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 reimply 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

Supplementary Material *Software (Source Code)*: `https://github.com/alexander-nadel/intel_sat_solver`; archived at `swh:1:rev:b5c43319c7bf98bf39593fde909d1379d12cbe21`

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++20. 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 Ψ, a MaxSAT solver returns a model (solution) to F which minimizes Ψ. In *unweighted* MaxSAT, Ψ has degree 1. In *anytime* MaxSAT, the solver is required to generate a series of solutions improving w.r.t Ψ. Anytimeness 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.

25th International Conference on Theory and Applications of Satisfiability Testing (SAT 2022).
Editors: Kuldeep S. Meel and Ofer Strichman; Article No. 8; pp. 8:1–8:23
Leibniz International Proceedings in Informatics
LIPICS Schloss Dagstuhl – Leibniz-Zentrum für Informatik, Dagstuhl Publishing, Germany

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 > 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.

2 Preliminaries

A literal l is either a Boolean variable v or its negation $\neg v$; l is called *positive* in the former case and *negative* in the latter. We denote $\mathit{var}(v)=\mathit{var}(\neg v)=v$. A clause is a disjunction of literals. The SAT solver maintains each clause $C = \left[c_1, c_2, \ldots, c_{|C|}\right]$ as an *ordered* sequence 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 $\left[c_i, c_{i+1}, \ldots, c_{|C|}\right]$ by $C_{[i\ldots]}$ (where $i \geqslant 1$ and $C_{[i\ldots]}$ is empty if $i > |C|$). When the literal order is irrelevant, we use curly brackets in the clause definition: $C = \left\{c_1, c_2, \ldots, c_{|C|}\right\}$. 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 σ, which, for every variable v, holds its current value $\sigma(v) \in \{U, \top, \bot\}$. A literal/variable l is *assigned* iff $\sigma(var(l)) \neq U$, 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* iff $\sigma(v) = \top/\bot$, 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, Unisat, Falsified). *A clause C is* unit *iff $\#U(C) = 1$ and $\#\bot(C) = |C| - 1$. C is* unisat *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)* $dl(v) \leqslant d$ (where $dl(\neg v) \equiv dl(v)$). 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 $F \wedge A$, 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* Π, 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 Π), in which case we say that c_1 is *implied* in the *parent clause* C. The algorithm sets $dl(c_1)$ to $maxl(C_{[2\ldots]})$, where $maxl(D)$ is the maximal level 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 $i \in \{1,2\}$ (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 $dl(c_i) \geqslant maxl(C_{[2\ldots]})$ and $\forall j > 2 : c_j$ is falsified.

2.1.2 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 $lit(v)$ 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 $dl(d_1) = d > dl(d_2) \geqslant maxl(D_{[2\ldots]})$. 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 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 $C = \{\neg l_1, l_2, l_3\} \wedge D = \{\neg l_3, l_4\}$. Assume there are no new input clauses until the next invocation 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 AddClause

ILB requires enabling 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 $c_i : dl(c_i) \geqslant maxl(C)$, 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 unisat (recall Def. 1), where the satisfied literal is l. We watch l. If there exists $c_i \neq l : dl(c_i) \geqslant maxl(C)$, c_i is also watched. The only remaining non-trivial case led us to introducing reimplication, which we apply to complete 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 $dl(l)$.

▶ **Definition 2** (Missed Lower Implication). *A clause C is a* Missed Lower Implication (Lower Implication) *iff C is unisat and, assuming c_i is the satisfied literal in C, it holds that: $\forall j \neq i : dl(c_i) > dl(c_j)$, where $1 \leqslant i, j \leqslant |C|$.*

3.3.1 AddClause

Assume 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 $maxl(C_{[2\ldots]})$. 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, `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 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_3 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.

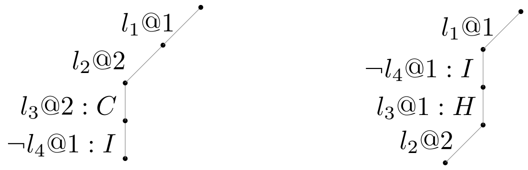
(a) $C = \{\neg l_1, l_2, l_3\} \wedge D = \{\neg l_3, l_4\}$.

(b) $C \wedge D \wedge (E = \{\neg l_2, \neg l_1\})$.

