Introducing Intel® SAT Solver

Alexander Nadel
Intel Corporation, Haifa, Israel

Abstract

We introduce Intel® SAT Solver (IntelSAT) – a new open-source CDCL SAT solver, written from scratch. IntelSAT is optimized for applications which generate many mostly satisfiable incremental SAT queries. We apply the following Incremental Lazy Backtracking (ILB) principle: in-between incremental queries, backtrack only when necessary and to the highest possible decision level. ILB is enabled by a novel reimplication procedure, which can reimplies an assigned literal at a lower level without backtracking. Reimplication also helped us to restore the following two properties, lost in the modern solvers with the introduction of chronological backtracking: no assigned literal can be implied at a lower level, conflict analysis always starts with a clause falsified at the lowest possible level. In addition, we apply some new heuristics. Integrating IntelSAT into the MaxSAT solver TT-Open-WBO-Inc resulted in a significant performance boost on incomplete unweighted MaxSAT Evaluation benchmarks and improved the state-of-the-art in anytime unweighted MaxSAT solving.

2012 ACM Subject Classification Mathematics of computing -> Solvers

Keywords and phrases SAT, CDCL, MaxSAT

Digital Object Identifier 10.4230/LIPIcs.SAT.2022.8


1 Introduction

The results of recent SAT competitions demonstrate a substantial progress in the performance of Conflict-Driven-Clause-Learning (CDCL) SAT solvers [15] on non-incremental benchmarks. However, some of the most widely-used SAT-based algorithms, such as state-of-the-art MaxSAT [8] and model checking [21,25] algorithms, need solving a series of related SAT instances, which require the underlying SAT solver to be incremental [22,49]. Surprisingly, it was recently shown in [32] that a state-of-the-art SAT solver RLNT exhibits no significant performance gains over Glucose 3.0 [4], released in 2013, on three prominent incremental SAT applications (with mostly satisfiable, mostly unsatisfiable and mixed SAT queries), despite RLNT being substantially more efficient on SAT competition benchmarks.

We introduce Intel® SAT Solver (IntelSAT) – a new open-source CDCL SAT solver, written from scratch in C++. Unlike other modern solvers, IntelSAT is optimized for incremental SAT applications which generate many mostly satisfiable SAT queries. A prominent example of such an application is anytime unweighted MaxSAT (included, under the name Satisfiability-based MaxSAT, in the three applications, analyzed in [32]).

MaxSAT [8] is a well-studied and widely-used optimization problem. Given a Boolean formula $F$ and a linear Pseudo-Boolean function $\Psi$, a MaxSAT solver returns a model (solution) to $F$ which minimizes $\Psi$. In unweighted MaxSAT, $\Psi$ has degree 1. In anytime MaxSAT, the solver is required to generate a series of solutions improving w.r.t. $\Psi$. Anytime-ness can be crucial, especially in industrial usage, where an approximate solution can often do, while reaching a timeout without any solution is not an option [1,30,34,41-43].

The baseline algorithm for anytime MaxSAT is Linear Search SAT-UNSAT (LSU) [10]. For unweighted MaxSAT, the leading solvers SATLike-c [17,33] and TT-Open-WBO-Inc [44-46] combine LSU with Mrs. Beaver [40] and Polosat [43] algorithms. The resulting flow tends to generate a lot of mostly satisfiable incremental SAT queries sharing many of the assumptions with few clauses added in-between.
In addition to [32], there is also the following evidence for the lack of performance progress in SAT solving for anytime unweighted MaxSAT. The anytime MaxSAT solver SATLike-c [33] was submitted to the latest MaxSAT Evaluation 2021 (MSE’21) in two versions, the difference being the SAT solver used for the initial SAT query – the SAT Competition 2020 winner Kissat [12] or the older Glucose 4.1. The Glucose-based version performed better and won both relevant (60- and 300-second unweighted incomplete) categories.

We introduce and apply in IntelSAT the following Incremental Lazy Backtracking (ILB) principle: in-between incremental queries, backtrack only when necessary and to the highest possible decision level. In contrast, a standard incremental solver processes new input clauses and solving requests only at decision level 0. ILB is intended to save unnecessary recreation of the same or similar trail [37,50,57] under the similar incremental queries.

Consider a situation, when the user provides a new input clause $C$ at level $\geq 0$, where $C$ contains one satisfied literal $l$ while the rest are falsified, and $l$ is assigned at a decision level higher than the other literals in $C$. We call such a clause a missed lower implication. Intuitively, $l$ should have been implied in $C$. Handling missed lower implications turned out to be the main challenge in supporting ILB. We solved it with our novel reimplication procedure, which fixes missed lower implications iteratively without backtracking.

Independently of ILB, with reimplication, we were able to design Boolean Constraint Propagation (BCP) so as to ensure the following two properties:
(a) lowest implication: no assigned literal can be implied at a lower decision level (that is, no missed lower implications exist after BCP), and
(b) lowest conflict: in case of a conflict, a clause falsified at the lowest possible decision level is returned.

These properties trivially hold without Chronological Backtracking (CB) [50], but were lost in modern CB-enabled solvers, since those propagate simultaneously at several decision levels. To support reimplication and our BCP procedure, we modified core BCP invariants and implemented the trail as a doubly linked list (rather than the standard stack).

Furthermore, we apply new heuristics in different solver components, including: query-driven tuning (tune various heuristics, based on the SAT query type), subsumption-based flipped clause filtering (conflict analysis) and incremental score reboot (decision heuristic).

Integrating IntelSAT into the anytime MaxSAT solver TT-Open-WBO-Inc resulted in a significant performance boost on MaxSAT Evaluation 2020 (MSE’20) and MSE’21 benchmarks and improved the state-of-the-art in anytime unweighted MaxSAT solving.

We would like to emphasize that IntelSAT’s algorithms are applicable not only in the context of anytime unweighted MaxSAT. Specifically, both BCP with reimplication and subsumption-based flipped clause filtering are expected to help generic SAT solving, while query-driven tuning and incremental score reboot are relevant for generic incremental SAT solving. We leave integrating our algorithms into other solvers and testing their impact on additional applications to future work.

The rest of this paper is organized as follows. Sect. 2 presents preliminaries. Sect. 3 introduces core IntelSAT algorithms, focusing on reimplication, BCP and their correctness. Sect. 4 highlights some of the other algorithms and heuristics. Sect. 5 is about experimental results. Sect. 6 concludes our work. Appendix A completes the correctness proofs.
of literals (of pairwise different variables). We denote clauses by capital letters and their literals by the corresponding small letters. For a clause $C$, we denote $C$’s sub-sequence $[c_i, c_{i+1}, \ldots, c_{|C|}]$ by $C_{[i..]}$ (where $i \geq 1$ and $C_{[i..]}$ is empty if $i > |C|$). When the literal order is irrelevant, we use curly brackets in the clause definition: $C = \{c_1, c_2, \ldots, c_{|C|}\}$. A clause $C$ subsumes the clause $D$, if $\forall c_i \in C : c_i \in D$, in which case $D$ is implied by $C$.

The solver maintains the current assignment $\sigma$, which, for every variable $v$, holds its current value $\sigma(v) \in \{U, \top, \bot\}$. A literal/variable $l$ is assigned if $\sigma(var(l)) \neq U$, otherwise it is unassigned. Given an assigned variable $v$, its assigned literal $lit(v)$ is $v/\neg v$, if $\sigma(v) = \top/\bot$, respectively. A variable $v$ is satisfied/falsified if $\sigma(v) = \top$, respectively. A negative literal $\neg v$ is satisfied/falsified iff $v$ is falsified/satisfied, respectively. Assigning a negative literal $\neg v$ the value $\top/\bot$ amounts to assigning $\bot/\top$ to $v$, respectively. Flipping a literal $l$, assigned $\top/\bot$, means unassigning it and then assigning it $\bot/\top$, respectively. A literal $l$ is non-falsified, if $\sigma(var(l)) \neq \bot$. Given a clause $C$, $\#\top(C)/\#\bot(C)/\#U(C)/\#NF(C)$ stand for the number of $C$’s literals, which are satisfied/falsified/unassigned/non-falsified, respectively.

