CDCL with Less Destructive Backtracking through Partial Ordering

Anthony Monnet, Roger Villemaire
Université du Québec à Montréal
Montreal, Quebec, Canada
anthonymonnet@aol.fr, villemaire.roger@uqam.ca

Abstract

Conflict-driven clause learning is currently the most efficient complete algorithm for satisfiability solving. However, a conflict-directed backtrack deletes potentially large portions of the current assignment that have no direct relation with the conflict. In this paper, we show that the CDCL algorithm can be generalized with a partial ordering on decision levels. This allows keeping levels that would otherwise be undone during backtracking under the usual total ordering. We implement partial ordering CDCL in a state-of-the-art CDCL solver and show that it significantly ameliorates satisfiability solving on some series of benchmarks.

1 Introduction

Conflict-driven clause learning (CDCL) [13] is a very efficient algorithm for solving the propositional satisfiability problem, currently used in virtually all complete state-of-the-art SAT solvers. For each conflict, it deduces a new clause that will allow an early detection of future similar conflicts, thus helping to prune the search space. It also performs a conflict-directed backtracking, which may undo several decision levels at once in order to return faster to the cause of the conflict and propagate this new learnt clause as early as possible in the search tree.

Despite this approach was proved very effective, each conflict-directed backtrack deletes a possibly large amount of instantiations that have no direct connection with the detected conflict. Indeed, by definition, none of the deleted levels contains any variable from the conflict, except for the conflict level itself. In the worst case, these levels could even belong to a distinct connected component of the problem, meaning that they can't be affected by the conflict and the resulting assertion, even indirectly. This results in a partial loss of previous search work, which may delay the discovery of a model or of another conflict. CDCL may have to rebuild this part of the search and reprocess all propagations. Given that propagations are the most time-consuming task of SAT solving, it is natural to try avoiding the destruction of instantiations that are still consistent with the current state. Several methods have been conceived to tackle this issue and minimize the amount of unrelated instantiations that are deleted, for instance tree decompositions [9, 3, 11, 5, 15] and phase saving [18].

In this paper, we propose a novel variation of the CDCL algorithm that detects instantiations that would be undone by the regular algorithm but can be safely retained. This is achieved by relaxing the ordering between decision levels. Indeed, with the usual total order, conflict-directed backtracking must delete all levels above the assertion level in order to return to that level and propagate the conflict clause. We show that this total ordering is not required to maintain essential properties of the algorithm, and that a partial ordering reflecting dependencies between decision levels can be used instead. As a consequence, instantiations are only deleted by the conflict-directed backtracking if they actually interfere with the conflict resolution. Partial order backtracking [7, 14, 4] has previously been described for the Constraint Satisfaction Problem (CSP), but to the best of our knowledge, it has never been used in the

context of SAT solving, moreover within the CDCL algorithm. This is the main contribution of our paper.

We also provide experimental results obtained by implementing partial order CDCL (PO-CDCL) in a state-of-the-art CDCL solver. We show that although PO-CDCL is not efficient on all SAT benchmarks, it seems to significantly reduce the solving trace on instances with a low partial order density, and that some benchmark series have a consistantly low density. Thus PO-CDCL manages to solve these series faster than the original CDCL solver.

The rest of this paper is organized as follows: section 2 summarizes the CDCL algorithm. Section 3 quickly introduces previous related works, namely tree decompositions, phase saving and partial order CSP. Section 4 presents the algorithm of partial order CDCL and gives the proof of some of its essential properties. Finally, section 5 shows and analyzes experimental results obtained by our implementation of PO-CDCL.

2 Conflict-Driven Clause Learning

Let $\mathcal{V}$ be a set of variables and $\mathcal{L} = \{v, \neg v \,|\, v \in \mathcal{V}\}$ the set of literals on $\mathcal{V}$. A propositional formula in conjunctive normal form $\mathcal{F}(\mathcal{V}, \mathcal{C})$ is defined by a set $\mathcal{V}$ of variables and a set $\mathcal{C}$ of clauses on $\mathcal{V}$, each clause $c \in \mathcal{C}$ being a set of literals. An assignment $\sigma \subset \mathcal{L}$ is a set of non-conflicting literals considered true. σ can be extended and interpreted as a partial function associating boolean values to variables, literals, clauses and formulas. If σ is defined on $v \in \mathcal{V}$, we will say that v is instantiated by σ; if it isn't, we will note $\sigma(v) = \text{undef}$. A total assignment σ on $\mathcal{V}$ is a model of the formula $\mathcal{F}(\mathcal{V}, \mathcal{C})$ iff $\sigma(\mathcal{F}(\mathcal{V}, \mathcal{C})) = \text{true}$. Given a formula, the SAT problem consists in determining whether it is satisfiable, i.e. whether it has at least one model.