Figure 1 Reimplication in AddClause. Fig. 1a shows a solver invocation trace (without any conflicts), given the clauses $C = \{\neg l_1, l_2, l_3\} \wedge D = \{\neg l_3, l_4\}$, where $l@x$ means that l is the decision literal at level x and $l@x : C$ 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 $dl(\neg l_2)$. 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) Conflict. **(b)** CB. **(c)** `IntelSAT`.

Figure 2 Reimplication in BCP. A SAT solver is invoked over F, which includes $C = \{\neg l_2, l_3\}$, $D = \{\neg l_1, \neg l_4, l_5\}$, $E = \{\neg l_1, \neg l_4, \neg l_5\}$ and $H = \{l_4, l_3\}$. Consider the conflict in Fig. 2a. The algorithm will learn $I = \{\neg l_1, \neg l_4\}$. Hence, $\neg l_4$ will be implied at level 1 after backtracking. Assume a solver which implements CB [50] is used (we refer to it simply as *CB* throughout this example). CB backtracks to level 2 and places $\neg l_4$ after the literals assigned at level 2, as shown in Fig. 2b. At this point, H becomes a missed lower implication: l_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 l_3's parent. Instead, `IntelSAT` places $\neg l_4$ 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: $J = \{\neg l_3, l_4, l_5\}$ and $K = \{\neg l_3, l_4, \neg l_5\}$. There will be a conflict immediately after propagating $\neg l_4$ for both CB and `IntelSAT`. CB will learn $L = \{l_4, \neg l_3\}$, while `IntelSAT` will learn $M = \{l_4\}$, which subsumes L. Assume now that, in addition to J and K, the clauses $N = \{\neg l_2, l_4, l_6\}$ and $P = \{\neg l_2, l_4, \neg l_6\}$ also belonged to F. In this case, there would be *two* potential conflicts after propagating $\neg l_4$. For `IntelSAT`, the two potential learnt clauses would be M or $Q = \{\neg l_2, l_4\}$. `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 l has a satisfied cached literal h, then h can no longer be higher than l. 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).

▶ **Definition 3** (c_i-Stable Clause, Stable Clause)**.** *For $i \in \{1, 2\}$, a clause C is c_i-*Stable *if:*
(a) $\sigma(c_i) \neq \bot$*, or*
(b) $\exists \langle h, C\rangle \in WL(c_i) : \sigma(h) = \top$ *and* $dl(h) \leqslant dl(c_i)$*.*
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 Π.

▶ **Definition 4** (c_i-BCP-Registered)**.** *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.

▶ **Definition 5** (c_i-BCP-Safe, BCP-Safe)**.** *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 Λ^R, so that reimplication could fix them one by one (where some clauses in Λ^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) $\sigma(c_1) = \top$*, and*
(d) $h(c_2, C) = c_1$*, and*
(e) $dl(c_2) \geqslant maxl(C_{[3\ldots]})$.

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 Λ^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 Λ^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 σ, 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 $C \in F \backslash \Lambda^R$ 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 Λ^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 $Q \in \Lambda^R : maxl(Q_{[2\ldots]}) \leqslant maxl(R_{[2\ldots]})$). Otherwise, line 18 fixes R (recall Cor. 7).

Algorithm 1 Reimply and Other Functions.

```
1: function Assign(Literal $l$, Parent clause $P$, Decision level $dl$)
2:     if $l = \neg var(l)$ then $\sigma(var(l)) := \bot$ else $\sigma(var(l)) := \top$
3:     $\Pi.push\_back(l); dl(l) := dl; Parent(var(l)) := P$
4: function Reassign(Literal $l$, Parent clause $P$, Level $dl$)
5:     $dl(var(l)) := dl; Parent(var(l)) := P$
6: function Swap(Clause $C$, $c_{1 \leqslant i \leqslant |C|}$, $c_{1 \leqslant j \leqslant |C|}$) $\{t := c_i; c_i := c_j; c_j := t\}$
7: function UpdateWatched($C$, $o \in \{c_1, c_2\}$, $n \in C$)
8:     Swap($C$, $o$, $n$)    ▷ if $n \in \{c_1, c_2\}$, the two lines below update $h(n, C)$ to the other watch
9:     if $n \notin \{c_1, c_2\}$ then $WL(o) := WL(o) \setminus \langle \_, C \rangle$ else $WL(n) := WL(n) \setminus \langle \_, C \rangle$
10:    $WL(n) = WL(n) \cup \{\langle c_{i \in \{1,2\}} : c_i \neq n, C \rangle\}$
11: function Backtrack($b$)
12:     while $dl(l) > b$ for the last literal in the trail $l$ do $\sigma(var(l)) := U$
13:     $d := b$
14: function Reimply
15:     while $\Lambda^R$ is not empty do
16:         Clause $R := \Lambda^R.back(); \Lambda^R.pop()$
17:         if $R$ is BCP-Safe then continue    ▷ Otherwise, $R$ is a lower implication
18:         Reassign($r_1$, $R$, $dl(r_2)$)    ▷ Fixing $R$ to make it BCP-Safe
19:         if $\neg r_1 \in \Pi$ then continue
20:         for $\langle h, C \rangle \in WL(\neg r_1)$ do    ▷ By the end of the iteration, $C$ will either be BCP-Safe or reimplication-registered or falsification-registered
21:             if $C \in \Lambda^R$ or $C \in \Lambda^F$ then continue
22:             if $h$ is satisfied at $dl(h) \leqslant dl(r_1)$ then continue    ▷ No clause visit!
23:             if $c_1 \neq h$ then UpdateWatched($C$, $c_1$, $h$)
24:             if $\exists i > 2 : \sigma(c_i) \neq \bot$ then $x := i$ else $x := a : \forall j > 1 : dl(c_a) \geqslant dl(c_j)$
25:             UpdateWatched($C$, $c_2$, $c_x$)    ▷ Assigns $h(c_2, C) := h$, even if $c_2 \equiv c_x$
26:             if $C$ is a lower implication then $\Lambda^R.push\_back(C)$
```

