Université de Liège - Faculté des Sciences Appliquées

Chronological vs. Non-Chronological Backtracking in SMT

Travail de fin d'études réalisé en vue de l'obtention du grade de master Ingénieur Civil en Informatique par Coutelier Robin

Promoteur Académique: Pr. Fontaine Pascal

Année académique 2022-2023

Abstract

SMT solvers are a powerful tool used in many verification applications. They are designed to check the satisfiability of a logic formula in a given set of theories. The core of many SMT solvers is a fine-tuned SAT solver enhanced with more expressive theory reasoners. Recently, SAT solvers have seen a different paradigm emerge with the re-introduction of Chronological Backtracking. This new approach has already shown some improvements on state-of-the-art standalone SAT solvers. In the following, we detail the functioning of CDCL solvers and the assumptions that non-chronological backtracking enables. After that, we study what it takes to convert a standard SAT solver to support chronological backtracking. We discuss both necessary and optional changes to the solver and present algorithms for each of them. We implement these changes on the SAT solver of `veriT`, an SMT solver written in C. Finally, we consider some details when adapting the SMT solver to support chronological backtracking. We present ideas for future work and discuss the potential benefits of this new approach.

Contents

Introduction

SMT solving is a powerful technique used in many applications such as model checking, program verification and theorem proving [2, 26]. A typical SMT solver will use a SAT solver to find models of a weakened version of the input problem, then refine the search by adding new propositional constraints to the SAT solver. This incremental approach has the property of asking mostly satisfiable queries to the SAT solver. Nadel et Al. and Möhle et al. [22, 19, 21] re-introduced and formalized a different paradigm for SAT solving that performs particularly well on satisfiable instances. This paradigm is called Chronological Backtracking (CB). In this thesis, we first detail the standard DPLL and CDCL algorithms used by modern SAT solvers. Then, we present the key ideas behind chronological backtracking as portrayed in the literature. We then explain the algorithms used in `veriT` to convert the SAT solver to CB. To that effect, we present a weak and strong version of chronological backtracking. Weak chronological backtracking only implements a minimal set of changes to obtain a sound and complete SAT solver. Strong chronological backtracking adds some complications to ensure that no implication is missed. This includes what [21] calls "missed lower implications" and how it is handled under constraints of minimal change to the core data structures of the SAT solver. We provide efficient algorithms to preserve the topological order of the trail after re-implication of literals. Finally, we discuss the impact chronological backtracking will have on the SMT solver. We explore some ideas on enhancement of the SAT solver's interface to allow for more efficient SMT solving. We explore some potential pitfalls of the conversion and how the interface between the SAT and SMT solver can be modified to avoid them.

The SAT solver of `veriT` was modified to implement both the weak and strong chronological backtracking algorithms. The code is theoretically motivated and empirically checked. However, the conversion of the SMT solver still remains theoretical. In future work, we intend to implement chronological backtracking in the new modular SMT solver: `modulariT`.

Chapter 1

Background Knowledge

1.1 Logic Notations

In the following thesis, we use some notations from logic. While they are mostly standard in the community, we still define them here for the sake of clarity and self-containment. We also define particular sets that are used to lighten the notations in this document. In the following document, we assume some basic understanding of propositional and first-order logic. We also assume the reader has some knowledge of programming, data structures and graph theory.

Ordered sets. Some sets used in this work are specified to be *ordered* when they are defined (see τ, ω... in Sect. 2.3). An ordered set $\mathcal{S}$ is a finite set that supports standard operations, but also the $\textsc{Pop}(\mathcal{S})$ (resp. $\textsc{Top}(\mathcal{S})$) and $\textsc{Dequeue}(\mathcal{S})$ (resp. $\textsc{First}(\mathcal{S})$) operations that respectively remove (resp. read) the last and first element added to the set. In an ordered set $\mathcal{S} = \{s_1, s_2\}$, we assume that s_1 is before s_2. These sets are constructed using $\cup$ as the concatenation symbol. For example, $\{s_1, s_2\} \cup \{s_3, s_4\} = \{s_1, s_2, s_3, s_4\}$ and $\{s_3, s_4\} \cup \{s_1, s_2\} = \{s_3, s_4, s_1, s_2\}$. The set subtraction symbol removes elements of the set, but keeps the relative order of the remaining elements. For example, $\{s_1, s_2, s_3, s_4\} \setminus \{s_2, s_3\} = \{s_1, s_4\}$.

Conjunctive and disjunctive sets. A conjunctive (resp. disjunctive) set is a mathematical object used to represent formulas. In this document, we refer to a conjunctive (resp. disjunctive) set of formulas to represent a formula that is a logical conjunction (resp. disjunction) of the formulas in the set. For example, the conjunctive set $\{a, b\}$ represents the formula $a \wedge b$. The disjunctive set $\{a, b\}$ represents the formula $a \vee b$. This notation is used to simplify the operations on formulas. The union of conjunctive (resp. disjunctive) sets is another conjunctive (resp. disjunctive) set. Conjunctive and disjunctive sets cannot be combined with set operations. However, they can be combined with the logical operators $\wedge$ and $\vee$. For example, let $\mathcal{S}^d = \{a, b\}$ be a disjunctive set and $\mathcal{S}^c = \{c, d\}$ be a conjunctive set, then $\mathcal{S}^d \wedge \mathcal{S}^c$ gives a disjunctive set of conjunctive sets $\{\{a, c, d\}, \{b, c, d\}\}$ or conjunctive set of disjunctive sets $\{\{a, b\}, \{c\}, \{d\}\}$ equivalent to the formula $(a \vee b) \wedge c \wedge d$. The semantic of formulas is described in the following paragraphs.

Propositional logic. In the context of propositional logic, we use the standard propositional vocabulary. In particular, we define a *variable* v as a Boolean variable, i.e., a variable that can take the value `true` ($\top$) or `false` ($\bot$). For SAT solving, a variable can also be `undefined`, meaning that it has not been assigned a value yet. A *partial assignment* π is a set of variables that have been assigned a value. We consider (partial) assignments as conjunctive sets of literals. A *literal* is a variable v or its negation $\neg v$. A literal ℓ can be *satisfied*, *falsified* or *undefined* by an assignment π. A literal ℓ is *satisfied* by π iff $\ell \in \pi$. Conversely, ℓ is *falsified* by π iff $\neg\ell \in \pi$. Finally, ℓ is *undefined* by π iff it is neither satisfied nor falsified. Assigning a literal ℓ to `true` means that $\ell \in \pi$, if $\ell = \neg v$, then assigning ℓ to `true` is equivalent to assigning v to `false`.

A *clause* is a disjunction of literals (e.g., $\ell_1 \vee \cdots \vee \ell_n$). It is written with a potentially subscripted or primed capital letter (usually but not limited to C). A clause is treated as a disjunctive set of literals. A clause C is *satisfied* by an assignment π if at least one of its literals is satisfied by π. A clause C is *falsified* if all its literals are falsified by π. A clause C is made *unit* by an assignment π if it is not satisfied by π and C has only one undefined literal. A clause C is made *unisat* by an assignment π if only one literal is satisfied by π and all the other literals are falsified.

A propositional *formula* is a logical expression that combines literals and logical connectives. More formally,

- a variable is a propositional formula;
- if ϕ is a propositional formula, then $\neg\phi$ is also a propositional formula;
- if ϕ and ψ are propositional formulas, then $(\phi \wedge \psi)$ and $(\phi \vee \psi)$ are also propositional formulas.

An interpretation $\mathcal{I}$ of a formula ϕ is a total mapping of all variables of ϕ to truth values. It is a set of mappings of the sort $v \leftarrow \top$ or $v \leftarrow \bot$ where v is a variable of ϕ. A model of a formula ϕ is an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \phi$. That is, $\mathcal{I}$ makes ϕ `true`. The symbol $\models$ reads *models* and is used to denote the entailment relation. $\phi \models \psi$ means that all models of ϕ are also models of ψ. $\mathcal{I} \models \phi$ means that $\mathcal{I}$ is a model of ϕ. Two formulas ϕ and ψ are logically equivalent if they have exactly the same models. That is $\phi \models \psi$ and $\psi \models \phi$. The truthfulness of a formula ϕ is defined by the semantic of the logical connectives detailed below.

A variable v is made `true` by an interpretation $\mathcal{I}$ if $v \leftarrow \top \in \mathcal{I}$. Conversely, v is made `false` by $\mathcal{I}$ if $v \leftarrow \bot \in \mathcal{I}$. The logical negation of a formula ϕ is written $\neg\phi$. An interpretation $\mathcal{I}$ makes the negation of a formula ϕ `true` (resp. `false`) iff it makes ϕ `false` (resp. `true`). We can easily see that $\neg\neg\phi$ is logically equivalent to ϕ. The logical disjunction of two formulas ϕ and ψ is written $(\phi \vee \psi)$. An interpretation makes the disjunction of two formulas ϕ and ψ `true` iff the interpretation makes at least one of ϕ and ψ `true`. The logical conjunction of two formulas ϕ and ψ is written $(\phi \wedge \psi)$. An interpretation makes the conjunction of two formulas ϕ and ψ `true` iff it makes both ϕ and ψ `true`.

An assignment π can also be used as an interpretation of the formula ϕ if the set of variables of π is identical to the set of variables in ϕ and $\pi \wedge \phi \not\models \bot$. An interpretation can be built from a full assignment π by assigning all the variables v of π to `true` if $v \in \pi$ and to false if $\neg v \in \pi$. More formally, $\mathcal{I} = \{v \leftarrow \top \mid v \in \pi\} \cup \{v \leftarrow \bot \mid \neg v \in \pi\}$. In the following, when it is stated that an assignment is a model of a formula, it is implied that the interpretation built from the assignment is a model of the formula.

Remark 1.1.1. The formula definition given above is not minimal. Every formula can be generated with the negation and disjunction operators only. The conjunction operator is syntactic sugar to simplify the writing of formulas. Indeed, $\phi \wedge \psi$ is logically equivalent to $\neg(\neg\phi \vee \neg\psi)$.

We say that a formula ϕ is satisfiable if it has at least one model. That is, there exists an interpretation $\mathcal{I}$ such that $\mathcal{I} \models \phi$. Conversely, we say that a formula ϕ is unsatisfiable if it has no model. It is also written $\phi \models \bot$. All models of ϕ are also models of $\bot$, but $\bot$ does not have any model, therefore ϕ is unsatisfiable. A formula is valid if it is `true` for all interpretations ($\top \models \phi$). The duality property states that a formula ϕ is valid if and only if $\neg\phi$ is unsatisfiable. Two formulas are equisatisfiable if both are satisfiable, or both are unsatisfiable.

First-order logic. In the second part of this thesis, we need a more expressive logic to discuss SMT. In particular, we will focus on First-Order Logic (FOL). We use the notations from [1] and refer to an infinite set of sorts S, and an infinite set of sorted variables X. A formula Φ is a first-order formula with a signature Σ composed of a set of sort symbols $\Sigma^S \subseteq S$, a set of predicate symbols Σ^P, a set of function symbols Σ^F and a total mapping from Σ^P to $\left(\Sigma^S\right)^*$ and from Σ^F to $\left(\Sigma^S\right)^* \times \Sigma^S$. The symbol $*$ is interpreted as a Kleene star, i.e., the set of all finite sequences of elements of Σ^S. These mappings correspond to the arity of predicate and function symbols. A function f with arity n has a mapping $\sigma_1 \ldots \sigma_n \mapsto \sigma$ (where $\sigma_1, \ldots, \sigma_n, \sigma \in \Sigma^S$), that is, a function that takes n arguments of sorts $\sigma_1, \ldots, \sigma_n$ and returns a value of sort σ. A function symbol with arity 0 is a constant symbol. Predicates behave similarly but do not have a return sort. A predicate P with arity n has a mapping $\sigma_1 \ldots \sigma_n$ and selects a subset of the domain of $\sigma_1 \times \cdots \times \sigma_n$. A predicate symbol with arity 0 is a propositional variable. In general, we use the symbols P, Q for predicates and f, g for functions. Constants (nullary functions) are differentiated from non-nullary functions by the use of the symbols a, b, c.

We also distinguish between *interpreted* and *uninterpreted* predicates and functions. An interpreted predicate (resp. function) is a predicate (resp. function) that has a meaning in a theory T. For example, the equality predicate $=$ is interpreted and takes the subset of pairs $(e, e') \in \sigma \times \sigma$ for which the e and e' are equal. The function $+$ is interpreted in the theory of arithmetic. An uninterpreted predicate (resp. function) is a predicate (resp. function) that is free of any theory. For example, the predicate P and function f are uninterpreted.

A term is a variable or a function symbol applied to a sequence of terms (of the correct sort and the correct number of terms). For example, $f(a), x, g(f(c), x)$ are terms. An atom is a predicate symbol applied to a sequence of terms. A literal is an atom or its negation. A formula is a literal, a disjunction or conjunction of formulas, the negation of a formula, or a quantified formula. If Φ and Ψ are FOL formulas, then $\neg\Phi, \Phi \vee \Psi, \Phi \wedge \Psi, \forall \boldsymbol{x} \Phi$ and $\exists \boldsymbol{x} \Phi$ are all formulas.

A first-order interpretation $\mathcal{I}$ is a total mapping from (1) the set of sorts $\sigma \in \Sigma^S$ to non-empty sets $\mathcal{I}(\sigma) \subseteq \mathcal{D}(\sigma)$ (where $\mathcal{D}(\sigma)$ is the domain of σ), (2) the set of variables $x \in \Sigma^X$ to elements of their sort σ, $\mathcal{I}(x) \in \mathcal{I}(\sigma)$, (3) the set of symbols $f \in \Sigma^F$ to a set of total functions $\mathcal{I}(f) : \sigma_1 \times \cdots \times \sigma_n \mapsto \sigma$ and (4) from the set of symbols $p \in \Sigma^P$ to subsets of the domain of p, $\mathcal{I}(p) \subseteq \sigma_1 \times \cdots \times \sigma_n$. Interpreted predicates and functions have predefined mappings and are fixed in the interpretation (e.g., the equality predicate cannot be interpreted as a different set as the one provisioned by the theory of equality). An interpretation $\mathcal{I}$ is a model of a formula Φ if $\mathcal{I} \models \Phi$. That is, $\mathcal{I}$ makes Φ `true`. The rules of logical connectives are identical to propositional logic. An interpretation $\mathcal{I}$ makes an atom $P(t_1, \ldots, t_n)$ `true` if $(\mathcal{I}(t_1), \ldots, \mathcal{I}(t_n)) \in \mathcal{I}(P)$. A term $f(t_1, \ldots, t_n)$ is interpreted depth first as the return value of the function f with arguments $t_1, \ldots, t_n$. The existential quantifier $\exists$ is interpreted as: $\mathcal{I} \models \exists(x \in \sigma).\ \Phi$ iff $\mathcal{I} \models x = x' \land \Phi$ for some fixed $x' \in \sigma$.

Remark 1.1.2. Similar to the duality of conjunctions and disjunctions, the universal quantifier can be expressed with an existential quantifier. Indeed, $\forall x \Phi$ is equivalent to $\neg \exists x \neg \Phi$.

Syntactic Sugar. To make formulas more readable, we will ignore some parenthesis inside formulas. In the definition above, a conjunction of three variables a, b, c is built as $(a \land (b \land c))$. Since conjunction and disjunction are associative, this is of course equivalent to $((a \land b) \land c)$. Furthermore, the outside parenthesis are meaningless and also removed. We will write $a \land b \land c$ instead. The same applies to disjunctions. Furthermore, when considering quantifiers, we will use the dot symbol . to indicate the start of a parenthesis that ends at the end of the current block. For example, $\forall x \in X.\ \phi(x)$ should be interpreted as $\forall x (x \in X \Rightarrow \phi(x))$. If the quantifier is restricted to a local block, then the dot symbol is also restricted to that block. For example $\forall x \in X.\ \phi(x) \land \exists y \in Y.\ \psi(x, y)$ is interpreted as $\forall x\ (x \in X \Rightarrow (\phi(x) \land \exists y\ (y \in Y \land \psi(x, y))))$.

1.2 Resolution Rule

The resolution rule is a well-known inference rule in logic. It is defined by [15] as a function that takes two clauses C_1 and C_2 and one literal ℓ as argument, and, if $\ell \in C_1$ and $\neg\ell \in C_2$, returns a new clause that is a logical consequence of C_1 and C_2. This new clause is $C' = C_1 \setminus \{\ell\} \cup C_2 \setminus \{\neg\ell\}$. The resolution rule is complete for propositional logic [13], meaning that if a formula ϕ is unsatisfiable, then there exists a resolution refutation of ϕ. That is, there is set of resolution steps from ϕ that lead to the empty clause (provided that ϕ is in CNF, defined in Sect. 2.1). And if a formula ϕ is satisfiable, then a saturated set of clauses can be found in bounded time. That is, a set of clauses that cannot be uniquely resolved further, and that does not contain the empty clause.

$$\frac{C_1 = \ell \lor D_1 \qquad C_2 = \neg\ell \lor D_2}{C' = D_1 \lor D_2}\ (\ell)$$

The resolution rule is sound. Indeed, two possible assignments of ℓ exist in any interpretation. Either ℓ is assigned `false`, then D_1 must be satisfied to satisfy C_1. Or ℓ is assigned true, then D_2 must be satisfied to satisfy C_2. In both cases, $D_1 \lor D_2$ must be satisfied to satisfy $C_1 \land C_2$. Therefore, $C_1 \land C_2 \models C'$.

The resolution rule will become relevant in Sect. 2.3.3 for conflict analysis.

Chapter 2

SAT Solving

2.1 Boolean Satisfiability Problem

Boolean Satisfiability (SAT) is an NP-complete problem that consists in finding a satisfying assignment of Boolean (also called propositional) variables given a Boolean formula. It is often used in practice, because it is a generalization of many other problems. For several applications, using state-of-the-art SAT solvers on an appropriate encoding appeared more effective than using traditional specialized algorithms. Indeed, a lot of clever techniques have been developed to solve SAT instances, bringing a lot of engineering effort that cannot always be matched by specialized algorithms. For example, [25, 6] shows that SAT solvers can efficiently solve the subsumption and subsumption resolution problem in first-order saturation algorithms. SMT solvers use SAT solving as a backbone to solve more complex and expressive problems [1, 4, 8].

Example 2.1.1. The formula $(a \lor b \lor \neg c) \land (\neg a \lor \neg b \lor c) \land (\neg a \lor b \lor c)$ is satisfiable with the assignment $\{\neg a, b, c\}$ since every clause is satisfied by either $\neg a$ or b. SAT solvers are able to find such an assignment, not only on toy examples, but on propositional formulas with thousands of variables and clauses.

Conjunctive Normal Form. For practical reasons, it is often desirable to have formulas in a standard form. For SAT solvers, this preferred form is the Conjunctive Normal Form (CNF). A formula is in CNF if it is a conjunction of clauses, where a clause is a disjunction of literals. A literal is a variable or its negation. (The Ex. 2.1.1 is already in CNF.)

Theorem 2.1.1 (Tseitin [28])**.** *Any Boolean formula can be converted into an equisatisfiable formula in CNF in linear time.*

The conversion uses the following equivalences to push the negations inwards and distribute the disjunctions over the conjunctions. The two first equivalences are the definition of the logical implication and equivalence. The next two are De Morgan's laws. The last two equivalences are distributivity laws.

Formula	**Towards CNF**
$\phi \Rightarrow \psi$	$\neg\phi \vee \psi$
$\phi \Leftrightarrow \psi$	$(\phi \vee \neg\psi) \wedge (\neg\phi \vee \psi)$
$\neg(\phi \wedge \psi)$	$\neg\phi \vee \neg\psi$
$\neg(\phi \vee \psi)$	$\neg\phi \wedge \neg\psi$
$\ell \vee (\phi \wedge \psi)$	$(\ell \vee \phi) \wedge (\ell \vee \psi)$
$\ell \wedge (\phi \vee \psi)$	$(\ell \wedge \phi) \vee (\ell \wedge \psi)$

Table 2.1: Logical equivalences to transform a formula into CNF. The formulas ϕ and ψ are arbitrary formulas, and ℓ is a literal.

A formula of the type $\phi_1 \vee \cdots \vee \phi_n$ cannot be transformed into CNF with the equivalences of Table 2.1. Therefore, we need to use the Tseitin transformation. New variables $t_1, \ldots, t_n$ are introduced for each subformula ϕ_i. The subformula ϕ_i is replaced by the variable t_i and the clause $t_i \Leftrightarrow \phi_i$ is added to the formula. On the above example, this method gives the equisatisfiable formula:

$$(t_1 \vee \ldots \vee t_n) \wedge (t_1 \Leftrightarrow \phi_1) \wedge \ldots \wedge (t_n \Leftrightarrow \phi_n)$$

The components $(t_i \Leftrightarrow \phi_i)$ can themselves be transformed into CNF using the Table 2.1 second equivalence. This yields the formula $(t_i \vee \neg\phi_i) \wedge (\neg t_i \vee \phi_i)$. The procedure repeats on the subformulas ϕ_i and $\neg\phi_i$ until the whole formula is in CNF.

In the rest of this thesis, we assume without loss of generality that the formulas are in CNF.

2.2 DPLL Algorithm

The Davis-Putnam-Logemann-Loveland (DPLL) algorithm [7] is a procedure that decides the satisfiability of a formula in CNF by recursively searching for a satisfying assignment. The algorithm is composed of two main parts: the Boolean constraint propagation (BCP) and the search.

A partial assignment is kept during the search. It is a *conjunctive set* (as defined in Sect. 1.1) π of literals that are assumed to be `true`. The partial assignment is assumed to be consistent, i.e., it does not contain both a variable and its negation. Let $\mathcal{V}$ be the set of variables in a propositional formula φ. If every variable in $\mathcal{V}$ is assigned to `true` or `false` and $\pi \models \varphi$, then π is a model of φ and a solution was found. The partial assignment can be understood as a formula that consists of clauses with a unique literal. Recall that the formula φ is assumed to be in CNF; therefore φ can be seen as a conjunctive set of clauses. It is also assumed that trivial clauses are removed from φ, i.e., φ contains no clause with a variable and its negation. Clauses also do not contain twice the same literal. These properties of φ are assumed to hold for the rest of this document.

2.2.1 Boolean Constraint Propagation

The BCP algorithm is a procedure that propagates the consequences of the current assignment π. BCP searches for unit clauses under assumption π, i.e., clauses with no satisfied literal and exactly one unassigned literal. If such a clause is found, BCP first checks that assigning the literal to `true` does not falsify any clause in φ. If it does, then a conflict is found and BCP returns $\bot$. Otherwise, the literal is assigned to `true`. This process is repeated until no more unit clauses are found. Literals that are assigned by BCP are called *implied* literals. If no conflict is detected, the procedure returns the partial assignment π.

Example 2.2.1. Consider the formula $\varphi = \{C_1, C_2, C_3, C_4\}$ with

$$\begin{array}{ll} C_1 = \neg b \lor c \lor d & C_2 = \neg a \lor b \lor \neg c \\ C_3 = \neg b \lor c & C_4 = \neg b \lor \neg c \end{array}$$

and the partial assignment $\pi = \{\neg b, c\}$. The clause C_2 is unit, and only $\neg a$ satisfies it. $\pi \cup \{\neg a\} = \{\neg b, c, \neg a\}$ does not falsify any clause; therefore, $\neg a$ can be added to π.

2.2.2 Search

First, BCP is performed on the formula. If a conflict is detected, then the search returns $\bot$ and backtracks. If all variables are assigned, then the search returns the model. Otherwise, it chooses one free variable and assigns it arbitrarily to `true` or `false`. This variable is called a *decision*. The search then continues recursively. If the first recursive call returns $\bot$, then the chosen variable is now assigned to `false` and the search is tried again. If the other call also returns $\bot$, then the search backtracks by returning $\bot$. When one of the calls returns a model, the search returns the model. Intuitively, $\textsc{Search}(\varphi, \pi)$ returns a model of φ under assumption π if it exists, and $\bot$ if $\varphi \land \pi \models \bot$. Starting with $\pi = \emptyset$ answers whether φ is satisfiable.

An overview of the algorithm is given in Alg. 1. It is, of course, possible to improve it by removing the recursive calls and implementing the search with a stack of literals. Alg. 1

Algorithm 1 DPLL search algorithm overview

```
procedure BCP($\varphi, \pi$)
    for unit clause $C$ in $\varphi$ by $\pi$ do
        $\ell \leftarrow C \setminus \pi$                                   ▷ The unassigned literal
        if $\exists C' \in \varphi.\ C' \wedge (\pi \cup \{\ell\}) \models \bot$ then    ▷ Conflict detected if $\ell$ is added to $\pi$
            return $\bot$
        $\pi \leftarrow \pi \cup \{\ell\}$
    return $\pi$
procedure SEARCH($\varphi, \pi$)
    $\pi \leftarrow$ BCP($\varphi, \pi$)
    if $\pi = \bot$ then
        return $\bot$
    if $\forall v \in \mathcal{V}.\ v \in \pi \vee \neg v \in \pi$ then           ▷ All variables are assigned
        return $\pi$
    $v \leftarrow$ DECIDE($\varphi, \pi$)                               ▷ Choose a free variable
    $\pi' \leftarrow$ SEARCH($\varphi, \pi \cup \{v\}$)
    if $\pi' \neq \bot$ then
        return $\pi'$
    return SEARCH($\varphi, \pi \cup \{\neg v\}$)
procedure DPLL($\varphi$)
    return SEARCH($\varphi, \emptyset$)
```

provides only a high-level overview of DPLL. Decisions are remembered implicitly in the recursive calls. Using a stack, the decision would be discriminated from implied literals such that backtracking can stop at the right point. A more sophisticated algorithm for BCP is also discussed in Sect. 2.3.

Example 2.2.2. Consider the same formula as in Ex. 2.2.1.

$$\begin{array}{ll} C_1 = \neg b \vee c \vee d & C_2 = \neg a \vee b \vee \neg c \\ C_3 = \neg b \vee c & C_4 = \neg b \vee \neg c \end{array}$$

One can see the search as a tree exploration. Nodes are partial assignments, and edges are new literal assignments. Fig. 2.2.1 displays a search tree that DPLL could follow. For the sake of readability, only the newly added literal is displayed in the node. The partial assignment is the conjunction of all the literals on the path from the root to the node. Decisions with the positive polarity are displayed on the left branches and the right branches are the negated choices. Vertical branches are propagations through BCP. Dots are branches that were not explored because the search terminated. DPLL progressively builds a model in a depth-first manner until a conflict is found, in which case it goes back to the last decision and resumes the search with the other polarity of the decision. Choosing b leads to a conflict between C_3 and C_4. The search backtracks and changes the polarity of the decision to $\neg b$. No unit clause is detected and c is decided. BCP safely propagates $\neg a$ and d is decided. The search terminates because every variable has been assigned. The assignment is a model and is returned.

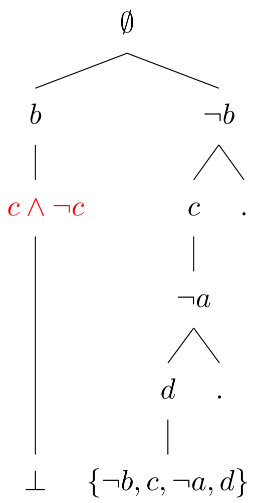


Figure 2.2.1: Tree exploration of the DPLL algorithm on Ex. 2.2.1

2.2.3 Soundness, Completeness and Complexity

DPLL is an effective procedure to solve the SAT problem. It is sound and complete. That is, answers provided by DPLL are always correct, and it will terminate in a finite time. If DPLL returns a model, then the search found a complete assignment that does not falsify any clause. Since all variables are assigned, it must be that every clause is satisfied, and the answer from DPLL is indeed a model. Otherwise, DPLL has systematically searched every partial assignment and concluded that each of them falsifies at least one clause. Therefore, the formula is unsatisfiable, and the procedure is sound. The procedure also terminates since there is a finite number of assignments to explore. The search never explores twice the same partial assignment since during every call of Search, at least one decision changes polarity, and literals cannot be assigned multiple times in the same partial assignment.

2.2.4 Discussion of the DPLL Algorithm

While the DPLL algorithm is sound and complete, it is sometimes not efficient. Indeed, if a subset of variables of $\mathcal{V}$ is involved in a conflict, the algorithm might explore the same conflict structure several times.

