version of the Max-SAT problem, a weight is associated to each clause and the goal is to find an assignment which maximizes (minimizes) the sum of the weights of the satisfied (falsified) clauses. There are other variants of Max-SAT (partial and weighted partial) which are not considered in this paper.

Among the complete methods for solving Max-SAT, Branch and Bound (BnB) algorithms (e.g. WMAXSATZ (Li et al. 2009; Li, Manyà, and Planes 2006; 2007)) have shown there efficiency, especially on random and crafted instances. One of the most critical components of BnB solvers is the estimation of the lower bound (LB). On the one hand it is applied very often and thus it takes a large part in the solvers execution time and on the other hand the quality of the LB estimation leads the backtrack and thus determines the number of explored nodes of the search tree.

The LB estimation consists in counting the weights of the clauses which will be falsified when extending the current partial interpretation. To do so, efficient BnB solvers use unit propagation based methods to detect disjoint inconsistent subsets (set of clauses which cannot be all satisfied). Each detected inconsistent subset (IS) is then transformed to ensure it will be counted only once. The transformations applied on IS have the particularity to remove the IS clauses from the formula. Consequently, propagated variables must be frequently unset.

To the best of our knowledge, all the existing Max-SAT implementations using unit propagation store only the first clause (first predecessor) causing the propagation of each variable (the others are simply ignored). When such a clause is removed from the formula, the propagated variable must be unset. To ensure that any other previously ignored variable's predecessor is now considered, all the assignments made after the variable propagation must also be undone. Thus, the propagations are undone in reverse chronological order. In the Max-SAT context, unit propagation is intensively used and the clauses are frequently removed from the formula. Thus this scheme can cause many useless redundant propagation steps.

We first present in this paper a new propagation scheme which takes into consideration all the propagation sources of the variables rather than only the first one as it is usually done. To the best of our knowledge, this subject has only been studied in the SAT context from a theoretical point of view (Van Gelder 2011) and in a very limited way for improving the backjump level (Audemard et al. 2008) of Conflict Driven Clause Learning SAT solvers (Marques-Silva and Sakallah 1999). This propagation scheme allows solvers to maintain propagated literals in a non-chronological way and thus to make less useless redundant propagation steps. We discuss of the advantages and drawbacks of this scheme and present its implementation in a BnB Max-SAT solver. To the best of our knowledge, the multiple predecessors scheme (MPS) has never been implemented before, neither for SAT nor Max-SAT. In the second part of this paper, we propose to exploit the information available with MPS to reduce the size of the IS build by BnB solvers. We present a heuristic which choose among the predecessors of the variables which participate to the conflict the ones which must be added to the IS. Eventually, we present and discuss the experimental study we have performed.

Definitions and Notations

A weighted formula Φ in conjunctive normal form (CNF) defined on a set of propositional variables $X = \{x_1, \ldots, x_n\}$ is a conjunction of weighted clauses. A weighted clause c_j is a weighted disjunction of literals and a literal l is a variable x_i or its negation $\overline{x}_i$. Alternatively, a weighted formula can be represented as a multiset of weighted clauses $\Phi = \{c_1, \ldots, c_m\}$ and a weighted clause as a tuple $c_j = (\{l_{j_1}, \ldots, l_{j_k}\}, w_j)$ with $\{l_{j_1}, \ldots, l_{j_k}\}$ a set of literals and $w_j > 0$ the weight of the clause. Unweighted formulas and clauses are weighted ones with all the clause weights set to 1. We denote the number of clauses of Φ by $|\Phi|$ and the number of literals of c_j by $|c_j|$.

An assignment can be represented as a set I of literals which cannot contain both a literal and its negation. If x_i is assigned to $true$ (resp. $false$) then $x_i \in I$ (resp. $\overline{x}_i \in I$). I is a complete assignment if $|I| = n$ and it is partial otherwise. A literal l is said to be satisfied by an assignment I if $l \in I$ and falsified if $\bar{l} \in I$. A variable which does not appear either positively or negatively in I is unassigned. A clause is satisfied by I if at least one of its literals is satisfied, and it is falsified if all its literals are falsified. By convention, an empty clause (denoted by $\Box$) is always falsified. A subset ψ of Φ is inconsistent if there is no assignment which satisfies all its clauses. For a unit assignment $I = \{l\}$, we denote by $\Phi|_I$ the formula obtained by applying I on Φ. Formally: $\Phi|_I = \{c_j \mid c_j \in \Phi, \{l, \bar{l}\} \cap c_j = \emptyset\} \cup \{c_j/\{\bar{l}\} \mid c_j \in \Phi, \bar{l} \in c_j\}$. This notation can be extended to any assignment $I = \{l_1, l_2, \ldots, l_k\}$ as follows: $\Phi|_I = (\ldots((\Phi|_{\{l_1\}})|_{\{l_2\}})\ldots|_{\{l_k\}})$. Solving the weighted Max-SAT problem consists in finding a complete assignment which maximizes the sum of the weights of the satisfied clauses of Φ. Two formulas are equivalent for (weighted) Max-SAT iff they have the same sum of falsified clause weights for each assignment.