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 Λ^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 Λ^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 $maxl(C)$. A backtrack-contradicting clause C is* Contradicting *if $maxl(C) = d$ (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 Λ^F, where, normally, all the falsification-registered clauses are contradicting. They might cease being contradicting on two occasions:

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

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

In both these cases, we invoke the function F2C.

3.6.1 F2C

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 Λ^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 Λ^F. Backtracking also renders any contradicting clauses BCP-Safe. Line 12 flips the now unassigned literal in every unit clause in Λ^F. At this point, all the clauses in F, including those in Λ^F, become BCP-Safe. Hence, line 13 empties Λ^F, and 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 Λ^F at line 12, which also renders these clauses BCP-Safe. Therefore, F2C returns with only contradicting clauses in Λ^F, the rest having been turned BCP-Safe.

Algorithm 2 Boolean Constraint Propagation (BCP).

```
function F2C
    if all the falsification-registered clauses are contradicting or $\Lambda^F$ is empty then return
    for $C \in \Lambda^F$ do
        UpdateWatched($C$, $c_1$, $c_i \in C : dl(c_i) = maxl(C)$)
        UpdateWatched($C$, $c_2$, $c_i \in C_{[2\ldots]} : dl(c_i) = maxl(C_{[2\ldots]})$)
    if $\exists C \in \Lambda^F : C$ is backtrack-contradicting then
        Backtrack(lowest possible level, such that $\exists C \in \Lambda^F : C$ is backtrack-contradicting)
    if there were no fake clauses in the beginning then
        Remove any BCP-Safe clauses from $\Lambda^F$ and return
    if $\exists C \in \Lambda^F : C$ is falsified then
        Backtrack(highest possible level, such that $\nexists C \in \Lambda^F : C$ is falsified)
    for $C \in \Lambda^F : C$ is unit do Assign($c_1$, $C$, $dl(c_2)$)
    Remove any BCP-Safe clauses from $\Lambda^F$
function BCP
    Reimply()
    while $\Pi$ is not empty do
        $l := \Pi.back(); \Pi.pop()$
        if $\neg l$ is non-falsified then continue
        for $\langle h, C \rangle \in WL(\neg l)$ do
            if $c_2 \neq \neg l$ then Swap($C$, $c_1$, $c_2$)          ▷ Ensure $c_2 \equiv \neg l$ for simplicity
            if $h$ is satisfied at $dl(h) \leqslant dl(l)$ then continue
            if $\#NF(C_{[3\ldots]}) > 0$ then                 ▷ A non-falsified unwatched literal exists
                Let $c_i \in C_{[3\ldots]}$ be a non-falsified literal in $C_{[3\ldots]}$
                UpdateWatched($C$, $c_2$, $c_i$)
            else                                    ▷ Now all the literals in $C_{[2\ldots]}$ are falsified
                if $c_1$ is falsified then                                      ▷ $C$ is falsified
                    $\Lambda^F.push\_back(C)$
                    if $C$ is not contradicting then
                        F2C()
                        $\Pi.push\_back(l)$
                        break
                else                                                     ▷ $C$ is unit or unisat
                    UpdateWatched($C$, $c_2$, $c_i \in C_{[2\ldots]} : dl(c_i) = maxl(C_{[2\ldots]})$)
                    if $c_1$ is unassigned then                                      ▷ $C$ is unit
                        Assign($c_1$, $C$, $dl(c_2)$)                                ▷ Implication
                    else if $dl(c_1) > dl(c_2)$ then                               ▷ Reimplication
                        $\Lambda^R.push\_back(C)$
                        $\Pi.push\_back(l)$
                        Reimply()
                        F2C()
                        break
    if $\Lambda^F$ is empty return $\top$ 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 Π. 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 $c_2 \equiv \neg l$ (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 $WL(\neg l)$, 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 $C_{[2\ldots]}$ 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 $dl(c_1) \leqslant dl(c_2)$, 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 $C > D$ iff:

(a) $l(C) < 16$ and $l(D) \geqslant 16$, or
(b) $l(C) \geqslant 16$ and $l(D) \geqslant 16$ and $a(C) \geqslant a(D)$, or
(c) $l(C) < 16$ and $l(D) < 16$ and $(l(C)/11 < l(D)/11$ or $(l(C)/11 = l(D)/11$ and $a(C) \geqslant a(D)))$, 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 Ψ and the model μ, the *score* per instance is: (1 + 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) the default `TT-Open-WBO-Inc-21` with Glucose 4.1 (`TT21G`),
(b) `TT-Open-WBO-Inc-21` with `IntelSAT` (`TT21I`),
(c) `SATLike-c` (`SL21`): the winner of MSE'21,
(d) `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) `NoILB`: disable ILB by backtracking to 0 before Solve and AddClause,
(b) `NoFLP`: disable flipped recording,
(c) `NoFLPFilt`: disable subsumption-based flipped clause filtering,
(d) `NoInSR`: disable incremental score reboot,
(e) `BtT0` and `BtT100`: apply backtracking with $T = 0$ and $T = 100$, respectively, for all the queries (to study the impact of query-driven tuning; recall that the default version uses different T'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) Run SAT to the get the initial model μ.
   (b) Invoke SATLike–a Stochastic Local Search (SLS) algorithm [17]–to improve μ (where `SL20`, `TT21G` and `TT21I` use an identical version of SATLike, while `SL21` has some undocumented novelties).
   (c) Switch to the main SAT-based flow to improve μ further.

Table 1 Experimental Results Summary.

Solver	MSE'20					MSE'21				
	300	240	180	120	60	300	240	180	120	60
`TT21I`	.953	.944	.933	.914	.880	.941	.933	.901	.874	.837
`TT21G`	.933	.923	.911	.899	.874	.912	.901	.888	.872	.837
`SL21`	.918	.913	.900	.887	.866	.913	.910	.897	.874	.843
`SL20`	.908	.900	.890	.877	.849	.908	.892	.884	.863	.815
`NoILB`	.931	.925	.916	.902	.870	.922	.911	.886	.865	.839
`NoFLP`	.950	.938	.924	.901	.868	.915	.903	.882	.862	.826
`NoFLPFilt`	.931	.925	.918	.899	.869	.915	.902	.882	.857	.818
`NoInSR`	.941	.935	.925	.909	.871	.926	.917	.893	.875	.833
`BtT0`	.935	.921	.915	.901	.864	.921	.901	.885	.867	.821
`BtT100`	.943	.936	.928	.913	.874	.924	.910	.888	.867	.836

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 reimply 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

1 Carlos Ansótegui, Felip Manyà, Jesus Ojeda, Josep M. Salvia, and Eduard Torres. Incomplete maxsat approaches for combinatorial testing. *CoRR*, abs/2105.12552, 2021. `arXiv:2105.12552`.

2 Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. RCL: Reduce learnt clauses. `https://baldur.iti.kit.edu/sat-race-2010/descriptions/solver_10.pdf`, 2010. Online; accessed 23 December 2021.

3 Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009. URL: `http://ijcai.org/Proceedings/09/Papers/074.pdf`.

4 Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018. `doi:10.1142/S0218213018400018`.

5 Fahiem Bacchus. MaxSat Lib. `https://www.cs.toronto.edu/maxsat-lib/`. Online; accessed 5 January 2022.

6 Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors. *MaxSAT Evaluation 2020: Solver and Benchmark Descriptions*, Department of Computer Science Report Series B. University of Helsinki, 2020.

7 Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors. *MaxSAT Evaluation 2021: Solver and Benchmark Descriptions*, Department of Computer Science Report Series B. University of Helsinki, 2021.

8 Fahiem Bacchus, Matti Järvisalo, and Ruben Martins. Maximum satisfiabiliy. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 929–991. IOS Press, 2021. `doi:10.3233/FAIA201008`.

9 Paul Beame, Henry A. Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *CoRR*, abs/1107.0044, 2011. `arXiv:1107.0044`.

10 Daniel Le Berre and Anne Parrain. The sat4j library, release 2.2. *JSAT*, 7(2-3):59–6, 2010. URL: `http://jsat.ewi.tudelft.nl/content/volume7/JSAT7_4_LeBerre.pdf`, `doi:10.3233/sat190075`.

11 Armin Biere. Adaptive restart strategies for conflict driven SAT solvers. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 28–33. Springer, 2008. `doi:10.1007/978-3-540-79719-7_4`.