The CDCL algorithm [13] determinates the satisfiability of a formula through a combination of depth-first search and inference. Algorithm 1 presents a pseudocode of CDCL. The search

Algorithm 1 CDCL

```
1:  σ ← ∅  /* begin with the empty assignment */
2:  loop
3:      c ←Propagate /* propagate new instantiations */
4:      if c ≠ NIL then /* a conflict was found during propagations */
5:          if λ = 0 then /* conflict at decision level 0 */
6:              return false /* F is unsatisfiable */
7:          else
8:              γ ← Analyze(c) /* infer the conflict clause γ */
9:              a ← AssertionLevel(γ, λ)
10:             Backtrack(a) /* backtrack to assertion level */
11:             λ ← a /* a becomes the current level */
12:             C ← C ∪ {γ} /* γ is learnt */
13:             PropagateAssertion(γ)
14:     else /* no conflict during propagations */
15:         if all variables are instantiated then
16:             return σ /* σ is a model of F */
17:         else
18:             λ ← NewLevel
19:             Decide(λ)
```

Algorithm 2 AssertionLevel(γ, λ) *[CDCL]*

$a \leftarrow \max(\{\lambda(l) \,|\, l \in \gamma\} \setminus \{\lambda\})$
return a

Algorithm 3 Backtrack(a) *[CDCL]*

for $v \in \mathcal{V} \,|\, \lambda(v) > a)$ **do**
    $\sigma(v) \leftarrow$ undef

part of the algorithm is conducted by repeatedly choosing instantiations to add to the current assignment σ (through the procedure Decide) until either all variables are instantiated or a conflict is reached. Conflicts are solved by undoing some of the last search choices.

The inference engine used within CDCL is the unit propagation rule: for any clause $c = \{l_1, \ldots, l_i\}$ such that $\sigma(l_1) = \sigma(l_2) = \ldots = \sigma(l_{i-1}) =$ false and $\sigma(l_i) =$ undef, c entails l_i under σ so l_i is added to σ. c is called the antecedent of l_i, noted $\alpha(l_i) = c$. Unit propagation is exhaustively applied to all unit clauses by procedure Propagate before making any new decision. The n^{th} decision and all unit propagations it entails form the n^{th} decision level of the search; all propagations which were deduced without any decision belong to decision level 0. We will note $\lambda(v)$ the decision level of a variable v and λ the current level of the search.

Propagate encounters a conflicts if it finds a clause c for which all literals are false under the current assignment σ. CDCL Analyzes this conflict and its reasons to produce a conflict clause γ which is also falsified by σ but only has one literal of current decision level λ. If $\lambda = 0$, then the conflict can't be avoided and $\mathcal{F}$ is unsatisfiable. Else γ defines an AssertionLevel a, which is the second largest decision level in this clause (Alg. 2). CDCL performs a Backtrack to the assertion level by entirely deleting all decision levels above a (Alg. 3). γ becomes unit, is propagated by PropagateAssertion, and Propagate is called again to deduce all possible inferences from this new instantiation. When a call to Propagate exhausts all unit propagations without encountering any conflict, a new decision is taken. If all variables have already been instantiated then σ is a model of $\mathcal{F}$.

All modern CDCL solvers implement unit propagation using watched literals [16], a method allowing a very efficient detection of unit propagations. Its pseudocode is shown by Alg. 4. As long as a clause has at least two literals that aren't false under σ, it can't be propagated. Therefore, for each clause c, CDCL keeps track of two of its literals $\omega(c) = \{w_1, w_2\} \subseteq c$. For each new propagation l, CDCL checks all clauses c where $\neg l$ is watched. If the second watched literal w is true under σ, then c is true and obviously can't be propagated; CDCL doesn't need to replace $\neg l$. Else, CDCL looks for another non-false literal w' to watch instead of $\neg l$. If it can't find one, either w is false (and c is a conflict), or w is undefined. In the latter case, c is unit and w is added to σ.

Checking clauses for propagations (lines 4 to 17 of Alg. 4) is the innermost loop of the Propagate procedure, which is generally by far the procedure in which the most time is spent during solving. Because of this, we will use in the rest of the paper the number of clause checks as a secondary indicator of solving efficiency, less implementation-dependent than solving time.