Example 2.2.3. Let φ be the conjunctive set of the clauses $C_1, \ldots, C_6$:

$$\begin{array}{lll} C_1 = a \vee b & C_2 = \neg b \vee c & C_3 = \neg a \vee \neg c \\ C_4 = \neg a \vee b \vee c & C_5 = a \vee \neg b \vee \neg c & C_6 = d \vee e \end{array}$$

From the Fig. 2.2.2, it is obvious that the conflict on variables a, b, c is explored multiple times. Indeed, if the order of variable decisions is not good, then the algorithm is highly inefficient. Some heuristics exist to choose the next variable to assign. It is, however, beyond the topic of this thesis to discuss them here. See [17, 18] for more information about decision heuristics.

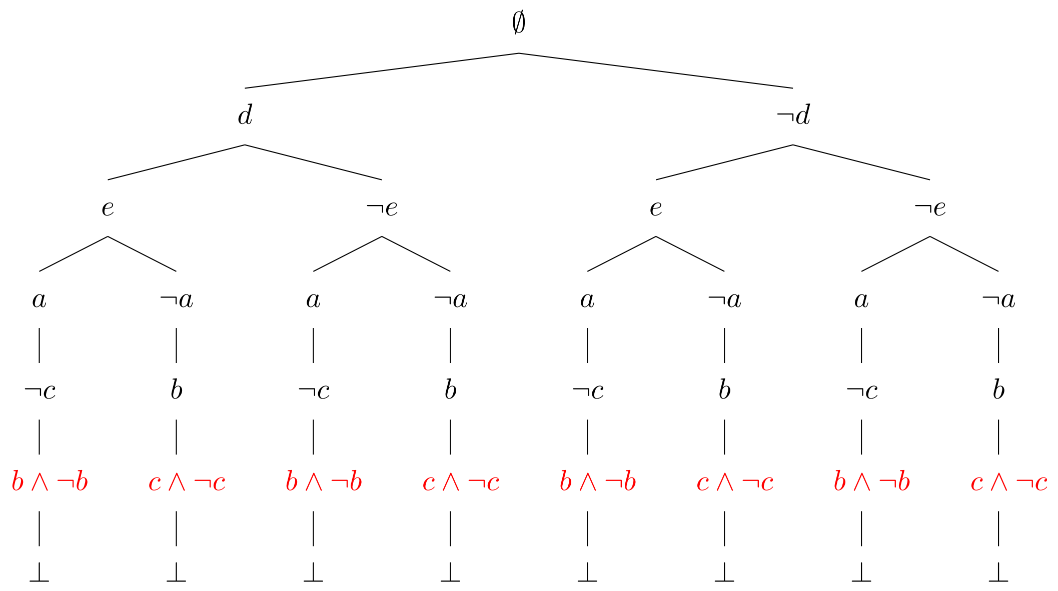


Figure 2.2.2: Tree exploration of the DPLL algorithm on Ex. 2.2.3. The same conflicting structure is repeated multiple times.

2.3 CDCL Algorithm

Motivated by the weakness of the DPLL algorithm to learn conflict patterns, Conflict Driven Clause Learning (CDCL) was introduced. The CDCL algorithm seeks to learn clauses to prune the search space and prevent conflicts from re-emerging. It is the basis of most modern SAT solvers. As described in [11], a CDCL solver is composed of four main parts: Boolean constraint propagation (BCP), conflict analysis and learning, backtracking, and decision heuristics.

In CDCL the order of assignment of literals is important: the order is no longer implicitly recorded in recursive calls, as it was in DPLL. The partial assignment is usually split into two disjoint *conjunctive ordered sets* (defined in Sect. 1.1) (1) the trail τ containing fully propagated literals (2) the propagation queue ω containing literals waiting to be propagated. A propagated literal has already been checked for implications and conflicts by BCP. Each literal in ω is implied by some clause in φ and τ. That is, $\varphi \wedge \tau \models \omega$. By abuse of notation, and to make algorithms more readable, decisions are also pushed on ω, but they are not consequences of $\varphi \wedge \tau$. To be more precise, $\varphi \wedge \tau \models \omega$ when ω does not contain any decision. The trail and the propagation queue compose the partial assignment $\pi = \tau \cup \omega$.

To ensure that the τ is usable, and that BCP did not miss any conflict, we enforce Inv. *trail sanity*.

Invariant 2.3.1 (Trail sanity). The trail τ does not falsify a clause in φ.

Stating that $\ell \in \tau$ means that the literal ℓ is assigned to `true` and is satisfied by τ. Similarly, $\neg\ell \in \tau$ means that the literal ℓ is falsified. Unassigned literals are neither satisfied nor falsified. The notation $\tau \models \varphi$ means that the formula φ is satisfied by the trail τ.

Remark 2.3.1. In practice, both the trail τ and the propagation queue ω often share the same memory space. Since the propagation process dequeues the literals from ω and pushes them on τ, only a single stack is needed, with a pointer indicating where τ ends and ω begins. This detail becomes relevant in Sect. 4.2.4.

The trail is divided in decision levels. The decision level of a literal ℓ is written $\delta(\ell)$ and is the number of decisions that were made before ℓ can be assigned. The decision level of a trail τ (resp. clause C) is written $\delta(\tau)$ (resp. $\delta(C)$) and is the maximum decision level of the literals in τ (resp. C). The set of decisions in the trail is written τ^d. A decision level of 0 indicates that the literal is a direct consequence of the formula φ. Unassigned literals have a decision level of ∞. We assume that the decision level of a literal is independent of its polarity. $\delta(\ell) = \delta(\neg\ell)$ regardless of whether $\ell \in \tau \cup \omega$ or $\neg\ell \in \tau \cup \omega$. Literals in the assignment which are not decisions ($\tau \cup \omega \setminus \tau^d$) are called *implied* literals. Literals in the trail τ are *propagated* literals.

Remark 2.3.2. A more precise definition of the decision level of a literal $\delta(\ell)$ will be necessary in Chap. Chronological Backtracking. The more accurate definition of $\delta(\ell)$ is given by the implication graph of the trail which is introduced later. However, this detail is not necessary as long as the solver uses non-chronological backtracking. In this chapter, we assume that this is the case, and this simple definition is sufficient.

Example 2.3.1. Assume that during an execution of the CDCL algorithm, the decision $\neg\boldsymbol{a}$ is made, then BCP implies and propagates b and c, finally, the solver decides $\neg\boldsymbol{d}$. The trail is $\tau = \{\neg\boldsymbol{a}, b, c, \neg\boldsymbol{d}\}$. The decision level of $\neg\boldsymbol{a}, b$ and c is 1 and the decision level of $\neg\boldsymbol{d}$ is 2.

2.3.1 CDCL Algorithm Overview

Alg. 2 gives an overview of the CDCL algorithm. It is composed of four main components.

BCP. Boolean Constraint Propagation (BCP) is the same as in the DPLL algorithm. BCP implies and propagates the direct consequences of the current assignment on the formula. It searches for unit clauses, i.e., clauses with no satisfied literal and one unassigned literal, if such a clause is found, the unassigned literal is assigned to `true` and pushed on ω. When a literal is done being propagated, it is pushed on the trail τ and removed from ω. This process is repeated until the propagation queue ω is empty. If a conflict is found, BCP now returns the conflicting clause instead of $\bot$, otherwise, it returns $\top$.

Conflict analysis (learning). When BCP finds a conflict (i.e., $\exists C \in \varphi.\ C \wedge \tau \models \bot$), conflict analysis searches for the pattern of the conflict. It finds a clause with exactly one literal at the highest decision level that prevents the conflict from happening again. This clause is sometimes called the conflict clause, but in this paper it is called the *learned* clause to avoid confusion with the clause that caused the conflict.

The solver backtracks to any decision level before the conflict. The learned clause is then added to the clause set φ. If the solver backtracks between the highest and second-highest level of the learned clause, the learned clause becomes unit, and BCP propagates the highest literal of the learned clause. As shall be discussed later, the choice of backtracking level is important. In non-chronological backtracking (NCB), which is the focus of this chapter, the backtracking level is always the second-highest decision level of the learned clause. In the next chapters, we present another backtracking method and its impact of the overall algorithm.

Backtracking. Backtracking is the procedure that undoes the assignments that were made after a determined level. It is done by popping literals from the trail τ until the desired level is reached. It also clears the propagation queue ω.

Decision. The decision procedure consists in choosing a free variable to which a value is assigned. This newly assigned variable is not a consequence of $\varphi \wedge \tau$; therefore the literals assigned by the DECIDE() procedure are called *decisions*.

Algorithm 2 CDCL algorithm overview

```
1: $\tau \leftarrow \emptyset$                          ▷ The trail
2: $\varphi \leftarrow \emptyset$                          ▷ The set of clauses
3: $\omega \leftarrow \emptyset$                          ▷ The propagation queue
4: procedure CDCL($\varphi'$)
5:     $\varphi \leftarrow \varphi'$                       ▷ Initialize the set of clauses
6:     while $\top$ do
7:         $C \leftarrow$ BCP()                ▷ $C$ is the conflicting clause
8:         if $C \neq \top$ then               ▷ Conflict detected
9:             if $\delta(C) = 0$ then         ▷ Conflict without any decision
10:                return $\bot$
11:            $C' \leftarrow$ ANALYZE($C$)
12:            $\ell \leftarrow$ the highest literal in $C'$
13:            BACKTRACK($\delta(C' \setminus \{\ell\})$)   ▷ Backtrack to the second-highest decision
                                                             level in $C'$
14:            $\omega \leftarrow \omega \cup \{\ell\}$
15:            $\varphi \leftarrow \varphi \cup \{C'\}$      ▷ $C'$ is no longer conflicting after backtrack-
                                                             ing
16:            continue
17:        $\ell \leftarrow$ DECIDE()
18:        if $\ell = \bot$ then               ▷ No more literals to propagate
19:            return $\tau$
20:        $\omega \leftarrow \omega \cup \{\ell\}$         ▷ The propagation queue should be empty
                                                             at this point
```

2.3.2 Boolean Constraint Propagation

The Boolean constraint propagation is the same as in the DPLL algorithm. However, only a very high-level view was given in the previous chapter. Watched literals is a commonly used method introduced in [20] to reduce the number of accesses to clauses when propagating literals. The idea is to only keep track of two literals in each clause called the watched literals. During the propagation of a literal ℓ, all clauses watched by the literal $\neg\ell$ are checked for unit propagation and conflict. The list keeping track of all clauses watched by a literal is called the watch list.

Without loss of generality, we assume that the watched literals are the first two literals in a clause. That is, for a clause C, c_1 and c_2 are its watched literals. Formally, a clause C is watched by two literals c_1 and c_2 if $C = c_1 \vee c_2 \vee \dots$ and complies to the following invariant:

Invariant 2.3.2 (Watched literals)*.* Let a clause $C \in \varphi$ watched by c_1 and c_2. If c_1 or c_2 is falsified by τ, then the other literal is satisfied by $\tau \cup \omega$.

Invariant 2.3.3 (Watch lists)*.* Each clause C in φ such that $|C| \geq 2$ is watched by two literals. Their corresponding watch lists contain the clause.

Because of Inv. *watched literals* and Inv. *watch lists*, BCP only needs to check the watch list of the negation of the propagated literal as described in Alg. 3. Essentially, when BCP attempts to falsify one of the watched literals of C by propagating ℓ, one of three things can happen. Without loss of generality, let $c_1 = \neg\ell$ and c_2 the other watched literal.

1. $\neg c_2 \in \tau$. The clause is conflicting with $\tau \cup \{\ell\}$ and ℓ cannot be propagated. The conflict analysis is triggered.
2. $c_2 \in \tau$. The clause is satisfied and is left untouched.
3. c_2 is unassigned. BCP then searches for a new literal in C to replace c_1 as watched literal. If no satisfied or unassigned literal is found, the clause is unit and BCP adds c_2 to ω to restore Inv. *watched literals*.

Properties of Alg. 3. The condition on line 9 of the algorithm is sufficient to state that the clause is conflicting. Indeed, $\neg c_2$ must have been propagated before ℓ. Therefore, if there had been another literal ℓ' in the clause C that was not assigned false, c_2 would have been replaced at step 16 when propagating $\neg c_2$

This observation can lead to some improvement on the algorithm. Indeed, since we just showed that no replacement literal was found during the propagation of ℓ', then C must have been detected as a unit clause at step 20. Therefore, $\neg\ell$ must have been implied and enqueued in ω before ℓ was propagated to ensure that $\tau \cup \omega$ satisfy C. The conflict could have been detected at that moment. In Alg. 3, it is possible to have both $\ell \in \omega$ and $\neg\ell \in \omega$, which we know will lead to a conflict later. We can enforce a new invariant that prevents that from happening and saves time.

Invariant 2.3.4 (Assignment coherence)*.* Let a literal ℓ. If $\ell \in \tau \cup \omega$, then $\neg\ell \notin \tau \cup \omega$.

Algorithm 3 BCP with watched literals

```
1:  procedure BCP()
2:      while $\omega \neq \emptyset$ do
3:          $\ell \leftarrow$ FIRST($\omega$)
4:          $c_1 \leftarrow \neg\ell$
5:          for $C$ in $\varphi$ watched by $c_1$ do
6:              $c_2 \leftarrow$ the other watched literal of $C$
7:              if $c_2 \in \tau$ then
8:                  continue                    ▷ Clause is satisfied
9:              if $\neg c_2 \in \tau$ then
10:                 return $C$                  ▷ Clause is falsified by $\tau \cup \{\ell\}$
11:             $r \leftarrow \bot$                ▷ $r$ will be the new watched literal
12:             for $\ell'$ in $C \setminus \{c_1, c_2\}$ do
13:                 if $\neg\ell' \notin \tau$ then      ▷ $\ell'$ is either unassigned or assigned to true
14:                     $r \leftarrow \ell'$            ▷ Found a new watched literal
15:                     break
16:             if $r \neq \bot$ then
17:                 $c_1$ stops watching $C$
18:                 $r$ starts watching $C$
19:                 continue
20:             $\omega \leftarrow \omega \cup \{c_2\}$     ▷ Because $c_1$ is falsified and Inv. *watched literals*, $c_2$ is must be satisfied by $\tau \cup \omega$
21:         $\omega \leftarrow \omega \setminus \{\ell\}$
22:         $\tau \leftarrow \tau \cup \{\ell\}$
23:     return $\top$
```

Modification of the algorithm. By now using the union of the trail and the propagation queue $\tau \cup \omega$, as shown in blue in Alg. 4, BCP is able to detect conflicts earlier. The only possible cost of this modification is that if $\neg c_2 \in \tau$, BCP has to check that there is no replacement literal for c_1 in C before concluding that there is a conflict. This was not necessary in Alg. 3 because BCP was sure that if c_2 was assigned `false`, then there was no replacement literal in C. However, as we shall see later (cfr. Rem 2.3.3), this is not possible. Therefore, this optimization is free.

Blockers. Another common optimization is the use of blockers. The idea is to keep track of one satisfied literal if it is found while searching for a replacement in Alg. 4. This literal is called the blocker. If a blocker is found, it is not necessary to change the watch list. This is shown in green in Alg. 4. With blockers, Inv. *watched literals* can be relaxed to Inv. *blocked watched literals* to preserve Inv. *trail sanity*.

Invariant 2.3.5 (Blocked watched literals). Let a clause $C \in \varphi$ watched by c_1 and c_2. If c_1 or c_2 is falsified by τ, then C is satisfied by $\tau \cup \omega$.

If a conflict is now detected, then both watched literals are falsified by ω. Indeed, if one of the watched literal had been falsified by τ, then the clause must be satisfied by $\tau \cup \omega$ because of Inv. *blocked watched literals*. In this new setting, Inv. *assignment coherence* is respected, and it is not possible to have both ℓ and $\neg\ell$ in $\tau \cup \omega$. Therefore, if both watched literals are falsified, then $\neg c_1 \in \omega$ and $\neg c_2 \in \omega$ or there is a blocker. However, blockers are satisfied contradicting the premise that the clause is conflicting. Therefore, $\neg c_1 \in \omega$ and $\neg c_2 \in \omega$.

Algorithm 4 BCP with watched literals - improved

```
procedure BCP()
    while $\omega \neq \emptyset$ do
        $\ell \leftarrow$ First($\omega$)
        $c_1 \leftarrow \neg\ell$
        for $C$ in $\varphi$ watched by $c_1$ do
            if $C$ has blocker satisfied by $\tau \cup \omega$ then
                continue
            $c_2 \leftarrow$ the other watched literal of $C$
            if $c_2 \in \tau \cup \omega$ then                ▷ $c_2$ is or will be assigned to true
                continue                                       ▷ Clause is satisfied
            $r \leftarrow \bot$
            for $\ell'$ in $C \setminus \{c_1, c_2\}$ do
                if $\neg\ell' \notin \tau \cup \omega$ then
                    $r \leftarrow \ell'$
                    break
            if $r \neq \bot$ then
                if $r \in \tau \cup \omega$ then
                    SetBlocker($C, r$)
                    continue                                   ▷ No need to change the watch lists
                $c_1$ stops watching $C$
                $r$ starts watching $C$
                continue
            if $\neg c_2 \in \tau \cup \omega$ then           ▷ Clause is conflicting with $\tau \cup \omega$
                return $C$
            $\omega \leftarrow \omega \cup \{c_2\}$            ▷ Clause is unit
        $\omega \leftarrow \omega \setminus \{\ell\}$
        $\tau \leftarrow \tau \cup \{\ell\}$
    return $\top$
```

2.3.3 Conflict Analysis

Implication Graphs

In the context of SAT solving, an implication graph is a graph whose vertices are literals and whose edges are implications between literals. The reason for the propagation of a literal ℓ is the unit clause C that required the literal to be implied because $C \setminus \{\ell\}$ is falsified by $\tau \cup \omega$. There is an implication between a literal ℓ' and ℓ if $\neg\ell' \in C$ where C is the reason for propagating ℓ. In an implication graph, the reason for a propagation ℓ can be built from the negation of parents of ℓ in the graph and ℓ. Fig. 2.3.1 shows such a graph. For example, the literal $\neg v_8$ was implied by $\neg v_7$ and v_6. The reason for $\neg v_8$ is the clause $C_5 = v_7 \vee \neg v_6 \vee \neg v_8$. Literals in vertices without parents are decisions. That is, vertices without inbound edges.

$$\begin{aligned}
C_1 &= v_1 \lor v_2\\
C_2 &= \neg v_2 \lor \neg v_3\\
C_3 &= \neg v_2 \lor \neg v_4 \lor \neg v_5\\
C_4 &= v_3 \lor v_5 \lor v_6\\
C_5 &= v_7 \lor \neg v_4 \lor \neg v_8\\
C_6 &= \neg v_4 \lor v_8 \lor v_9\\
C_7 &= v_{10} \lor \neg v_9 \lor v_{11}\\
C_8 &= \neg v_{11} \lor v_8 \lor \neg v_{12}\\
C_9 &= v_{12} \lor \neg v_{13}\\
C_{10} &= v_7 \lor v_{12} \lor v_{14}\\
C_{11} &= \neg v_6 \lor v_{12} \lor v_{15}\\
C_{12} &= v_{13} \lor \neg v_{14} \lor \neg v_{16}\\
C_{13} &= \neg v_{15} \lor \neg v_{14} \lor v_{16}
\end{aligned}$$

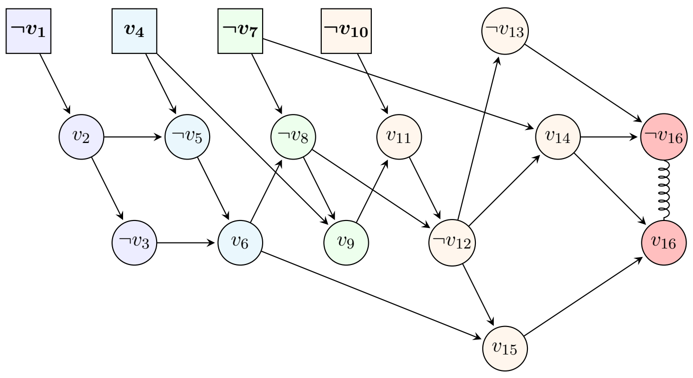


Figure 2.3.1: Example of an implication graph leading to a conflict (adapted from [13]). Literal levels are color-coded from blue (level 1) to orange (level 4). Red literals are conflicting.

Implication graphs are a useful tool for understanding the behavior of SAT solvers.

SAT solvers build such graphs during the search. The graph is initialized as a pair of empty sets $G = (V, E) = (\emptyset, \emptyset)$. When a literal ℓ is added to ω, the graph is updated as follows:

- $V \leftarrow V \cup \{\ell\}$
- If ℓ is added to ω because the clause $\ell \lor \ell'_1 \lor \cdots \lor \ell'_n$ is unit under assignment $\tau \cup \omega$, then $\neg\ell'_1, \ldots, \neg\ell'_n$ are already in V and $E \leftarrow E \cup \{(\neg\ell'_1, \ell), \ldots, (\neg\ell'_n, \ell)\}$.

When backtracking to level δ, the implication graph removes all vertices reachable from the decisions above level δ and the edges connected to them. For example, in Fig. 2.3.1, backtracking to level 2 would remove all the green, orange and red vertices and the edges connected to them.

In practice, the SAT solver only needs to remember the reason for the propagation of each literal. If no such reason exists, then the literal is a decision and does not have parents. The order in which literals are added to the implication graph is identical to the order of literals in the trail. By construction, it is easy to see that (1) the implication graph is acyclic and (2) the trail follows a topological order of the implication graph. The second property is used when performing conflict analysis.

Conflict Analysis Goals

During conflict analysis, the solver searches a clause C' (different from the conflicting clause C) that is a logical consequence of the formula φ and is unsatisfiable under the current assignment $\tau \cup \omega$. More precisely, the clause C' must be falsified before the conflict C arose. For example, if the last falsified literal of the C is located at position p in $\tau \cup \omega$, then the last literal of the new clause C' must be falsified before p. Otherwise, it will not prevent the solver from going through the same conflict again. This new clause C' is called the learned clause. C' is then added to the clause set φ once backtracking made it safe to do so (without violating Invs. *trail sanity* and *blocked watched literals*).

The objective of this operation is to learn the most general clause possible that prevents the solver from going through the same conflict again. The intuition is that the learned clause prunes parts of the search space that would always lead to a conflict. If the learned clause has a unique literal at the maximum decision level, then backtracking the conflict makes the learned clause unit. Therefore, the search will be able to resume without needing a decision. This is a property which we want to achieve.

Naive conflict analysis

The simplest conflict analysis is to take the negation of all the literals in the trail τ to generate the learned clause. While this sounds reasonable, it is not a good solution because it is highly inefficient. It only prunes the current branch of the search tree and does not have a unique literal at the maximum decision level.

A better solution can be found by observing that all literals that are not decisions are logical consequences of the decisions. If we denote τ^d the trail restricted to the decisions, we have $\varphi \land \tau^d \models \tau \cup \omega$. Therefore, the learned clause can be generated by taking the negation of all the literals in τ^d.

While this approach works and generates valid clauses, it does not learn the structure of the conflict. Indeed, the learned clause may contain a lot of literals that are irrelevant. This would be mostly equivalent to DPLL. The solver would backtrack one level at a time, just like DPLL, and would be forced to change the polarity of the last decision, just like DPLL. In short, the CDCL solver would behave like an over-complicated DPLL solver that adds clauses instead of simply changing the polarity of the last decision.

Example 2.3.2. Consider the formula φ comprising the clauses

$$\begin{array}{lll} C_1 = a \lor b & C_2 = a \lor \neg b & C_3 = \neg a \lor b \\ C_4 = \neg a \lor \neg b & C_5 = c \lor d \lor \cdots \lor z & \end{array}$$

If the solver decides to start assigning literals with $\tau = \{\boldsymbol{c}, \boldsymbol{d}, \dots, \boldsymbol{z}, \boldsymbol{a}, b\}$, the learned clause would contain all the literals in τ except b. However, the conflict only involves a and b. A lot of unnecessary literals are added to the learned clause.

First Unique Implication Point

When a conflict is detected, a clause $C = \ell_1 \lor \cdots \lor \ell_n$ is falsified by $\tau \cup \omega$. That is, in the current assignment $\tau \cup \omega$, all the literals in C are assigned to `false`. We also know that there are two types of literals in the trail: decisions, and implied literals. Implied literals are literals that were assigned to `true` because there was no other way to satisfy a clause in φ. This is the reason for the propagation.

For any ℓ in C such that $\neg\ell$ is implied (not decisions), we therefore know that there exists a clause $C' = \neg\ell \lor \ell'_2 \lor \cdots \lor \ell'_{n'}$ such that $\neg\ell \in \tau \cup \omega$ and $\{\neg\ell'_2, \dots, \neg\ell'_{n'}\} \subseteq (\tau \cup \omega)$. We can use the resolution rule (see Sect. 1.2) between C and C' to generate a new clause.

$$\frac{C = \ell \lor \ell_2 \lor \cdots \lor \ell_n \qquad C' = \neg\ell \lor \ell'_2 \lor \cdots \lor \ell'_{n'}}{C'' = \ell_2 \lor \cdots \lor \ell_n \lor \ell'_2 \lor \cdots \lor \ell'_{n'}}$$

The clause C'' is a logical consequence of φ and is unsatisfiable under the current assignment $\tau \cup \omega$. Indeed, $\ell_2 \vee \cdots \vee \ell_n$ is unsatisfiable since C is falsified. $\ell'_2 \vee \cdots \vee \ell'_{n'}$ is unsatisfiable since it is the falsified part of the reason of $\neg\ell$, that is, $\neg\ell$ was the only satisfied literal in C'.

It might be the case that some literals in C' are also in C. In this case, the resolution may generate reasonably small clauses.

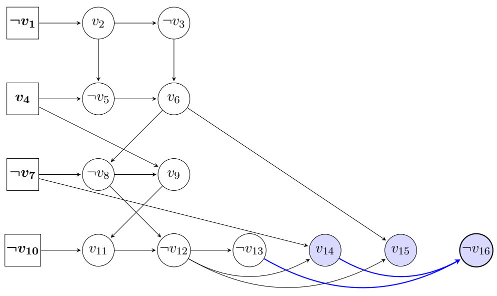
If this procedure is applied iteratively to remove literals at the highest decision level until only one remains, the learned clause is called a Unique Implication Point (UIP) [10]. A UIP is a clause that is a logical consequence of φ, is unsatisfiable under the current assignment $\tau \cup \omega$, and has a unique literal at the maximum decision level. The first UIP is the closest to the conflict and is a good heuristic to find the most general clause possible.

Remark 2.3.3. In NCB, the conflict clause always has at least two literals at the current decision level. Indeed, if it had not been the case, the last literal would have been propagated at a lower decision level. Furthermore, we discussed earlier that the two watched literals of a conflict must be falsified by ω, and ω has all its literals at the highest level. This is not necessarily the case in chronological backtracking as we shall see in the next chapter.

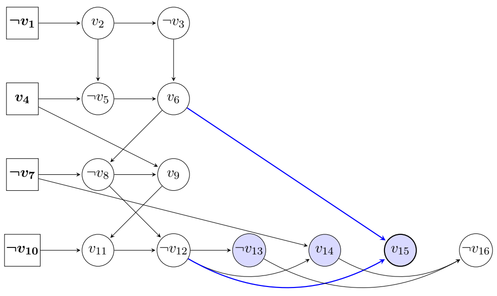
Remark 2.3.4. To obtain a UIP C', it is necessary that the last literal ℓ of the conflict C in the assignment $\tau \cup \omega$ does not belong to C'. Indeed, we know that there are at least two literals in C at the highest level. By contradiction, if we tried and preserve $\ell \in C'$, we would need to remove all the other literals at level $\delta(\ell)$. This is not possible since replacing them by their reason can only go as far as the decision at level $\delta(\ell)$ that cannot be replaced. Therefore, we would be stuck with ℓ and at least one other literal at decision level $\delta(\ell)$. This would not qualify as a UIP. Furthermore, when replacing any literal by its parent in the implication graph, the replacement literals are always located before in the assignment (because $\tau \cup \omega$ is a topological ordering with respect to the implication graph). Therefore, a UIP has all its literals strictly before ℓ in the assignment and can be used as a learned clause.