Inconsistencies Detection and Handling

BnB solvers explore the whole search space and compare, at each node of the search tree, the current sum of the falsified clause weights plus an (under-)estimation of the ones which will become falsified (the lower bound, LB) to the best solution found so far (the upper bound, UB). If LB $\geq$ UB, then no better solution can be found by extending the current branch and they perform a backtrack. The estimation of the remaining inconsistencies is critical in two ways. Firstly, it is applied very often and its computing time has an important impact on a solver's efficiency. Secondly, its quality determines the number of explored nodes. Simplistically, this estimation can be divided into two distinct (but closely linked) parts: (1) the detection of the disjoint inconsistent subsets of clauses and (2) their treatment. We describe these two elements in the rest of this section.

Detecting Inconsistent Subsets

Recent BnB Max-SAT solvers apply unit propagation (UP) based methods to detect inconsistent subsets (more precisely simulated unit propagation (Li, Manyà, and Planes 2005) and failed literals (Li, Manyà, and Planes 2006)). For each unit clause $\{l\}$, they remove all the occurrences of $\bar{l}$ from the clauses and all the clauses containing l. This process is repeated until an empty clause (a conflict) is found or no more unit clause remains. Unit clauses $\{l\}$ are called l predecessors and the clauses which are reduced by l are its successors. When an empty clause is found by UP, an inconsistent subset (IS) of the formula can be built by analyzing the propagation steps which have led to the conflict.

Transforming Inconsistent Subsets

Once detected by UP, IS are transformed to ensure that they are counted only once. Two transformations are actually applied by recent BnB solvers. The first one consists in simply removing the clauses of the IS from the formula. It is fast but the resulting formula is not equivalent to the original one and may contains less inconsistent subsets. The second transformation is close to the clause learning mechanism of modern SAT solvers (Marques-Silva and Sakallah 1999). It consists in applying several max-resolution steps (the Max-SAT version of the SAT resolution (Bonet, Levy, and Manyà 2007; Heras and Larrosa 2006; Larrosa and Heras 2005)) between the clauses of the IS. Note that both these transformations remove the original clauses of the IS from the formula.

First Predecessor Scheme

To the best of our knowledge, all the existing Max-SAT solvers (as well as the SAT ones) use the first predecessor scheme (FPS): they only consider the first predecessor of each propagated variable. They memorize the propagation steps by an implication graph, which can be defined as follows (see for instance (Marques-Silva and Sakallah 1999) for a definition in the SAT context).

Definition 1 (Implication Graph). *Let $\Phi = \{c_1, \ldots, c_m\}$ be a (weighted) CNF formula defined on a set of Boolean variables $X = \{x_1, \ldots, x_n\}$ and I a partial assignment (with both decisions and propagations) of the variables of X. We assume that there can be only one falsified clause, i.e. UP is stopped when a conflict is discovered. An implication* *graph is a directed labeled acyclic graph $G = (V, A)$ with:*

$$\begin{aligned} V &= \{l \in I\} \cup \{\Diamond_{c_i} \textit{ s.t. } \exists c_i \in \Phi, |c_i| = 1\} \cup \\ &\quad \{\Box \textit{ if } \exists c_j \in \Phi \textit{ falsified by } I\} \\ A &= \{(l, l', c_k) \textit{ s.t. } \exists c_k \in \Phi \textit{ which is reduced by } l \textit{ and} \\ &\quad \textit{which is the first predecessor of } l'\} \cup \\ &\quad \{(\Diamond_{c_p}, l, c_p) \textit{ s.t. } \exists c_p = \{l\} \in \Phi\} \cup \\ &\quad \{(l, \Box, c_q) \textit{ s.t. } \exists c_q \in \Phi \textit{ falsified by } I \textit{ and } \bar{l} \in c_q\} \end{aligned}$$

We use the two special nodes $\Diamond$ and $\Box$ to represent respectively the initial vertices of the unit clauses and the terminal one of the falsified clause. For clarity reason, we hide the nodes $\Diamond$ in the graphical representation of the implication graphs. Each arc is labeled with the clause it comes from.

As we have seen in the previous section, BnB Max-SAT solvers frequently remove clauses from the formula causing the unassignment of propagated variables. If the first predecessor clause of a propagated literal l is removed then l must be unassigned, whether or not it has other predecessors. Moreover, all the propagations which have been made afterwards must be undone to ensure that an eventual other predecessor of l is not ignored. Among the undone propagations, some may still have a valid predecessor after l unassignment and will be immediately re-propagated. Thus FPS can cause unnecessary unassignments and reassignments. The following example illustrates this situation.