**Definition 1 (Unit, Unsat, Falsified).** A clause $C$ is unit iff $\#U(C) = 1$ and $\#\bot(C) = |C| - 1$. $C$ is unsat iff $\#\top(C) = 1$ and $\#\bot(C) = |C| - 1$. $C$ is falsified iff $\#\bot(C) = |C|$.

### 2.1 Incremental CDCL SAT Solving Review

Since Minisat [22], the two basic API functions of an incremental CDCL SAT solver are AddClause(Clause $C$) and Solve(Literals $A$).

The solver maintains the current decision level (current level) $d$, initialized to 0. Whenever a variable $v$ is assigned, it is associated with its decision level (level) $dl(v) \leq d$ (where $dl(\neg v) = dl(v)$). We assume an order relation between assigned literals and variables, induced by their levels (e.g., literal $l$ is higher than literal $q$ iff $dl(l) > dl(q)$).

If $|C| > 1$, AddClause($C$) adds the given clause $C$ to the set of clauses $F$ and other data structures. If $|C| = 1$, the literal $c_1$ is assigned $\top$ at level 0.

Solve($A$) returns the satisfiability status (SAT or UNSAT) of the formula $F \land A$, where $A$ is a set (conjunction) of the so-called assumptions, which hold only for the current Solve invocation. Solve carries out backtrack search. Any newly assigned literal $l$ is pushed to:

(a) the trail, which contains all the assigned literals, and

(b) the propagation stack $\Pi$, which contains literals to be propagated by Boolean Constraint Propagation (BCP).

#### 2.1.1 Boolean Constraint Propagation (BCP)

For every literal $l \in \Pi$, BCP propagates $l$’s value as follows.

BCP visits any clause which might become unit as the result of $l$’s assignment. Assume that a unit clause $C$ is identified; assume WLOG that $c_1$ is the only unassigned literal in $C$. Then, BCP assigns $c_1 := \top$ (also pushing $c_1$ to $\Pi$), in which case we say that $c_1$ is implied in the parent clause $C$. The algorithm sets $dl(c_1)$ to $\maxl(C_{[2..]})$, where $\maxl(D)$ is the maximal order amongst $D$’s assigned literals or 0 if $D$ is empty. Then, BCP continues propagating $l$. BCP might also encounter a falsified clause $C$, in which event we say that a conflict at conflict level $\maxl(C)$ occurs, and BCP returns $C$.

Since Chaff [38], to identify unit and falsified clauses efficiently, the algorithm watches two literals in every clause $C$, where the watched literals (watches) are the first two literals in the clause: $c_1$ and $c_2$. For every literal $l$, the solver maintains its Watch List ($WL$): $WL(l)$. Since [19], every $WL(l)$ element is a pair $\langle h \neq l \in C, C \rangle$ containing a clause $C$ where $l$ is watched and $l$’s cached literal $h \in C : h \neq l$, denoted by $h(l, C)$. To propagate $\neg l$, BCP goes
over WL(l). Assume some \langle h, C \rangle \in WL(l) is visited. If h is satisfied, BCP skips the satisfied clause, thus saving potential cache misses. Otherwise, C is visited and tested for being unit or falsified. BCP ensures that, by its end, the following WL invariants hold for every C and i ∈ \{1, 2\} (if a conflict interrupts BCP, the invariants might be broken for unvisited clauses, but backtracking following conflict analysis, reviewed in Sect. 2.1.2, restores them):
(a) \( c_i \) is non-falsified or \( h(c_i, C) \) is satisfied, or
(b) \( c_i \) is falsified and \( dl(c_i) \geq maxl(C_{[2..]}) \) and \( \forall j > 2 : c_j \) is falsified.

### 3.1 Solve

**SOLVE(A)** starts off by running BCP. In case of a conflict, there is a global contradiction, hence the solver will return UNSAT from that moment on. Otherwise, **SOLVE** embarks on the backtrack search. At each new level \( d \), **SOLVE** heuristically chooses an unassigned decision variable \( v \) and assigns it either \( \top \) or \( \bot \), where \( \textit{lit}(v) \) is called the decision literal at level \( d \). If unassigned assumptions exist, an unassigned assumption is always chosen as the next decision literal. This simple strategy ensures that **SOLVE** satisfies all the assumptions, whenever possible. If a falsified assumption is discovered, the solver returns UNSAT.

BCP is invoked after every decision. If there is no conflict, the solver moves on to a new level \( d + 1 \), otherwise it enters conflict analysis. An iteration of the conflict analysis loop derives the so-called asserting learnt clause \( D \), where it holds that: \( D \) is implied by \( F \), \( D \) is falsified and \( dl(d_1) = d > dl(d_2) \geq maxl(D_{[2..]}) \). The algorithm adds \( D \) to \( F \) (unless \( |D| = 1 \) and backtracks to level \( b \), where \( b \in \{dl(d_2), dl(d_2) + 1, \ldots, d - 1\} \) (if \( |D| = 1 \), assume \( dl(d_2) = 0 \)). If CB is not applied, \( b \) always equals \( dl(d_2) \).

Backtracking to level \( e \) unassigns all the literals assigned at levels \( > e \) and removes them from the trail. It also updates \( d \) to \( e \). In addition, if CB is applied, backtracking reassigns any out-of-order literals (that is, literals whose level \( < e \)), which are then repropagated by the next BCP. The WL invariants are maintained under backtracking with no action required.

After backtracking, \( D \) becomes unit and \( d_1 \) is assigned \( \top \), followed by BCP. Normally, conflict analysis loop goes on until BCP does not identify a conflict anymore, in which case the solver increments \( d \) and continues to a new decision. Otherwise, the conflict analysis loop might derive a global contradiction or conclude that an assumption is flipped, in which cases the solver returns UNSAT.

After **SOLVE** is completed, the solver backtracks to the global decision level 0 and waits for new clauses and incremental invocations.

### 3 Core CDCL Algorithms in Intel® SAT Solver

This section introduces our implementation of the core CDCL algorithms. Specifically, Sects. 3.1 and 3.2 are about **SOLVE** and ** AddClause**, respectively. Sect 3.3 makes the case for reimplication. Sect 3.4 presents our formal framework. Sects. 3.5 and 3.6 introduce our reimplication and BCP algorithms, respectively. We provide arguments for the correctness of our algorithms while presenting them, yet complete the proofs in Appendix A.