Finding the first UIP can be done in linear time with Alg. 5. The algorithm starts from the conflict clause C and iteratively replaces literals from the highest decision level with the falsified literals from the reason of that literal. In the implication graph, this is equivalent to replacing the current node with its direct predecessors. The algorithm stops when there is only one literal at the maximum decision level (line 6). To obtain the most relevant clause, the algorithm removes literals in the reverse order in which they were added to the trail. The example on Fig. 2.3.1 was analyzed with Alg. 5 and the steps are shown in Fig. 2.3.2. The negation of the literals belonging to the conflict clause in the current step are in blue. At first (Fig. 2.3.2a), the learned clause is identical to the conflict clause. Then (Fig. 2.3.2b), the last literal on the trail ($\neg v_{16}$) is replaced by the negation of its parents v_{13} and $\neg v_{14}$. $\neg v_{14}$ was already in the clause and is not added again. The process continues with v_{15} (Fig. 2.3.2c) until $\neg v_{12}$ (Fig. 2.3.2e). There is only one literal at decision level $\delta(\tau)$ left and the analysis is complete. The learned clause is $\neg v_6 \vee v_7 \vee v_{12}$. The procedure is equivalent to using the resolution rule on literals in the reverse order of the trail:

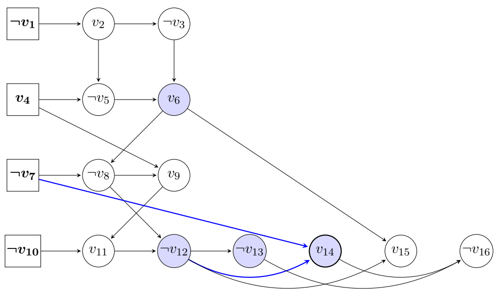
$$\dfrac{\dfrac{\dfrac{\dfrac{\neg v_{14} \vee \neg v_{15} \vee v_{16} \qquad v_{13} \vee \neg v_{14} \vee \neg v_{16}}{v_{13} \vee \neg v_{14} \vee \neg v_{15}} \qquad \neg v_6 \vee v_{12} \vee v_{15}}{\neg v_6 \vee v_{12} \vee v_{13} \vee \neg v_{14}} \qquad \boldsymbol{v_7} \vee v_{12} \vee v_{13} \vee v_{14}}{\neg v_6 \vee \boldsymbol{v_7} \vee v_{12} \vee v_{13}} \qquad v_{12} \vee \neg v_{13}}{\neg v_6 \vee \boldsymbol{v_7} \vee v_{12}}$$



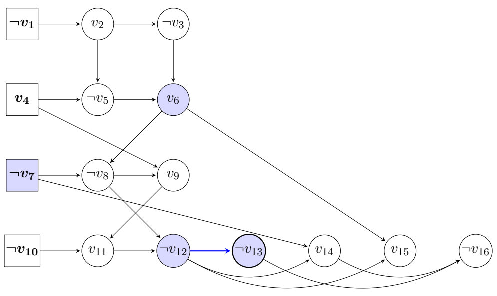
(a) $i = 15, n = 3, L = \{\neg v_{14}, \neg v_{15}, v_{16}\}$



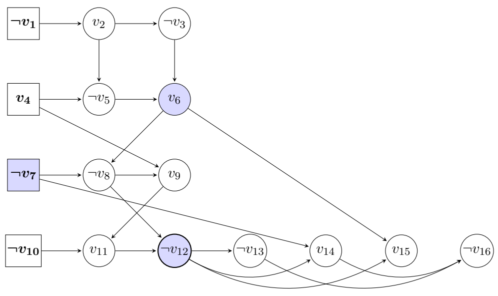
(b) $i = 14, n = 3, L = \{v_{13}, \neg v_{14}, \neg v_{15}\}$



(c) $i = 13, n = 3, L = \{\neg v_6, v_{12}, v_{13}, \neg v_{14}\}$



(d) $i = 12, n = 2, L = \{\neg v_6, v_7, v_{12}, v_{13}\}$



(e) $i = 11, n = 1, L = \{\neg v_6, v_7, v_{12}\}$

Figure 2.3.2: Execution of the conflict analysis on the implication graph of Fig. 2.3.1. The literals were re organized to fit the trail order (left to right, top to bottom). The negation of literals marked in blue belong to the current conflict clause.

Algorithm 5 First UIP conflict analysis

```
1: procedure Analyze($C$)
2:     $\pi \leftarrow \tau \cup \omega$                          ▷ $\pi$ the array version of $\tau \cup \omega$
3:     $C' \leftarrow C$                              ▷ $C'$ is the current learned clause
4:     $n \leftarrow$ the number of literals in $C$ at $\delta(\tau)$
5:     $i \leftarrow |\pi| - 1$                         ▷ $i$ is the index of the last literal in trail
6:     while $n > 1$ do
7:         while $\neg\pi[i] \notin C'$ do             ▷ Find the last literal of $C'$ in $\pi$
8:             $i \leftarrow i - 1$
9:         $C' \leftarrow C' \setminus \{\neg\pi[i]\}$               ▷ Remove the literal $\pi[i]$
10:        $n \leftarrow n - 1$
11:        $C'' \leftarrow$ GetReason($\pi[i]$)          ▷ Get the clause that implied $\pi[i]$
12:        for $\ell \in C'' \setminus \{\pi[i]\}$ do
13:            if $\ell \in C'$ then
14:                continue                        ▷ Avoid duplicates
15:            $C' \leftarrow C' \cup \{\ell\}$
16:            if $\delta(\ell) = \delta(\tau)$ then
17:                $n \leftarrow n + 1$
18:    return $C'$
```

An important assumption of Alg. 5 as described in [10] is that the literals in $\tau \cup \omega$ are ordered in a topological order with respect to the implication graph. In NCB, this is self-evident because the trail is built in a topological order as discussed earlier. However, as we shall see in Sect. 4.2.3, this detail becomes important when switching to Strong Chronological Backtracking (SCB). If this assumption is broken, then the exploration order of literals may miss some literal at the highest level (see i in Alg. 5 line 8). Some literals in C' could be added behind i and not be replaced. Several traversals of the highest level would be necessary, and the returned clause might not be the first UIP.

2.3.4 Learning and Backtracking

Previously, the procedure for conflict analysis was established. Once the first UIP is found, it must be added to the formula. However, it cannot be done directly without breaking Inv. *trail sanity*. Therefore, the solver first backtracks to a decision level before the conflict. The learned clause is no longer conflicting, but is still unit under τ (if backtracking only unassigned the highest literal in the learned clause). So, following Inv. *watched literals* (or Inv. *blocked watched literals*), the unassigned literal is added to the propagation queue ω. Finally, the clause can safely be added to the formula and the solver can resume the search. The learned clause prevents the solver from meeting the same conflict again.

When using watched literals, backtracking is almost free. The watched literals are only updated when they are falsified by the trail. When backtracking, no such thing is done. Furthermore, conflicting and unit clauses see their watched literals unassigned during backtracking. Indeed, conflicting clauses must have both watched literals at the highest decision level, and unit clauses must have one watched literal in ω at the highest level too. When backtracking, the watched literals are unassigned, meaning that the solver recovers sane watched literals.

The backtracking procedure shown on Alg. 6 is rather simple. It consists in popping the trail τ until the desired level is reached. It also clears the propagation queue ω since any literal in ω must be at level $\delta(\tau)$. Backtracking preserves Inv. *trail sanity*. Indeed, backtracking does not assign any new literals. Therefore, it cannot create any falsified clause. It also preserves Inv. *watched literals* (or Inv. *blocked watched literals*). Watched literals are always updated when they are falsified. Therefore, each update pushes the level of watched literals up. This ensures that falsified watched literals are at the highest level in the clause, or that the clause is satisfied at a level lower than the watched literals. Therefore, backtracking is guaranteed to either backtrack the watched literals, or preserve the satisfied literal.

Algorithm 6 Backtracking

```
1: procedure BACKTRACK($\delta$)                                ▷ $\delta < \delta(\tau)$
2:     $\omega \leftarrow \emptyset$
3:     while $\delta(\tau) > \delta$ do
4:         POP($\tau$)
```

Example 2.3.3. If during the conflict analysis, the clause $a \vee b \vee c$ is found, and the literals were assigned at level $\delta(\neg a) = 5$, $\delta(\neg b) = 3$ and $\delta(\neg c) = 2$, a solver using NCB backtracks to level 3. The learned clause becomes unit, and a is implied at level 3.

2.3.5 Decision

DECIDE() is the procedure that searches an unassigned literal to propagate when no inference can be used. It is usually done using some heuristic that keeps track of a score for each literal. The literal with the highest score is propagated. [11] uses the "activity" of variables. The activity of a variable is increased when it is involved in a conflict. Periodically, all variables' activity is decreased to avoid arithmetic overflow. Literals that were assigned with the DECIDE() procedure are called *decisions*. Literals that were propagated by BCP after a decision are at the same decision level as the decision (in NCB).

2.3.6 Purging Clauses

During the execution of the SAT solver, clauses with exactly one literal may be discovered (either as input or as learned clauses). Some literals will be propagated at level 0. That is, no decision is required to infer these literals. In that case, literals at level 0 can be considered as "facts". Once in a while, the solver performs some sort of garbage collection, also called purge. All clauses satisfied at level 0 can be removed, and all literals falsified at level 0 can be removed from clauses (since they can never be `true`).

2.3.7 Soundness and Complete

The CDCL algorithm is sound and complete. Soundness is guaranteed by the fact that the trail τ can never falsify any clause. Indeed, before adding a literal ℓ to τ, BCP first checks that adding τ does not falsify any clause. If it does, then ℓ is not added to τ and conflict analysis is performed. Therefore, if a trail τ contains all the variables in φ and does not falsify any clause in φ, then τ is a model and φ is satisfiable. If φ is unsatisfiable, it means that a clause was falsified at level 0. Since the conflict analysis only generates clauses that are implied by φ, the conflict can only be implied by φ without any decision and φ is unsatisfiable.

Termination is guaranteed by the fact that conflict analysis never generates a clause that is already in φ. Indeed, the learned clause is conflicting with the partial assignment $\tau \cup \omega$, before any other clause in φ. Therefore, the learned clause is not in φ. There are finitely many clauses that can be generated with a finite number of variables. If n is the number of variables, then there are 3^n possible clauses. Each variable can either, be in the clause with positive polarity, with negative polarity or not be in the clause at all (3 choices for each variable). If φ is unsatisfiable, then a conflict at level 0 must eventually occur ($a \land \neg a$ would be such a conflict). If φ is satisfiable, then the solver will eventually find a model. In the worst case, the solver generates all possible clauses that are implied by φ and then DECIDE() and BCP() will find a model.

These proof sketches are not very rigorous. If the reader is interested, [3, 12] provide a verified implementation of CDCL in Isabelle/HOL [24].

2.3.8 Advantage over DPLL

Let us apply CDCL on the motivating example Ex. 2.2.3 to display the strength of conflict analysis. The formula is $\varphi = \{C_1, C_2, C_3, C_4, C_5, C_6\}$ with

$$\begin{array}{lll} C_1 = a \lor b & C_2 = \neg b \lor c & C_3 = \neg a \lor \neg c \\ C_4 = \neg a \lor b \lor c & C_5 = a \lor \neg b \lor \neg c & C_6 = d \lor e \end{array}$$

Let us use the same decision ordering as in Ex. 2.2.3. During the search, the first conflict is discovered when propagating b with the trail $\tau = \{\boldsymbol{d}, \boldsymbol{e}, \boldsymbol{a}, \neg c\}$. The conflict is caused by clause $C_2 = \neg b \lor c$. However, we know that the reason for b was C_4. The conflict analysis procedure learns the clause $C_7 = \neg a$.

$$\dfrac{\dfrac{\neg b \lor c \qquad \neg a \lor b \lor c}{\neg a \lor c} \qquad \neg a \lor \neg c}{C_7 = \neg a}$$

The second-highest level of C_7 is 0, and the solver backtracks to level 0, adds the C_7 to φ then propagates $\neg a$. Because of C_1, b is implied. Then $\neg c$ because of C_5. We now have a conflict with C_2. However, no decision was made on the trail; therefore the conflict clause is $C_8 = \bot$. The problem is UNSAT. Those steps are summarized in Fig. 2.3.3. CDCL explored much fewer nodes than DPLL because it was able to identify that the conflict was local to the variables a, b and c.

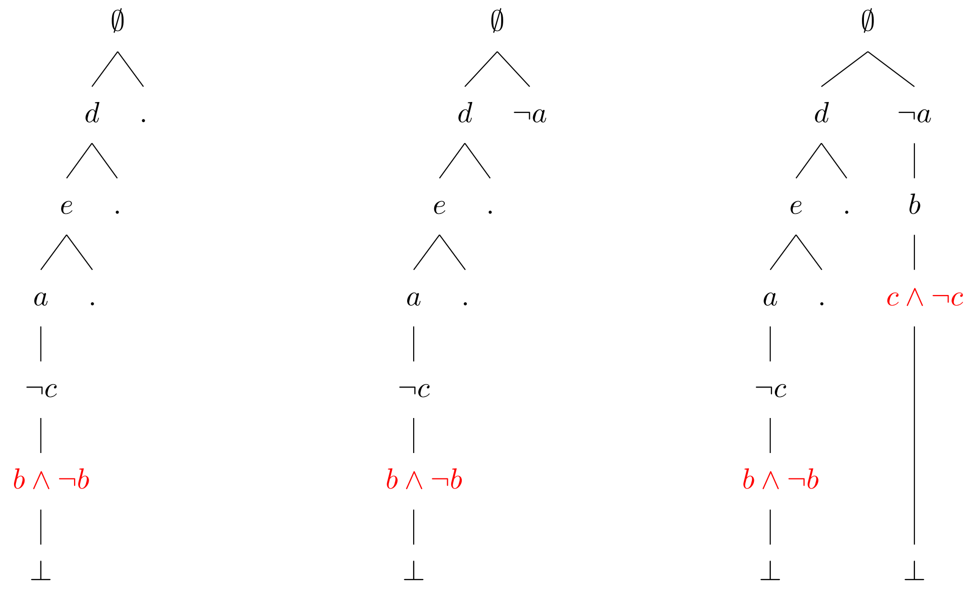


Figure 2.3.3: CDCL with first UIP on Ex. 2.2.3

Chapter 3

Chronological Backtracking

Previously, we learned that the traditional CDCL algorithm backtracks to the second-highest decision level of a learned clause. This is called *non-chronological backtracking.* In this chapter, we learn about a new paradigm: *chronological backtracking.* This method was introduced by [22] and formalized by [19]. We first cover the key ideas behind chronological backtracking and present the required modifications to the CDCL algorithm as presented by the authors of [22, 19]. In the next chapter, we present the details of implementation that were not discussed in the literature (as far as the authors are aware). Some modifications were done on the algorithms to better fit the needs of the implementation in `veriT` and to comply with the previously used notations.

3.1 Motivation

Non-Chronological Backtracking (NCB) has been an undisputed technique for CDCL SAT solvers for a long time. However, Nadel et al. [22] introduced the idea of removing the minimum number of literals possible from the trail and stopping backtracking as soon as the learned clause becomes unit. That is, backtrack to the highest level in the learned clause minus one. They showed that it could be beneficial to undo the trail less aggressively. Their implementation performed significantly better on some problems without huge drawbacks. Chronological Backtracking (CB) performs particularly well on satisfiable instances, which is an interesting property for this thesis since in SMT, most instances are satisfiable. When using NCB, conflicts may undo a large portion of the stack that may need to be recomputed later. This is expensive work.

Example 3.1.1. Consider the example from [22]: $\varphi = \varphi_1 \wedge (a \vee \neg b) \wedge (a \vee b)$, where φ_1 is an arbitrarily large satisfiable formula whose variables are disjoint from a and b. A traditional NCB solver would first try to solve φ_1 since the number of occurrences of variables is higher than for a and b. When the solver is done with solving φ_1, it would then make a decision. It may decide $\neg a$, leading to a conflict. The solver learns the clause a, then backtracks to the second-highest literal in the clause, that is, 0. All the work done to solve φ_1 is lost (aside from learned clauses) and has to be redone.

In the example above, a chronological backtracking solver would simply backtrack $\neg a$, propagate a, then make a decision on b. In that particular example, it would save a lot of time.

This simple idea, however, requires some modifications to the core CDCL algorithm. This change of backtracking paradigm impacts all the main components of CDCL.

3.2 Modifications on the CDCL Algorithm

Alg. 7 shows the modifications necessary to convert a SAT solver from NCB to CB as presented by [19]. The main difference is that the solver no longer backtracks to the second-highest level in the clause but to the highest level minus one. This means that literals can be implied and propagated at a lower level than the highest level on the trail. An example is discussed in Sect. 3.2.2. Another important difference is that it is now possible to have conflicts with only one literal at the highest level. In this case, it is not necessary to trigger conflict analysis since the conflict already satisfies the conditions of the first UIP (that is, only one literal falsified at the highest level). In that case, the solver simply backtracks to the highest level minus one and propagates the highest literal in the clause. This is shown in lines 9 to 14 in Alg. 7. An example is also shown on Fig. 3.2.2 and discussed in Sect. 3.2.2.

Previously, the assignment level of literals was implicitly determined. In NCB, any new literal is assigned at level $\delta(\tau)$ if it is implied, and $\delta(\tau) + 1$ if it is a decision. In CB, the level must be set according to the level of the reason for the literal as shown on lines 12 and 20 or Alg. 7. The following algorithms explicitly set the decision levels when necessary. Unassigned literals have a decision level of ∞.

In Rem. 2.3.2, we mentioned that the definition of decision levels held as long as we stayed in the non-chronological backtracking setting. This is no longer the case, and the definition must be updated. The decision level of a literal $\delta(\ell)$ is the number of decisions before ℓ plus one if ℓ is a decision, and the highest decision level of predecessors of ℓ in the implication graph. It is computed by the maximum level of falsified literals in the reason for the implication of ℓ. In NCB, this definition is equivalent to the one given in the previous chapter. Indeed, since a decision is taken only when no implication can be made, no more literals can be implied by the last decision until a new clause is added. When a learned clause is added, the solver backtracks to the second-highest level of the clause and still implies the literal at the highest level in τ. Therefore, both definitions are equivalent in NCB. From this point onward, we consider the second definition to be the correct interpretation of the decision level.

3.2.1 Backtracking

The backtracking procedure is also modified. Since the trail may be out of order with respect to the decision level, it is necessary to preserve the literals lower than the backtracking level. This is done using an auxiliary queue β that contains the literals that are lower than the backtracking level. Alg. 8 is the backtracking algorithm advocated by [22] adjusted to our notations. However, it lacks some details that are discussed in the next chapter.

3.2.2 Broken Invariants

While it is expected and documented by [19] that the conversion to CB breaks some invariants, it may be worth mentioning the main invariant that is broken by CB. In NCB, the trail is a sequence of literals of monotonically increasing decision levels. In CB, that is no longer the case. Right after a conflict, the learned clause may not involve some decision levels between

Algorithm 7 CDCL with chronological backtracking

```
 1: $\tau \leftarrow \emptyset$                                    ▷ The trail
 2: $\omega \leftarrow \emptyset$                                    ▷ The propagation queue
 3: $\varphi \leftarrow \emptyset$                                    ▷ The set of learned clauses
 4: procedure CDCL($\varphi'$)
 5:     $\varphi \leftarrow \varphi'$                                ▷ Initialize the set of clauses
 6:     while $\top$ do
 7:         $C \leftarrow$ BCP()
 8:         if $C \neq \top$ then                          ▷ Conflict detected
 9:             if $C$ has only one literal at level $\delta(C)$ then
10:                 $\ell \leftarrow$ the highest literal in $C$
11:                 Backtrack($\delta(C) - 1$)
12:                 $\delta(\ell) \leftarrow \delta(C \setminus \{\ell\})$
13:                 $\omega \leftarrow \omega \cup \{\ell\}$
14:                 continue
15:             if $\delta(C) = 0$ then
16:                 return $\bot$
17:             $C' \leftarrow$ Analyze($C$)
18:             Backtrack($\delta(C') - 1$)
19:             $\ell \leftarrow$ the highest literal in $C'$
20:             $\delta(\ell) \leftarrow \delta(C' \setminus \{\ell\})$
21:             $\omega \leftarrow \omega \cup \{\ell\}$
22:             $\varphi \leftarrow \varphi \cup \{C'\}$
23:             continue
24:         $\ell \leftarrow$ Decide()
25:         if $\ell = \bot$ then                          ▷ No more literals to propagate
26:             return $\tau$
27:         $\delta(\ell) \leftarrow \delta(\tau) + 1$
28:         $\omega \leftarrow \omega \cup \{\ell\}$
```

Algorithm 8 Backtracking with chronological backtracking

```
 1: procedure Backtrack($\delta$)
 2:     $\beta \leftarrow \emptyset$
 3:     $\omega \leftarrow \emptyset$
 4:     while $\delta(\tau) > \delta$ do
 5:         $\ell \leftarrow$ Pop($\tau$)
 6:         if $\delta(\ell) < \delta$ then
 7:             $\beta \leftarrow \{\ell\} \cup \beta$
 8:         else
 9:             $\delta(\ell) \leftarrow \infty$
10:     $\tau \leftarrow \tau \cup \beta$
```

the highest and second-highest levels. Therefore, the solver can propagate literals that are lower than the current decision level. An example of this is shown in Fig. 3.2.1.

In CB, the trail no longer behaves like a stack. Indeed, literals can be removed from the center of the trail. This needs to be taken into account when CDCL is embedded in an SMT solver.

How to read trail diagrams. When using chronological backtracking, one needs to know the level of every literal in the partial assignment. This is why, from this point onward, we use diagrams such as shown in Fig. 3.2.1. The bold literals are decisions. Each step level represents a decision level. The literals are positioned in the relative order in which they were added to the trail (until further notice, see Sect. 4.2.4). Literals that were implied are annotated with the reason for propagation below the bottom line. The reason for implication is the unit clause that was used to infer the literal. For visual reasons, literals falsified by $\tau \cup \omega$ are shown in red. The two first literals of the clauses are the watched literals. Conflicts are marked at the end of the trail by a $\perp$ symbol and the reason for conflict is also annotated below the bottom line.

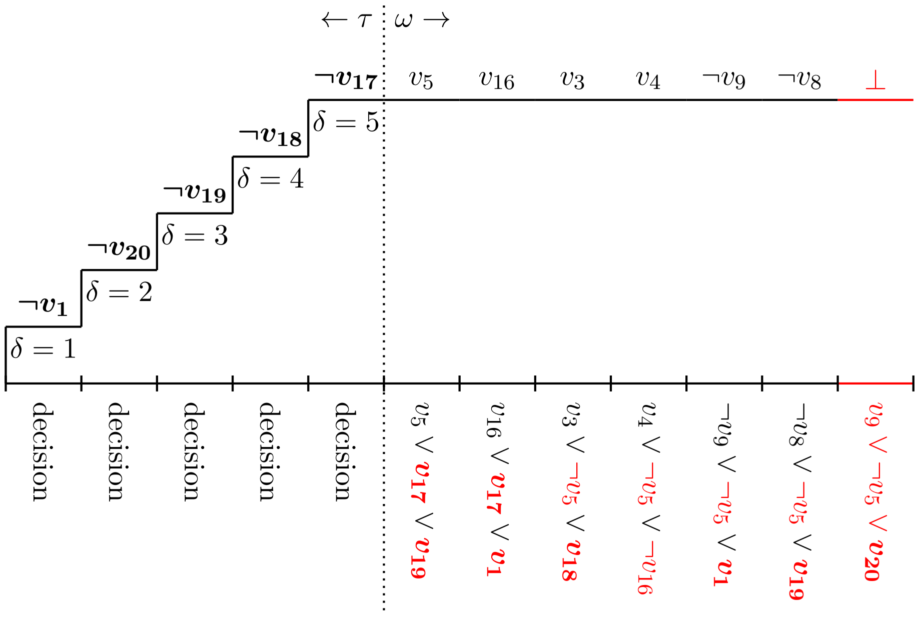
The dotted line marks the current propagation progression. Literals on the left are in the trail τ, and literals on the right are in the propagation queue ω. If no such line is present, then the trail is entirely in τ. When segments of the trail are marked as dashed, it means that the solver implied more literals that are irrelevant to the example at hand, and were cut out for conciseness. No assumptions should be made about the levels of the literals that were cut out.

In general, the diagrams are taken from real runs of the modified `veriT`-SAT solver over the SATLIB [16] benchmarks from the Uniform Random-3-SAT[1] category. The goal is to obtain realistic examples that show real behaviors of the algorithms. Instances are labeled `uf` for satisfiable instances, and `uuf` for unsatisfiable instances. Then, the number of variables is appended to the label, and finally the problem number is placed after the hyphen. For example, `uf20-0003.cnf` is the third satisfiable instance with 20 variables. However, some examples were handcrafted to illustrate a simpler point. In that case, the clauses are provided in the statement and the variables are named $a, b, c, \ldots$ as opposed to $v_1, v_2, v_3, \ldots$ for the SATLIB instances.

Chronological backtracking at work. In the example shown in Fig. 3.2.1, a conflict is detected at level 5. The conflict clause is $C = v_9 \vee \neg v_5 \vee v_{20}$. The highest level in the clause is $\delta(v_9) = \delta(\neg v_5) = 5$. The conflict analysis learns the clause $D = \neg v_5 \vee v_{20} \vee v_1$. The highest literal in D is $\neg v_5$ and $\delta(\neg v_5) = 5$. The second-highest literal is v_{20} and $\delta(\neg v_{20}) = 2$. Therefore, the solver backtracks to level 4 and propagates $\neg v_5$ at level 2. In NCB, the solver would have backtracked to level 2, undoing more work.

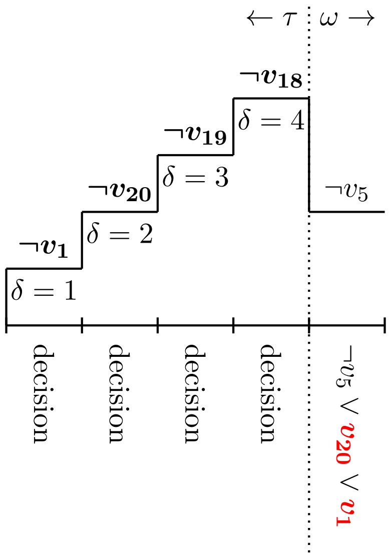
Unique literal at highest level. As briefly mentioned in Sect. 3.2, since new literals can be implied at a lower level than previous literals, it is possible that the conflict clause contains only one literal at the highest level. Indeed, the clause may be unit under $\tau \cup \omega$ at a level δ (one literal at level δ) with the unassigned literal ℓ. But before the clause is detected to be

[1]https://www.cs.ubc.ca/ hoos/SATLIB/benchm.html



$\leftarrow \tau$ $\omega \rightarrow$
$\neg \boldsymbol{v_{17}}$ v_5 v_{16} v_3 v_4 $\neg v_9$ $\neg v_8$ $\bot$
$\neg \boldsymbol{v_{18}}$
$\neg \boldsymbol{v_{19}}$
$\neg \boldsymbol{v_{20}}$
$\neg \boldsymbol{v_1}$
$\delta = 1$
$\delta = 2$
$\delta = 3$
$\delta = 4$
$\delta = 5$
decision
decision
decision
decision
decision
$v_5 \vee \boldsymbol{v_{17}} \vee \boldsymbol{v_{19}}$
$v_{16} \vee \boldsymbol{v_{17}} \vee \boldsymbol{v_1}$
$v_3 \vee \neg v_5 \vee \boldsymbol{v_{18}}$
$v_4 \vee \neg v_5 \vee \neg v_{16}$
$\neg v_9 \vee \neg v_5 \vee \boldsymbol{v_1}$
$\neg v_8 \vee \neg v_5 \vee \boldsymbol{v_{19}}$
$v_9 \vee \neg v_5 \vee \boldsymbol{v_{20}}$

(a) A conflict is detected at level 5. The clause $\neg v_5 \vee v_{20} \vee v_1$ was learned.