Example 1. *Let us consider the unweighted formula $\Phi_1 = \{c_1, \ldots, c_{10}\}$ with $c_1 = \{x_1\}$, $c_2 = \{\overline{x}_1, x_2\}$, $c_3 = \{\overline{x}_1, \overline{x}_2, x_3\}$, $c_4 = \{\overline{x}_2, x_4\}$, $c_5 = \{x_5\}$, $c_6 = \{\overline{x}_5, x_2\}$, $c_7 = \{x_6\}$, $c_8 = \{\overline{x}_6, x_7\}$, $c_9 = \{\overline{x}_6, x_3\}$ and $c_{10} = \{\overline{x}_6, \overline{x}_7, \overline{x}_3\}$. The application of UP on Φ_1 (with the UP* ordering (Li, Manyà, and Planes 2006)) leads to the sequence of propagations $< x_1@c_1, x_2@c_2, \ldots, x_5@c_5, x_6@c_7, x_7@c_8 >$ (meaning that x_1 is propagated by clause c_1, then x_2 by c_2, etc.). The clause c_{10} is empty. Fig. 1 shows the corresponding implication graph. Note that the clauses c_6 and c_9, which are predecessors (but not the first ones) of respectively x_2 and x_3 are not represented in the implication graph. The set of clauses $\psi_1 = \{c_1, c_2, c_3, c_7, c_8, c_{10}\}$ which have led by propagation to the conflict (i.e. to the empty clause c_{10}) is an inconsistent subset of Φ_1. If the clauses of ψ_1 are removed from the formula, then all the propagations caused by these clauses must be undone. The less recent ones is $x_1@c_1$ and since the propagations are undone in reverse chronological order in FPS, all the propagations are undone. We obtain the formula $\Phi'_1 = \{c_4, \ldots, c_6, c_9\}$, and the application of UP on Φ'_1 leads to sequence of propagation $< x_5@c_5, x_2@c_6, x_4@c_4 >$. Note that these three variables have been consecutively unassigned and reassigned.*

Multiple Predecessors Scheme

To the best of our knowledge, all the Max-SAT complete solvers use the first predecessor scheme (FPS): they only consider the first unit clauses causing the propagation of the variables. We propose in this section to consider all the predecessors of the variables. The resulting multiple predecessors scheme (MPS) deeply changes the way the unit propagation is applied by Max-SAT solvers. In MPS, all the predecessors of each variables are stored rather than only the first one. Propagated variables are undone only when they have no more predecessor. This way, propagations can be undone in a non-chronological order and fewer unnecessary propagation steps are performed. Moreover, the conflicts are detected earlier in the multiple predecessors scheme. If a not yet propagated variable has predecessors of both polarities, then at least one of them will be falsified when assigning the variable.

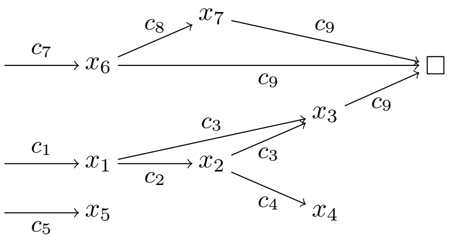

Figure 1: Implication graph of the formula Φ_1 from Example 1. Nodes are the propagated literals and arrows are labeled with the clauses causing the propagations. Note that the clauses c_6 and c_9 are not the first predecessors of the variables they propagate (respectively x_2 and x_3) thus they are not represented in the implication graph.

In the rest of this section, we first give some basic definitions on the multiple predecessors scheme and we discuss its advantages and drawbacks and its implementation in BnB Max-SAT solvers.

Full Implication Graph

In a FPS based solver, the sequence of propagation steps is usually modeled by the implication graph: each instantiated variable is represented by a node (root nodes represent the decisions while the other nodes represent the propagations) and arrows link the reasons of the propagations (the falsified literals of the clauses) to their consequences (the propagated literals). Such a representation is not suited to a MPS based solver since it is not possible to distinguish several predecessors of a literal from the multiple reasons (the falsified literals) of a single one. We thus define a new structure to model the propagation steps in MPS.

Definition 2 (Full Implication Graph). *Let $\Phi = \{c_1, \ldots, c_m\}$ be a (weighted) CNF formula defined on a set of Boolean variables $X = \{x_1, \ldots, x_n\}$ and I a partial assignment (with both decisions and propagations) of the variables of X. We assume that there can be only one falsified clause, i.e. UP is stopped when a conflict is discovered. A full implication graph is an AND/OR directed acyclic graph $G = (V_{or}, V_{and}, A)$ where V_{or} is the set of the OR nodes which represent the assigned variables, V_{and} is the set of the AND nodes which represent the unit clauses and A is the set of arrows which link the unit clauses (the predecessors) to the variables they propagate and the assigned variables* *to the clause they reduce (the successors). Formally:*

$$\begin{aligned} V_{or} &= \{l \in I\} \cup \{l \textit{ s.t. } \exists c_j = \{l\} \in \Phi|_I\} \\ V_{and} &= \{c_k \in \Phi \textit{ s.t. } c_k \textit{ contains only one literal not} \\ &\quad \textit{falsified by } I\} \\ A &= \{(l, c_p) \textit{ s.t. } c_p \in V_{and} \textit{ is a successor of } l \in I\} \\ &\quad \{(c_q, l) \textit{ s.t. } c_q \in V_{and} \textit{ is a predecessor of } l \in I\} \end{aligned}$$