#### 3.1 Solve

Our implementation of **SOLVE** is mostly standard. The differences stem from applying ILB. First, consider the end of a **SOLVE** invocation. If the result is SAT, we do not backtrack. Assume now that a falsified assumption \( l \) is discovered by BCP. We let BCP complete the propagation. If no conflict follows, we return UNSAT without backtracking. Otherwise, if a falsified clause \( C \) is discovered, we backtrack to level \( maxl(C) - 1 \) to make sure the trail is consistent with \( F \) and return UNSAT.
Now consider the beginning of a non-initial \texttt{Solve} \((A) \) invocation. Observe that if one or more of the first decision literals appear in \( A \), backtracking can be saved. Let \( k \) be the lowest level, whose decision literal does not appear in \( A \). We backtrack to level \( k - 1 \) instead of \( 0 \). For an example, consider Fig. 1a (ignore Fig. 1’s caption for now, except for the first sentence after the title). It shows a trace of a satisfiable SAT solver invocation without conflicts over the formula \( C \equiv \{\neg l_1, l_2, l_3\} \land D = \{\neg l_3, l_4\} \). Assume there are no new input clauses until the next invocation \texttt{Solve}\((\{l_3, l_1\})\). We backtrack to level \( 1 \) (rather than \( 0 \)), since the assumption \( l_1 \) already serves as the first decision literal. The impact of backtracking to level \( k - 1 \) instead of \( 0 \) is similar to that of trail savings [28], except that our scheme works independently of whether any skipped literals (\( l_1 \) in our case) were previously assumptions or just happened to be chosen as first decision literals by the variable decision heuristic.

### 3.2 \texttt{AddClause}

ILB requires enabling \texttt{AddClause}(\( C \)) at an arbitrary level. We still ensure that the WL invariants hold for \( C \) by the end of function as follows.

If \( \#NF(\ C) > 1 \), we safely watch any two non-falsified literals. If \( C \) is unit, we watch the unassigned literal \( l \) and a falsified \( c_i : dl(c_i) \geq \maxl(C) \), and then assign \( l \) at \( \maxl(C) \). If \( C \) is falsified, we backtrack to \( \maxl(C) - 1 \) to unassign one or more of \( C \)’s literals and proceed as above. Let now \( C \) be unsat (recall Def. 1), where the satisfied literal is \( l \). We watch \( l \). If there exists \( c_i \neq l : dl(c_i) \geq \maxl(C) \), \( c_i \) is also watched. The only remaining non-trivial case led us to introducing reimplication, which we apply to complete \texttt{AddClause}.

### 3.3 The Case for Reimplication

Def. 2 formalizes the notion of a missed lower implication. Intuitively, if a missed lower implication clause \( C \) exists, its satisfied literal \( l \in C \) could be implied in \( C \) at a level lower than its current level \( dl(l) \).

\begin{definition}[Missed Lower Implication] A clause \( C \) is a Missed Lower Implication (Lower Implication) iff \( C \) is unsat and, assuming \( c_i \) is the satisfied literal in \( C \), it holds that: \( \forall j \neq i : dl(c_i) > dl(c_j) \), where \( 1 \leq i, j \leq |C| \).
\end{definition}

### 3.3.1 \texttt{AddClause}

Assume \texttt{AddClause} is provided with a missed lower implication \( C \), where, WLOG, the satisfied literal is \( c_1 \). To eliminate the lower implication, an ILB-driven solution could backtrack to \( dl(c_1) - 1 \) to make \( C \) unit, assign \( c_1 \) at \( \maxl(C) \) and propagate by rerunning BCP. However, that would unassign intermediate decision levels. Our key observation is that one can abstain from backtracking and propagating altogether. Instead, we fix \( C \) by reassigning \( c_1 \) in \( C \) at \( \maxl(C[2..]) \). Fixing might generate more missed lower implications and break the WL invariants for some clauses. Our reimplication procedure, introduced in Sect. 3.5, iteratively fixes the new lower implications and restores the WL invariants.

To handle reimplication (and CB) efficiently, \texttt{IntelSAT} implements the trail as a doubly linked list (rather than the standard stack) with pointers to the end of each decision level and to the list’s beginning. This allows us to easily remove literals from anywhere in the trail and append literals to the end of any level.

Fig. 1 presents an example of handling a lower implication in \texttt{AddClause}.
3.3.2 BCP

Our BCP procedure uses reimplication to ensure lowest implication and lowest conflict (recall Sect 1). Fig. 2 shows how lower implications are handled in a standard solver vs. IntelSAT.

Fig. 2 illustrates that CB might cause the solver learn a weaker clause, subsumed by one that would be learnt without CB for the same conflict. This is because, with CB, the solver does not meet the lowest implication property: unlike IntelSAT, it does not decrease the levels of the assigned literals by reimplying them in lower implications whenever required, while there are no missed lower implications if CB is not used. For example, in Fig. 2, the analysis of the second conflict by the CB-enabled solver involves two decision levels instead of one (since $l_2$ is not reimplied in the lower implication $H$), which results in learning the clause $L = \{l_4, \neg l_3\}$ instead of $M = \{l_4\}$, where $M$ would be learnt by both a standard solver without CB and IntelSAT with CB (since IntelSAT applies reimplication).

Fig. 2 also illustrates that BCP may discover more than one conflict. Pre-CB solvers stopped at the first conflict due to lack of a better heuristic [24]. With CB, BCP can identify several conflicts at different levels, but the current solvers still stop at the first conflict, even if it is not a lowest one. Note that reassignment and repropagation of out-of-order literals after backtracking guarantees that lower conflicts are discovered by the conflict analysis loop before taking any further decisions (and that implications are not missed). However, while working correctly, current CB-enabled solvers learn unnecessary clauses and lose time due to their failure to always start conflict analysis with a lowest conflict. Reimplication enables BCP restoring the lowest conflict property and makes out-of-order literal management obsolete.

\[ l_1 \oplus 1 \]
\[ \neg l_2 \oplus 2 \]
\[ l_3 \oplus 2 : C \]
\[ l_4 \oplus 2 : D \]

\[ l_1 \oplus 1 \]
\[ \neg l_2 \oplus 1 : E \]
\[ l_3 \oplus 1 : C \]
\[ l_4 \oplus 1 : D \]

(a) $C = \{\neg l_1, l_2, l_3\} \land D = \{\neg l_3, l_4\}$.  
(b) $C \land D \land (E = \{\neg l_2, \neg l_1\})$.

**Figure 1** Reimplication in AddClause. Fig. 1a shows a solver invocation trace (without any conflicts), given the clauses $C = \{\neg l_1, l_2, l_3\} \land D = \{\neg l_3, l_4\}$, where $l$ is the decision literal at level $x$ and $\forall x : C$ means that $l$ was implied at level $x$ in the parent clause $C$. The solver returns SAT, but, in the ILB-driven IntelSAT, it does not backtrack. Assume it receives a new input clause $E = \{\neg l_2, \neg l_1\}$. $E$ is a lower implication, since it has one satisfied literal $\neg l_2$, while the rest ($\neg l_1$ only here) are falsified at a level lower than $dl(l_2)$. In our case, $\neg l_2$ is a decision literal, which should now be reimplied at level 1. This can be accomplished by either backtracking to level 1, implying $\neg l_2$ in $E$ and propagating or, instead, by applying our novel reimplication procedure. In our simple case, both alternatives result in the same trace, shown in Fig. 1b.

3.4 Formal Framework

The correctness of CDCL SAT solvers has been proven in several different frameworks [37, 39, 51] (e.g., [39] proves it with or without CB). However, all the proofs implicitly assume that BCP never misses falsified clauses (for correctness) and unit clauses (for performance), which, after the introduction of cached literals and CB, is no longer self-evident. To argue that, in IntelSAT, BCP terminates, does not miss falsified and unit clauses and guarantees lowest implication and lowest conflict, we need to introduce a formal framework.
A clause while propagating we request
Assume now that the following two clauses also belong to
At this point, H becomes a missed lower implication: \( I_3 \) could have been implied in H at level 1, instead of its current implication in C at level 2. However, CB continues using C as \( I_1 \)'s parent. Instead, IntelSAT places \( \neg I_4 \) at the end of level 1 (using our doubly linked list trail) and runs BCP, which identifies H as a lower implication and invokes reimplication to create the trace in Fig. 2c. Assume now that the following two clauses also belong to F: \( J = \{\neg I_3, I_2, I_1\} \) and \( K = \{\neg I_3, I_4, \neg I_5\} \). There will be a conflict immediately after propagating \( \neg I_4 \) for both CB and IntelSAT. CB will learn \( L = \{I_4, \neg I_5\} \), while IntelSAT will learn \( M = \{I_4\} \), which subsumes L. Assume now that, in addition to J and K, the clauses \( N = \{\neg I_2, I_4, \neg I_5\} \) and \( P = \{\neg I_2, I_4, \neg I_5\} \) also belonged to F. In this case, there would be two potential conflicts after propagating \( \neg I_4 \). For IntelSAT, the two potential learnt clauses would be \( M \) or \( Q = \{\neg I_2, I_4\} \). IntelSAT would choose M, since it occurs at a lower level. CB would stop at the first (arbitrary) conflict.