$\leftarrow \tau$ $\omega \rightarrow$
$\neg \boldsymbol{v_{18}}$
$\neg \boldsymbol{v_{19}}$
$\neg \boldsymbol{v_{20}}$
$\neg \boldsymbol{v_1}$
$\neg v_5$
$\delta = 1$
$\delta = 2$
$\delta = 3$
$\delta = 4$
decision
decision
decision
decision
$\neg v_5 \vee \boldsymbol{v_{20}} \vee \boldsymbol{v_1}$

(b) The solver backtracks to level $4 = 5 - 1$. The learned clause $\neg v_5 \vee v_{20} \vee v_1$ becomes unit at level 2. Therefore, $\neg v_5$ is implied at level 2.

Figure 3.2.1: An example of CB at work (from `uf20-0001.cnf`). $\neg v_5$ is implied and propagated at a lower level than $\neg \boldsymbol{v_{18}}$. The monotonicity of decision levels in the trail is broken.

unit, the solver enqueues $\neg\ell$ at a level $\delta' < \delta$. The clause becomes a conflict, but has only one literal at the highest level. Therefore, backtracking is enough to make the clause unit and continue the propagation. An example is shown in Fig. 3.2.2. This special case is caused because it is possible to propagate literals at arbitrary levels (lower than $\delta(\tau)$). Breaking the monotonicity invariant already shows some interesting properties.

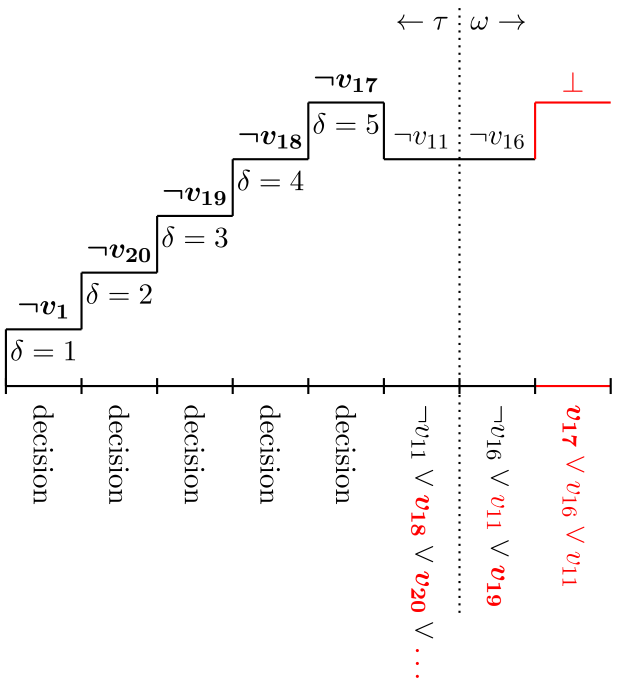

$\leftarrow \tau$
$\omega \rightarrow$
$\neg \boldsymbol{v_{17}}$
$\neg \boldsymbol{v_{18}}$
$\neg \boldsymbol{v_{19}}$
$\neg \boldsymbol{v_{20}}$
$\neg \boldsymbol{v_1}$
$\delta = 5$
$\delta = 4$
$\delta = 3$
$\delta = 2$
$\delta = 1$
$\neg v_{11}$
$\neg v_{16}$
$\bot$
decision
decision
decision
decision
decision
$\neg v_{11} \lor \boldsymbol{v_{18}} \lor \boldsymbol{v_{20}} \lor \ldots$
$\neg v_{16} \lor v_{11} \lor \boldsymbol{v_{19}}$
$\boldsymbol{v_{17}} \lor v_{16} \lor v_{11}$

Figure 3.2.2: An example of a conflict with only one literal at the highest level (from `uf20-0016.cnf`). The clause $v_{17} \lor v_{16} \lor v_{11}$ is conflicting at level 5, but only contains the literal v_{17} at that level. It is therefore not necessary to trigger conflict analysis. The solver backtracks to level 4 and propagates v_{17}.

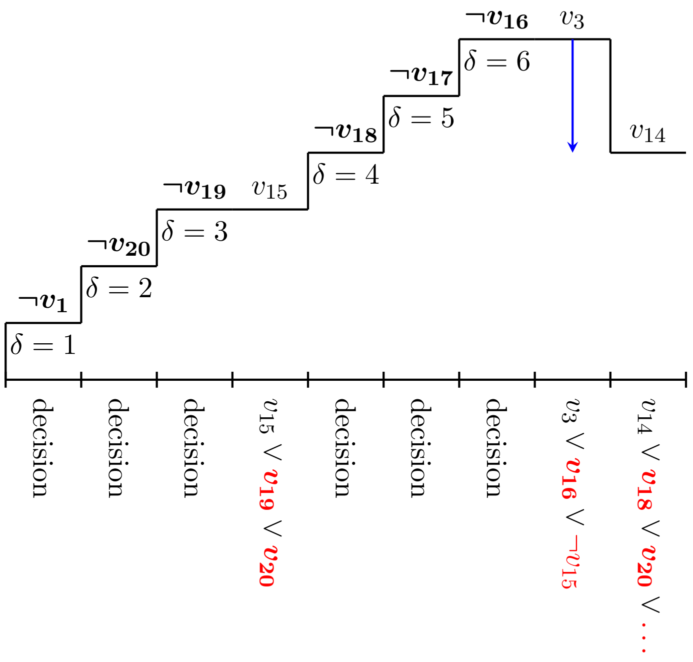
3.3 Missed Lower Implications

During propagation, it is now possible to generate a trail that satisfies a clause by only one literal at a level higher than all the other literals in the clause. This is called a missed lower implication [21]. The satisfying literal should have been implied at a lower level (since the clause is unit at that level), but was not since the clause became unisat after it was satisfied. An example of a missed lower implication is given in Fig. 3.3.1.

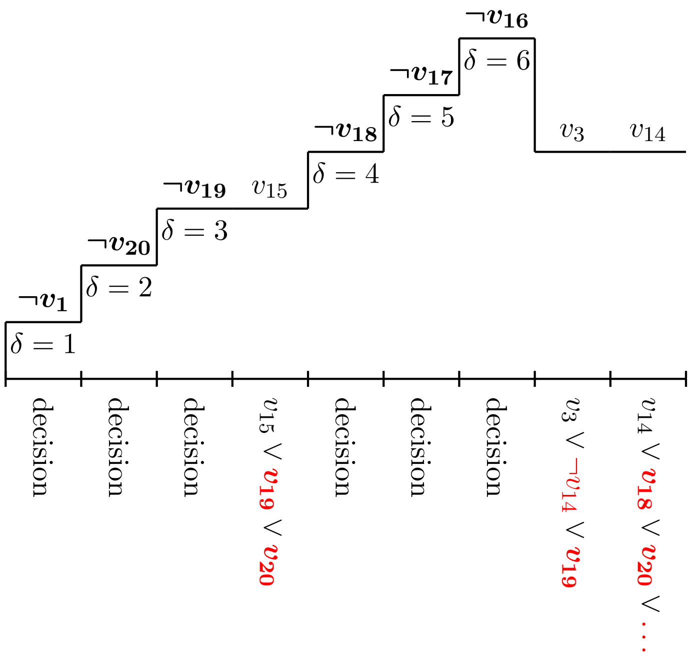
Missed lower implications are a missed opportunity to refine the exploration of the search space. While soundness and completeness are not impaired by overlooking this issue, efficiency is. Missed lower implications are literals that may be backtracked at a higher level than they should. In which case, an implication will be missed. But since the clause is unit, it must be watched by the satisfying literal. If it is backtracked and propagated with the opposite polarity, BCP will notice the conflict. This is inefficient since backtracking literals that are still valid is a waste of time. Also, ensuring that the literals are as deep as possible in the trail is a good way to prune the largest number of branches in the search tree when learning.

A re-implication procedure modifies the levels and reasons of missed lower implications. Updating those levels may, in turn, reveal more missed lower implications. The same way propagation is performed until a fixed point is reached, re-implication is performed until no more missed lower implications are found.

If missed lower implications are not resolved, Inv. *blocked watched literals* no longer holds. Indeed, if a clause C is unisat by a literal ℓ at level δ and $\delta(C \setminus \ell) < \delta$, then if the solver backtracks to level $\delta - 1$, C will see one of its watched literals falsified but will not be satisfied. This is a violation of Inv. *blocked watched literals*.



(a) A missed lower implication is detected with the clause $v_3 \lor \neg v_{14} \lor v_{19}$



(b) The literal reason and level of v_3 is revised to recover from the missed lower implication.

Figure 3.3.1: Missed lower implication example (from `uf20-0010.cnf`). While propagating v_{14}, the clause $v_3 \lor \neg v_{14} \lor v_{19}$ is examined. It is unisat and the highest falsified literal is at level 4. However, the unique satisfied literal is at level 6. Therefore, this is a missed lower implication and v_3 should be implied at a lower level. The missed lower implication should be resolved. Otherwise, backtracking to level 4 or 5 would create an undetected unit clause, breaking Inv. *blocked watched literals*.

Chapter 4

Chronological Backtracking in `veriT`

In this work, we intend to convert the NCB SAT solver from `veriT` to CB. The first step is to convert the standalone SAT solver `veriT`-SAT. In Chap. SMT solving, we consider the impact of this change on the SMT solver. This process reveals insights on details that were not covered in the literature on CB (as far as we are aware). In this chapter, we discuss the attention points and complications one might encounter during this process. In particular, we discuss the changes CB brings to the watched literal management. The re-implication procedure suggested by [21] is adapted to fit the needs of `veriT`. We also discuss a few broken non-core assumptions that may not be unique to `veriT`.

This chapter is divided in three parts. The first one discusses changes that are necessary for soundness. The second part addresses techniques used to avoid missing propagations and improve performance. The last part puts all the pieces together and provides a detailed algorithm in agreement with the lessons learned from the first two sections. The algorithms discussed in this chapter have been implemented in `veriT` and empirically tested on the SATLIB [16] benchmarks.

Recommendations for CB conversion. Before starting any changes, we advocate for first defining a set of invariants that should be preserved by the conversion. Then, implement *invariant checkers* to ensure that they are respected. This helps to detect any broken assumption that the solver may have and is very useful for debugging. They are discussed throughout this chapter. All invariants mentioned in this chapter were used to empirically check the correctness of the conversion.

Preserved invariants. While some invariants are modified by the conversion, a few remain and are advised to be checked during the implementation process. Some may seem trivial but are spelled out for the sake of completeness.

Invariant 4.0.1 (Trail sanity). The trail τ does not falsify a clause in φ.

Invariant 4.0.2 (Trail level ordering). For each level $\delta \in \{1, \ldots, \delta(\tau)\}$, the first literal of $\tau \cup \omega$ at level δ is a decision.

Invariant 4.0.3 (Watch list completeness). Each clause C in φ such that $|C| \geq 2$ is watched by two literals. Their corresponding watch lists contain the clause.

Weak vs. strong chronological backtracking. In this document, we discuss two different approaches to chronological backtracking. The first is called *weak chronological backtracking* and consists in the minimum changes required to convert a NCB solver to CB. The second is called *strong chronological backtracking* and is a more radical approach that requires more changes to the solver. The main difference between the two is that the strong approach does not allow the solver to miss any implication. That is, no clause can be made unit by the trail τ.

4.1 Necessary Modifications - Weak Chronological Backtracking

4.1.1 Greedy Conflict Resolution

During BCP, as many other solvers, `veriT` uses a greedy approach to conflict detection. That is, it stops at the first conflict encountered. This premature stopping criterion creates a situation where the last literal is not fully propagated. Furthermore, `veriT` adds literals to the trail before finishing the propagation, temporarily breaking Inv. *watched literals*. This is done to simplify the code. In NCB, it is not a problem since after the conflict resolution, the last literal in τ is guaranteed to be backtracked. Indeed, the last literal is always at the highest decision level. However, this approach is not compatible with CB. Indeed, since literals can be propagated at an almost arbitrary decision level, the highest literal in the conflicting clause may not be the last to be propagated. An example is shown on Fig. 4.1.1.

Solution 1: premature stop with restart. The first proposed solution is to stop prematurely the search while propagating ℓ (just as before). If a conflict is found, the solver checks after backtracking if the top of the trail τ is still ℓ. In this case, the solver removes ℓ from τ and pushes it at the front of ω. Then the solver resumes. If the trail order is important, then it is also important that ℓ is pushed at the front of ω and not at the back. Trail ordering is discussed in more detail in Sect. 4.2.4. This is shown on Figure 4.1.2.

Solution 2: propagate until the end. The second solution consists in continuing the propagation until the end. BCP then returns the lowest conflict. This ensures that all the other conflicts will be backtracked. Indeed, if the highest literal in the lowest conflict is backtracked, then the highest literal in all the other conflicts will be backtracked too.

This approach has the advantage to only perform at most one conflict analysis (some conflicts do not require conflict analysis) and one backtracking in a row, saving some time. However, the additional learned clauses created by the *restart* method may be useful later in the search. Furthermore, the second approach requires performing more steps when propagating a literal generating a conflict. Indeed, the solver must always fully propagate a literal, while premature stopping may be sufficient.

We have implemented both methods in the weak chronological backtracking algorithm. Preliminary results show that the first approach is slightly faster than the second one. However, the difference is not significant enough to draw any strong conclusion. This becomes irrelevant when implementing strong chronological backtracking, thus we will not discuss this further.

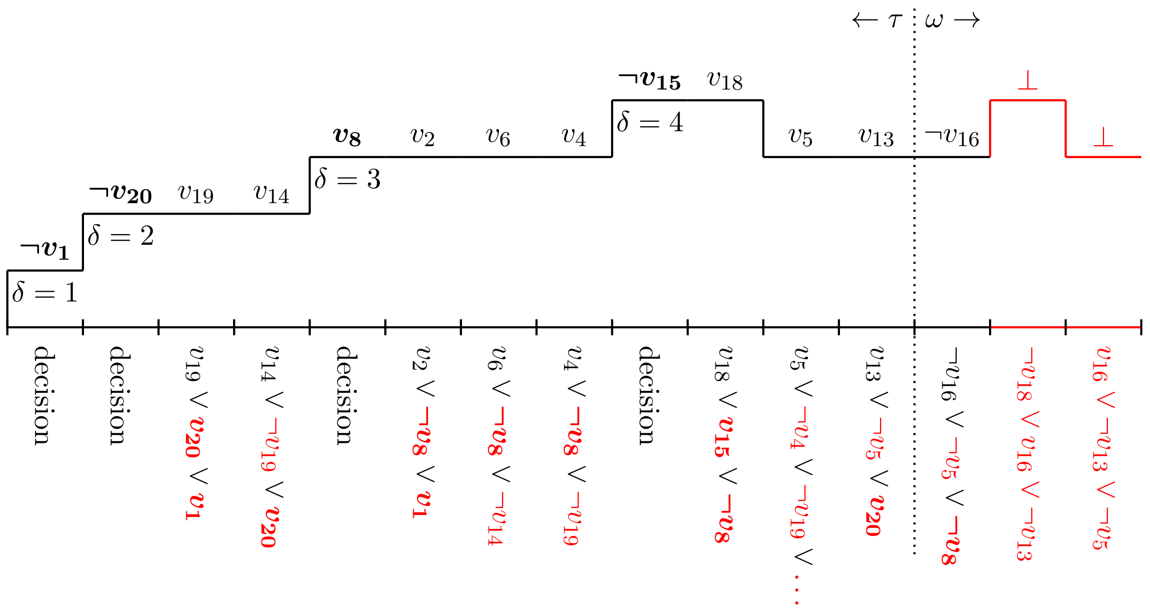

$\leftarrow \tau$
$\omega \rightarrow$
$\neg v_1$
$\delta = 1$
$\neg v_{20}$ v_{19} v_{14}
$\delta = 2$
v_8 v_2 v_6 v_4
$\delta = 3$
$\neg v_{15}$ v_{18}
$\delta = 4$
v_5 v_{13} $\neg v_{16}$
$\bot$
$\bot$
decision
decision
$v_{19} \lor v_{20} \lor v_1$
$v_{14} \lor \neg v_{19} \lor v_{20}$
decision
$v_2 \lor \neg v_8 \lor v_1$
$v_6 \lor \neg v_8 \lor \neg v_{14}$
$v_4 \lor \neg v_8 \lor \neg v_{19}$
decision
$v_{18} \lor v_{15} \lor \neg v_8$
$v_5 \lor \neg v_4 \lor \neg v_{19} \lor \dots$
$v_{13} \lor \neg v_5 \lor v_{20}$
$\neg v_{16} \lor \neg v_5 \lor \neg v_8$
$\neg v_{18} \lor v_{16} \lor \neg v_{13}$
$v_{16} \lor \neg v_{13} \lor \neg v_5$

Figure 4.1.1: Example of multiple conflicts (from `uf20-0736.cnf`). v_{13} was added to τ at the start of its propagation to simplify the NCB code. When the solver propagates v_{13}, it detects a conflict on $\neg v_{18} \lor v_{16} \lor \neg v_{13}$. The highest literal in the conflict is $\neg v_{18}$ at level 4. Therefore, if the solver stops the propagation, it would backtrack to level 3 and the conflict on $v_{16} \lor \neg v_{13} \lor \neg v_5$ would be undetected (because v_{13} was not fully propagated). Since v_{13} was added to τ prematurely and was not backtracked, it is partially propagated and violates Inv. *trail sanity*.

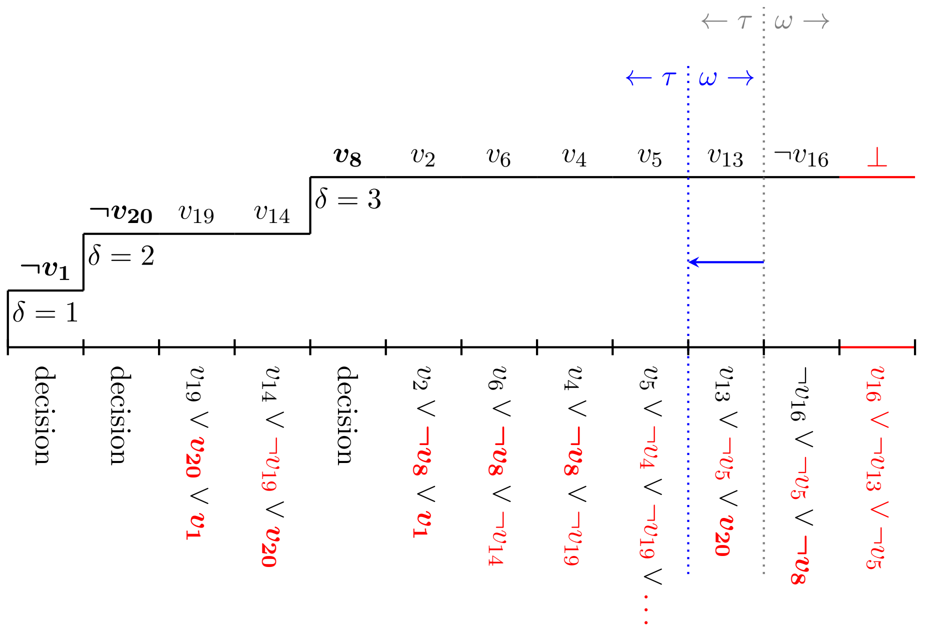

$\leftarrow \tau$
$\omega \rightarrow$
$\leftarrow \tau$
$\omega \rightarrow$
$\neg v_1$
$\delta = 1$
$\neg v_{20}$ v_{19} v_{14}
$\delta = 2$
v_8 v_2 v_6 v_4 v_5 v_{13} $\neg v_{16}$
$\delta = 3$
$\bot$
decision
decision
$v_{19} \lor v_{20} \lor v_1$
$v_{14} \lor \neg v_{19} \lor v_{20}$
decision
$v_2 \lor \neg v_8 \lor v_1$
$v_6 \lor \neg v_8 \lor \neg v_{14}$
$v_4 \lor \neg v_8 \lor \neg v_{19}$
$v_5 \lor \neg v_4 \lor \neg v_{19} \lor \dots$
$v_{13} \lor \neg v_5 \lor v_{20}$
$\neg v_{16} \lor \neg v_5 \lor \neg v_8$
$v_{16} \lor \neg v_{13} \lor \neg v_5$

Figure 4.1.2: After backtracking, v_{13} is still on top of τ. It is therefore moved from τ to ω and the propagation resumes. It now detects the second conflict and backtrack again.

4.1.2 Watched Literals

The usage of watched literals makes a very strong assumption: the order of propagation is the reverse order of backtracking. That is, if ℓ is propagated before ℓ', then it will be backtracked after ℓ'. This property is broken by CB. Indeed, the propagation may propagate ℓ at a higher level than ℓ', but before ℓ'. Therefore, ℓ may be backtracked before ℓ'.

This seemingly small detail actually breaks a lot of important properties on which the watched literals rely. When clauses are conflicting, the solver expects that the watched literals will be backtracked after conflict resolution. This assumption is no longer true in CB. Therefore, the solver must ensure that this property is recovered before losing sight of the clause. Whenever a conflict is detected, the conflicting clause (learned clauses included) ensures that at least one watched literal of the conflict will be backtracked. That is, at least one of the watched literals is at the highest decision level in the conflict.

Invariant 4.1.1 (Watch literal levels). Let a clause $C \in \varphi$ watched by c_1 and c_2. If C becomes conflicting when propagating $\neg c_1$ or $\neg c_2$, then $\delta(c_1) = \delta(C) \vee \delta(c_2) = \delta(C)$.

In `veriT`, the data structure used to store and update watch lists follows the idea of [11]. That is, the watch list is an array that is being explored with two pointers: a reading and a writing head. The reading head is used to find the candidate clause, and the writing head is used to write the clause back to the watch list if the clause remains watched. The algorithm is as shown in Alg. 9.

Practical aspects of the watch lists and their modification during the execution mean that special attention is required when replacing the watched literals of conflicting clauses to ensure that the watch lists remain valid. We need to be careful when modifying a list on which we are iterating. More specifically, while the read and write heads are different, there may be twice the same clause in the watch list. This is not a difficult technical challenge, but could be easily missed when tempering with the watch lists.

Remark 4.1.1. Pay attention that Alg. 9 comes from a NCB solver. It is not compatible with CB, but this insight is used later when designing the new BCP algorithm. For the sake of readability, this implementation detail is omitted in the following algorithms. We trust the reader to keep this in mind if they wish to implement the algorithms.

Invariant 4.1.2 (Weak watched literals). Let a clause $C \in \varphi$ watched by c_1 and c_2. Either c_1 or c_2 is not falsified by τ. Formally: $\neg c_1 \notin \tau \vee \neg c_2 \notin \tau$

Inv. *weak watched literals* is a weakening of Inv. *watched literals* that allows unit clauses to slip through. Indeed, Inv. *watched literals* ensures that no unit nor falsified clause can exist in φ under τ. As is discussed in Sect. 4.2.3, this invariant is difficult to maintain in CB. Inv. *weak watched literals* is sufficient to ensure that the solver does not miss any conflict, guaranteeing soundness. If at least one of the watched literals is not falsified in τ, then, to have a conflict, the other watched literal must be falsified by ω, and the solver will detect at least one conflict when propagating it (their may be several conflicts with one propagation).

4.1.3 Blockers

When using CB, blocker literals can no longer be used as easily as before. Indeed, the blockers made the assumption that, because they were implied before the propagation of the negation

Algorithm 9 BCP with read-write pointers in NCB

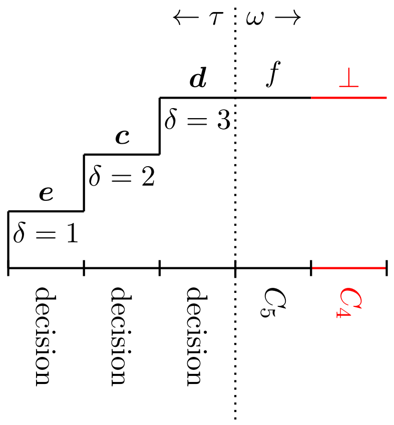
```
procedure BCP()
    while $\omega \neq \emptyset$ do
        $\ell \leftarrow$ FIRST($\omega$)
        $c_1 \leftarrow \neg\ell$
        $j \leftarrow 0$
        $n \leftarrow$ the number of clauses watched by $c_1$
        for $i \leftarrow 0$ to $n-1$ do
            $C \leftarrow$ the $i^{th}$ clause watched by $c_1$
            if $C$ has satisfied blocker in $\tau \vee \omega$ then
                $j^{th}$ watched clause of $c_1 \leftarrow C$        ▷ Keep the clause in the watch list
                $j \leftarrow j+1$        ▷ Move the writing head
                continue
            $c_2 \leftarrow$ the other watched literal of $C$
            if $c_2 \in \tau \cup \omega$ then
                $j^{th}$ watched clause of $c_1 \leftarrow C$        ▷ Keep the clause in the watch list
                $j \leftarrow j+1$        ▷ Move the writing head
                continue        ▷ Clause is satisfied
            $r \leftarrow \bot$
            for $\ell'$ in $C \setminus \{c_1, c_2\}$ do
                if $\neg\ell' \notin \tau \cup \omega$ then
                    $r \leftarrow \ell'$
                    break
            if $r \neq \bot$ then
                if $r \in \tau \cup \omega$ then
                    SETBLOCKER($C, r$)
                    $j^{th}$ watched clause of $c_1 \leftarrow C$        ▷ Keep the clause in the watch list
                    $j \leftarrow j+1$        ▷ Move the writing head
                    continue
                else
                    $r$ starts watching $C$        ▷ By not copying $C$ to the watch list, we are effectively removing it
            if $\neg c_2 \in \tau \cup \omega$ then        ▷ Clause is conflicting
                copy the watch list $[i$ to $n]$ to $[j$ to $n-(i-j)]$
                return $C$
            $j^{th}$ watched clause of $c_1 \leftarrow C$        ▷ Keep the clause in the watch list
            $j \leftarrow j+1$        ▷ Move the writing head
            $\omega \leftarrow \omega \cup \{c_2\}$        ▷ Clause is unit
        truncate the watch list of $c_1$ to $j$ clauses
        $\omega \leftarrow \omega \setminus \{\ell\}$
        $\tau \leftarrow \tau \cup \{\ell\}$
    return $\top$
```

of the watched literals, they will be backtracked after. As discussed earlier, this is no longer the case.

Example 4.1.1. Consider the clauses $C_1, \ldots, C_5$ and the trail τ shown in Fig. 4.1.3a. A conflict is detected at level 4. The solver learns the clause $C_6 = \neg e \vee \neg d$. The propagation then continues.

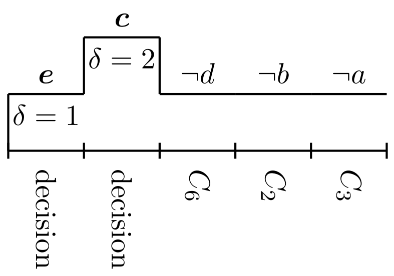
When propagating $\neg b$ and $\neg a$, the clause $C_1 = a \vee b \vee c$ is watched by a and b, but is blocked by c that is satisfied. Therefore, the watched literals remain unchanged. If now a conflict backtracks to level 1 or level 2, then nothing prevents the solver from propagating $\neg c$ and not notice the conflict on C_1. This is a violation of Inv. *trail sanity*.

$C_1 = a \vee b \vee c \qquad C_4 = \neg e \vee \neg d \vee \neg f$
$C_2 = \neg b \vee d \qquad C_5 = \neg d \vee f$
$C_3 = \neg a \vee d$



(a) A conflict occurs when propagating f, C_4 is conflicting with $\tau \cup \omega$, and conflict analysis is triggered.

$C_1 = a \vee b \vee c \qquad C_4 = \neg e \vee \neg d \vee \neg f$
$C_2 = \neg b \vee d \qquad C_5 = \neg d \vee f$
$C_3 = \neg a \vee d \qquad C_6 = \neg d \vee \neg e$



(b) After backtracking and performing BCP again, C_1 is watched by two falsified literals. If c is backtracked and $\neg c$ is propagated instead, then C_1 is not detected as a conflict.

Figure 4.1.3: Example of the failure of standard blockers in CB (handcrafted example). Blocker cannot be used in CB without other assumptions. c is at a higher decision level than a and b, therefore c does not qualify as a weakened blocker.