Example 2. *Let us consider the formula Φ_1 from Example 1. The application of UP with MPS leads to the same six first propagation steps done in Example 1: $< x_1@c_1, x_2@c_2, \dots, x_5@c_5, x_6@c_7 >$. At this point, variable x_7 has two predecessors of opposite polarities, c_8 and c_{10}, which cannot be both satisfied. Fig. 2 shows the full implication graph obtained in a MPS based solver. One can note that the second predecessors of x_2 and x_3 (respectively c_6 and c_9) are represented in the full implication graph. As in the previous example, we can build the inconsistent subset $\psi_1 = \{c_1, c_2, c_3, c_7, c_8, c_{10}\}$ by taking the first predecessor of each propagated variable which have led to the conflict. If ψ_1 is removed from the formula, the variables x_1, x_3, x_6 and x_7 have no more predecessor and must be unset. Note that x_5, x_2 and x_4 remain propagated since they all have still at least one predecessor.*

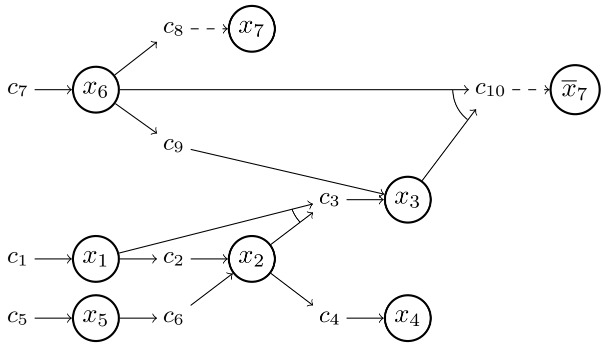

Figure 2: Full implication graph of the formula Φ_1 from Example 2. The circled nodes are the propagated variables while the uncircled nodes are unit clauses. Note that contrary to the implication graph of Fig 1, the predecessors c_6 of x_2 and c_9 of x_3 are represented.

Cycle Problem

The multiple predecessors scheme has the drawback of producing cycles in the full implication graph. There is a cycle in the full implication graph when a propagated variable x_i causes, by one or more propagation steps, a new unit clause to be one of its own predecessors. In such a case, when all the other predecessors of x_i are removed, x_i remains propagated. However, falsifying x_i does not falsify any clause, because the propagations depending on x_i are undone and so is its own remaining predecessor. Thus, taking these cycles into account leads to false conflict detection.

Example 3. *Let $\Phi_2 = \{c_1, c_2, \dots, c_6\}$ be a CNF formula, with $c_1 = \{x_1\}$, $c_2 = \{x_2\}$, $c_3 = \{\overline{x}_1, x_3\}$, $c_4 = \{\overline{x}_3, x_1\}$, $c_5 = \{\overline{x}_3, x_4\}$ and $c_6 = \{\overline{x}_2, \overline{x}_4\}$. x_4 has predecessors of both polarities and thus Φ_2 is inconsistent. Fig. 3 shows the full implication graph of Φ_2. If we remove c_1 from Φ_2 to obtain Φ'_2, x_1 is still propagated since it has a predecessor c_4, and the conflict is still present in the full implication graph (see Fig. 4). However, Φ'_2 is not inconsistent anymore and the assignment $I =< \overline{x}_1, x_2, \overline{x}_3, \overline{x}_4 >$ satisfies all its clauses.*

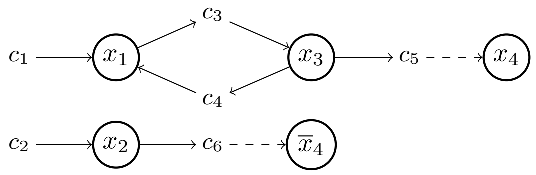

Figure 3: Full implication graph of the formula Φ_2 from Example 3. A cycle is present between x_1 and x_3 and it is kept up by clause c_1 which propagates x_1.

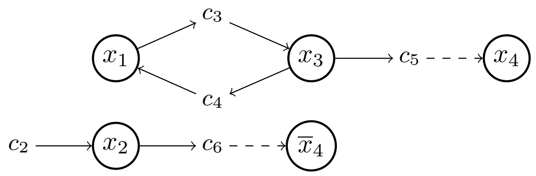

Figure 4: Full implication graph of the formula Φ'_2 from Example 3. Again, the cycle between x_1 and x_3 is present but it is not kept up by any other predecessors.