12 Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Heisinger. CaDiCaL, Kissat, Paracooba, Plingeling and Treengeling entering the SAT Competition 2020. In Tomas Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Competition 2020 – Solver and Benchmark Descriptions*, volume B-2020-1 of *Department of Computer Science Report Series B*, pages 51–53. University of Helsinki, 2020.

13 Armin Biere and Mathias Fleury. Chasing target phases. In *Pragmatics of SAT (PoS)*, 2020.

14 Armin Biere and Andreas Fröhlich. Evaluating CDCL variable scoring schemes. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 405–422. Springer, 2015. `doi:10.1007/978-3-319-24318-4_29`.

15 Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2021.

16 Armin Biere, Matti Järvisalo, and Benjamin Kiesl. Preprocessing in sat solving. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 336 of *Frontiers in Artificial Intelligence and Applications*, pages 391–435. IOS Press, 2021. `doi:10.3233/FAIA200992`.

17 Shaowei Cai and Zhendong Lei. Old techniques in new ways: Clause weighting, unit propagation and hybridization for maximum satisfiability. *Artif. Intell.*, 287:103354, 2020. `doi:10.1016/j.artint.2020.103354`.

18 Shaowei Cai and Xindi Zhang. Deep cooperation of CDCL and local search for SAT. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 64–81. Springer, 2021. `doi:10.1007/978-3-030-80223-3_6`.

19 Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. Cache conscious data structures for boolean satisfiability solvers. *J. Satisf. Boolean Model. Comput.*, 6(1-3):99–120, 2009. `doi:10.3233/sat190064`.

20 Nachum Dershowitz, Ziyad Hanna, and Alexander Nadel. Towards a better understanding of the functionality of a conflict-driven SAT solver. In João Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10th International Conference, Lisbon, Portugal, May 28-31, 2007, Proceedings*, volume 4501 of *Lecture Notes in Computer Science*, pages 287–293. Springer, 2007. `doi:10.1007/978-3-540-72788-0_27`.

21 Rohit Dureja, Arie Gurfinkel, Alexander Ivrii, and Yakir Vizel. IC3 with internal signals. In *Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021*, pages 63–71. IEEE, 2021. `doi:10.34727/2021/isbn.978-3-85448-046-4_14`.

22 Niklas Eén and Niklas Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2003. `doi:10.1007/978-3-540-24605-3_37`.

23 Nick Feng and Fahiem Bacchus. Clause size reduction with all-uip learning. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 28–45. Springer, 2020. `doi:10.1007/978-3-030-51825-7_3`.

24 Andrea Formisano and Flavio Vella. On multiple learning schemata in conflict driven solvers. In Stefano Bistarelli and Andrea Formisano, editors, *Proceedings of the 15th Italian Conference on Theoretical Computer Science, Perugia, Italy, September 17-19, 2014*, volume 1231 of *CEUR Workshop Proceedings*, pages 133–146. CEUR-WS.org, 2014. URL: `http://ceur-ws.org/Vol-1231/long10.pdf`.

25 Alberto Griggio and Marco Roveri. Comparing different variants of the ic3 algorithm for hardware model checking. *IEEE Trans. Comput. Aided Des. Integr. Circuits Syst.*, 35(6):1026–1039, 2016. `doi:10.1109/TCAD.2015.2481869`.

26 Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. Learning for dynamic subsumption. *Int. J. Artif. Intell. Tools*, 19(4):511–529, 2010. `doi:10.1142/S0218213010000303`.

27 HyoJung Han and Fabio Somenzi. On-the-fly clause improvement. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 209–222. Springer, 2009. `doi:10.1007/978-3-642-02777-2_21`.

28 Randy Hickey and Fahiem Bacchus. Speeding up assumption-based SAT. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 164–182. Springer, 2019. `doi:10.1007/978-3-030-24258-9_11`.

29 Randy Hickey and Fahiem Bacchus. Trail saving on backtrack. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 46–61. Springer, 2020. `doi:10.1007/978-3-030-51825-7_4`.

30 Hao Hu, Mohamed Siala, Emmanuel Hebrard, and Marie-José Huguet. Learning optimal decision trees with maxsat and its integration in adaboost. In Christian Bessiere, editor, *Proceedings of the Twenty-Ninth International Joint Conference on Artificial Intelligence, IJCAI 2020*, pages 1170–1176. ijcai.org, 2020. `doi:10.24963/ijcai.2020/163`.