We disallow missed lower implications through strengthening the WL invariant for cached literals as follows: if a falsified watch \( I \) has a satisfied cached literal \( h \), then \( h \) can no longer be higher than \( I \). Def. 3 formally defines a \( c_i \)-stable/stable clause (where we make the following standard assumption, satisfied by construction: \( \forall C \in F : C \) appears once and only once in \( WL(c_1) \) and \( WL(c_2) \); see Appendix A for further details).

\begin{definition} [\( c_i \)-Stable Clause, Stable Clause] \end{definition}
For \( i \in \{1, 2\} \), a clause \( C \) is \( c_i \)-Stable if:
\begin{enumerate}[(a)]
\item \( \sigma(c_i) \neq \bot \), or
\item \( \exists \langle h, C \rangle \in WL(c_i) : \sigma(h) = \top \) and \( dl(h) \leq dl(c_i) \).
\end{enumerate}
A clause \( C \) is Stable iff \( \forall i \in \{1, 2\} : C \) is \( c_i \)-Stable.

Our algorithms ensure that, by the end of BCP without a conflict, all the clauses are stable. However, we need to allow some intermediate exceptions.

A clause ceases to be \( c_i \)-stable when an unassigned watch \( c_i \) is assigned \( \bot \). In this case, we request \( c_i \) to register for BCP, that is, \( \neg c_i \) is to be added to the propagation stack \( \Pi \).

\begin{definition} [\( c_i \)-BCP-Registered] \end{definition}
For \( i \in \{1, 2\} \), a clause \( C \) is \( c_i \)-BCP-Registered if \( \neg c_i \in \Pi \).

Intuitively, it is safe for a clause \( C \) to become \( c_i \)-BCP-Registered, since BCP will visit \( C \) while propagating \( \neg c_i \) and make \( C \) \( c_i \)-stable, if possible, or, otherwise, discover that \( C \) is falsified. Hence, if a clause is either \( c_i \)-stable or \( c_i \)-BCP-Registered, we call it \( c_i \)-BCP-Safe.

\begin{definition} [\( c_i \)-BCP-Safe, BCP-Safe] \end{definition}
For \( i \in \{1, 2\} \), a clause \( C \) is \( c_i \)-BCP-Safe if \( C \) is either \( c_i \)-stable or \( c_i \)-BCP-Registered. A clause \( C \) is BCP-Safe if \( \forall i \in \{1, 2\} : C \) is \( c_i \)-BCP-Safe.
A lower implication $C$ is not $c_i$-stable for the falsified watch $c_i$. We store lower implications in a dedicated stack $\Lambda^R$, so that reimplication could fix them one by one (where some clauses in $\Lambda^R$ could already be inadvertently fixed). Def. 6 lists the five conditions under which a clause is considered to be registered for reimplication.

**Definition 6** (Reimplication-Registered). A clause $C$ is Reimplication-Registered if:

(a) $C \in \Lambda^R$, and
(b) $C$ is unisat, and
(c) $\sigma(c_1) = T$, and
(d) $h(c_2, C) = c_1$, and
(e) $dl(c_2) \geq maxi(C_{3\ldots})$.

The conditions in Def. 6 are intended to ensure that a reimplication-registered clause is either already BCP-Safe or is a fixable lower implication; see Cor. 7.

**Corollary 7** (Fixing). A reimplication-registered clause $C$ is either:

(a) BCP-Safe, or
(b) a lower implication, in which case $C$ can be fixed to become BCP-Safe by applying REASSIGN($c_1, C, dl(c_2)$); see Alg. 1, line 4, where fixing lowers (the level of) $c_1$.

Finally, we store any falsified clauses in a stack $\Lambda^F$ to ensure they are not missed by BCP. Def. 8 defines clauses registered for falsification.

**Definition 8** (Falsification-Registered). A clause $C$ is Falsification-Registered if $C \in \Lambda^F$ and $C$ is falsified.

### 3.5 Reimplication

REIMPLY, shown in Alg. 1, line 14, eliminates any lower implications and guarantees that all the non-falsification-registered clauses are BCP-Safe. It removes one clause $R \in \Lambda^R$ at a time in a while-loop until $\Lambda^R$ becomes empty. Each while-loop iteration fixes $R$, if required, which makes $R$ BCP-Safe, but might render other clauses in $WL(\neg r_1)$ BCP-Unsafe. Hence, the algorithm goes over $WL(\neg r_1)$ and ensures every clause is either reimplication-registered (to be handled later by our loop), falsification-registered or BCP-Safe.

Notably, REIMPLY never modifies the assignment $\sigma$, but only the levels and the parents of assigned literals. In addition, REIMPLY might decrease the levels of assigned literals, but it never increases them. Finally, during REIMPLY, some levels might "collapse" and disappear (if their decision literal is reimplied at a lower level).

Lemma 9 is pivotal for arguing about REIMPLY’s correctness; see Appendix A.1 for further details. We show that Lemma 9 holds while presenting REIMPLY below.

**Lemma 9** (REIMPLY Loop Invariants). The following are two loop invariants for the while-loop in Alg. 1, line 15:

1. Every clause $C \in F \setminus \Lambda^R$ is either BCP-Safe or falsification-registered.
2. Every clause $C \in \Lambda^R$ is reimplication-registered.

We assume the invariants hold at the beginning of a while-loop iteration and show that they also hold at its end. Each iteration removes one clause $R$ from $\Lambda^R$. By Cor. 7 and our loop invariant, $R$ is either BCP-Safe or a fixable lower implication. Line 17 skips $R$, if it is BCP-Safe (optionally, one could also skip $R$, if $r_1 = q_1$ for $Q \in \Lambda^R: maxi(Q_{[2\ldots]}) \leq maxi(R_{[2\ldots]})$). Otherwise, line 18 fixes $R$ (recall Cor. 7).
Reassigning \( r_1 \) to fix \( R \) might render a BCP-Safe clause \( C \) BCP-unsafe, but only if the falsified literal \( \neg r_1 \) is watched by \( C \) (it can be easily verified that lowering an unwatched or satisfied literal has no such impact). Hence, we visit every pair \( \langle h, C \rangle \in WL(\neg r_1) \) and make sure \( C \) is BCP-Safe, reimplication-registered or falsification-registered.

Line 19 skips visiting \( \neg r_1 \)'s WL if \( \neg r_1 \in \Pi \), since, by Def. 4, all the clauses in \( WL(\neg r_1) \) are \( \neg r_1 \)-BCP-Registered, hence they are still BCP-Safe after the reassignment. Intuitively, they can be skipped, since BCP will visit them anyway.