A workaround to avoid the detection of false conflicts is to ignore the predecessors which may create cycles. A simple way to do so is to use the level of the variables and clauses in the full implication graph to ignore the reversing arrows.

Definition 3 (level in a full implication graph). *The level of a variable x_i in a full implication graph can be defined as follows: $level(x_i) =$*

$$\begin{cases} 0, \textit{ if } x_i \textit{ is a decision} \\ max\{level(c) \mid c \textit{ predecessor of } x_i\}, \textit{ if } x_i \textit{ is propagated} \\ +\infty, \textit{ otherwise} \end{cases}$$

The level of a literal l is the level of its variable and the level of a clause c_j can be defined as:

$$level(c_j) = max(\{level(l) \mid l \in c_j, l \textit{ falsified }\}) + 1$$

Proposition 1. *The predecessor c_j of a propagated variable x_i cannot create any cycle if $level(x_i) \geq level(c_j)$.*

When a new predecessor is of a higher level than the variable that it propagates, then it is simply ignored. This way, and at a low computational cost, a solver can use the multiple predecessor scheme without the cycles drawback. It should be noted that we tried to detect cycles by exploring the full implication graph when a new predecessor of higher level is added or when no more active predecessor remains (to avoid undoing propagation if not necessary). In both cases, the time consumed in analyzing the implication graph was higher than its benefits.

Implementation and Complexity

We have implemented MPS as a core component of our experimental solver MSSOLVER. Each variable keeps two statically allocated lists for its positive and negative predecessors. When a clause c_j becomes a unit clause, MSSOLVER computes its level and identifies the propagated variable x_i (the only non-falsified variable of c_j). Then it adds c_j to the appropriate predecessors list of x_i. The level of the variables are computed when they are assigned. Eventually, when a conflict is detected (a variable has predecessors of both polarities), MSSOLVER builds the corresponding inconsistent subset by analyzing the full implication graph and transforms it as other performing BnB solvers do. Propagated variables are unset only when they have no more predecessor.

MPS does not change the overall complexity of our algorithm. Since there cannot be more predecessors than the number of clauses of the instance, the worst case complexity of the IS computation mechanisms (unit propagation, inconsistent subset building and transformation) remains unchanged. In practice however more clauses will be examined during the unit propagation process and when building the inconsistent subset. Moreover, the underlying data structures must be adapted to support non-chronological propagations.

Reducing the Inconsistent Subset Sizes

We present in this section how MPS can be used to influence the characteristics of the inconsistent subsets built by BnB solvers. The structure and properties (size, size of their clauses, etc.) of the inconsistent subsets generated when analyzing conflicts have an important impact on the solver's ability to detect remaining conflicts. Especially, reducing the size of the IS allows more clauses to be available for unit propagation. It may improve the estimation of the lower bound and thus reduce the number of explored nodes of the search tree.

When FPS based solvers find a conflict (an empty clause), they build a corresponding inconsistent subset (Algorithm 1) by analyzing the sequence of propagation steps which have led to the conflict. For each propagated variable of this sequence, its predecessor is added to the inconsistent subset. Since only the first predecessor of each propagated variable is known, only one inconsistent subset can be built. The following functions are used in the algorithm: `select_and_remove_variable(`Q`)` selects and removes the variable of higher level from the queue Q and `first_predecessor(`v`)` return the first predecessor of a variable v.

MPS based solvers can choose, for each variable which has led to a conflict, the predecessor they add to the corresponding inconsistent subset. Thus MPS based solvers can influence the structure of the generated inconsistent subsets. We propose here a very simple heuristic (Algorithm 2) to select the predecessors used in the inconsistent subsets. For each propagated variable which leads to the conflict, the solver adds to the inconsistent subset the predecessor which adds the fewest new propagated variables to the queue Q. We use the following functions: `predecessors(`*x,pol*`)`

Algorithm 1: Inconsistent subset building in a first predecessor scheme

```
Data: A CNF formula $\Phi$, an assignment $I$ and a falsified
      clause $c$.
Result: $IS$ an inconsistent subset of $\Phi$.
1 begin
2     $IS \leftarrow \{c\}$;
3     $Q \leftarrow \{$propagated variables of $c\}$;
4     while $Q \neq \emptyset$ do
5         $v \leftarrow$select_and_remove_variable($Q$);
6         $c \leftarrow$first_predecessor($v$);
7         $IS \leftarrow IS \cup \{c\}$;
8         $Q \leftarrow Q \cup \{$propagated variables of $c/\{v\}\}$;
9     return $IS$;
```

returns the sets of the predecessors of the variable x with polarity *pol*, `select_smallest_couple(`*A,B*`)` returns the couple of clauses (c_1, c_2) from two sets of clauses A, B such as $\forall(c_1', c_2') \in A \times B, |c_1 \cup c_2| \leq |c_1' \cup c_2'|$ and `value` (I, v) return the value of the variable v in the assignment I. We refer to this heuristic as the smallest intermediary resolvent (SIR) heuristic in the rest of this paper. It should be noted that the SIR heuristic does not necessarily produces the smallest possible resolvent.