31 Stepan Kochemazov. Improving implementation of SAT competitions 2017-2019 winners. In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing - SAT 2020 - 23rd International Conference, Alghero, Italy, July 3-10, 2020, Proceedings*, volume 12178 of *Lecture Notes in Computer Science*, pages 139–148. Springer, 2020. `doi:10.1007/978-3-030-51825-7_11`.

32 Stepan Kochemazov, Alexey Ignatiev, and João Marques-Silva. Assessing progress in SAT solvers through the lens of incremental SAT. In Chu-Min Li and Felip Manyà, editors, *Theory and Applications of Satisfiability Testing - SAT 2021 - 24th International Conference, Barcelona, Spain, July 5-9, 2021, Proceedings*, volume 12831 of *Lecture Notes in Computer Science*, pages 280–298. Springer, 2021. `doi:10.1007/978-3-030-80223-3_20`.

33 Zhendong Lei, Shaowei Cai, Fei Geng, Dongxu Wang, Yongrong Peng, Dongdong Wan, Yiping Deng, and Pinyan Lu. SATLike-c: Solver description. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors, *MaxSAT Evaluation 2021*, Department of Computer Science Report Series B, pages 19–20. University of Helsinki, 2021.

34 Alexandre Lemos, Pedro T. Monteiro, and Inês Lynce. Minimal perturbation in university timetabling with maximum satisfiability. In Emmanuel Hebrard and Nysret Musliu, editors, *Integration of Constraint Programming, Artificial Intelligence, and Operations Research - 17th International Conference, CPAIOR 2020, Vienna, Austria, September 21-24, 2020, Proceedings*, volume 12296 of *Lecture Notes in Computer Science*, pages 317–333. Springer, 2020. `doi:10.1007/978-3-030-58942-4_21`.

35 Chu-Min Li, Fan Xiao, Mao Luo, Felip Manyà, Zhipeng Lü, and Yu Li. Clause vivification by unit propagation in CDCL SAT solvers. *Artif. Intell.*, 279, 2020. `doi:10.1016/j.artint.2019.103197`.

36 Jia Hui Liang, Chanseok Oh, Vijay Ganesh, Krzysztof Czarnecki, and Pascal Poupart. MapleCOMSPS, MapleCOMSPS_LRB, MapleCOMSPS_CHB. In Matti Juhani Järvisalo, editor, *Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions*, Department of Computer Science Series of Publications B, page 52, Finland, 2016. University of Helsinki.

37 Sibylle Möhle and Armin Biere. Backing backtracking. In Mikolás Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing - SAT 2019 - 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9-12, 2019, Proceedings*, volume 11628 of *Lecture Notes in Computer Science*, pages 250–266. Springer, 2019. `doi:10.1007/978-3-030-24258-9_18`.

38 Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*, pages 530–535. ACM, 2001. `doi:10.1145/378239.379017`.

39 Alexander Nadel. *Understanding and Improving a Modern SAT Solver.* Dissertation, Tel Aviv University, 2009. URL: `https://tau-primo.hosted.exlibrisgroup.com/permalink/f/bqa2g2/972TAU_ALMA21235249890004146`.

40 Alexander Nadel. Solving maxsat with bit-vector optimization. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 54–72. Springer, 2018. `doi:10.1007/978-3-319-94144-8_4`.

41 Alexander Nadel. Anytime weighted maxsat with improved polarity selection and bit-vector optimization. In Clark W. Barrett and Jin Yang, editors, *2019 Formal Methods in Computer Aided Design, FMCAD 2019, San Jose, CA, USA, October 22-25, 2019*, pages 193–202. IEEE, 2019. `doi:10.23919/FMCAD.2019.8894273`.

42 Alexander Nadel. Anytime algorithms for maxsat and beyond. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, page 1. IEEE, 2020. `doi:10.34727/2020/isbn.978-3-85448-042-6_1`.

43 Alexander Nadel. On optimizing a generic function in SAT. In *2020 Formal Methods in Computer Aided Design, FMCAD 2020, Haifa, Israel, September 21-24, 2020*, pages 205–213. IEEE, 2020. `doi:10.34727/2020/isbn.978-3-85448-042-6_28`.

44 Alexander Nadel. Polarity and variable selection heuristics for SAT-based anytime MaxSAT. *J. Satisf. Boolean Model. Comput.*, 12:17–22, 2020.

45 Alexander Nadel. TT-Open-WBO-Inc-20: an anytime maxsat solver entering MSE'20. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors, *MaxSAT Evaluation 2020*, Department of Computer Science Report Series B, pages 32–33. University of Helsinki, 2020.