Otherwise, we go over every pair \( \langle h, C \rangle \in WL(\neg r_1) \). If \( C \) is already in \( \Lambda^R \), it is skipped at line 21, since it will be visited later by our loop. Line 21 also skips any falsification-registered clauses (those will be handled by BCP). Otherwise, if the cached literal \( h \) is already satisfied not higher than \( r_1 \), \( C \) can be skipped, since it remains BCP-Safe (line 22). Note that clause visit is not required in these cases.

In the remaining case, we visit \( C \). Lines 23–25 fix \( C \)'s watches to enable rendering \( C \) either BCP-Safe or reimplication-registered. Specifically, line 23 ensures the first watch \( c_1 \) has the satisfied literal \( h \). Line 24 sets \( x \) to the current index of the second watch-to-be: a non-falsified literal, if one exists, or, otherwise, a highest falsified literal. Line 25 updates the second watch.

Finally, line 26 pushes \( C \) to \( \Lambda^R \), if \( C \) is a lower implication. By construction, \( C \) becomes either BCP-Safe or reimplication-registered.
3.6 Boolean Constraint Propagation (BCP)

We introduce our BCP algorithm starting with falsified clause processing. Consider Def.10, which categorizes falsification-registered clauses. Note that fake clauses have only one highest literal \( l \). When such clauses are encountered by a standard CB-enabled solver, it backtracks, flips \( l \) and reruns BCP, as elaborated in [37,50].

**Definition 10 (Contradicting, Backtrack-Contradicting, Fake).** A falsification-registered clause \( C \) is Backtrack-Contradicting if at least two literals in \( C \) are assigned at level \( \text{max}_l(C) \). A backtrack-contradicting clause \( C \) is Contradicting if \( \text{max}_l(C) = d \) (recall that \( d \) is the current decision level). A falsified clause \( C \) is Fake if it is not backtrack-contradicting.

Our BCP algorithm guarantees that all the falsified clauses (if any) are contradicting after BCP (thus, it does not miss falsified clauses and ensures lowest conflict). To that end, BCP registers all the falsified clauses for falsification by pushing them to \( \Lambda^F \), where, normally, all the falsification-registered clauses are contradicting. They might cease being contradicting on two occasions:

(a) if \( \Lambda^F \) is non-empty and \( \text{REIMPLY} \) is invoked, since \( \text{REIMPLY} \) might lower the literals in falsification-registered clauses, or

(b) after a new non-contradicting falsified clause is discovered and added to \( \Lambda^F \).

In both these cases, we invoke the function \( \text{F2C} \).

3.6.1 \( \text{F2C} \)

\( \text{F2C} \), shown in Alg.2, line 1 makes every falsification-registered clause either BCP-Safe or contradicting.

Line 2 returns, if all the falsification-registered clauses are already contradicting or no falsification-registered clauses exist. Lines 3–5 watch two highest literals in every falsification-registered clause to facilitate rendering them BCP-Safe following backtracking.

Assume there are no fake clauses. Line 7 backtracks to the lowest possible level, such that at least one backtrack-contradicting clause still exists. Observe that this operation renders every backtrack-contradicting clause either BCP-Safe (by unassigning both watches) or contradicting. Hence, we can now remove any BCP-Safe clauses from \( \Lambda^F \) and return (line 9).

From now on, assume there is at least one fake clause. If any backtrack-contradicting clauses exist, line 7 still backtracks rendering every backtrack-contradicting clause either BCP-Safe or contradicting. After the backtracking, a fake clause may still be falsified; otherwise, it may have become unit or BCP-Safe.

Assume any falsified fake clauses remain. Line 11 turns at least one of them unit and the rest either unit or BCP-Safe by backtracking to the highest possible level ensuring no falsified clauses in \( \Lambda^F \). Backtracking also renders any contradicting clauses BCP-Safe. Line 12 flips the now unassigned literal in every unit clause in \( \Lambda^F \). At this point, all the clauses in \( F \), including those in \( \Lambda^F \), become BCP-Safe. Hence, line 13 empties \( \Lambda^F \), and \( \text{F2C} \) returns.

If no falsified fake clauses remain after backtracking at line 7, backtracking at line 11 is skipped. Hence, at least one contradicting clause remains. We still need to flip the now unassigned literal in any fake-turned-unit clauses in \( \Lambda^F \) at line 12, which also renders these clauses BCP-Safe. Therefore, \( \text{F2C} \) returns with only contradicting clauses in \( \Lambda^F \), the rest having been turned BCP-Safe.
Algorithm 2 Boolean Constraint Propagation (BCP).

```python
1: function F2C
2:     if all the falsification-registered clauses are contradicting or \( \Lambda^F \) is empty then return
3:     for \( C \in \Lambda^F \) do
4:         UpdateWatched(\( C, c_1, c_i \in C : dl(c_i) = \maxl(C) \))
5:         UpdateWatched(\( C, c_2, c_i \in C[2...] : dl(c_i) = \maxl(C[2...]) \))
6:     if \( \exists C \in \Lambda^F : C \) is backtrack-contradicting then
7:         BACKTRACK(lowest possible level, such that \( \exists C \in \Lambda^F : C \) is backtrack-contradicting)
8:     if there were no fake clauses in the beginning then
9:         Remove any BCP-Safe clauses from \( \Lambda^F \) and return
10:    if \( \exists C \in \Lambda^F : C \) is falsified then
11:        BACKTRACK(highest possible level, such that \( \exists C \in \Lambda^F : C \) is falsified)
12:     for \( C \in \Lambda^F : C \) is unit do Assign(\( c_1, C, dl(c_1) \))
13:     Remove any BCP-Safe clauses from \( \Lambda^F \)
14: function BCP
15:     Reimply()
16:     while \( \Pi \) is not empty do
17:         \( l := \Pi.back(); \Pi.pop() \)
18:     if \( \neg l \) is non-falsified then continue
19:     for \( h, C \in WL(\neg l) \) do
20:         if \( c_2 \neq \neg l \) then Swap(\( C, c_1, c_2 \))\footnote{Ensure \( c_2 = \neg l \) for simplicity}
21:         if \( h \) is satisfied at \( dl(h) \leq dl(l) \) then continue
22:     if \( \#NF(C[3...]) > 0 \) then \footnote{A non-falsified unwatched literal exists}
23:         Let \( c_i \in C[3...] \) be a non-falsified literal in \( C[3...] \)
24:         UpdateWatched(\( C, c_2, c_i \))
25:     else \footnote{Now all the literals in \( C[2...] \) are falsified}
26:         if \( c_1 \) is falsified then \footnote{\( C \) is falsified}
27:             \( \Lambda^F.push_back(C) \)
28:         if \( C \) is not contradicting then
29:             F2C()
30:             \( \Pi.push_back(l) \)
31:             break
32:     else \footnote{\( C \) is unit or unsat}
33:         UpdateWatched(\( C, c_2, c_i \in C[2...] : dl(c_i) = \maxl(C[2...]) \))
34:         if \( c_1 \) is unassigned then \footnote{\( C \) is unit}
35:             Assign(\( c_1, C, dl(c_2) \))\footnote{Implication}
36:         else if \( dl(c_1) > dl(c_2) \) then \footnote{Reimplication}
37:             \( \Lambda^R.push_back(C) \)
38:             \( \Pi.push_back(l) \)
39:             \( \text{Reimply()} \)
40:             F2C()
41:             break
42:     if \( \Lambda^F \) is empty return T else return any \( C \in \Lambda^F \)
```
3.6.2 BCP

BCP (Alg. 2, line 14) starts by invoking Reimply to fix any reimplication-registered clauses. The main while-loop (line 16) propagates literals in $\Pi$. An iteration starts with fetching the currently propagated literal $l$ and skipping it if $\neg l$ is non-falsified, followed by a for-loop (line 19), which goes over $WL(\neg l)$ to propagate $l$.