Algorithm 2: Inconsistent subset building in a multiple predecessors scheme with the smallest resolvent heuristic

```
Data: A CNF formula $\Phi$, an assignment $I$ and a variable $v$
      with predecessors of both polarities.
Result: $IS$ an inconsistent subset of $\Phi$.
1  begin
2      $(c_1, c_2)$ =select_smallest_couple(
       predecessors($v$,$true$), predecessors($v$,
       $false$));
3      $IS \leftarrow \{c_1, c_2\}$;
4      $Q \leftarrow \{$propagated variables of $c_1\} \cup$
       $\{$propagated variables of $c_2\}$;
5      while $Q \neq \emptyset$ do
6          $v \leftarrow$select_and_remove_variable($Q$);
7          $(Q, c) \leftarrow$select_smallest_couple($\{Q\}$,
           predecessors($v$, value($I, v$)));
8          $IS \leftarrow IS \cup \{c\}$;
9          $Q \leftarrow Q \cup \{$propagated variables of $c/\{v\}\}$;
10     return $IS$;
```

Example 4. *Let us consider the formula Φ from the Example 1. In a first predecessor scheme (Fig. 1), the inconsistent subset computed from the conflict by considering only the first predecessors of each propagated variables is $\psi_1 = \{c_1, c_2, c_3, c_7, c_8, c_{10}\}$. In a multiple predecessors scheme (Fig. 2), the smallest resolvent heuristic works as follow. It start by adding to the queue Q the falsified literals (x_3 and x_7) of the predecessors of both polarities (c_8 and c_{10}) of the conflicting variable x_7. Then, it picks the variables of $Q = \{x_3, x_6\}$ of higher level, x_3. x_3 has two predecessors c_3 and c_9. The heuristic chooses the predecessor which adds* *the less new literals to the queue Q: c_9. Only one literal x_6 remains in Q, which have a single predecessor c_7. Thus the inconsistent subset built is $\psi_2 = \{c_7, \ldots, c_{10}\}$ which contains two clauses less than ψ_1. These clauses can be used to continue applying unit propagation and can potentially lead to the detection of new conflicts.*

Experimental Study

The propagation scheme is an important part of BnB solvers. It determines the way the propagations are done and undone, how the conflicts are analyzed and the underlying data structures. In our MPS implementation these parts represent 40 to 50% of the source code and in average 60 to 70% of the execution time. The performances of a variant of our solver implementing FPS would be highly dependent of the quality of the implementation. Rather than comparing our solver to such a variant, we have evaluated the potency of the multiple predecessor scheme by three set of experiments. We have first estimated the percentage of propagation steps saved thanks to MPS. Then, we have evaluated the impact of the IS building heuristic on the solver behavior. Finally, we have compared our solver to the best performing BnB solvers of the last Max-SAT Competition.

The tests presented in this section are performed on all the random and crafted instances of the Max-SAT and Weighted Max-SAT categories of the Max-SAT Competition 2013[1]. We include neither (weighted) Partial Max-SAT instances nor industrial ones in our experiments. Even if the results presented in this paper can naturally be extended to these instance categories, our solver MSSOLVER does not handle them efficiently. A performing BnB solver for (weighted) Partial Max-SAT must handle both the soft and the hard parts of the instances. Thus, it must include SAT mechanisms such as nogood learning, activity-based branching heuristic or backjumping and our solver currently does not. For the industrial instances, solvers must have a very efficient memory management. To the best of our knowledge, none of the best performing BnB solvers (including ours) handles huge industrial instances efficiently. The experiments are performed on a cluster of servers equipped with Intel Xeon 2.4 Ghz processors, 24 Gb of RAM and running under a GNU/Linux operating system. The cutoff time for each instance is fixed to 1800 seconds.

Measuring of the Saved Propagations

We have first estimated the percentage of propagation steps saved thanks to MPS. In a dedicated variant of our solver, we have kept in parallel of MPS a chronologically ordered list of the propagations to simulate the first predecessor scheme. When the predecessor of a propagated variable is remove from the formula, the variable would have been unassigned in a FPS based solver. In our solver, if the variable has more than one predecessor, then it is not unassigned. That's what we call the "directly" saved propagation steps. We have also measured the propagation steps which would have been undone due to the reverse-chronological order of the unassignment by counting the number of propagated variables which still have predecessor and which have been propagated after the less recent propagation to be undone. We call these last saved propagation steps "indirect".