Therefore, we need a weakened blocker that is not taken into account except if the watched literals are at a higher level than the blocker. In our implementation, we let this weakened blocker aside and do not use it. Otherwise, Inv. *weak watched literals* would be violated. In theory, it is possible to use, but ignoring this complication is simpler and does not seem to have a significant impact on performance. We therefore disabled blockers altogether. Future work could investigate the use of weakened blockers.

4.1.4 Purge

In chronological backtracking (without missed re-implication), it is possible that a clause is satisfied by a watched literal ℓ ($\delta(\ell) > 0$) and that its second watched literal is falsified at level 0. In theory, this is not a problem for either completeness and soundness. However, the PURGE procedure from `veriT` makes the assumption that no watched literal can be falsified at level 0 unless the clause is conflicting at level 0 (UNSAT) or satisfied at level 0. The PURGE procedure should not remove literals from clauses without paying attention to the watch lists. Since the assumption that clauses cannot be watched by literals at level 0 no longer holds in CB, the procedure must be modified to take into account this possibility. While this is not hard to fix (by watching a new literal if the removed literal was removed), it is an attention point that must be kept in mind when implementing CB.

4.2 Recommended Modifications - Strong Chronological Backtracking

The modifications described in the previous section are necessary to have a functioning solver. However, a lot of unit clauses are missed by solely implementing these changes. We here present optimizations to prevent missing such clauses and restore an invariant from NCB.

Invariant 4.2.1 (Watched literals). Let a clause $C \in \varphi$ watched by c_1 and c_2. If c_1 or c_2 is falsified by τ, then the other literal is satisfied by $\tau \cup \omega$.

Inv. *watched literals* ensures that no clause is made unit by the trail τ. Indeed, if a clause C is unit, then all the literals but one are falsified, therefore, at least one of the watched literals must be falsified. If the other watched literal is not satisfied, then Inv. *watched literals* is violated. Otherwise, the unassigned literal is not watched. In that case, both watched literals are falsified and Inv. *watched literals* is violated. Therefore, maintaining Inv. *watched literals* ensures that no clause is unit, nor conflicting. It is the same invariant as in NCB.

Remark 4.2.1. As for NCB, we recover an interesting property of Inv. *watched literals*. If it is satisfied, then a clause C conflicting with $\tau \cup \omega$ mean that the two watched literals of C are in the propagation queue ω. This will no longer be true after the clause is detected as a conflict since we want to preserve Inv. *watched literal levels* and might change the watched literals for that reason.

Inv. *watched literals* is not necessary for completeness and soundness of the solver; a weaker version such as Inv. *weak watched literals* is enough. However, not respecting it certainly hurts the performance of the solver. Indeed, completeness is not lost since the literals that should have been implied could still be propagated later. Soundness is not lost either, since the reasons for the lost literals will still be detected as conflicts if the missed implications are propagated with the other polarity. But missing inferences is the difference between a clever exploration and a pointless wander in the search space.

4.2.1 Backtracking

While [22] provides a backtracking algorithm, as discussed previously, it lacks some details. In particular, it does not suggest recovering the propagating queue ω. The Alg. 8 can be modified to account for this as shown in Alg. 10.

It is important to note that the queue ω is not emptied like in NCB. The literals in ω that are lower than the backtracking level still need to be propagated after backtracking. The literals that are higher than the backtracking level must be filtered out from ω. Not only is it a good idea to preserve literals when possible, but it is also necessary to not violate Inv. *watched literals*. Indeed, if ω is emptied, then the propagations that were implied before the conflict are lost and some clauses may be made unit by removing the satisfied literal in ω and keeping the falsified watched literal in τ.

This new backtracking algorithm preserves Inv. *watched literals* provided that, if a watched literal ℓ is satisfied at level $\delta(\ell)$, the other watched literal is not falsified, or is falsified at a level higher than $\delta(\ell)$. Indeed, if the other watched literal is not falsified, then backtracking ℓ only does not violate Inv. *watched literals*. If the other watched literal is falsified at a higher level than $\delta(\ell)$, then it will be backtracked as well, and the invariant is preserved. This assumption is not trivial to satisfy, and is discussed later in this section. This can be expressed as Inv. *satisfied watched literal level*. Inv. *satisfied watched literal level* is trivially satisfied in NCB (due to the order of propagations) but must be enforced in CB.

Invariant 4.2.2 (Satisfied watched literal level). Let a clause $C \in \varphi$ be watched by c_1 and c_2. If $c_1 \in \tau$ and $\neg c_2 \in \tau$, then $\delta(c_1) \leq \delta(c_2)$.

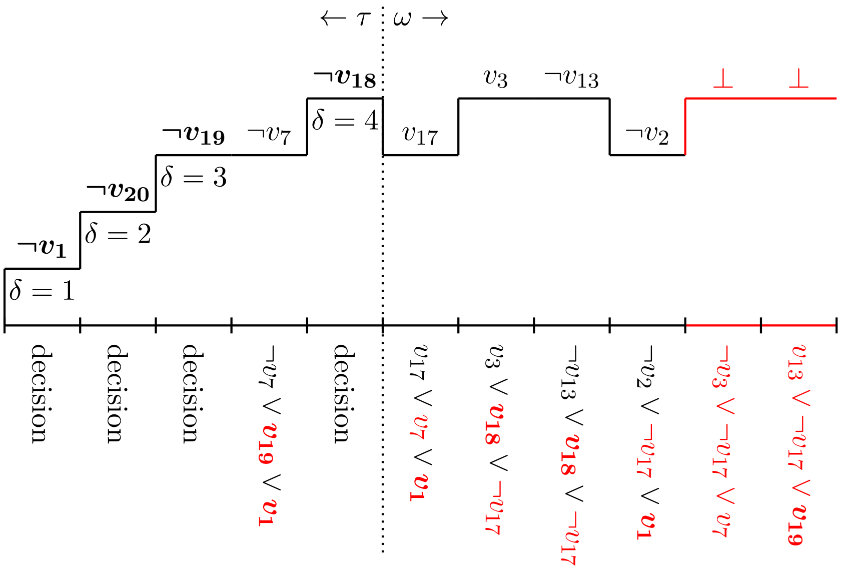
Algorithm 10 Backtracking with Chronological Backtracking

```
1: procedure BACKTRACK($\delta$)
2:     $\beta \leftarrow \emptyset$
3:     while $\delta(\tau) > \delta$ do
4:         $\ell \leftarrow$ POP($\tau$)
5:         if $\delta(\ell) < \delta$ then
6:             $\beta \leftarrow \{\ell\} \cup \beta$
7:     $\tau \leftarrow \tau \cup \beta$
8:     $\beta \leftarrow \emptyset$
9:     while $\omega \neq \emptyset$ do
10:        $\ell \leftarrow$ DEQUEUE($\omega$)
11:        if $\delta(\ell) < \delta$ then
12:            $\beta \leftarrow \beta \cup \{\ell\}$
13:    $\omega \leftarrow \beta$
```

4.2.2 Multiple Unit Clauses

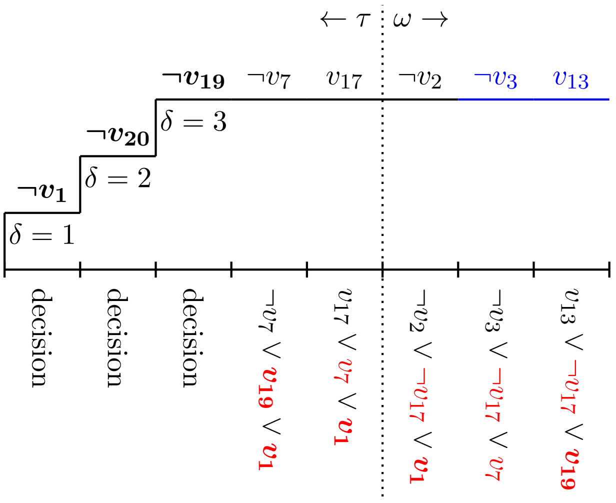
When the solver detects a conflict, it is not always required to perform conflict analysis. Indeed, if the conflicting clause already has a unique literal at level $\delta(C)$, then backtracking to $\delta(C) - 1$ creates a unit clause. [19] suggests to simply backtrack and propagate the literal because C is now unit. It is actually possible to generate multiple unit clauses at once. Fig. 4.2.1 shows such an instance. Therefore, remembering all the conflicts can become useful to check whether some of them became unit clauses after backtracking. In the next subsection, we see that it is actually necessary to remember the conflicts anyway, so this optimization is not costly.

Since conflicts have both watched literals in the propagation queue ω, after backtracking, the clause could be made unit by $\tau \cup \omega$, but not by τ alone. This is acceptable for Inv. *watched* *literals*. Since the conflicts are stored anyway, we might as well take advantage of them. The impact on performance of this optimization is not yet clear and is left for future work.



← τ ⋮ ω →
$\neg v_1$
$\delta = 1$
$\neg v_{20}$
$\delta = 2$
$\neg v_{19}$
$\neg v_7$
$\delta = 3$
$\neg v_{18}$
$\delta = 4$
v_{17}
v_3
$\neg v_{13}$
$\neg v_2$
$\perp$
$\perp$
decision
decision
decision
$\neg v_7 \lor v_{19} \lor v_1$
decision
$v_{17} \lor v_7 \lor v_1$
$v_3 \lor v_{18} \lor \neg v_{17}$
$\neg v_{13} \lor v_{18} \lor \neg v_{17}$
$\neg v_2 \lor \neg v_{17} \lor v_1$
$\neg v_3 \lor \neg v_{17} \lor v_7$
$v_{13} \lor \neg v_{17} \lor v_{19}$

(a) Before backtracking. Two conflicts are found.



← τ ⋮ ω →
$\neg v_1$
$\delta = 1$
$\neg v_{20}$
$\delta = 2$
$\neg v_{19}$
$\neg v_7$
v_{17}
$\delta = 3$
$\neg v_2$
$\neg v_3$
v_{13}
decision
decision
decision
$\neg v_7 \lor v_{19} \lor v_1$
$v_{17} \lor v_7 \lor v_1$
$\neg v_2 \lor \neg v_{17} \lor v_1$
$\neg v_3 \lor \neg v_{17} \lor v_7$
$v_{13} \lor \neg v_{17} \lor v_{19}$

(b) After backtracking and repairing the conflicts. v_{17} can safely be added to the trail.

Figure 4.2.1: Example of multiple unit clauses (from `uf20-0651.cnf`). There are two conflicts at the lowest conflict level (4) when propagating v_{17}. Assume that the clause $\neg v_3 \lor \neg v_{17} \lor v_1$ is returned. It has a single literal at the highest level. Therefore, following the idea described in Sect. 3.2, the solver backtracks to level 3, then propagates v_3. However, we should not forget to propagate v_{13} since $v_{13} \lor \neg v_{17} \lor v_{19}$ also became unit after backtracking.

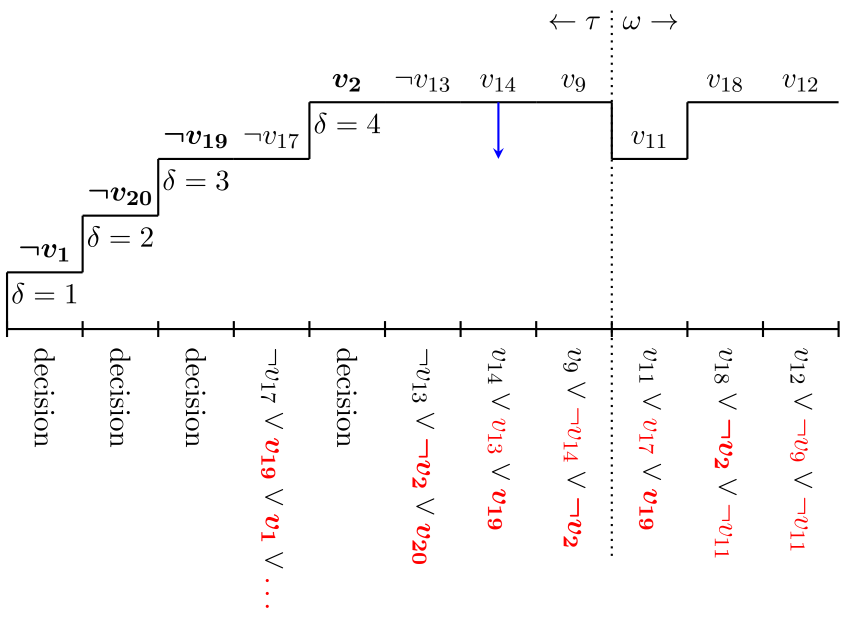

$\leftarrow \tau \vdots \omega \rightarrow$
$\boldsymbol{v_2}$ $\neg v_{13}$ v_{14} v_9
v_{18} v_{12}
$\neg \boldsymbol{v_{19}}$ $\neg v_{17}$
$\delta = 4$
v_{11}
$\neg \boldsymbol{v_{20}}$
$\delta = 3$
$\neg \boldsymbol{v_1}$
$\delta = 2$
$\delta = 1$
decision
decision
decision
$\neg v_{17} \vee \boldsymbol{v_{19}} \vee \boldsymbol{v_1} \vee \ldots$
decision
$\neg v_{13} \vee \neg \boldsymbol{v_2} \vee \boldsymbol{v_{20}}$
$v_{14} \vee v_{13} \vee \boldsymbol{v_{19}}$
$v_9 \vee \neg v_{14} \vee \neg \boldsymbol{v_2}$
$v_{11} \vee v_{17} \vee \boldsymbol{v_{19}}$
$v_{18} \vee \neg \boldsymbol{v_2} \vee \neg v_{11}$
$v_{12} \vee \neg v_9 \vee \neg v_{11}$

Figure 4.2.2: The literal v_{14} is a missed lower implication because the clause $v_{14} \vee \neg v_{11} \vee v_{20}$ became unit at decision level 3 when v_{11} was being propagated.

4.2.3 Missed Lower Implications

As discussed in the previous chapter, it is now possible to generate a trail that satisfies a clause by only one literal at a level higher than all the other literals in the clause. This is called a missed lower implication. The satisfying literal should have been implied at a lower level (since the clause is unit at that level). It was not the case because the clause became unit after it was satisfied. An example of a missed implication is given in Fig. 4.2.2. The goal of fixing missed lower implications is to obtain a trail that satisfies Inv. *satisfied watched literal level*. Indeed, if a clause is satisfied by a literal at level δ, then the other watched literal must not be falsified or belong to a level higher than δ. If it is not the case, then the clause might not satisfy Inv. *watched literals* after backtracking. Every clause that would not satisfy Inv. *satisfied watched literal level* after adding ℓ to τ becomes a missed lower implication before violating the invariant. More concretely, a missed lower implication has at least one literal in ω. Therefore, the clause is not yet made unisat by τ. If missed lower implications are fixed, then the invariant can remain satisfied.

Missed lower implications were documented by [21]. However, their approach is impractical in the context of `veriT`. Indeed, their implementation changes the trail into a linked list structure. However, in `veriT`, it is not really possible to change the trail data structure without a major rewrite of the code. The solver counts over 1300 occurrences of calls to the structure of the SAT solver. Therefore, the method from [21] must be adapted to the current structure of the solver.

Detecting missed lower implications. In order to detect missed lower implications, we introduce a property of clauses that can be enforced when necessary. This is a variation of what [21] refers to as BCP-safety conditions. Previously, the order of the watched literals was not relevant, if C was watched by c_1 and c_2, then c_1 and c_2 could be swapped without any consequences. However, in this new framework, the order of watched literals becomes important. The following property can be imposed:

Property 4.2.1 (Literal ordering). Let $C \in \varphi$ be a clause watched by c_1 and c_2 in that specific order.

- c_1 is unassigned and $\neg c_2 \in \tau \cup \omega$ iff C is made unit by $\tau \cup \omega$ and $\delta(c_2) = \delta(C \setminus \{c_1\})$.
- $\neg c_1 \in \tau \cup \omega$ iff the clause is conflicting and $\delta(c_1) = \delta(C)$, $\delta(c_2) = \delta(C \setminus \{c_1\})$. That is, c_1 and c_2 are respectively the highest and second-highest literals in C.
- $c_1 \in \tau \cup \omega$ and $\neg c_2 \in \tau \cup \omega$ iff the clause is unisat and $\delta(c_2) = \delta(C \setminus \{c_1\})$.

In the following algorithms, we enforce Prop. *literal ordering* with a call to the RestoreLiteralOrder(C) procedure. Detecting conflicts, unit clauses and missed lower implications becomes trivial when Prop. *literal ordering* holds. However, it can be expensive to maintain. Detecting a missed lower implication simply checks that the first watched literal is satisfied and that the second watched literal is falsified at a lower level. If it is the case, then the clause is added to the *re-implication* queue ρ.

When enforcing Prop. *literal ordering* on clauses that see one of their watched literal falsified, and after checking for conflicts, unit clauses or missed lower implications, we ensure the following property:

Property 4.2.2 (Registered missed lower implications). Let a clause $C \in \varphi$ watched by c_1 and c_2 in that order, ℓ be implied but not yet propagated ($\ell \in \omega \wedge \ell \notin \tau$), and $c_1 = \neg\ell \vee c_2 = \neg\ell$. After Prop. *literal ordering* is enforced on C, if $c_1 \in \tau \cup \omega$ and $\neg c_2 \in \tau \cup \omega$, then $\delta(c_1) \leq \delta(c_2)$ or $C \in \rho$.

After re-implication, Prop. *registered missed lower implications* has the additional constraint that $\rho = \emptyset$. Therefore, we have: For every clause C watched by $\neg\ell$, if $c_1 \in \tau \cup \omega$ and $\neg c_2 \in \tau \cup \omega$, then $\delta(c_1) \leq \delta(c_2)$. Since $\ell \in \omega$, ℓ can safely be added to τ without breaking Inv. *satisfied watched literal level*. If $\delta(c_1) > \delta(c_2)$ before re-implication, the clause C is added to ρ, and c_1 is re-implied at level $\delta(c_2)$. Therefore, $\delta(c_1) \leq \delta(c_2)$ holds after re-implication.

Remark 4.2.2. Using the Prop. *literal ordering* breaks the property that states that conflicting clauses see both their watched literals falsified in ω. Indeed, `RestoreLiteralOrder` changes the watched literals to be the highest in the clause (when necessary). It is possible that the highest literal in the clause is not in ω but in τ. Therefore, the property no longer holds. Instead, we can state that conflicts have at least two literals falsified in ω. Since the watched literals were just moved around, the literals falsified in ω are still falsified in ω. This trade off gives us the guarantee that both watched literals will be backtracked if the clause is conflicting.

Dealing with missed lower implications. Once the solver finds a set ρ of reasons C for missed lower implications ℓ, it resolves them. Iteratively, each missed lower implication has its level and reason updated to match the new reason C for the propagation. In turn, clauses watched by $\neg\ell$ can themselves generate missed lower implications. Therefore, each clause in the watch list is checked for the re-implication criterion. This process goes on until the re-implication queue is empty. The procedure is terminating since the levels are bounded by zero, and the levels of literals always decrease.

During the process of checking clauses in the watch list of $\neg\ell$, Prop. *literal ordering* must be enforced again since the level of literals has changed. RestoreLiteralOrder must therefore be efficient since it is done very frequently.

Level of conflicts. Since the level of literals is subject to change when the re-implication procedure is called, we cannot search for the lowest conflict in a greedy fashion (that is, just keep the lowest level conflict and replace it if we find a lower conflict). Instead, we keep track of the set of conflicts found during the search and search for the lowest conflict after the entire literal is propagated, and the re-implication procedure is finished. The set of conflicts is called κ.

Level collapsing. It is possible that a missed lower implication involves a decision literal. In this case, an entire level is collapsed, and the decision levels above must be updated. It might be that, if the decision literal ℓ is re-implied several levels lower than previously, some literals in the collapsed level could be implied at level higher than $\delta(\ell)$. This case is handled by the ReImply procedure. Therefore, collapsing a level δ can only trivially bring down every level greater than or equal to δ by exactly one. All the literals strictly higher than δ must be brought down by one level, and literals at level δ can be descended by one level too. Since a decision is going to disappear, the above decision must fill the void. This is done by the LevelCollapse(δ) procedure in Alg. 11.

Algorithm 11 Level collapse

```
1: procedure LevelCollapse($\delta$)
2:     for $l \in \tau \cup \omega$ do
3:         if $\delta(l) \geq \delta$ then
4:             $\delta(l) \leftarrow \delta(l) - 1$
```

Re-implying in propagation queue. It is possible that the re-implied literal is not yet on the trail τ but is still in the propagation queue ω. In that case, when the re-implication procedure runs, it naturally propagates the literal, which can then be removed from ω and put in τ. In practice, re-implying a literal is the same procedure as to propagate it for the first time. One could assume that the re-implication differs from propagation by the fact that re-implication cannot detect conflicts. That statement is wrong. We just established that it is possible that the re-implication propagates a literal for the first time. Therefore, it might detect a conflict. However, if the literal was already propagated in τ and still watches a clause, then the clause must have been a conflict or a unit clause. In that case, conflict analysis would have been triggered earlier, or another literal would have been added to ω to satisfy the unit clause. Therefore, the re-implication of literals in τ cannot detect conflicts. This does not justify the fact to duplicate the propagation procedure.

Remark 4.2.3. In practice, the statement above is not perfectly accurate. It is possible that re-implying a propagated literal encounters a conflict, but this conflict has been discovered before. For example (in Fig. 4.2.3), the conflict $v_5 \vee v_9 \vee v_{16}$ was detected when propagating $\neg v_{16}$. But in addition to the conflict, the solver also noticed some missed lower implications.

When it re-implied $\neg v_9$ ($\neg v_9$ in τ), the conflict was met again. Notice that the distinction between τ and ω is no longer marked by the dotted line. This is because the assignment must be reordered before re-implication. It is discussed in the next subsection. For the sake of this example, the only necessary piece of information to understand the example is that $\boxed{\textit{boxed}}$ literals are in ω and the others are in τ.

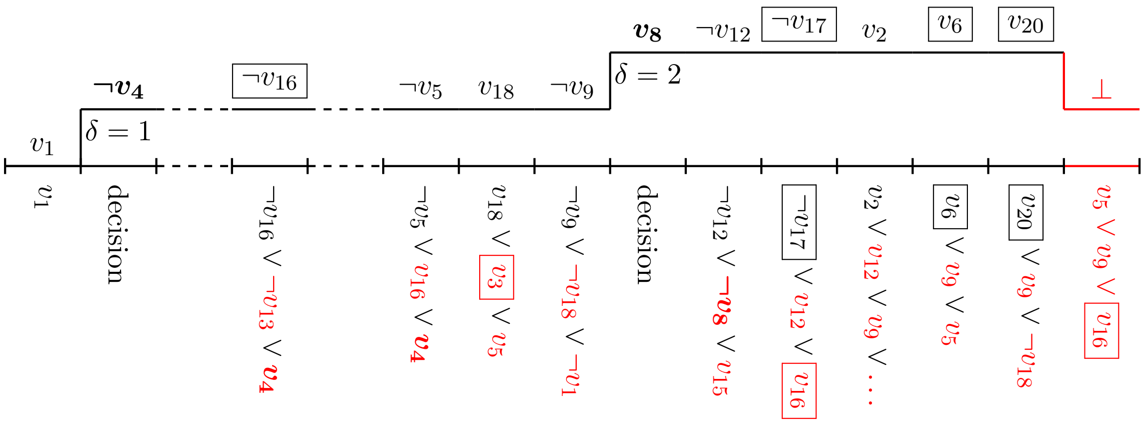
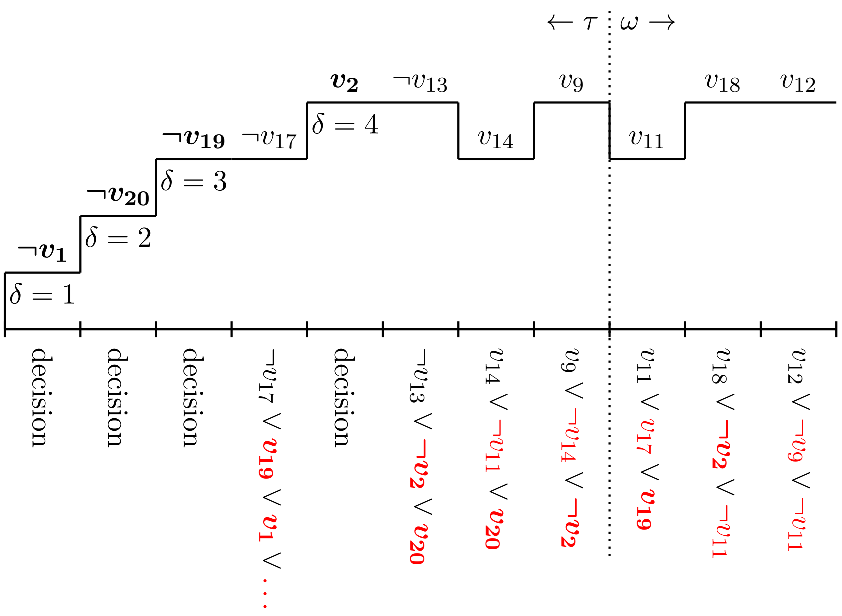

v_1
$\delta = 1$
$\neg \boldsymbol{v_4}$
$\boxed{\neg v_{16}}$
$\neg v_5$
v_{18}
$\neg v_9$
$\delta = 2$
$\boldsymbol{v_8}$
$\neg v_{12}$
$\boxed{\neg v_{17}}$
v_2
$\boxed{v_6}$
$\boxed{v_{20}}$
$\perp$
v_1
decision
$\neg v_{16} \vee \neg v_{13} \vee \boldsymbol{v_4}$
$\neg v_5 \vee v_{16} \vee \boldsymbol{v_4}$
$v_{18} \vee \boxed{v_3} \vee v_5$
$\neg v_9 \vee \neg v_{18} \vee \neg v_1$
decision
$\neg v_{12} \vee \neg \boldsymbol{v_8} \vee v_{15}$
$\boxed{\neg v_{17}} \vee v_{12} \vee \boxed{v_{16}}$
$v_2 \vee v_{12} \vee v_9 \vee \ldots$
$\boxed{v_6} \vee v_9 \vee v_5$
$\boxed{v_{20}} \vee v_9 \vee \neg v_{18}$
$v_5 \vee v_9 \vee \boxed{v_{16}}$

Figure 4.2.3: Example of multiple detections of identical conflict (from `uf20-0355.cnf`). While propagating $\neg v_{16}$ at level 1, the solver noticed a conflict $v_5 \vee v_9 \vee v_{16}$ and some re-implications. When $\neg v_9$ and $\neg v_5$ were re-implied, the solver found the same conflict.

Remark 4.2.4. A reader with a keen eye might notice that the conflict does not have two literals falsified in ω. This is because when the conflict was originally discovered, $\neg v_5$ and $\neg v_{16}$ were still in ω. When $\neg v_5$ was re-implied, $\neg v_5$ was moved to τ since it was considered to be propagated. This is why the diagram does not show $\neg v_5$ in ω. Since the conflict is registered in κ, $\neg v_5$ is going to be backtracked in any case. Prematurely adding it to τ makes the implementation simpler without making a difference after recovering from the conflict.

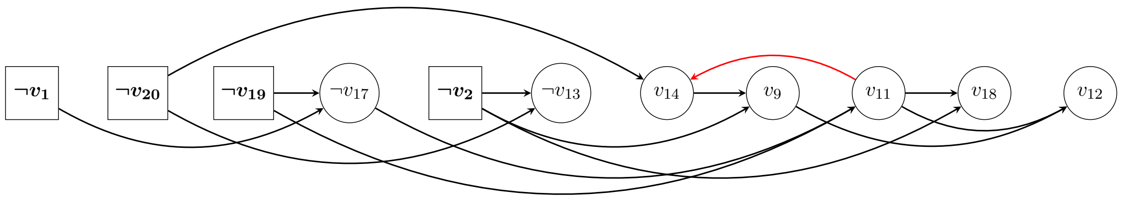
4.2.4 Topological Consistency

When we discussed conflict analysis, we briefly mentioned that the algorithm used to compute the first UIP makes the assumption that the trail is in topological order with respect to the implication graph. In other words, a literal ℓ at position p in the trail $\tau \cup \omega$ must have a reason C whose literals are all located before p in the trail $\tau \cup \omega$. In NCB, this is trivially true. Indeed, the trail behaves exactly like a stack. Only the tail is modified either by removing literals, which does not break the topological order, or by pushing literals. Pushed literals are either decisions, which have no parent in the graph (therefore not breaking the property), or implied literals, which are implied by a clause that is unit under the assignment $\tau \cup \omega$; therefore following the topological order. In CB without re-implication, this property also holds. The only difference is that literals can be removed in the middle of the assignment. In general, this could be an issue; however, literals are removed from the highest levels to the lowest; and if one literal at level δ is removed, then all literals above level δ are removed as well. Furthermore, literals at a lower level than δ cannot depend on literals above δ by construction. Therefore, after removing every literal above δ, the topological order of the remaining assignment persists.