Assume WLOG $c_2 = \neg l$ (guaranteed by line 20). Every for-loop iteration ensures that, by its end, the currently visited clause $C$ will either be $c_2$-stable or contradicting.

If the cached literal is satisfied not higher than $l$, the clause is skipped being $c_2$-stable (line 21). Lines 22–24 handle the easy case, when there is a non-falsified unwatched literal. Otherwise, all the literals, but possibly $c_1$, are falsified.

If $c_1$ is falsified (line 26), $C$ is falsified. We register $C$ for falsification and, if $C$ is not already contradicting, invoke F2C to render $C$ BCP-Safe (if $C$ is fake) or contradicting (if $C$ is backtrack-contradicting). F2C might modify $WL(l)$, hence it is safer to repropagate $l$ later. Thus, we re-register $l$ for BCP and break the loop.

The only remaining case is when $C$ is either unit or unisat (line 32). Line 33 updates $c_2$ to the highest literal in $C_{c_1}[\ldots]$ and assigns $h(c_2, C) := c_1$. If $C$ is unit, we imply $c_1$ in $C$ rendering $C$ $c_2$-stable. Otherwise, $C$ is unisat. If $dl(c_1) \leq dl(c_2)$, $C$ is already $c_2$-stable, in which case we are done. Otherwise, $C$ is a lower implication. We register $C$ for reimplication (line 37) and invoke Reimply to fix it, but not before re-registering $l$ for BCP (line 38) to guarantee correctness, since, otherwise, the yet unvisited clauses in $WL(\neg l)$ would not be BCP-Safe. Finally, we invoke F2C and break the loop, since $l$ will be repropagated later.

BCP returns either $\top$ (no conflict) or an arbitrary contradicting clause. Appendix A.3 argues for the correctness of BCP.

4 IntelSAT’s Algorithms and Heuristics

First, we mention some of the algorithms we are not using: we do not differentiate between satisfiable and unsatisfiable stages [52], since we expect most of the queries to be satisfiable; inprocessing [16] and vivification [35] are too heavy; rephasing [13,18] would be overridden by anytime MaxSAT’s TORC polarity selection heuristic [41,44]; trail saving [29] and all-UIP recording [23] had been implemented, but did not improve the performance in our setting.

Next, we describe some of IntelSAT’s algorithms and heuristics, including the novel query-driven tuning, subsumption-based flipped clause filtering and incremental score reboot.

4.1 Query-driven Tuning

Modern SAT solvers often switch between different heuristics within the same solver component, where the criterion for switching is carefully chosen, based on empirical data. For example, [31] compares various criteria for switching between decision heuristics in a MapleCOMSPS [36]-grounded SAT solver, driven by run-time, conflict count and propagation count and converges to a deterministic conflict count-driven criterion. Another example is a restart count-driven criterion regulating the all-UIP conflict clause recording algorithm [23].

We found that tuning various heuristics per SAT query type (that is, Solve query type) is often beneficial in our setting. One can distinguish between three types of SAT queries in SAT-based anytime unweighted MaxSAT algorithms: initial, short-incremental – an incremental query with the conflict threshold of 1000 (used by Mrs. Beaver and Polosat), and normal-incremental (used by LSU). Below, we provide several examples of choosing the heuristic, based on the current SAT query type. Sect. 5 demonstrates the positive impact of query-driven tuning on the performance of IntelSAT within the backtracking heuristic.
4.2 Conflict Analysis

We always record the standard asserting 1UIP clause [38,55], enhanced by minimization [9,56] and binary resolution [2,4]. We also apply on-the-fly subsumption [26,27] during the 1UIP clause generation.

In addition, we apply an enhanced version of flipped recording [20,39,53]. Specifically, we sometimes record the Latest-Flipped-1UIP (LF1) clause—the 1UIP clause w.r.t a fake decision level, created by marking the latest literal flipped by conflict analysis as a decision. Our algorithm is outlined below.

If a flipped literal exists, we generate (but not yet record) an LF1 clause $L$. Then, we apply our subsumption-based flipped clause filtering—a seemingly minor but nevertheless a performance-crucial improvement: we abandon LF1 recording, if $L$ is subsumed by the standard 1UIP clause (note that the standard 1UIP variable might belong to $L$). Otherwise, $L$ is recorded after it is enhanced by minimization and binary resolution.

4.3 Decision Heuristic

We apply the Exponential Variable State Independent Decaying Sum (EVSIDS) variable decision heuristic [4,11,22] (see [14] for EVSIDS review). Recall that whenever a variable is visited during conflict analysis, EVSIDS adds the value $(1/f)^i$ to its score, where $f$ is the so-called variable activity decay factor and $i$ is the current conflict number. We increase $f$ by 0.01 every 5000th conflict starting at 0.95 until it reaches 0.99.

Furthermore, we apply the following new incremental score reboot heuristic: we re-initialize $f$ to 0.95 before every normal-incremental query (observe the query driven-tuning). Our heuristic causes the scores to increase by a larger factor at the beginning of such queries, which, intuitively, simulates a score reboot.

4.4 Backtracking and Restarting

Let score-based backtracking be the backtracking heuristic, which backtracks to the level containing the variable with the highest EVSIDS score [37]. We apply score-based backtracking, if the gap between the current level $d$ and the would-be Non-Chronological Backtracking (NCB) level $e$ is higher than $T$, where $T = 0$ for normal-incremental queries and $T = 100$ for the two other query types (observe the query driven-tuning again). Otherwise, we apply NCB. In addition, if the resulting backtrack level is lower than that of the highest assumption $l_a$, we always backtrack to $dl(l_a)$ instead.

We use local restarts [54] on top of a simple arithmetic restart strategy (with the conflict threshold of 1000).

4.5 Clause Deletion

Since COMiniSatPS [52], most solvers divide the learnt clauses into 3 tiers—Core, Tier2 and Local—based on their LBD score [3]. Clauses are considered for deletion only from Local, where the clauses are sorted by activity: the lower the clause is, the more likely it is to be deleted. Clauses are moved between the tiers, based on their activity; see [31] for details.

For simplicity, we maintain the learnt clauses in a single tier. We simulate a tiered strategy by changing the sorting criterion as follows.
Let \( l(C) \) be the LBD score of clause \( C \) and \( a(C) \) be \( C \)'s activity score. Then \( C > D \) if:
\[(a) \ l(C) < 16 \text{ and } l(D) \geq 16, \text{ or} \]
\[(b) \ l(C) \geq 16 \text{ and } l(D) \geq 16 \text{ and } a(C) \leq a(D), \text{ or} \]
\[(c) \ l(C) < 16 \text{ and } l(D) < 16 \text{ and } (l(C)/11 < l(D)/11 \text{ or } (l(C)/11 = l(D)/11 \text{ and } a(C) \geq a(D))), \] where division is truncated.

Additionally, clauses with the LBD score of 2 or lower are never deleted.

## 5 Experimental Results

IntelSAT is available on github under the open-source MIT license [48]. We have integrated IntelSAT into TT-Open-WBO-Inc-21 [46]—the runner-up in both the relevant (60- and 300-second unweighted incomplete) categories at MSE’21 [7] ( downloadable at [47]).

We used two benchmark sets from the incomplete unweighted categories of MSE’20 [6] and MSE’21, respectively. Similarly to MSE’21, we compared solvers by their average score, where, given the linear Pseudo-Boolean function to minimize \( \Psi \) and the model \( \mu \), the score per instance is: 
\[
(1 + \text{the best known result}) / (1 + \Psi(\mu)).
\]
Note that the higher the score is, the better. We took the best known results for the MSE’20 and MSE’21 sets from [5] and [7] (in the detailed per-benchmark results), respectively.