46 Alexander Nadel. TT-Open-WBO-Inc-21: an anytime maxsat solver entering MSE'21. In Fahiem Bacchus, Jeremias Berg, Matti Järvisalo, and Ruben Martins, editors, *MaxSAT Evaluation 2021*, Department of Computer Science Report Series B, pages 21–22. University of Helsinki, 2021.

47 Alexander Nadel. TT-Open-WBO-Inc-21 with Intel® SAT Solver. `https://drive.google.com/file/d/1taWT1Kp16xzXp9FWKHzQwjwo80v_0hoh/view?usp=sharing`, 2022.

48 Alexander Nadel. Intel® SAT Solver. `https://github.com/alexander-nadel/intel_sat_solver`, 2022.

49 Alexander Nadel and Vadim Ryvchin. Efficient SAT solving under assumptions. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference, Trento, Italy, June 17-20, 2012. Proceedings*, volume 7317 of *Lecture Notes in Computer Science*, pages 242–255. Springer, 2012. `doi:10.1007/978-3-642-31612-8_19`.

50 Alexander Nadel and Vadim Ryvchin. Chronological backtracking. In Olaf Beyersdorff and Christoph M. Wintersteiger, editors, *Theory and Applications of Satisfiability Testing - SAT 2018 - 21st International Conference, SAT 2018, Held as Part of the Federated Logic Conference, FloC 2018, Oxford, UK, July 9-12, 2018, Proceedings*, volume 10929 of *Lecture Notes in Computer Science*, pages 111–121. Springer, 2018. `doi:10.1007/978-3-319-94144-8_7`.

51 Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT modulo theories: From an abstract davis–putnam–logemann–loveland procedure to dpll(T). *J. ACM*, 53(6):937–977, 2006. `doi:10.1145/1217856.1217859`.

52 Chanseok Oh. Between SAT and UNSAT: the fundamental difference in CDCL SAT. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 307–323. Springer, 2015. `doi:10.1007/978-3-319-24318-4_23`.

53 Vadim Ryvchin and Alexander Nadel. Flipped recording. In *Pragmatics of SAT (PoS)*, 2019.

54 Vadim Ryvchin and Ofer Strichman. Local restarts. In Hans Kleine Büning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008, 11th International Conference, SAT 2008, Guangzhou, China, May 12-15, 2008. Proceedings*, volume 4996 of *Lecture Notes in Computer Science*, pages 271–276. Springer, 2008. `doi:10.1007/978-3-540-79719-7_25`.

55 João P. Marques Silva and Karem A. Sakallah. GRASP - a new search algorithm for satisfiability. In Rob A. Rutenbar and Ralph H. J. M. Otten, editors, *Proceedings of the 1996 IEEE/ACM International Conference on Computer-Aided Design, ICCAD 1996, San Jose, CA, USA, November 10-14, 1996*, pages 220–227. IEEE Computer Society / ACM, 1996. `doi:10.1109/ICCAD.1996.569607`.

56 Niklas Sörensson and Armin Biere. Minimizing learned clauses. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 237–243. Springer, 2009. `doi:10.1007/978-3-642-02777-2_23`.

57 Peter van der Tak, Antonio Ramos, and Marijn Heule. Reusing the assignment trail in CDCL solvers. *J. Satisf. Boolean Model. Comput.*, 7(4):133–138, 2011. `doi:10.3233/sat190082`.

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).

▶ **Lemma 11** (WL Correctness). *If L is the set of all the literals, then:*
(a) $\forall C \in F : \forall i \in \{1, 2\} : \exists \langle h \neq c_i \in C, C \rangle \in WL(c_i)$*, and*
(b) $\sum_{l \in L} |WL(l)| = 2 \times |F|$.

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).

▶ **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.*

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 $h(c_1, C)$ 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, $h(c_1, C)$ remains satisfied at $dl(c_1)$. ◀

A.1 Reimply

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

▶ **Lemma 13** (Reimply Termination). *If every $C \in \Lambda^R$ is reimplication-registered, Reimply terminates and empties Λ^R.*

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 Λ^R. Otherwise, R is not BCP-Safe and, by Cor. 7, Reassign() at line 18 lowers r_1. The size of Λ^R and the level of every literal are lower bounded by 0, where literal levels are never increased. Hence, eventually, Λ^R will be emptied and Reimply will terminate. ◀

▶ **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.*

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, Λ^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.