$\leftarrow \tau$
$\omega \rightarrow$
$\neg\boldsymbol{v_1}$
$\neg\boldsymbol{v_{20}}$
$\neg\boldsymbol{v_{19}}$
$\neg v_{17}$
$\boldsymbol{v_2}$
$\neg v_{13}$
v_{14}
v_9
v_{11}
v_{18}
v_{12}
$\delta = 1$
$\delta = 2$
$\delta = 3$
$\delta = 4$
decision
decision
decision
$\neg v_{17} \lor \boldsymbol{v_{19}} \lor \boldsymbol{v_1} \lor \ldots$
decision
$\neg v_{13} \lor \neg\boldsymbol{v_2} \lor \boldsymbol{v_{20}}$
$v_{14} \lor \neg v_{11} \lor \boldsymbol{v_{20}}$
$v_9 \lor \neg v_{14} \lor \neg\boldsymbol{v_2}$
$v_{11} \lor v_{17} \lor \boldsymbol{v_{19}}$
$v_{18} \lor \neg\boldsymbol{v_2} \lor \neg v_{11}$
$v_{12} \lor \neg v_9 \lor \neg v_{11}$

(a) The trail after re-implying v_{14} at level 3. The reason for v_{14} is now $v_{14} \lor \neg v_{11} \lor v_{20}$.



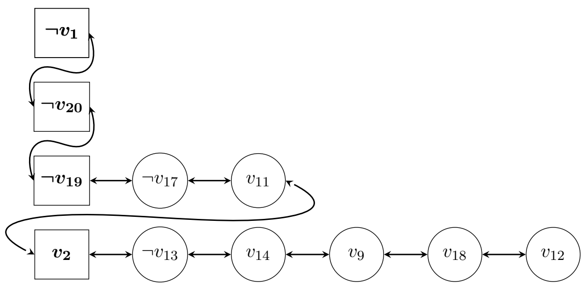
(b) Implication graph of the trail in Fig. 4.2.4a.

Figure 4.2.4: Fixing the missed lower implications in Fig. 4.2.2 messes up the topological order. The reason for v_{14} now depends on v_{11}, which is located after v_{14} in the trail.

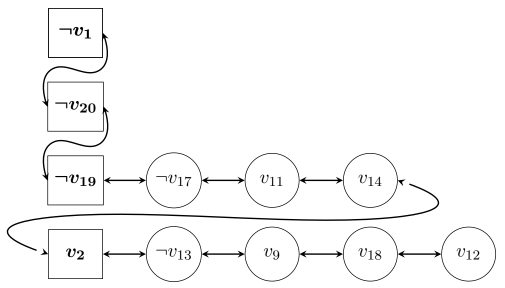
When adding re-implication, this property no longer holds. Indeed, when a literal ℓ is re-implied from level δ to δ' ($\delta > \delta'$), it means that a clause C is unisat with ℓ at level δ'. The clause C can only have become unisat after ℓ was added to the propagation queue ω, otherwise, ℓ would have been propagated earlier. Generally (see Rem 4.2.5), that means that one of the literals in C is located after ℓ in $\tau \cup \omega$. Therefore, changing the reason of ℓ to be C breaks the topological order. An example of this can be seen on Fig. 4.2.4. Literal v_{14} is being re-implied. The reason for v_{14} no longer respects the topological order and depends on v_{11} that is positioned after v_{14} in the trail.

Remark 4.2.5. Although it is possible that ℓ is implied twice at different levels during another literal's propagation, leading to a re-implication without breaking the topological order, this is not usually the case.

To overcome this problem, two solutions can be considered: (1) We modify the conflict analysis to no longer need the topological order. (2) We restore the topological order. Since it is usually dangerous to blindly break invariants that can be preserved without too much computation costs, and we know from [21] that restoring the order is a fine strategy, we



(a) Trail before re-implication.



(b) After re-implying $\neg v_{14}$, the trail is updated. The order of literals is indeed a topological sort of the implication graph.

Figure 4.2.5: Trail represented as a doubly linked list. Squares represent decisions, circles represent implied literals. A pointer to each decision is kept in a decision stack for fast access.

decided to implement the second solution. Some future work could consist in implementing the first approach. In Chap. SMT solving, we discuss the fact that it might be a preferable idea in the context of SMT.

Invariant 4.2.3 (Topological consistency). The assignment $\tau \cup \omega$ is a topological order of the implication graph of $\tau \cup \omega$.

Solution with Linked Lists [21]

[21] changes the paradigm of the trail by representing it as a doubly linked list with pointers to every decision. This structure allows for cheap insertion and deletion of elements in the middle of the trail. The main advantage is that it allows to

- keep the trail ordered by decision level,
- cheaply preserve the Inv. *topological consistency*,
- cheaply backtrack to a given level.

When re-implying a literal, it suffices to extract the literal from its position in the linked list, and to insert it at the end of its new level. This is done in $O(1)$ if the location of the literal is known in the list. The example of Fig. 4.2.4 is illustrated on Fig. 4.2.5.

Reordering the Assignment

Since `veriT` is not a standalone SAT solver, we cannot change the data structure of the trail. However, we can still modify the semantics of the data structure, and reorder the trail to preserve Inv. *topological consistency*. The problem is that we cannot simply change the order of the trail since, as discussed previously, the trail τ and the propagating queue ω share the same memory space (a C array). The only distinction between the two is a pointer to the first element of ω. Obviously, changing the order of the trail might interleave propagated and non-propagated literals. Therefore, we need a new way of interpreting the data structure. In practice, most algorithms treat the full partial assignment $\pi = \tau \cup \omega$ together. Therefore, we can change the meaning of the array to be $\pi = \tau \cup \omega$ in an order that would suit other needs. However, we still need to be able to distinguish between τ and ω. To do so, we add a circular queue that keeps track of the literals that are in ω. The meaning of ω remains unchanged, the only difference is that τ is no longer directly accessible, and should be accessed as $\tau = \pi \setminus \omega$. Changing the meaning of the array does not have any impact on the interaction between the SAT solver and the rest of the program, since the SMT solver only queries the partial assignment when all the literals in ω have been propagated ($\omega = \emptyset$). That is, the SMT solver queries the partial assignment when $\pi = \tau$. However, changing the order of literals impacts the behavior of the SMT solver because it assumes that the partial assignment behaves like a stack. This would be true no matter the data structure used (linked list or array).

The goal of the reordering is to ensure that once a literal is re-implied, its position still follows the topological order. To do so, we shall proceed in two steps. First, before any re-implication, we sort the trail by decision level. This ensures that any re-implication does not break the topological order. Indeed, since all the literal at a decision level δ' are before any literal at level δ if $\delta > \delta'$, if a literal is re-implied from level δ to δ', it is located after all the literals that could be in the reason for the re-implication. Then, during successive re-implication, we wish to keep the topological and decision level order. To that effect, we *sink* the newly re-implied (and newly implied) literals to be on top of their new level.

The assignment reordering can be done in linear time with respect to the length of the partial assignment with Alg. 12. It is important that this sorting algorithm is stable, i.e., at the same level, the literals remain in the same order. An example of the sinking procedure is shown in Fig. 4.2.6. First, the trail must be reordered by levels. Then, the literal v_{18} is re-implied at level 1, and sunk to the appropriate position.

Reordering the trail is used to obtain the property Prop. *assignment ordering*. This is used during re-implication to ensure that re-implying a literal does not break the topological order.

Property 4.2.3 (Assignment ordering). The assignment $\pi = \tau \cup \omega$ is ordered by decision levels.

Remark 4.2.6. To make this procedure faster, the location of each decision in the stack is remembered. In ReorderAssignment(), this is the goal fulfilled by Δ'. It requires a bit of bookkeeping, but with that information, we get two advantages: (1) The new location of a sunk literal is known directly. It replaces the decision one level above its new level. (2) Searching the initial position of the sunk literal is cheaper if we know that it is located after the decision of its current level. This detail is not represented in the algorithm, but it is implemented in `veriT`.

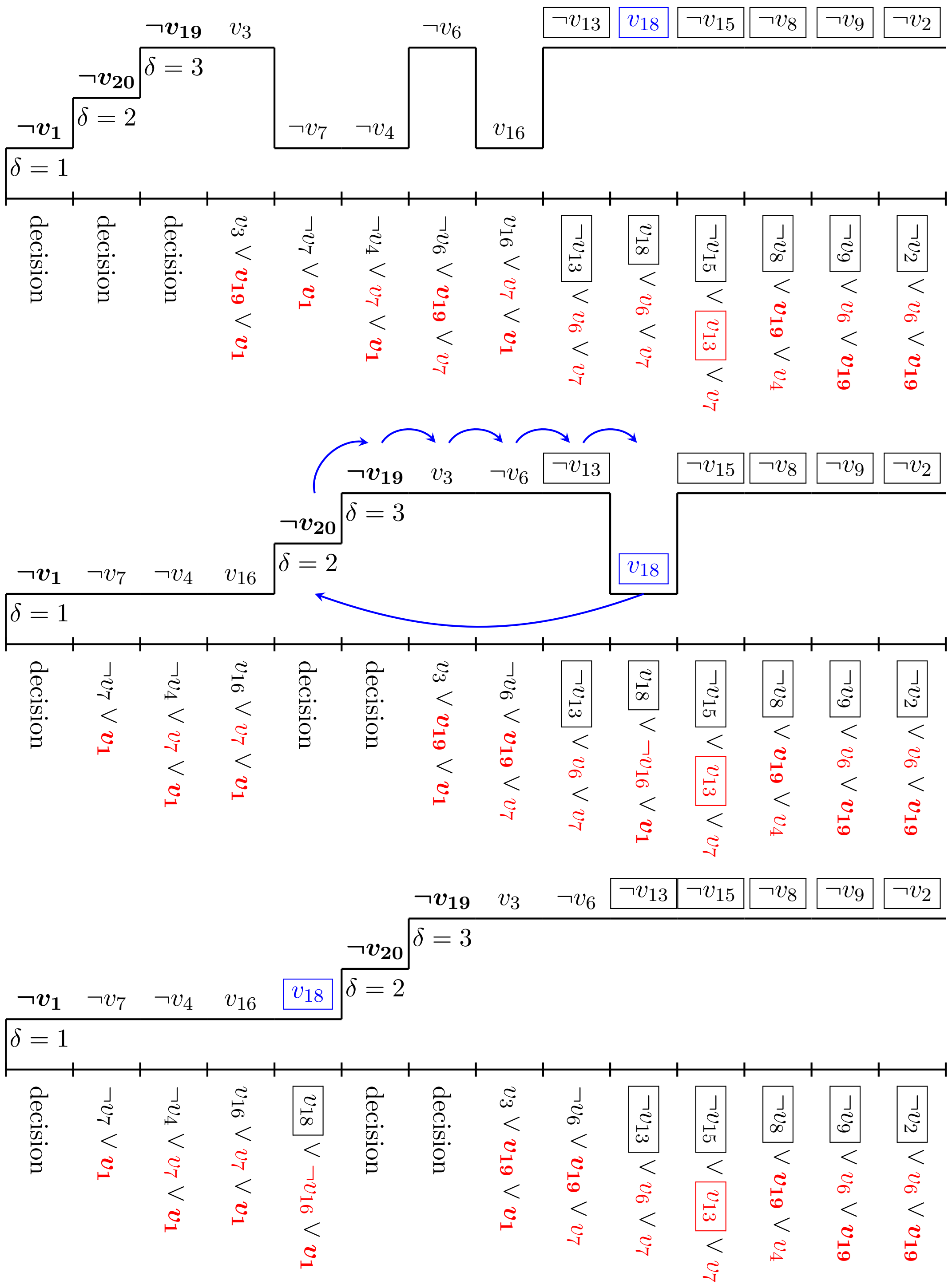


Figure 4.2.6: Example of a sinking literal (from `uf20-0024.cnf`). The literal v_{18} is re-implied and must be sunk to preserve the order of the assignment.

Algorithm 12 Reorder the partial assignment

```
1:  procedure ReorderAssignment()
2:      $\Delta \leftarrow$ array of zeros of length $\delta(\pi) + 1$
3:      for $\ell \in \pi$ do
4:          $\Delta[\delta(\ell)] \leftarrow \Delta[\delta(\ell)] + 1$
5:                                              ▷ $\Delta$ contains the number of literals at each level
6:      $\Delta' \leftarrow \Delta$
7:      for $\delta \in 1, \dots, \delta(\pi)$ do
8:          $\Delta'[\delta - 1] \leftarrow \Delta'[\delta - 1] + \Delta'[\delta]$   ▷ $\Delta'[\delta]$ contains the index of the first literal of level $(\delta + 1)$ (if $\delta < \delta(\pi)$)
9:      $\pi' \leftarrow \pi$
10:     for $\ell \in \pi'$ do
11:         $\pi[\Delta'[\delta(\ell)] - \Delta[\delta(\ell)]] \leftarrow \ell$
12:         $\Delta[\delta(\ell)] \leftarrow \Delta[\delta(\ell)] - 1$
13: procedure SinkLiteral($\ell, \pi$)
14:     $\pi \leftarrow \pi \setminus \{\ell\}$
15:     $\pi_{\text{left}} \leftarrow \{\ell' \in \pi : \delta(\ell') \leq \delta(\ell)\}$
16:     $\pi_{\text{right}} \leftarrow \pi \setminus \pi_{\text{left}}$
17:     $\pi \leftarrow \pi_{\text{left}} \cup \{\ell\} \cup \pi_{\text{right}}$
```

4.3 Complete Algorithm

In this section, we introduce the algorithms used in `veriT` to convert the SAT solver to chronological backtracking. We follow both the required and recommended changes to obtain the final algorithm.

CDCL. CDCL is modified to take into account the new meaning for the partial assignment π, and new data structures used to store the propagation queue ω, the re-implication queue ρ and the set of conflicts κ. The algorithm is presented in Alg. 13. Similarly to NCB CDCL, the loop first calls BCP, then if no conflict is found, it decides a new literal. If no such literal exist, then π is complete and a model was found. However, if some conflict is found, there might be several in the set κ. This is handled by the RepairConflict() procedure. If no conflict is found, the algorithm takes a decision and continues.

Backtracking. Since the data structure used changed, the Backtracking procedure must also be adapted. Alg. 14 shows the small changes required. This is not very sophisticated but was added for the sake of completeness. In essence, we no longer consider the trail and the propagation queue separately, but consider the partial assignment as a whole. At the end, we still need to filter out backtracked literals in the propagation queue.

Conflict handling. It is possible that several conflicts are detected at the same time. In this case, we need to search for the lowest conflict C, such that resolving it also resolves all the others in κ. As described in Alg. 7, it is possible that C is already a valid learned clause, that is, it has only one literal at level $\delta(C)$. In which case the conflict analysis is not triggered.

Algorithm 13 CDCL with missed lower implication and multiple conflict detection

```
1: $\varphi \leftarrow \emptyset$                          ▷ The set of clauses
2: $\pi \leftarrow \emptyset$                          ▷ The partial assignment
3: $\omega \leftarrow \emptyset$                          ▷ The propagation queue
4: $\rho \leftarrow \emptyset$                          ▷ The re-implication queue
5: $\kappa \leftarrow \emptyset$                          ▷ The set of conflicts
6: procedure CDCL($\varphi'$)
7:     $\varphi \leftarrow \varphi'$                      ▷ Set the input clauses
8:     while $\top$ do
9:         BCP()
10:        if $\kappa = \emptyset$ then              ▷ No conflict detected
11:            $\ell \leftarrow$ DECIDE()
12:            if $\ell = \bot$ then              ▷ No more literals to propagate
13:                return $\pi$
14:            $\delta(l) \leftarrow \delta(\pi) + 1$
15:            $\pi^d \leftarrow \pi^d \cup \{\ell\}$          ▷ Add $\ell$ to the decision set. $\pi^d$ does not have
                                              a proper data structure but is a metadata
                                              of $\pi$.
16:            $\omega \leftarrow \omega \cup \{\ell\}$
17:            $\pi \leftarrow \pi \cup \{\ell\}$
18:        if REPAIRCONFLICT() $= \bot$ then    ▷ Conflict at level 0
19:            return $\bot$
```

Algorithm 14 Backtracking with Chronological Backtracking

```
1: procedure BACKTRACK($\delta$)
2:     $\beta \leftarrow \emptyset$
3:     while $\delta(\pi) > \delta$ do
4:         $\ell \leftarrow$ POP($\pi$)
5:         if $\delta(\ell) < \delta$ then
6:             $\beta \leftarrow \{\ell\} \cup \beta$
7:         else
8:             $\delta(\ell) \leftarrow \infty$
9:     $\pi \leftarrow \pi \cup \beta$
10:    $\omega \leftarrow \omega \cap \pi$                  ▷ Remove all literals that are no longer in $\pi$
```

If it is triggered, a first backtracking step is performed to set up the partial assignment π for analysis.

A difference with Alg. 7 resides in the fact that the set κ can now contain several unit clauses. (The learned clause qualifies as a conflicting clause.) Therefore, we need to search for all unit clauses in κ and propagate them. This is done in lines 9 to 21 of Alg. 15.

Remark 4.3.1. It is possible that a literal ℓ is implied by two clauses C_1 and C_2 ($\delta(C_1) > \delta(C_2)$) that were conflicting before backtracking. ℓ might be implied by C_1 because it became unit, then C_2 detects a missed lower implication and re-implies ℓ. In this case, re-implication is almost free. Indeed, it is not possible that one of the falsified literal in C_2 was propagated after ℓ. Literals that may be propagated after ℓ all have a reason that was in κ. Furthermore, if a literal ℓ' is in a clause $C \in \kappa$, then no clause in κ can contain $\neg\ell'$ (all clauses in κ were conflicting with the same assignment π). Therefore, flipping the polarity of one literal in $C \in \kappa$ cannot falsify a literal in another clause in κ. The order of literals propagated by the procedure does not matter, and the reason and level for ℓ can be changed without further modification to the assignment. The re-implication queue ρ is not needed here.

Algorithm 15 Conflict handling with Chronological Backtracking

```
 1: procedure RepairConflict()
 2:     $C \leftarrow$ LowestConflict($\kappa$)            ▷ Get the lowest conflict in $\kappa$
 3:     if $\delta(C) = 0$ then                          ▷ Conflict at level 0
 4:         return $\bot$
 5:     if $C$ has more than one literal at level $\delta(C)$ then
 6:         Backtrack($\delta(C)$)                       ▷ Clean $\pi$ such that conflict analysis is simpler
 7:         $\kappa \leftarrow \kappa \cup$ Analyze($C$)
 8:     Backtrack($\delta(C) - 1$)
 9:     for $C' \in \kappa$ do                           ▷ Search all unit clauses for propagation
10:         RestoreLiteralOrder($C'$)                    ▷ $c'_1$ cannot be falsified
11:         $c'_1 \leftarrow$ the first watched literal of $C'$
12:         $c'_2 \leftarrow$ the second watched literal of $C'$
13:         if $c_1 \in \pi$ then
14:             if $\neg c'_2 \in \pi \wedge \delta(c'_1) > \delta(c'_2)$ then   ▷ Missed lower implication
15:                 $\delta(c'_1) \leftarrow \delta(c'_2)$
16:                 SetReason($c'_1, C'$)
17:         else if $\neg c'_2 \in \pi$ then             ▷ Unit clause
18:             $\delta(c'_1) \leftarrow \delta(C \setminus \{c'_1\})$
19:             SetReason($c'_1, C'$)
20:             $\omega \leftarrow \omega \cup \{c'_1\}$
21:             $\pi \leftarrow \pi \cup \{c'_1\}$
22:     return $\top$
```

Literal propagation. In this new algorithm, we separate the literal propagation from the BCP procedure to avoid code duplication and improve readability. In PropagateLiteral(ℓ, `sink`), the SAT solver checks all the clauses watched by the literal ℓ and searches for unit clauses, conflicts and missed lower implications. This method is presented in Alg. 16. In addition to the standard propagation procedure, the algorithm takes a Boolean argument `sink`. If `sink` is true, then literals added to ω are sunk to the end of their level in π. This is used to restore the order of the partial assignment during re-implication. SinkLiteral(ℓ) is presented in Alg. 12. If `sink` is true, then the partial assignment is assumed to be ordered by level. When adding the literal c_2, we need to restore this property.

As explained earlier, Prop. *literal ordering* makes it very easy to check for conflicts, unit clauses and missed lower implication. However, each clause must first restore the property before it can be checked. It seems like an expensive work, however, it is actually similar to the procedure called to replace the falsified watched literal in the clause. Despite being similar to the literal replacement step in NCB, a few factors make it more expensive: (1) Both watched literals can be modified by RestoreLiteralOrder(C), but the propagation only knows the location of the clause in one of the watch lists. Therefore, the other may need to be searched for. (2) The procedure must reason over the levels of literals in the clause. This is more expensive than simply checking the assignment of the literal. However, this is still a constant time operation, the coefficient is a bit higher nonetheless. (3) The procedure stops to enumerate literals in the clause when a satisfactory literal pair (c_1, c_2) is found (to satisfy Prop. *literal ordering*). This requires to look at more literals in the clause than finding a non-falsified literal. One idea for future work to mitigate this cost would be to let the clause in the replaced literal's watch list until it is read. When iterating over clauses in the watch list of ℓ, if neither watched literal is ℓ, then skip the clause. This will naturally remove it from the watch list without need to search for it. This idea is not implemented in `veriT` yet.

Algorithm 16 Literal propagation with missed lower implications and literal sinking

```
procedure PropagateLiteral($\ell$, sink)
    for $C$ in watched clauses of $\neg\ell$ do
        RestoreLiteralOrder($C$)
        $c_1 \leftarrow$ first watched literal of $C$
        $c_2 \leftarrow$ second watched literal of $C$
        if $\neg c_1 \in \pi$ then                              ▷ Conflict
            $\kappa \leftarrow \kappa \cup \{C\}$
        else if $c_1 \in \pi$ then                             ▷ Unisat clause
            if $\neg c_2 \in \pi \wedge \delta(c_1) > \delta(c_2)$ then    ▷ Missed lower implication
                $\rho \leftarrow \rho \cup \{C\}$
        else if $\neg c_2 \in \pi$ then                         ▷ Unit clause
            $\delta(c_1) \leftarrow \delta(C \setminus \{c_1\})$
            SetReason($c_1, C$)
            $\omega \leftarrow \omega \cup \{c_2\}$
            $\pi \leftarrow \pi \cup \{c_2\}$
            if sink then
                SinkLiteral($c_2$)                              ▷ Preserve the ordering
```

BCP. The BCP procedure must now be modified to take into account the re-implication. Rather than returning a single conflict, the conflicts detected by PropagateLiteral(ℓ, `sink`) are stored in κ and handled by RepairConflict(). After propagating a literal, BCP attempts to resolve any missed lower implication by running ReImply(). The new BCP is presented in Alg. 17. During BCP, there is no need to sink literals popped from ω. The order of literals on the stack respects the topological order of the implication graph by construction, and the level ordering is not important here.

Algorithm 17 BCP with missed lower implication and trail reordering

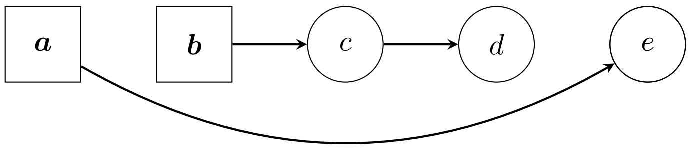
```
1: procedure BCP()
2:     while $\omega \neq \emptyset$ do
3:         $\ell \leftarrow$ First($\omega$)
4:         PropagateLiteral($\ell$, false)
5:         if $\rho \neq \emptyset$ then
6:             ReImply()
7:         if $\kappa \neq \emptyset$ then
8:             return
9:     $\omega \leftarrow \omega \setminus \{\ell\}$
```

Re-implication. The re-implication procedure described in Alg. 18 is similar to the one described by [21]. However, it was adapted to match our data structures and reordering aspects that they did not need to consider because of their doubly linked list structure. The first step is to reorder the assignment. Indeed, since the reason of literals is going to change, we need to ensure some order to find cheaply where the re-implied literal has to be moved. Furthermore, it is possible that simply moving one literal is not enough to restore Inv. *topological consistency*. For example, the implication graph shown on Fig. 4.3.1 cannot restore the topological order by moving c only. This is why the first step of re-implication is to order the trail by level. This is done by the method OrderAssignment(). Then, the re-implication queue is progressively emptied by re-implying the necessary literals. Once a literal is re-implied, it must be sunk, then propagated again, enabling the `sink` option. This ensures that the level ordering of the assignment is still enforced even if propagation implies new literals on ω.

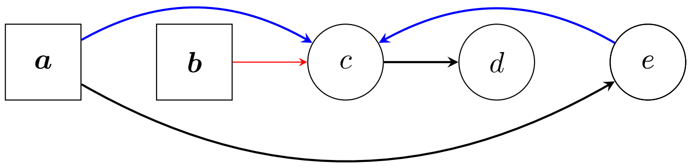
If the re-implied literal ℓ was a decision, then the LevelCollapse($\delta(\ell)$) procedure is called. The literal is no longer a decision and all the literals above must be re-implied after the decisions at level δ have been re-assigned at level $\delta - 1$. This process can be made more efficient by directly subtracting one level to all the literals above $\delta(\ell)$, and then re-implying ℓ. Some literals will not require further re-implication.

Re-implication at work. A lengthy example of execution of the re-implication procedure is shown in Figs. 4.3.2 and 4.3.3. For the sake of readability, we write that the re-implication queue contains literals, when it really contains clauses. When the re-implication queue contains a clause C that implies ℓ at a lower level, we write that $\ell \in \rho$. This abuse of notation allows to more concisely describe the example.

While propagating v_4, the clauses $v_{10} \vee \neg v_4 \vee \neg v_5$ and $v_{14} \vee \neg v_4 \vee \neg v_2$ were looked at. They are both missed lower implications and $\rho = \{v_{14}, v_{10}\}$. The re-implication procedure is triggered, and the assignment is reordered. v_{10} is re-implied at level 3. Since it was a decision, all the literals of level 5 are collapsed to level 4. The re-propagation of v_{10} detects that $\neg v_{15}$ and v_{11} can be re-implied at level 3 too. At this stage, the re-implication queue ρ contains



(a) Initial implication graph



(b) Implication graph after re-implying c with the reason $c \vee \neg a \vee \neg e$. The blue edges replace the red one. It is not possible to recover a topological order by moving c only.

Figure 4.3.1: Reordering the assignment is necessary to preserve the topological order.