We ran the solvers for 300 seconds and measured the average score at the following intervals (to simulate different timeouts): 60, 120, 180, 240 and 300 seconds. For all the experiments we used machines with 32Gb of memory running Intel® Xeon® processors of 3Ghz CPU frequency.

The first goal of our experiments was to analyze the impact of IntelSAT on the performance of TT-Open-WBO-Inc-21 and compare the resulting MaxSAT solver to other leading unweighted anytime MaxSAT solvers. To that end, we launched the following solvers:
\[(a) \ \text{the default TT-Open-WBO-Inc-21 with Glucose 4.1 (TT21G)}, \]
\[(b) \ \text{TT-Open-WBO-Inc-21 with IntelSAT (TT21I)}, \]
\[(c) \ \text{SATLike-c (SL21): the winner of MSE’21}, \]
\[(d) \ \text{SATLike-c-20 (SL20): the winner of MSE’20}. \]

Our second goal was to study the impact of a number of heuristics on TT21I by disabling each and running the resulting TT21I version, including:
\[(a) \ \text{NoILB}: \text{disable ILB by backtracking to 0 before SOLVE and ADDCLAUSE}, \]
\[(b) \ \text{NoFLP}: \text{disable flipped recording}, \]
\[(c) \ \text{NoFLPFilt}: \text{disable subsumption-based flipped clause filtering}, \]
\[(d) \ \text{NoInSR}: \text{disable incremental score reboot}, \]
\[(e) \ \text{BtT0 and BtT100}: \text{apply backtracking with } T = 0 \text{ and } T = 100, \text{ respectively, for all the queries (to study the impact of query-driven tuning; recall that the default version uses different } T \text{’s for normal-incremental queries vs. the other two query types)}.
\]

The results are summarized in Table 1. Our main conclusions are as follows:

1. TT21I outperforms TT21G and both the MSE’20 and MSE’21 winners SL20 and SL21, respectively, on the MSE’20 set for every timeout and on the MSE’21 set starting with the 180 second timeout.

To understand the results, recall that all the solvers we have used apply the following high-level flow:
\[(a) \ \text{Run SAT to get the initial model } \mu, \]
\[(b) \ \text{Invoke SATLike—a Stochastic Local Search (SLS) algorithm [17]—to improve } \mu \text{ (where SL20, TT21G and TT21I use an identical version of SATLike, while SL21 has some undocumented novelties)}., \]
\[(c) \ \text{Switch to the main SAT-based flow to improve } \mu \text{ further.} \]
Table 1 Experimental Results Summary.

<table>
<thead>
<tr>
<th>Solver</th>
<th>MSE'20</th>
<th>MSE'21</th>
</tr>
</thead>
<tbody>
<tr>
<td></td>
<td>300</td>
<td>240</td>
</tr>
<tr>
<td>TT21I</td>
<td>.953</td>
<td>.944</td>
</tr>
<tr>
<td>TT21G</td>
<td>.933</td>
<td>.923</td>
</tr>
<tr>
<td>SL21</td>
<td>.918</td>
<td>.913</td>
</tr>
<tr>
<td>SL20</td>
<td>.908</td>
<td>.900</td>
</tr>
<tr>
<td>NoILB</td>
<td>.931</td>
<td>.925</td>
</tr>
<tr>
<td>NoFLP</td>
<td>.950</td>
<td>.938</td>
</tr>
<tr>
<td>NoFLPfilt</td>
<td>.931</td>
<td>.925</td>
</tr>
<tr>
<td>NoInSR</td>
<td>.941</td>
<td>.935</td>
</tr>
<tr>
<td>BtT0</td>
<td>.935</td>
<td>.921</td>
</tr>
<tr>
<td>BtT100</td>
<td>.943</td>
<td>.936</td>
</tr>
</tbody>
</table>

For the smaller timeouts, the relatively mild impact of IntelSAT on MSE'20 set and the lack of impact on MSE'21 set is observed because, initially, the performance and quality are dominated by SLS, where the arbitrary quality of the initial SAT model is also a factor. As the time goes by, the SAT solver performance is becoming the dominant factor, hence our results are an evidence that IntelSAT improves the state-of-the-art in SAT for unweighted anytime MaxSAT.

2. ILB is essential, though, surprisingly, NoILB slightly outperforms the default TT21I for the 60-second timeout on the MSE'21 set.

3. Flipped recording is helpful across the board, where the contribution of subsumption-based flipped clause filtering is crucial.

4. Incremental score reboot is useful everywhere, with one surprising exception of the 120-second timeout on the MSE'21 set.

5. The results of BtT0 and BtT100 demonstrate the critical contribution of query-driven tuning to the performance of our backtracking heuristic and TT21I.

6 Conclusion

We introduced Intel® SAT Solver (IntelSAT) – a new open-source SAT solver, written from scratch, optimized for applications generating many mostly satisfiable incremental SAT queries, such as anytime unweighted MaxSAT.

IntelSAT applies Incremental Lazy Backtracking (ILB), based on a novel reimplication procedure, which can reimplies an assigned literal at a lower level without backtracking.

With reimplication, we were able to restore the following two properties, lost in modern solvers with the introduction of chronological backtracking:

(a) **lowest implication**: no assigned literal can be implied at a lower level (that is, no missed lower implications exist after BCP), and

(b) **lowest conflict**: in case of a conflict, a clause falsified at the lowest possible level is returned.

In addition, we applied and empirically verified the usefulness of new heuristics, including query-driven tuning, subsumption-based flipped clause filtering and incremental score reboot.

Integrating IntelSAT into the MaxSAT solver TT-Open-WBO-Inc resulted in boosting TT-Open-WBO-Inc’s performance on incomplete unweighted MaxSAT Evaluation benchmarks and improving the state-of-the-art in anytime unweighted MaxSAT solving.
References


A. Nadel 8:17


8:18 Introducing Intel® SAT Solver


A Correctness Proofs

Lemma 11 captures the following assumption we have been and continue to use implicitly: every clause $C \in F$ appears once and only once in the WL of each of its watched literals $c_1$ and $c_2$, and there is nothing else is the WL’s (assuming that clauses of size $< 2$ are not added to $F$).

\begin{definition}
Lemma 11 (WL Correctness). If $L$ is the set of all the literals, then:
\begin{enumerate}[label=(\alph*)]
\item $\forall C \in F : \forall i \in \{1, 2\} : \exists \langle h \neq c, C \rangle \in WL(c_i)$, and
\item $\sum_{l \in L} |WL(l)| = 2 \times |F|$.
\end{enumerate}
\end{definition}

Proof. By construction.

Lemma 12 states that the set of $c_i$-stable clauses is closed under our straightforward backtracking implementation BACKTRACK (Alg. 1, line 11).

\begin{definition}
Lemma 12 ($c_i$-Stable Set Closure under BACKTRACK). For $i \in \{1, 2\}$, a $c_i$-stable clause $C$ remains $c_i$-stable after BACKTRACK.
\end{definition}

Proof. The only non-trivial case is when the watch $c_i$ (WLOG, let it be $c_1$) is falsified, in which case the cached literal $h(p, C)$ must be satisfied not higher than $c_1$. Since BACKTRACK unassigns literals in decreasing order, it is guaranteed that it either unassigns $c_1$ or, otherwise, $h(p, C)$ remains satisfied at $\delta l(p, C)$.