Let us first recall that unit propagation is used very intensively by BnB Max-SAT solvers. In MSSOLVER, in average almost 2000 propagation steps are performed at each decision and the total average number of propagation steps per solved instance is roughly 60 million. The percentage of direct and indirect saved propagation steps is shown in Table 1, column PS. In average, MPS reduce of 24.1% the number of propagation steps made by MSSOLVER. The reduction can go up to almost 60% on some instance classes. Note that the direct propagation steps saved are in average inferior to 1%, while the indirect ones are greater than 23%.

IS Building Heuristic

We have implemented three variants of our solver MSSOLVER which differ by the way they choose the propagated variable predecessors when they build the inconsistent subsets:

- MSSOLVER^F picks the first predecessor.
- MSSOLVER^R picks randomly one predecessor.
- MSSOLVER^H picks predecessors according to the SIR heuristic presented above.

Table 1 compares the results obtained with these three variants. We can first observe that the SIR heuristic improves slightly the LB estimation, thus MSSOLVER^H make less decisions than the two other variants (-3% in average, columns D). Consequently, the average solving time is also reduced (respectively -7.3% and -9,2% compared to the ones of MSSOLVER^F and MSSOLVER^R, columns T).

The gain in solving time, although being significant, is not outstanding. A first possible explanation is that the SIR heuristic is naive. A more complex IS building heuristic may reduce more efficiently the IS sizes and thus improve further the solving time. Another possible explanation lie in the fact that the impact of the transformed IS inner structure on the unit propagation process is not well known. The IS size may not be the only important criterion to consider when choosing the propagated variables predecessors which are added to the IS. A thoughtful study of these interactions may lead to establishing a more efficient IS building heuristic.

Comparison with State of the Art

We have compared the variant of our solver using the SIR heuristic, MSSOLVER^H, to the two best performing BnB solvers of the Max-SAT Competition 2013: WMAXSATZ2009 and WMAXSATZ2013 (Li et al. 2009; Li, Manyà, and Planes 2006; 2007). The results (Table 2) show that our solver is quite competitive. It solves 41 instances more than WMAXSATZ2009 and 7 more than WMAXSATZ2013. In terms of solving time, our solver is respectively 58% and 31% faster. It should be noted however that MPS is not the only specificity of our solver over the state of the art ones. These results show that a solver using MPS can be competitive with the state of the art ones.

[1]Available from http://maxsat.ia.udl.cat:81

Table 1: Comparison of the IS building heuristic in MSSOLVER. The two first columns give the instances classes and the number of instances per class. The third column PS gives the average percentage of estimated saved propagations steps. For each tested variant of MSSOLVER, the columns S, D and T give respectively the number of solved instances, the average number of decisions and the average solving time. Columns marked with a star take in consideration only the instances solved by all solvers.

	instances classes	#	PS	MSSOLVER^H			MSSOLVER^F			MSSOLVER^R		
				S	D*	T	S	D*	T	S	D*	T
unweighted	crafted/bipartite	100	11.7%	100	34909	**95.8**	100	36451	100.3	100	37255	103.4
	crafted/maxcut	67	10.8%	56	177964	44.4	**57**	180331	74.4	56	176624	54.1
	random/highgirth	82	26.1%	6	4597721	**1098.6**	6	4639127	1113.9	6	4605430	1104.2
	random/max2sat	100	8.9%	100	38841	**79.3**	100	45887	93.1	100	52671	105.9
	random/max3sat	100	6.7%	100	426143	**297.3**	100	438460	306	100	437275	309.2
	random/min2sat	96	15.2%	96	979	**2.3**	96	1119	2.7	96	1168	2.8
weighted	crafted/frb	34	2.4%	14	210245	**37.5**	14	212541	**37.5**	14	222651	39.4
	crafted/ramsey	15	54.3%	4	157478	**56.4**	4	158424	**56.4**	4	157291	71.5
	crafted/wmaxcut	67	43%	**62**	20317	**31.1**	61	22539	35.7	61	19956	32.4
	random/wmax2sat	120	58.2%	120	3898	**48.9**	120	4165	53.2	120	4603	60.5
	random/wmax3sat	40	49.9%	40	44804	**122.3**	40	45959	127.5	40	45903	129.5
	Total	821	24.1%	698	135879	**100.3**	698	139803	108.2	697	140183	110.5

Table 2: Comparison of MSSOLVER^H to the two best performing BnB solvers of the Max-SAT Competition 2013. The two first columns give the instances classes and the number of instances per class. For each tested solver, the columns S, D and T give respectively the number of solved instances, the average number of decisions and the average solving time.