Algorithm 18 Reimplying literals

```
 1: procedure REIMPLY()
 2:     REORDERASSIGNMENT()
 3:     while $\rho \neq \emptyset$ do
 4:         $C \leftarrow$ POP($\rho$)                         ▷ We know that $C$ is unisat
 5:         RESTORELITERALORDER($C$)
 6:         $c_1 \leftarrow$ first watched literal of $C$     ▷ Uniquely satisfied literal
 7:         $c_2 \leftarrow$ second watched literal in $C$    ▷ Highest falsified literal
 8:         if $\delta(c_1) \leq \delta(c_2)$ then            ▷ The clause is not a missed lower implica-
                                                              tion anymore
 9:             continue
10:         $\delta \leftarrow \delta(c_1)$                   ▷ Remember to collapse the level if neces-
                                                              sary
11:         $\delta(c_1) \leftarrow \delta(c_2)$
12:         SETREASON($c_1, C$)
13:         SINKLITERAL($c_1$)
14:         if $c_1 \in \omega$ then
15:             $\omega \leftarrow \omega \setminus \{c_1\}$  ▷ No need to propagate it twice.
16:         PROPAGATELITERAL($c_1$, true)
17:         if $c_1 \in \pi^d$ then                       ▷ Decision literal
18:             $\pi^d \leftarrow \pi^d \setminus \{c_1\}$
19:             LEVELCOLLAPSE($\delta$)
```

reasons for $\{v_{14}, \neg v_{15}, v_{11}\}$. When re-implying v_{11}, two interesting behaviors occur: (1) v_{11} was in the propagation queue; re-implying it effectively propagates it. Therefore, it can be removed from ω. (2) $\neg v_{15}$ is detected as a missed lower implication once again. Another reason for $\neg v_{15}$ is pushed to ρ. $\rho = \{v_{14}, \neg v_{15}, \neg v_{15}\}$ with two different reasons for $\neg v_{15}$. This is why on Fig. 4.3.3b, the reason for $\neg v_{15}$ is different from before, while the previous reason was also a valid clause for re-implication. When the re-implication procedure meets another reason for $\neg v_{15}$, it is not re-implied again since the clause no longer represents a missed lower implication. Finally, v_{14} is re-implied, collapsing level 4 since it was a decision. The re-implication queue is empty and the procedure terminates.

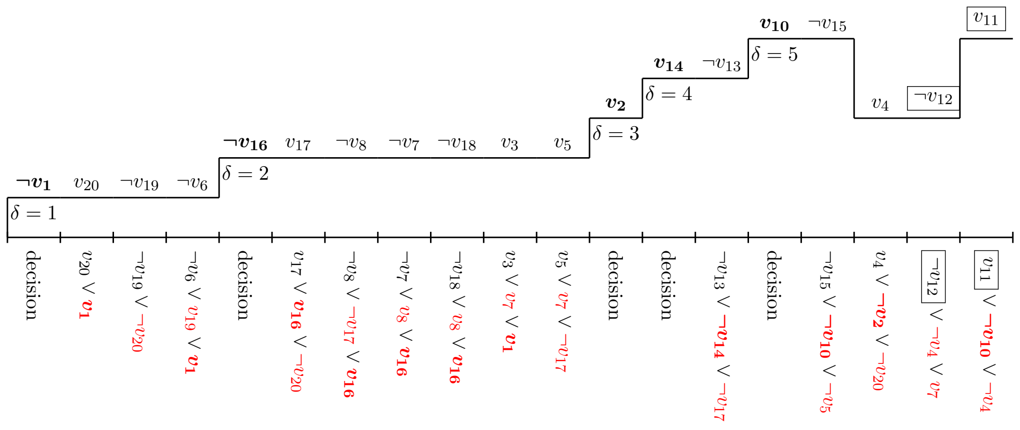
As can be seen in the example, the procedure could have been faster if both decisions were collapsed and re-implied before the other literals. This might be an interesting optimization to consider in the future.

Conflict analysis. Because of the reordering procedure, we have ensured that the trail remains in a topological order of the implication graph. However, it is still possible to have an interleaving of levels in the trail π. Therefore, conflict analysis must skip literals at lower levels. This small change is illustrated in Alg. 19. The conflict analysis procedure is otherwise unchanged. This modification is necessary regardless of the re-implication procedure. If no re-implication is performed, then the trail possibly still has interleaving of levels.

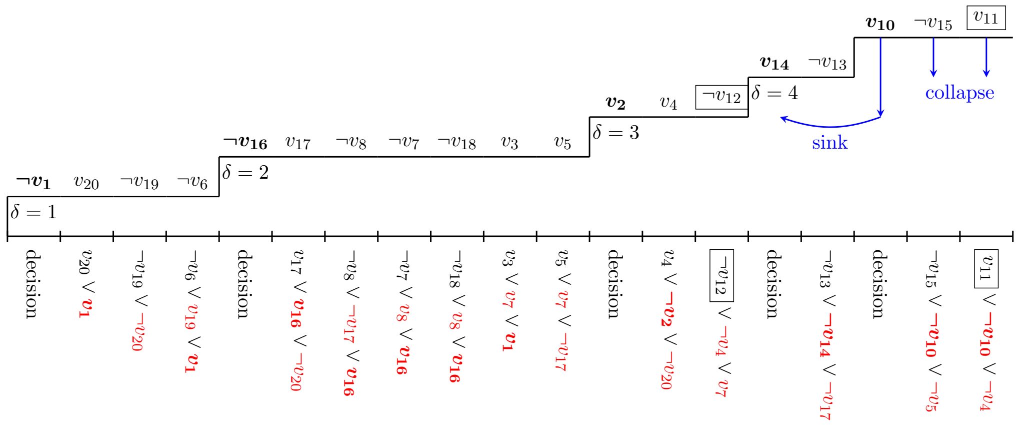
Algorithm 19 Conflict analysis with Chronological Backtracking

```
1:  procedure ANALYZE($C$)
2:      trail ← $\pi$                                   ▷ trail the array version of $\pi$
3:      $C' \leftarrow C$                                ▷ $C'$ is the current learned clause
4:      $n \leftarrow$ the number of literals in $C$ at $\delta(\tau)$
5:      $i \leftarrow |$trail$| - 1$                     ▷ $i$ is the index of the last literal in trail
6:      while $n > 1$ do
7:          while $\delta($trail$[i]) < \delta(C) \lor \neg$trail$[i] \notin C'$ do   ▷ Find the last literal of $C$ in trail
8:              $i \leftarrow i - 1$
9:          $C' \leftarrow C' \setminus \{\neg$trail$[i]\}$          ▷ Remove the literal trail$[i]$
10:         $n \leftarrow n - 1$
11:         $C' \leftarrow$ GETREASON(trail$[i]$)           ▷ Get the clause that implied trail$[i]$
12:         for $\ell \in C' \setminus \{$trail$[i]\}$ do
13:             $C' \leftarrow C' \cup \{\ell\}$              ▷ Avoid duplicates in practice
14:             if $\delta(\ell) = \delta(\tau)$ then
15:                 $n \leftarrow n + 1$
16:     return $C'$
```

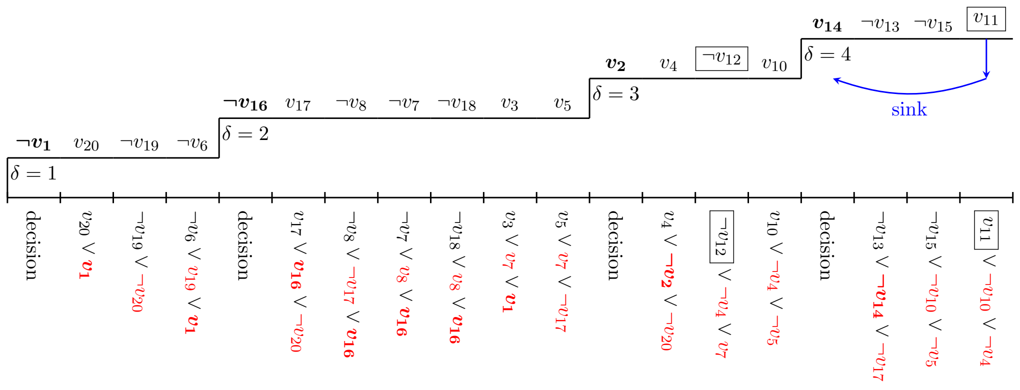
Purge. As a bonus, the assumption that watched literals cannot be falsified at level 0 unless the clause is conflicting at level 0 is restored. Therefore, the bug in the purge from `veriT` provoked by the conversion to CB was fixed naturally.



(a) Trail before re-implication. The re-implication queue ρ contains new reasons for $\{v_{14}, v_{10}\}$.

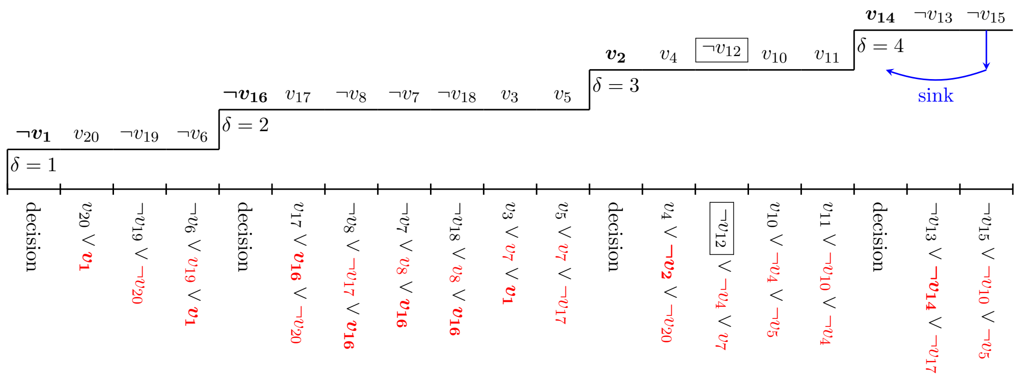


(b) Trail after reordering. v_{10} is waiting to be re-implied, collapsing level 5 to level 4. When re-implying v_{10}, v_{15} and $\neg v_{11}$ are detected to be a missed lower implication. $\rho = \{v_{14}, \neg v_{15}, v_{11}\}$.

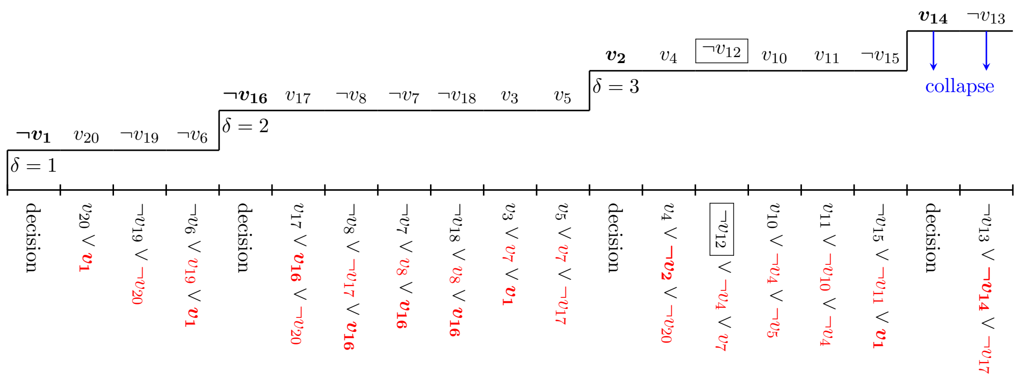


(c) After re-implication of v_{10}, v_{11} is re-implied. v_{11} was in the propagation queue, but was propagated during the re-implication. v_{11} can be removed from ω. $\rho = \{v_{14}, \neg v_{15}, \neg v_{15}\}$.

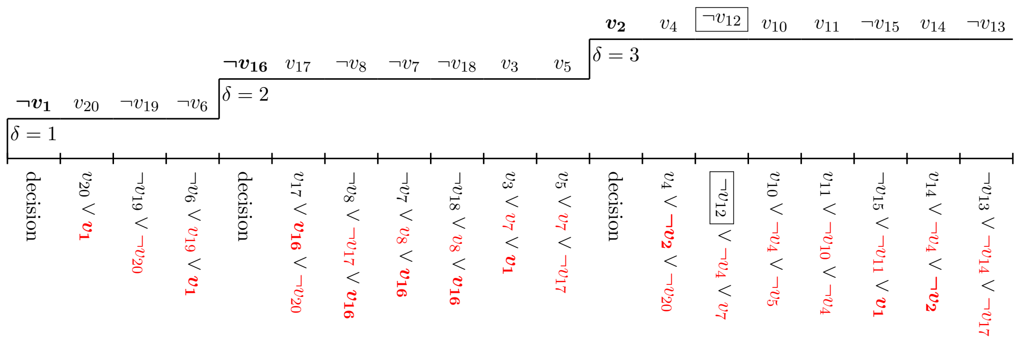
Figure 4.3.2: Example of re-implication: Part 1 (from problem `uf20-0083.cnf`).


$\neg v_1$ v_{20} $\neg v_{19}$ $\neg v_6$ $\neg v_{16}$ v_{17} $\neg v_8$ $\neg v_7$ $\neg v_{18}$ v_3 v_5 v_2 v_4 $\boxed{\neg v_{12}}$ v_{10} v_{11} v_{14} $\neg v_{13}$ $\neg v_{15}$
$\delta = 1$
$\delta = 2$
$\delta = 3$
$\delta = 4$
sink
decision
$v_{20} \lor v_1$
$\neg v_{19} \lor \neg v_{20}$
$\neg v_6 \lor v_{19} \lor v_1$
decision
$v_{17} \lor v_{16} \lor \neg v_{20}$
$\neg v_8 \lor \neg v_{17} \lor v_{16}$
$\neg v_7 \lor v_8 \lor v_{16}$
$\neg v_{18} \lor v_8 \lor v_{16}$
$v_3 \lor v_7 \lor v_1$
$v_5 \lor v_7 \lor \neg v_{17}$
decision
$v_4 \lor \neg v_2 \lor \neg v_{20}$
$\boxed{\neg v_{12}} \lor \neg v_4 \lor v_7$
$v_{10} \lor \neg v_4 \lor \neg v_5$
$v_{11} \lor \neg v_{10} \lor \neg v_4$
decision
$\neg v_{13} \lor \neg v_{14} \lor \neg v_{17}$
$\neg v_{15} \lor \neg v_{10} \lor \neg v_5$

(a) After the re-implication of v_{11}, $\neg v_{15}$ is re-implied. $\rho = \{v_{14}\}$


$\neg v_1$ v_{20} $\neg v_{19}$ $\neg v_6$ $\neg v_{16}$ v_{17} $\neg v_8$ $\neg v_7$ $\neg v_{18}$ v_3 v_5 v_2 v_4 $\boxed{\neg v_{12}}$ v_{10} v_{11} $\neg v_{15}$ v_{14} $\neg v_{13}$
$\delta = 1$
$\delta = 2$
$\delta = 3$
collapse
decision
$v_{20} \lor v_1$
$\neg v_{19} \lor \neg v_{20}$
$\neg v_6 \lor v_{19} \lor v_1$
decision
$v_{17} \lor v_{16} \lor \neg v_{20}$
$\neg v_8 \lor \neg v_{17} \lor v_{16}$
$\neg v_7 \lor v_8 \lor v_{16}$
$\neg v_{18} \lor v_8 \lor v_{16}$
$v_3 \lor v_7 \lor v_1$
$v_5 \lor v_7 \lor \neg v_{17}$
decision
$v_4 \lor \neg v_2 \lor \neg v_{20}$
$\boxed{\neg v_{12}} \lor \neg v_4 \lor v_7$
$v_{10} \lor \neg v_4 \lor \neg v_5$
$v_{11} \lor \neg v_{10} \lor \neg v_4$
$\neg v_{15} \lor \neg v_{11} \lor v_1$
decision
$\neg v_{13} \lor \neg v_{14} \lor \neg v_{17}$

(b) Finally, v_{14} is re-implied and level 4 is collapsed. $\rho = \emptyset$.


$\neg v_1$ v_{20} $\neg v_{19}$ $\neg v_6$ $\neg v_{16}$ v_{17} $\neg v_8$ $\neg v_7$ $\neg v_{18}$ v_3 v_5 v_2 v_4 $\boxed{\neg v_{12}}$ v_{10} v_{11} $\neg v_{15}$ v_{14} $\neg v_{13}$
$\delta = 1$
$\delta = 2$
$\delta = 3$
decision
$v_{20} \lor v_1$
$\neg v_{19} \lor \neg v_{20}$
$\neg v_6 \lor v_{19} \lor v_1$
decision
$v_{17} \lor v_{16} \lor \neg v_{20}$
$\neg v_8 \lor \neg v_{17} \lor v_{16}$
$\neg v_7 \lor v_8 \lor v_{16}$
$\neg v_{18} \lor v_8 \lor v_{16}$
$v_3 \lor v_7 \lor v_1$
$v_5 \lor v_7 \lor \neg v_{17}$
decision
$v_4 \lor \neg v_2 \lor \neg v_{20}$
$\boxed{\neg v_{12}} \lor \neg v_4 \lor v_7$
$v_{10} \lor \neg v_4 \lor \neg v_5$
$v_{11} \lor \neg v_{10} \lor \neg v_4$
$\neg v_{15} \lor \neg v_{11} \lor v_1$
$v_{14} \lor \neg v_4 \lor \neg v_2$
$\neg v_{13} \lor \neg v_{14} \lor \neg v_{17}$

(c) Re-implication is finished.

Figure 4.3.3: Example of re-implication: Part 2 (from problem `uf20-0083.cnf`).

Chapter 5

SMT Solving

The satisfiability problem is a major component of many system verification techniques. For example, checking whether an invariant $I(\boldsymbol{x})$ holds after a transition $T(\boldsymbol{x}, \boldsymbol{x'})$ is equivalent to checking whether the formula $[I(\boldsymbol{x}) \wedge T(\boldsymbol{x}, \boldsymbol{x'})] \Rightarrow I(\boldsymbol{x'})$ is valid. By duality, this problem can be reduced to checking whether the formula $I(\boldsymbol{x}) \wedge T(\boldsymbol{x}, \boldsymbol{x'}) \wedge \neg I(\boldsymbol{x'})$ is unsatisfiable. A model of such a formula can be interpreted as a bug instance. In practice, formulas expressed in real-world applications need a more expressive language than propositional logic. A user might need to declare non-Boolean variables such as real numbers, integers, arrays, etc. This is where SMT solvers come into play. Satisfiability Modulo Theories is an extension of SAT solving improving the expressiveness of formulas by adding useful theories [8]. A definition of SMT solving is given by [1]: *"Satisfiability Modulo Theories (SMT) refers to the problem of determining whether a first-order formula is satisfiable with respect to some logical theory."*

In this chapter, we briefly introduce the SMT problem, and discuss how it shapes the interface of the SAT solver running within the SMT solver. We also discuss some changes that need to be made when the SAT solver is converted to chronological backtracking. This chapter is mostly oriented towards possible exploration strategies for implementing such a solver. Unfortunately, an implementation is not yet complete. In the few weeks following the submission of this thesis, we will, however, implement a flexible CB and NCB SAT solver for a new project at the University of Liège. This project, led by Pr. Pascal Fontaine, is called `modulariT` and aims at creating a modular SMT solver in C++ for research purposes.

Core idea of SMT. Each atom in a FOL formula Φ is assigned a propositional variable v. Then, the formula Φ is translated into a propositional formula φ and fed to the SAT solver. The formula φ is often an under constraint propositional formula and has too many models. Each model of Φ is also a model of φ, but the opposite is not true. The SAT solver then tries to find a model of φ and return it to the SMT solver. If the SAT solver finds a model of φ, the SMT solver checks whether it is also a model of the theories T_i in the formula Φ. If it is, then the SMT solver returns the model to the user. Otherwise, the SMT solver creates a clause that prevents this propositional model from being found again, and provide it to the SAT solver. This procedure continues until φ becomes unsatisfiable, or until a model of Φ is found. In the context of SMT, for readability, we refer to a conflict as a propositional conflict, and a T-conflict as a conflict with respect to some theory T.

Example 5.0.1. Let us illustrate with a simple example. The only theory T considered here is equality. The theory reasoner only involves the *congruence closure* [23] algorithm. The formula Φ is the following:

$$\Phi \equiv (a = b \vee \neg P(c)) \wedge (P(f(a)) \vee \neg P(f(c))) \wedge c = f(c) \wedge P(c) \wedge \neg P(f(b))$$

We introduce one propositional variable for each atom in Φ:

$$\begin{array}{lll} v_{a=b} & v_{P(c)} & v_{P(f(a))} \\ v_{P(f(c))} & v_{c=f(c)} & v_{P(f(b))} \end{array}$$

and the propositional formula φ:

$$\varphi \equiv \left(v_{a=b} \vee \neg v_{P(c)}\right) \wedge \left(v_{P(f(a))} \vee \neg v_{P(f(c))}\right) \wedge v_{c=f(c)} \wedge v_{P(c)} \wedge \neg v_{P(f(b))}$$

One possible model of φ is $\pi = \{v_{a=b}, \neg v_{P(f(a))}, \neg v_{P(f(b))}, v_{c=f(c)}, v_{P(c)}, \neg v_{P(f(c))}\}$. This model is returned to the SMT solver. However, there is a T-conflict with the literals $c = f(c) \wedge P(c) \wedge \neg P(f(c))$. The clause $\neg v_{c=f(c)} \vee \neg v_{P(c)} \vee v_{P(f(c))}$ is added to φ and the SAT solver is called again. On the second run, the SAT solver provides the model $\pi = \{v_{a=b}, v_{P(f(a))}, \neg v_{P(f(b))}, v_{c=f(c)}, v_{P(c)}, v_{P(f(c))}\}$. Once again, the solver discovers a T-conflict with $a = b \wedge P(f(a)) \wedge \neg P(f(b))$ and adds the clause $\neg v_{a=b} \vee \neg v_{P(f(a))} \vee v_{P(f(b))}$ to φ. The SAT solver is called again, but is unable to find a model of φ. The SMT solver returns UNSAT. Indeed, this formula is equivalent to checking that the formula Φ' is valid.

$$\Phi' \equiv [(a = b \vee \neg P(c)) \wedge (P(f(a)) \vee \neg P(f(c))) \wedge c = f(c) \wedge P(c)] \Rightarrow P(f(b))$$

Since we have $c = f(c) \wedge P(c)$, we can imply that $P(f(c))$. Then, we remove falsified literals from disjunctions in Φ' and obtain the formula $a = b \wedge P(f(a)) \Rightarrow P(f(b))$. This formula is an instance of the congruence closure axiom for equality on unary predicates (Leibniz's law).

$$\forall xy.\ x = y \Rightarrow (P(x) \Leftrightarrow P(y))$$

with the substitution $x \mapsto f(a)$ and $y \mapsto f(b)$.

5.1 Components of an SMT solver

Fig. 5.1.1 shows the structure of a typical SMT solver. Roughly speaking, the SMT solver comprises a SAT solver, a theory reasoner and an instantiation module. The SAT solver is responsible for handling case splits and the Boolean structure of the formula. It finds a model for the propositional formulation of the FOL formula. Of course checking whether the Boolean structure of the formula is satisfiable is not enough. The theory reasoner is responsible for checking whether the model found by the SAT solver is also a model of the theories in the formula. It filters out unwanted models and constrains the SAT solver to not meet that conflict again by providing a T-conflicting clause. The theory reasoner and the SAT solver only handle ground terms (without quantifiers). However, it might sometimes be useful to have quantified formulas. For example, theory axioms that are not defined by an implemented theory are added to the formula; they often come with quantifiers. The instantiation module is responsible for generating ground instances of quantified formulas. After briefly exploring the impact of CB on the instantiation module, it did not seem to have any obvious dangers.

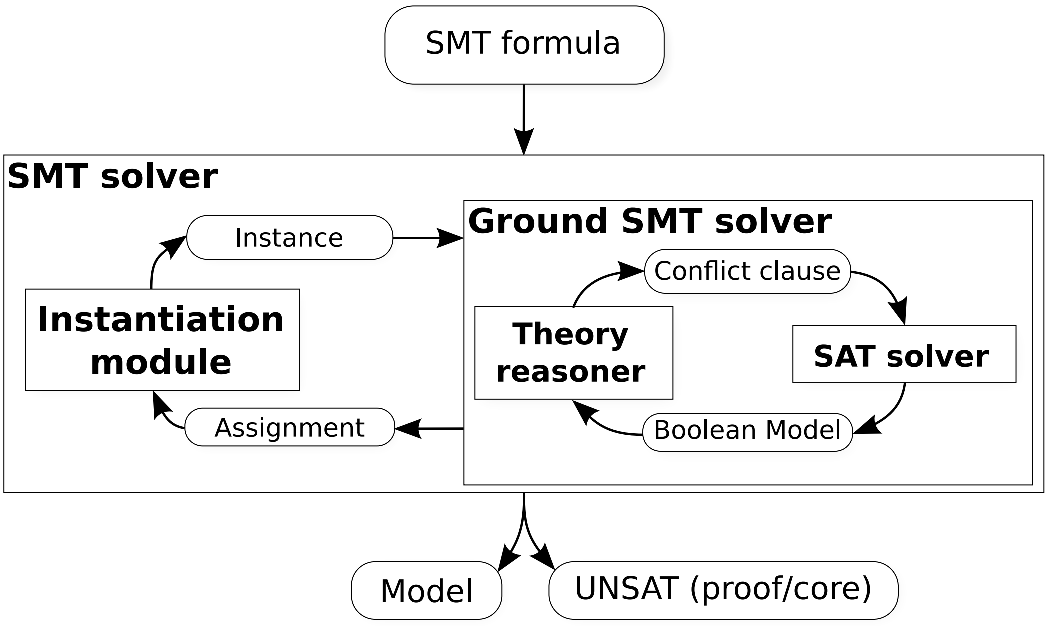


Figure 5.1.1: Schematic of a typical SMT solver (from [14]).

Therefore, we will not discuss it further but will keep in mind that it might be necessary to modify it as well in the future.

In this first section, we consider elements from a standard SMT solver whose SAT solver is based on NCB. Since this work is focused on the SAT part of the solver, we do not insist on details of each theory, but rather on the interaction between the SAT solver and the theory reasoner. The next section focuses on the modification that converting the SAT solver to CB leads to.

5.1.1 SAT Solver

In the following, each procedure from the SAT solver is prefixed with SAT_{...}. the meaning of τ, ω, π remain the same as for Chaps. SAT solving, Chronological Backtracking and Chronological Backtracking in `veriT`. Although, we assume that when the theory reasoner is running, literals in $\pi = \tau \cup \omega$ are fully propagated. That is, $\omega = \emptyset$ and $\pi = \tau$. Therefore, we use τ when the order of literals matters, and π when it does not.

Incremental solvers. The SAT solver in the SMT context is often modified to better fit the incremental nature of the problems it is required to solve. More specifically, when the theory reasoner discovers a T-conflict in the assignment π, it is not desirable to forget the state of the SAT solver and start from scratch. To that intent, the SAT solver is made incremental. That is, it supports the addition of new clauses during its execution. In the Ex. 5.0.1, the trail of the SAT solver before the conflict might be $\tau = \left\{v_{c=f(c)}, v_{P(c)}, \neg v_{P(f(b))}, v_{a=b}, \boldsymbol{\neg v_{P(f(a))}}, \neg v_{P(f(c))}\right\}$ with $\neg v_{P(f(a))}$ a decision. When the T-conflict clause $\neg v_{c=f(c)} \vee \neg v_{P(c)} \vee v_{P(f(c))}$ is added, the solver only backtracks the last decision level (that is, $\neg v_{P(f(a))}$ and $\neg v_{P(f(c))}$) and the trail becomes $\tau = \left\{v_{c=f(c)}, v_{P(c)}, \neg v_{P(f(b))}, v_{a=b}\right\}$. Then the SAT solver continues its execution from there, without needing to propagate again the literals at level 0, and find the model $\tau = \left\{v_{c=f(c)}, v_{P(c)}, \neg v_{P(f(b))}, v_{a=b}, v_{P(f(c))}, v_{P(f(a))}\right\}$. In this case, there are no decisions involved, and when the T-conflict clause $\neg v_{a=b} \vee \neg v_{P(f(a))} \vee v_{P(f(b))}$ is added, the SAT solver returns UNSAT because it has a conflict at level 0.

Sometimes, the clauses added to the SAT solver do not qualify as UIPs (see Sect. 2.3.3) and conflict analysis should be triggered before continuing the search. While conflict analysis is not necessary (even for non UIPs), it is a good way to continue the propagation without needing to make a decision. The T-conflict was not in the clause set φ before, but is conflicting, therefore narrowing the search; this is enough to ensure progress of the solver. However, this new constraint is not always enough to propagate. This is why conflict analysis is triggered like shown in Alg. 20. This function behaves very much like when a conflict was found inside the SAT solver. The T-conflict C is added to the clause set φ and must therefore handle C similarly. However, since the conflict was triggered outside the BCP loop, it does not have some properties such as having two falsified literals in the propagation queue ω.