▶ **Corollary 15** (BCP-Safe Closure under Backtrack)**.** *Any BCP-Safe clause remains BCP-Safe after Backtrack.*

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 Λ^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 Λ^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 Λ^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 Λ^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. *Λ^R is empty.*

Proof. While presenting the algorithm, we have demonstrated that the currently visited clause C becomes BCP-Safe or contradicting by the end of the for-loop iteration. The rest of the clauses are modified only during F2C and REIMPLY.

Consider the F2C invocation at line 29. By Lemma 16, it leaves BCP-Safe clauses BCP-Safe and renders any falsification-registered clauses either BCP-Safe or contradicting. The remaining potentially $\neg l$-unstable unvisited clauses in $WL(\neg l)$ are rendered $\neg l$-BCP-Registered (thus $\neg l$-BCP-Safe) explicitly at line 30, so one can safely break the loop.

Consider the REIMPLY invocation at line 39. We render all the remaining clauses in $WL(\neg l)$ BCP-Safe explicitly just before invoking REIMPLY by registering l for BCP (line 38). In addition, by our loop invariant, Λ^R is empty before adding C at line 37, and C becomes reimplication-registered by construction. Hence, by Lemma 14, all the clauses are either BCP-Safe or falsification-registered after REIMPLY, which is immediately followed by an F2C invocation (line 40) to make all the clauses BCP-Safe or contradicting. By Lemma 13, REIMPLY empties Λ^R as required. ◀

▶ **Lemma 20** (BCP While-Loop Invariant). *The following are two loop invariants for the while-loop in Alg. 2, line 16:*

1. *Every $C \in F$ is either BCP-Safe or contradicting, and*
2. *Λ^R is empty.*

Additionally, any single while-loop iteration terminates.

Proof. If our iteration is discontinued at line 18, the invariants still hold, since, although clauses in $WL(\neg l)$ become non-$\neg l$-BCP-Registered, they are $\neg l$-stable and thus BCP-Safe, since $\neg l$ is non-falsified. Other clauses and Λ^R are not modified.

Otherwise, the algorithm reaches the for-loop, where Lemma 19's invariants hold, since our lemma ensures that Λ^R is empty and every $C \in F$ is either BCP-Safe or contradicting with the exception of the clauses in $WL(\neg l)$, which might have become $\neg l$-unsafe, as required. Hence, Lemma 19's invariants still hold at the end of the for-loop iteration.

The for-loop terminates, since only REIMPLY and F2C might modify $WL(\neg l)$, but we break the for-loop immediately after applying these functions. By construction, the for-loop terminates after visiting all the clauses in $WL(\neg l)$ (if the algorithm breaks the for-loop, all the clauses in $WL(\neg l)$ are considered visited by Lemma 19). Therefore, by Lemma 19, all the clauses are either BCP-Safe or contradicting and Λ^R is empty by the end of the for-loop, and thus by the end of our while-loop iteration, which also terminates, as required. ◀

Our main Theorem 21 argues that BCP is correct.

▶ **Theorem 21** (BCP Correctness). *If every $C \in F \backslash \Lambda^R$ 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 REIMPLY. REIMPLY terminates and empties Λ^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 REIMPLY does not modify the assignment σ, and there are no falsification-registered clauses before 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, Π 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 BACKTRACK is applied by 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 Π, but the variables are not unassigned anymore, hence there is a finite number of potential assignments. Line 30 could push to Π, but only after invoking F2C in the presence of a non-contradicting falsification-registered clause, which, by Lemma 17, would decrease d. Finally, line 38 might push to Π, however the number of such operations is also finite, since 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 Λ^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 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.