	instances classes	#	WMAXSATZ2009			WMAXSATZ2013			MSSOLVER^H		
			S	D	T	S	D	T	S	D	T
unweighted	crafted/bipartite	100	99	527295	268.7	99	796983	282.3	**100**	34909	**95.8**
	crafted/maxcut	67	55	850803	97.4	55	755340	54.7	**56**	177964	**44.4**
	random/highgirth	82	0	-	-	0	-	-	**6**	4597721	1098.6
	random/max2sat	100	96	666713	288.1	100	523266	169.8	100	38841	**79.3**
	random/max3sat	100	97	2211487	381.7	100	1476192	**242.9**	100	426143	297.3
	random/min2sat	96	77	648900	185.5	96	22402	9.4	96	979	**2.3**
weighted	crafted/frb	34	9	1379041	12.2	14	1537566	62.8	14	210245	**37.5**
	crafted/ramsey	15	4	876667	93.4	4	549137	**52.6**	4	157478	56.4
	crafted/wmaxcut	67	61	75186	80.8	**63**	126254	73.5	62	19993	31.1
	random/wmax2sat	120	119	82064	288.9	120	81440	134.2	120	3898	**48.9**
	random/wmax3sat	40	40	328504	177.1	40	257175	130.7	40	44804	**122.3**
	Total	821	657	716729	240.2	691	541647	145	**698**	135686	**100.3**

Conclusion

We have presented in the first part of this paper a new propagation scheme, which takes into consideration all the predecessors of the variables. With this scheme, BnB Max-SAT solvers can apply unit propagation in a non-chronological way and thus make less useless propagation steps. We have shown experimentally that this scheme reduces significantly the number of propagation steps performed by our solver MSSOLVER.

In the second part of this paper, we have presented how the information available with the MPS scheme can be used to influence the characteristics of the inconsistent subsets produced by BnB Max-SAT solvers. The experimental results obtained show that it can be efficiently used to reduce the size of the IS and improve the LB quality.

In our opinion, the interest of the multiple predecessors scheme is not limited to BnB Max-SAT solvers. In the future, we will look how the information at disposal in MPS can be used to improve both Max-SAT and SAT complete solvers. We will also try to make our heuristic more robust or even develop new and more efficient heuristics. We will study the impact of the transformed IS characteristics on the behavior of BnB solvers, and especially on the IS detection capability.

References

Audemard, G.; Bordeaux, L.; Hamadi, Y.; Jabbour, S.; and Sais, L. 2008. A generalized framework for conflict analysis. In Kleine Bning, H., and Zhao, X., eds., *Theory and Applications of Satisfiability Testing SAT 2008*, volume 4996 of *Lecture Notes in Computer Science*. Springer Berlin Heidelberg. 21–27.

Bonet, M. L.; Levy, J.; and Manyà, F. 2007. Resolution for max-sat. *Artificial Intelligence* 171(8-9):606–618.

Heras, F., and Larrosa, J. 2006. New inference rules for efficient max-sat solving. In *Proceedings of the 21st national conference on Artificial intelligence - AAAI'06*, volume 1, 68–73. AAAI Press.

Larrosa, J., and Heras, F. 2005. Resolution in max-sat and its relation to local consistency in weighted csps. In Kaelbling, L. P., and Saffiotti, A., eds., *Proceedings of the Nineteenth International Joint Conference on Artificial Intelligence - IJCAI'05*, 193–198. Morgan Kaufmann Publishers Inc.

Li, C. M.; Manyà, F.; Mohamedou, N.; and Planes, J. 2009. Exploiting cycle structures in max-sat. In Kullmann, O., ed., *Theory and Applications of Satisfiability Testing - SAT 2009*, volume 5584 of *LNCS*. Springer Berlin / Heidelberg. 467–480.

Li, C. M.; Manyà, F.; and Planes, J. 2005. Exploiting unit propagation to compute lower bounds in branch and bound max-sat solvers. In van Beek, P., ed., *Principles and Practice of Constraint Programming - CP 2005*, volume 3709 of *LNCS*. Springer Berlin / Heidelberg. 403–414.

Li, C. M.; Manyà, F.; and Planes, J. 2006. Detecting disjoint inconsistent subformulas for computing lower bounds for max-sat. In *Proceedings of the 21st National Conference on Artificial Intelligence - AAAI 2006*, 86–91. AAAI Press.

Li, C. M.; Manyà, F.; and Planes, J. 2007. New inference rules for max-sat. *Journal of Artificial Intelligence Research* 30:321–359.

Marques-Silva, J. P., and Sakallah, K. A. 1999. Grasp: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers* 48(5):506–521.

Papadimitriou, C. H. 1994. *Computational complexity*. Addison-Wesley.

Strickland, D. M.; Barnes, E.; and Sokol, J. S. 2005. Optimal protein structure alignment using maximum cliques. *Operations Research* 53(3):389–402.

Van Gelder, A. 2011. Generalized conflict-clause strengthening for satisfiability solvers. In Sakallah, K., and Simon, L., eds., *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*. Springer Berlin / Heidelberg. 329–342.

Xu, H.; Rutenbar, R. A.; and Sakallah, K. 2002. sub-sat: a formulation for relaxed boolean satisfiability with applications in routing. In *Proceedings of the 2002 International Symposium on Physical Design - ISPD '02*, 182–187. ACM.