A.1 Reimply

Lemmas 13 and 14 argue for termination and soundness of REIMPLY.

\begin{definition}
Lemma 13 (REIMPLY Termination). If every $C \in \Lambda^R$ is reimplication-registered, REIMPLY terminates and empties $\Lambda^R$.
\end{definition}

Proof. Consider one iteration of the loop in Alg. 1, line 15. If it is discontinued at line 17, the iteration reduces the size of $\Lambda^R$. Otherwise, $R$ is not BCP-Safe and, by Cor. 7, REASSIGN() at line 18 lowers $r_1$. The size of $\Lambda^R$ and the level of every literal are lower bounded by 0, where literal levels are never increased. Hence, eventually, $\Lambda^R$ will be emptied and REIMPLY will terminate.

\begin{definition}
Lemma 14 (REIMPLY Soundness). If every clause $C \in F$ is either BCP-Safe or falsification-registered or reimplication-registered and every $C \in \Lambda^R$ is reimplication-registered at the beginning of REIMPLY, then every clause $C \in F$ will be BCP-Safe or falsification-registered after REIMPLY.
\end{definition}

Proof. Implied by Lemma 9 (recall Sect. 3.5), since:
1. Our condition ensures that Lemma 9’s loop invariants hold at the beginning of the first while-loop iteration, and
2. By Lemma 13, $\Lambda^R$ is emptied, hence all the clauses, except for any falsification-registered clauses, must become BCP-Safe.

A.2 F2C

We start off with Cor. 15, which follows from Lemma 12 and the fact that BACKTRACK does not change the status of $c_i$-BCP-Registered clauses.

Lemma 16 argues about the soundness of F2C, while the remaining lemmas will be useful for proving BCP correctness.

Lemma 16 (F2C Soundness). Every BCP-Safe clause remains BCP-Safe after F2C. Every falsification-registered clause becomes either BCP-Safe or contradicting.

Proof. Any BCP-Safe clauses will remain BCP-Safe, since F2C does not modify such clauses explicitly, while backtracking is safe by Cor. 15. As we have demonstrated while presenting F2C, F2C renders any falsification-registered clauses removed from $\Lambda^F$ BCP-Safe, while the remaining falsification-registered clauses are contradicting.

Lemma 17 (Backtracking in F2C). If there is at least one non-contradicting falsification-registered clause at the beginning of F2C, then F2C decreases the current decision level $d$.

Proof. Line 2 is skipped, since there is at least one non-contradicting falsification-registered clause. If some of the falsification-registered clauses are non-fake, backtracking at line 7 decreases $d$. Otherwise, backtracking at line 11 decreases $d$.

Lemma 18 (F2C Properties). If $\Lambda^F$ is non-empty at the beginning of an F2C invocation, then, by the end of F2C, it is either that:
1. There exists at least one contradicting clause, or
2. There exists a lowered flipped literal $l$, that is, an assigned literal $l$ which was assigned $\neg\sigma(l)$ at a higher level at the beginning of F2C.

Proof. Assume $\Lambda^F$ is non-empty at the beginning of an F2C invocation.

If all the falsification-registered clauses are contradicting, the algorithm returns at line 2 meeting our lemma’s condition 1. Otherwise, if there are no fake clauses, it backtracks so as to leave at least one contradicting clause in $\Lambda^F$ and returns, as required.

Assume now there are some fake clauses.

If after backtracking at line 7 any falsified clauses remain, line 11 renders at least one of them unit. Line 12 completes the flipping of the now unassigned literal in that unit clause, which is now assigned lower than previously by construction, as required by condition 2.

Finally, if no falsified fake clauses remain after backtracking at line 7, backtracking at line 11 is skipped. Hence, at least one contradicting clause remains, fulfilling condition 1.

A.3 BCP

The next two lemmas establish invariants for BCP’s for-loop and while-loop, respectively. In both lemmas, we show that if the corresponding invariants hold at the beginning of an iteration, they also hold at its end.

Lemma 19 (BCP For-Loop Invariant). The following are two loop invariants for the for-loop in Alg. 2, line 19:
1. Every $C \in F$ is either BCP-Safe or contradicting, except that any unvisited $C : \langle _, C \rangle \in WL(\neg l)$ might be $\neg l$-unstable (where, if the algorithm breaks the loop, all the clauses $C : \langle _, C \rangle \in WL(\neg l)$ are considered visited), and
2. $\Lambda^R$ is empty.
Introducing Intel® SAT Solver

Theorem 21 (BCP Correctness). If every \( C \in F \setminus \Lambda^R \) is BCP-Safe and every \( C \in \Lambda^R \) is reimplication-registered, then BCP terminates when every clause \( C \in F \) is either stable or contradicting.

Proof. BCP starts with invoking \textsc{Reimply}. \textsc{Reimply} terminates and empties \( \Lambda^R \) by Lemma 13. It also renders all the clauses BCP-Safe or falsification-registered by Lemma 14. In our case, all the clauses must be BCP-Safe, since \textsc{Reimply} does not modify the assignment \( \sigma \), and there are no falsification-registered clauses before \textsc{Reimply}. Hence, Lemma 20’s invariants hold at the beginning of the first while-loop iteration.

If the while-loop terminates, our theorem holds, since applying Lemma 20 iteratively renders all the clauses BCP-Safe or contradicting, where, for the loop to terminate, \( \Pi \) must be empty, thus all the clauses are stable or contradicting.

It is left to show that the while-loop terminates, where every single iteration terminates by Lemma 20.
Observe that the algorithm might reduce the current decision level \( d \) and unassign variables only when \textsc{Backtrack} is applied by \textsc{F2C}. However, this can occur only a finite number of times, since \( d \) is never increased and is lower bounded by 0.

Assume the algorithm reached the point, when \( d \) does not change and no variable is unassigned anymore. Each iteration decreases \(|\Pi|\) at line 17. We show that \(|\Pi|\) can only be increased for a finite number of times. Assign pushes to \( \Pi \), but the variables are not unassigned anymore, hence there is a finite number of potential assignments. Line 30 could push to \( \Pi \), but only after invoking \textsc{F2C} in the presence of a non-contradicting falsification-registered clause, which, by Lemma 17, would decrease \( d \). Finally, line 38 might push to \( \Pi \), however the number of such operations is also finite, since \textsc{Reimply} is bound to lower \( c_1 \) to fix the reimplication-registered clause \( C \), but the levels of variables are never increased and are lower bounded by 0.

Theorem 21 guarantees that, if every clause is either BCP-Safe or a reimplication-registered missed lower implication at the beginning, there are no missed unit clauses and no missed lower implications (since all the non-falsified clauses after BCP are stable). In addition, every falsified clause must be contradicting, therefore the lowest conflict property is guaranteed. Any falsified clause must belong to \( \Lambda^F \), therefore falsified clauses are not missed.

One could argue that the above properties would have held if BCP, e.g., had simply backtracked to level 0. We sketch a proof that BCP makes progress.

By Theorem 21, there are no unit clauses after BCP, hence, if BCP does not backtrack, it propagates all the unit clauses. By construction, BCP may backtrack only in \textsc{F2C}. Hence, by Lemma 18, each time BCP backtracks, it either flips a literal at a lower level (and propagates it by Theorem 21) or maintains at least one contradicting clause. These arguments demonstrate that every time BCP backtracks, it makes progress by either flipping a literal at a lower level and propagating (similarly to conflict analysis loop) or by simply maintaining a contradicting clause \( C \), where, in the latter case, if no backtracking and flipping occurs later in BCP, the conflict analysis loop will use \( C \) to backtrack, flip at a lower level and invoke BCP once again to propagate. Formalizing the arguments further would require extending our formal framework to reason about the whole CDCL SAT solving algorithm, which is outside of the scope of this paper.