Algorithm 20 Add a clause to φ

```
1: procedure SAT_AddClause($C$)
2:     if $C$ is a unit clause then
3:         $\ell \leftarrow$ the unassigned literal in $C$
4:         $\omega \leftarrow \omega \cup \{\ell\}$
5:     else if $C$ is conflicting with $\tau \cup \omega$ then
6:         if $\delta(C) = 0$ then
7:             return $\bot$
8:         if $C$ has only one literal at level $\delta(C)$ then
9:             $\ell \leftarrow$ the literal in $C$ at $\delta(C)$
10:            Backtrack($\delta(C \setminus \{\ell\})$)
11:            $\omega \leftarrow \omega \cup \{\ell\}$
12:        else
13:            Backtrack($\delta(C)$)
14:            $C' \leftarrow$ Analyze($C$)
15:            $\ell \leftarrow$ the highest literal in $C'$
16:            Backtrack($\delta(C' \setminus \{\ell\})$)
17:            $\omega \leftarrow \omega \cup \{\ell\}$
18:            $\varphi \leftarrow \varphi \cup \{C'\}$
19:    $\varphi \leftarrow \varphi \cup \{C\}$
20:    return $\top$
```

Partial solving. In practice, a T-conflict may be detected before the SAT solver finds a complete model. It would be a waste to solve a problem with a million variables when the theory reasoner can find a T-conflict after only a few propagations. Therefore, in practice, the SMT solver highjacks the CDCL loop and interrupts the search prematurely when a T-conflict is found. The solving interface of the SAT solver is modified to support these interruptions. In `veriT`, the SAT_propagate function performs everything that does not require a decision. That is, it propagates literals using BCP and performs conflict analysis when necessary. This procedure is shown on Alg. 21.

Algorithm 21 SAT_Propagate for SMT solving

```
1: procedure SAT_PROPAGATE
2:     while $\top$ do
3:         $C \leftarrow$ BCP()
4:         if $C = \top$ then                          ▷ No conflict detected
5:             return $\top$
6:         if $\delta(C) = 0$ then                     ▷ Conflict without any decision
7:             return $\bot$
8:         $C' \leftarrow$ ANALYZE($C$)
9:         $\ell \leftarrow$ the highest literal in $C'$
10:        BACKTRACK($\delta(C' \setminus \{\ell\})$)   ▷ Backtrack to the second-highest decision level in $C'$
11:        $\omega \leftarrow \omega \cup \{\ell\}$
12:        $\varphi \leftarrow \varphi \cup \{C'\}$
```

Providing hints. Since the SMT solver does not decompose the SAT search between decisions, it might be a good idea to take advantage of work done by the theory reasoner to find a conflict. It might be that it did not find a conflict, but found a literal that is T-implied by the current assignment. In that case, the SMT solver provides a *hint* to the SAT solver to guide the search. Hints can be handled in two ways. Either they are considered as decisions, or they are pushed at the last decision level. In the second case, the SMT solver must be able to give a reason for the propagation in order for conflict analysis to work. In general, this is done in a lazy manner. That is, instead of generating the reason directly when providing the hint, the theory reasoner waits until the SAT solver asks for it. Indeed, it is desirable to have the smallest possible reason for the propagation, and it can be expensive to compute. However, by using the lazy approach, it might not be necessary. A literal can be backtracked without being implicated in a conflict. Therefore, the theory reasoner does not always need to justify hints. If hints are considered as decisions, there is no need to pay attention to that detail since conflict analysis never tries to obtain the reason for a decision. Let τ^h be the set of hints provided to the SAT solver.

Solving an SMT problem can be summarized as in Alg. 22. The function SMT_SOLVE() is called by the user and calls the SAT solver. The SAT solver interface with the SMT solver consists in calls to SAT_PROPAGATE, SAT_ADDCLAUSE, SAT_DECIDE and the trail τ.

5.1.2 Theory Reasoner

The theory reasoner is charged of evaluating whether the propositional model is also a model in some predefined theories. In practice, a set of specific theories T_i is implemented in the SMT solver and are combined into a larger theory T. This allows for a lot of flexibility in the design of the SMT solver. Furthermore, it allows to keep the best possible complexity depending on the user's needs. For example, if the user only uses linear real arithmetic, there is no need to use a general non-linear arithmetic reasoner: "You do not pay for what you do not use".

Incrementalism. Because of the considerations about the SAT solver, efficient SMT solvers usually implement an incremental approach to theory reasoning as well. That is, the theory

Algorithm 22 Solving an SMT problem

```
1: $\Phi \leftarrow \emptyset$                                   ▷ The first-order formula to be solved
2: $\varphi \leftarrow \emptyset$                                   ▷ The SAT formula to be solved
3: $\tau \leftarrow \emptyset$                                   ▷ The current assignment
4: $\omega \leftarrow \emptyset$                                   ▷ The propagation queue
5: procedure SMT_SOLVE($\Phi'$)
6:     $\Phi \leftarrow$ CNF()                            ▷ Convert $\Phi'$ to CNF
7:     $\mathcal{V} \leftarrow \{v_{atom} \mapsto atom : atom \in \Phi\}$   ▷ Set of propositional variables to convert $\Phi$ to SAT
8:     $\varphi \leftarrow$ CONVERTTOSAT($\Phi, \mathcal{V}$)               ▷ Convert $\Phi$ to SAT
9:     while SAT_PROPAGATE() do
10:        $C \leftarrow$ FINDT-CONFLICT($\tau$)
11:        if $C \neq \top$ then
12:            SAT_ADDCLAUSE($C$)
13:            continue
14:        if $\exists \ell$ that is T-implied by $\tau$ then
15:            $\ell \leftarrow$ FINDT-IMPLICATION($\tau$)
16:            $\omega \leftarrow \omega \cup \{\ell\}$
17:            continue
18:        $\ell \leftarrow$ SAT_DECIDE()                    ▷ Decide a literal
19:        if $\ell = \bot$ then                        ▷ Every literal has been assigned
20:            return $\tau$                          ▷ SAT
21:        $\omega \leftarrow \omega \cup \{\ell\}$
22:    return $\bot$                                  ▷ UNSAT
```

reasoner is able to add formulas to the model and update upon the set of formulas it previously had. When the SAT solver propagates new literals, the theory reasoner only needs to update its knowledge of the problem with the new literals of the theory. Conversely, it should be able to backtrack a certain number of literals without losing its full state. Backtracking should be an inexpensive procedure.

When a T-conflict is detected, the theory reasoner should be able to generate a relatively small set of literals responsible for the conflict. This clause should be as small as possible since smaller clauses prune larger portions of the propositional search tree. In addition to conflict analysis, the theory reasoner should also be able to imply new literals to give as hints to the SAT solver. Furthermore, it should be able to provide a reason for the propagation of the literal. It may sometimes be cheaper to imply a literal than to give a small reason for it. The theory reasoner may use a lazy approach and only provide this reason when asked by the SAT solver.

Different theories. General-purpose SMT solvers usually implement a wide range of theories, from simple equality reasoner to higher-order theories. The most common theories include equality, linear and non-linear real or integer arithmetic, arrays, bit vectors, etc. The specific details of each theory are out of the scope of this work. However, it might be useful to understand a few concepts to understand what is possible in the context of SMT solving. The modules implementing these theories are called *decision procedures.*

A fast incremental algorithm for congruence closure is explained by [23]. The idea is to maintain a set of congruence classes and merge them when an equality is added to the set. If a disequality is added, then we check whether the two elements are in the same congruence class. If they are, then we have a T-conflict.

The decision procedures for linear real arithmetic generally use a simplex algorithm [9] to check the feasibility of a set of constraints. This algorithm is based on using a set of base variables to select active (tight) constraints. In a greedy fashion, the algorithm selects a variable to enter the base and a variable to leave the base. This is done until a feasible solution is found, or until no pivot can be found. In the latter case, the set of constraints is unsatisfiable.

The satisfiability of linear integer arithmetic (LIA) is NP-complete and is therefore not as efficient as linear real arithmetic. Decision procedures for LIA are outside the scope of this paper but curious readers can find more information in [5].

Combining theories. The general framework for combining decision procedures is SMT is the *Nelson-Oppen combination procedure* [27]. The idea is to declare new variables until every atom is pure in one theory. A *pure* atom is an atom that contains interpreted function and predicates from one unique theory. For example, $P(g(a) + 1)$ is not pure, and is replaced by $P(v_1), v_1 = v_2 + 1, v_2 = g(a)$. The first and third atoms are handled by congruence closure, while the second is handled by linear real arithmetic. Theories communicate through the equalities and disequalities between variables. Each decision procedure infers new equalities in turn until a fixed point is reached. If a T-conflict is found, then it is provided to the SAT solver. Otherwise, the SAT solver continues its search. Some theories such as non-linear real arithmetic can generate disjunctions of equalities (e.g., $v_1^2 = 1 \Rightarrow (v_1 = 1 \vee v_1 = -1)$). In that case, the decision procedure guesses an arrangement until a model is found, or all arrangements have been tried.

5.1.3 SAT Solver and Theory Reasoner

To achieve incrementalism, both the SAT solver and the theory reasoner keep track of a stack of literals τ and τ'. The SAT solver also keeps track of the lowest modification p of the trail τ since the last synchronization with the theory reasoner. This is done by simply keeping track of the lowest level that was backtracked during the propagation. When the SAT solver exits its propagation, the theory reasoner backtracks every literal above p in τ', then propagates the literals above p in τ. Once this is done, we say that the stacks are synchronized ($\tau = \tau'$), and $p \leftarrow |\tau|$.

5.2 Chronological Backtracking in SMT

This section briefly discusses the modifications that we considered necessary to the SMT solver when converting the SAT solver to CB. As explained earlier, these are only considerations and the implementation is not complete. Therefore, we have no empirical evidence that they will be efficient or even sufficient. However, this is a good starting point for the implementation of a CB SMT solver.

Incrementalism. The incremental nature of the SAT solver does not need a lot of modifications when using CB. Indeed, Alg. 20 already implements the main pitfalls of CB. The only necessary change would be to backtrack the SAT solver to the highest decision level of the T-conflict minus one, instead of the second-highest decision level. However, one might notice that the conflict clause, as opposed to NCB, is not necessary at the highest level in the trail. Therefore, to ease conflict analysis when necessary, the SAT solver first backtracks to the highest level of the T-conflict clause.

The added clause is always conflicting and cannot be a missed lower implication. Therefore, no special case must be considered for missed lower implications. However, we still want to ensure the fact that the watched literals are backtracked before the others. Therefore, we also need to impose that the two watched literals are the highest falsified literals in the clause.

Out of order backtracking. In the NCB setting, the trail behaves exactly like a stack. That is, literals are pushed and popped to and from the tail of the trail. When the solver backtracks, we are guaranteed that the first literal propagated after backtracking was not in the trail before. More specifically, we denote by $\Delta(\delta)$ the position of the first literal at level δ in the trail τ. When the solver backtracks to level δ, the trail becomes $\tau' \leftarrow \tau[0, \ldots, \Delta(\delta+1)-1]$, the literal pushed at position $\Delta(\delta+1)$ is guaranteed to not belong to τ since it was conflicting with τ. Therefore, the SMT solver also backtracks to the literal at position $(\Delta(\delta+1)-1)$. However, in the CB setting, this is no longer the case.

Let us illustrate with the example used to introduce multiple unit conflict. However, for the simplicity of the explanation, we consider that there is only one conflict as shown on Fig. 5.2.1. Let us consider that the conflicting clause is a T-conflict added by the theory reasoner (this was not the case in the real example). The trail was synchronized until literal $\neg v_2$. When backtracking, the SAT solver set the synchronization threshold p to $\Delta(4)$. While technically, the literal at position $\Delta(4)$ is different after backtracking, it was in the trail before. When possible, it would be desirable to preserve the literals that do not change. For some decision procedures such as the simplex for linear real arithmetic, removing literals can be done in any order. Therefore, it would be a waste to remove v_{17} and $\neg v_2$ just to add them again. In other theories, known algorithms strongly depend on the ordering of the trail, and there is no choice and the trail must behave like a stack. Congruence closure is such an algorithm. Therefore, in these decision procedures, we do not have a choice but to backtrack to the last literal that did not move in the trail, then propagate again the remainder of the stack. In the example, this would be $\neg v_7$.

With these considerations in mind, we would like to introduce an additional interface between the SAT solver and the theory reasoner. This interface would allow considering trail changes out of order as transactions. A transaction is a pair of sets (π^-, π^+) where π^- contains the literals that were removed from the partial assignment since the last synchronization, and π^+ are literals that were added to the partial assignment. Note that if a literal has seen its polarity change from ℓ to $\neg\ell$, then $\ell \in \pi^-$ and $\neg\ell \in \pi^+$. Furthermore, for efficiency reasons, it is desirable that $\pi^+ \cap \pi^- = \emptyset$. With this new interface, theories that are able to consider the partial assignment as an unordered set of literals can simply ignore literals that were moved around in the trail. Theories that require the trail to behave like a stack keep functioning as before.

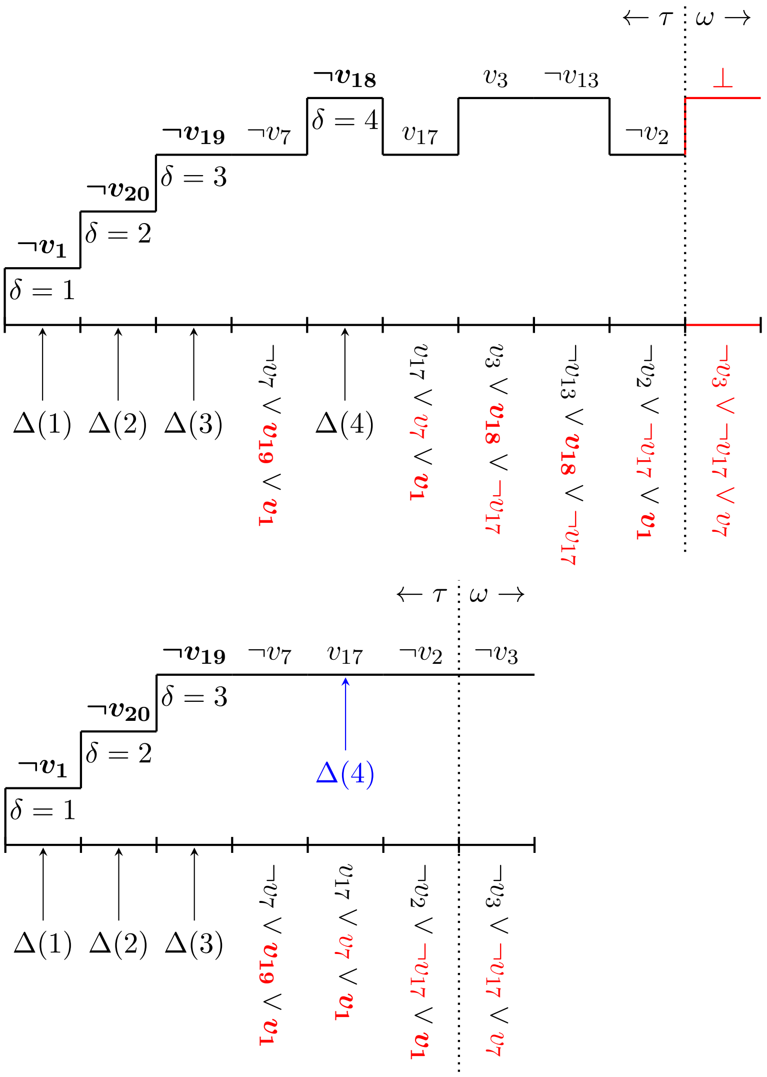

$\leftarrow \tau$ $\omega \rightarrow$
$\neg \boldsymbol{v_1}$ $\neg \boldsymbol{v_{20}}$ $\neg \boldsymbol{v_{19}}$ $\neg v_7$ $\neg \boldsymbol{v_{18}}$ v_{17} v_3 $\neg v_{13}$ $\neg v_2$ $\perp$
$\delta = 1$ $\delta = 2$ $\delta = 3$ $\delta = 4$
$\Delta(1)$ $\Delta(2)$ $\Delta(3)$ $\Delta(4)$
$\neg v_7 \vee \boldsymbol{v_{19}} \vee \boldsymbol{v_1}$
$v_{17} \vee v_7 \vee \boldsymbol{v_1}$
$v_3 \vee \boldsymbol{v_{18}} \vee \neg v_{17}$
$\neg v_{13} \vee \boldsymbol{v_{18}} \vee \neg v_{17}$
$\neg v_2 \vee \neg v_{17} \vee \boldsymbol{v_1}$
$\neg v_3 \vee \neg v_{17} \vee v_7$
$\leftarrow \tau$ $\omega \rightarrow$
$\neg \boldsymbol{v_1}$ $\neg \boldsymbol{v_{20}}$ $\neg \boldsymbol{v_{19}}$ $\neg v_7$ v_{17} $\neg v_2$ $\neg v_3$
$\delta = 1$ $\delta = 2$ $\delta = 3$
$\Delta(1)$ $\Delta(2)$ $\Delta(3)$ $\Delta(4)$
$\neg v_7 \vee \boldsymbol{v_{19}} \vee \boldsymbol{v_1}$
$v_{17} \vee v_7 \vee \boldsymbol{v_1}$
$\neg v_2 \vee \neg v_{17} \vee \boldsymbol{v_1}$
$\neg v_3 \vee \neg v_{17} \vee v_7$

Figure 5.2.1: Trail of the SAT solver in chronological backtracking (from `uf20-0651.cnf`). When backtracking, the synchronization threshold $\pi = \Delta(4)$. However, the literal at position $\Delta(4)$ was in the trail before, it was just moved. Note that the limit between τ and ω was artificially pushed compared to Fig. 4.2.1 to simulate that the conflict clause is a T-conflict.

Trail reordering. Because of the heavy changes on the trail order due to the missed lower implication procedure described in the previous chapter, it seems that the cost of such a modification to the algorithm is significant in SMT. This is why, in future work, we will try to find an alternative solution that does not require the trail to keep the topological order. More specifically, we will intend to find an alternative solution for conflict analysis. It might be worth building a topological order on the last level of the trail before performing conflict analysis. Or keep track of a topologically ordered trail in parallel with a chronologically ordered trail for conflict analysis and communication with the SMT module respectively. Both solutions will necessarily hurt the performance of the SAT solver, but may have advantages in the SMT context.

Remark 5.2.1. Note that the reordering issue is independent of the data structures used. No matter if we use doubly linked lists like [21] or a standard array, a change in the trail ordering will always be an issue in SMT. Simulating a stack will always require to perform much more work than necessary. Therefore, the reordering issue is not a data structure issue, but rather an algorithmic issue.

Hints

The introduction of hints does not seem to require a lot of modifications to the SAT solver's behavior when converting the solver to CB and remain sound. Indeed, if hints are added to the trail as a decision would, then its reason must be contained in the levels below. And if the hint is set at level $\delta(\tau)$, this property still holds. It is however possible that the hint is a T-missed lower implication. That is, it is propagated at level $\delta(\tau)$ while the T-reason for the implication is lower. However, due to the lazy approach of `veriT`, this is not detectable directly. The missed lower implication might be discovered when the hint is asked to be explained by the theory reasoner. In which case, it means that the hint is at level $\delta(C)$ where C is a (T-)conflict. Therefore, the hint will be backtracked anyway.

Despite the fact that it is not necessary to treat hints as missed lower implications, it might be useful to discover it when conflict analysis is triggered. When the hint is being explained by the theory reasoner, quickly realizing the hint is actually still valid after backtracking might be useful. Indeed, instead of backtracking it, the SAT solver, now aware of the T-reason for the hint might be able to imply it a lower level. Otherwise, the theory reasoner may need to imply the hint multiple times. Furthermore, when realizing that the hint is a missed lower implication, it does not need to be justified by the first UIP procedure, since it in facts belongs to a lower level, therefore reducing the length of learned clauses. Deciding whether the T-reason is added to the clause set φ or not is an engineering aspect that should be explored when the implementation of the `modulariT` SAT solver starts.

5.3 Status of the Implementation in `veriT`

While we tried to integrate the newly chronological backtracking SAT solver into `veriT`, a soundness bug has been detected but not yet identified. This is the reason why this chapter is mostly theoretical. As yet, we do not have empirical evidence that the modifications discussed earlier are sufficient or if another aspect of the solver needs to be modified. It might be that the bug in `veriT` is just an implementation detail, or it could be something more fundamental. `veriT` being a large code base, we were not able to perform thorough checking like for the standalone SAT solver. When running the problem putting `veriT` at fault, the SAT solver keeps behaving normally and respects the invariants discussed earlier. Which gives us strong confidence in believing that the problem lies in the communication between the SAT solver and the theory reasoner. However, we were required to pause the bug hunt and focus on the redaction of this document.

Chapter 6

Conclusion and Future Work

In this thesis, we have discussed the problem of propositional satisfiability, and explained the two main approaches to solving them: DPLL and CDCL. In standard CDCL, the backtracking mechanism called non-chronological backtracking is the dominating strategy. This method ensures that the algorithms follow a set of clean invariants that guarantee soundness and completeness. However, in the past few years, some researchers in the SAT community started advocating for the introduction of chronological backtracking. This new strategy uses a less aggressive backtracking method but comes with some complications to maintain the core invariants of CDCL. We discussed thoroughly what this seemly small change brings to the table. We implemented the change in the SAT solver of `veriT`. While the results show that the new implementation is functional and sound, preliminary benchmarks do not show any improvements when using missed lower implications in the context of SAT solving alone. After examining the changes necessary to the SAT solver, we briefly discussed what it entails in the context of SMT. From a first study, it seems that the modifications to keep soundness should not be too heavy on the SMT side provided that the SAT solver preserves some properties. We suggest a set of modifications to both the SAT solver and theory reasoner to take advantage of the new technique while minimizing drawbacks. However, the SMT solver `veriT` was not yet fully adapted to support CB. Further improvements may be more research heavy, but it is what makes this problem interesting.

Future work. As previously discussed, in the few weeks following the submission of this document, we will work on a SAT solver from scratch that will support both CB and NCB frameworks. This project follows the line of libraries that are currently being developed in a brand new modular SMT solver: `modulariT`. In addition to re-implementing the algorithms discussed in Chap. Chronological Backtracking in `veriT`, we will work on a new conflict analysis algorithm that does not require a complete reordering of the trail. We will also try to implement an efficient transaction interface to the SAT solver that will allow the SMT solver to consider the partial assignment as an unordered set. In addition, we are going to search for more efficient ways to enforce Prop. *literal ordering* which seems to be a bottleneck of the current implementation. Finally, we intend on performing thorough benchmarking to compare the performance of the different algorithms and ideas.

Bibliography

[1] Clark Barrett and Cesare Tinelli. Satisfiability modulo theories. In *Handbook of Model Checking*. Springer, 2018.

[2] Nikolaj S. Bjørner, Kenneth L. McMillan, and Andrey Rybalchenko. Program verification as satisfiability modulo theories. In Pascal Fontaine and Amit Goel, editors, *10th International Workshop on Satisfiability Modulo Theories, SMT 2012, Manchester, UK, June 30 - July 1, 2012*, volume 20 of *EPiC Series in Computing*, pages 3–11. EasyChair, 2012.

[3] Jasmin Christian Blanchette, Mathias Fleury, Peter Lammich, and Christoph Weidenbach. A verified SAT solver framework with learn, forget, restart, and incrementality. *J. Autom. Reason.*, 61(1-4):333–365, 2018.

[4] Thomas Bouton, Diego Caminha Barbosa De Oliveira, David Déharbe, and Pascal Fontaine. verit: An open, trustable and efficient smt-solver. In Renate A. Schmidt, editor, *Automated Deduction - CADE-22, 22nd International Conference on Automated Deduction, Montreal, Canada, August 2-7, 2009. Proceedings*, volume 5663 of *Lecture Notes in Computer Science*, pages 151–156. Springer, 2009.

[5] Martin Bromberger. *Decision Procedures for Linear Arithmetic. (Quelques procédures de décision pour l'arithmétique linéaire)*. PhD thesis, Saarland University, Saarbrücken, Germany, 2019.

[6] Robin Coutelier, Laura Kovács, Michael Rawson, and Jakob Rath. SAT-Based Subsumption Resolution (to appear). In *Automated Deduction - CADE-29, 29th International Conference on Automated Deduction, Rome, Italy, August 1-4, 2023. Proceedings.* Springer, 2023.

[7] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, 1960.

[8] Leonardo Mendonça de Moura and Nikolaj S. Bjørner. Z3: an efficient SMT solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems, 14th International Conference, TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, March 29-April 6, 2008. Proceedings*, volume 4963 of *Lecture Notes in Computer Science*, pages 337–340. Springer, 2008.

[9] Diego Caminha Barbosa De Oliveira and David Monniaux. Experiments on the feasibility of using a floating-point simplex in an smt solver. In *PAAR@ IJCAR*, pages 19–28, 2012.

[10] David Déharbe, Pascal Fontaine, Daniel Le Berre, and Bertrand Mazure. Computing prime implicants. In *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, October 20-23, 2013*, pages 46–52. IEEE, 2013.

[11] 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.

[12] Mathias Fleury. Optimizing a verified SAT solver. In Julia M. Badger and Kristin Yvonne Rozier, editors, *NASA Formal Methods - 11th International Symposium, NFM 2019, Houston, TX, USA, May 7-9, 2019, Proceedings*, volume 11460 of *Lecture Notes in Computer Science*, pages 148–165. Springer, 2019.

[13] Pascal Fontaine. Logic for computer science, 2021.

[14] Pascal Fontaine and Bernard Boigelot. Introduction to computer systems verification, 2023.

[15] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively p-simulate general propositional resolution. In Dieter Fox and Carla P. Gomes, editors, *Proceedings of the Twenty-Third AAAI Conference on Artificial Intelligence, AAAI 2008, Chicago, Illinois, USA, July 13-17, 2008*, pages 283–290. AAAI Press, 2008.

[16] Holger H Hoos and Thomas Stützle. Satlib: An online resource for research on sat. *Sat*, 2000:283–292, 2000.

[17] Michail G. Lagoudakis and Michael L. Littman. Learning to select branching rules in the DPLL procedure for satisfiability. *Electron. Notes Discret. Math.*, 9:344–359, 2001.

[18] Paolo Liberatore. On the complexity of choosing the branching literal in DPLL. *Artif. Intell.*, 116(1-2):315–326, 2000.

[19] 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.

[20] 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.

[21] Alexander Nadel. Introducing intel(r) SAT solver. In Kuldeep S. Meel and Ofer Strichman, editors, *25th International Conference on Theory and Applications of Satisfiability Testing, SAT 2022, August 2-5, 2022, Haifa, Israel*, volume 236 of *LIPIcs*, pages 8:1–8:23. Schloss Dagstuhl - Leibniz-Zentrum für Informatik, 2022.

[22] 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.

[23] Greg Nelson and Derek C. Oppen. Fast decision procedures based on congruence closure. *J. ACM*, 27(2):356–364, 1980.

[24] Tobias Nipkow, Lawrence C. Paulson, and Markus Wenzel. *Isabelle/HOL - A Proof Assistant for Higher-Order Logic*, volume 2283 of *Lecture Notes in Computer Science*. Springer, 2002.

[25] Jakob Rath, Armin Biere, and Laura Kovács. First-order subsumption via SAT solving. In Alberto Griggio and Neha Rungta, editors, *22nd Formal Methods in Computer-Aided Design, FMCAD 2022, Trento, Italy, October 17-21, 2022*, pages 160–169. IEEE, 2022.

[26] Cesare Tinelli. Smt-based model checking. In *NASA Formal Methods*, page 1, 2012.

[27] Cesare Tinelli and Mehdi T. Harandi. A new correctness proof of the {Nelson-Oppen} combination procedure. In Franz Baader and Klaus U. Schulz, editors, *Frontiers of Combining Systems, First International Workshop FroCoS 1996, Munich, Germany, March 26-29, 1996, Proceedings*, volume 3 of *Applied Logic Series*, pages 103–119. Kluwer Academic Publishers, 1996.

[28] Grigori S Tseitin. On the complexity of derivation in propositional calculus. *Automation of reasoning: 2: Classical papers on computational logic 1967–1970*, pages 466–483, 1983.