Note that the Backtrack procedure is described here as implemented in zChaff [16] and Glucose [1] for instance. One of the original CDCL solvers GRASP [13] uses a less destructive backtracking: it generally only deletes the last decision level and instantiates the assertion as a new "pseudo-decision". It only performs an actual conflict-directed backtracking when the decision at the conflict level is already a pseudo-decision itself. Although pseudo-decisions allow the use of less destructive backtracks, they are propagations stored in a pseudo-

Algorithm 4 PROPAGATE *[CDCL]*

```
1: Π ←{instantiations not yet propagated}
2: while {Π ≠ ∅} do
3:     choose l ∈ Π
4:     for c ∈ C | ¬l is watched in c do
5:         w ← the second watched literal in c
6:         if σ(w) ≠ true then
7:             Ω ← {l' ∈ c | σ(l') ≠ false} \ {w}
8:             /* Ω is the set of literals that could replace ¬l */
9:             if Ω = ∅ then /* no other literal in c can be watched */
10:                if σ(w) = undef then /* c is unit */
11:                    σ(w) ← true /* w is propagated by c*/
12:                    Π ← Π ∪ {w}
13:                else
14:                    return c /* c is a conflict */
15:            else
16:                choose w'∈Ω
17:                ω(c) ← {w, w'} /* w' is watched instead of ¬l */
18:    Π ← Π \ {l}
19: return NIL /* no conflict occured */
```

level without any other literal of their antecedent. Therefore they can be deleted without these causes, leaving an undetected unit clause. GRASP-type backtracks thus do not ensure that all possible unit propagations have been performed before taking a new decision, unlike backtracks used in zCHAFF and GLUCOSE. As a result, conflicts discovered by GRASP can involve clauses that were already unit several decision levels earlier, which means these conflicts could have been avoided much earlier in the search by an exhaustive unit propagation. As unit propagations are crucial for efficiently pruning the search space, we suspect that the incompleteness of unit propagations in GRASP is partly responsible for its lower performance wrt. zChaff ([16], Section 4.4.4. of [12]). Enforcing complete unit propagations within a GRASP backtrack type would require to exhaustively check all clauses after each conflict, which may be time expensive, and would still cause pseudo-decisions. In contrast, PO-CDCL aims to reduce the amount of instantiations undone during conflict-directed backtracking while keeping the exhaustive unit propagation property.

3 Related Works

Several methods have been proposed to directly or indirectly minimize the quantity of search progress lost during conflict-directed backtracking while solving SAT or CSP problems.

Some of them rely on tree decompositions [19] of the connectivity graph between variables. They constrain the order of decision variables so that the instance first breaks into several connected components, and then that the solving of one component can't undo instantiations in another component [9, 3, 11, 5, 10]. The main practical drawback is that challenging SAT problems are typically so large that computing a good decomposition becomes untractable [15].

Phase saving [18] is a more heuristic and very lightweight approach. It simply memorizes the last polarity assigned to a variable and reuses it if the variable is picked for a decision.

Phase saving actually doesn't prevent instantiations from being undone, but makes it possible to rediscover the deleted instantiations later and recover the search progress. This recovery however doesn't save the cost of repeating the time-consuming propagation phase.

A set of CSP solving algorithms was designed with the goal of undoing less search progress than conflict-directed backjumping (CBJ) by relaxing the order between variables. While CBJ resolves a conflict by deleting all instantiations and restoring all eliminated values from the culprit variable on (the most recent variable in the nogood), dynamic backtracking (DB) [7] only undoes the culprit variable and restores only eliminated values where the culprit variable was part of the nogood. Partial order backtracking (POB) [14] uses the same backtrack as DB and additionally allows to pick any variable in the nogood as the culprit variable. To ensure termination, it however progressively sets permanent order constraints between variables. Both algorithms were hybridated [8] and generalized [4].

Similarly to DB and POB, PO-CDCL undoes less search progress than regular conflict-directed algorithms, and like POB it allows some freedom in the choice of the assertion level. However, instead of setting definitive constraints on the order of variable instantiations, it sets local constraints on the order in which decision levels will be undone. Moreover, PO-CDCL is specifically adapted to various aspects of CDCL, such as the integration of unit propagations and the watched literal mechanism, which correctness implicitely relies on the total order between decision levels.

Finally, some techniques aim to enhance performances of SAT solvers by increasing the quantity of instantiations undone by backtracks [17, 2], which is a totally opposite strategy wrt. PO-CDCL.

4 Partial Order CDCL

This section introduces PO-CDCL, a generalization of the usual CDCL that relies on a partial order on decision levels during the search. In the first subsection, we will present the algorithm of PO-CDCL, and in the second we will show amongst others that it is correct and complete and that it terminates.

4.1 Algorithm

Algorithm 1, that we used to describe CDCL, remains the backbone of PO-CDCL, but some of its elements are modified.

In the original CDCL, decision levels are assumed to be totally ordered such that $i < j$ iff the decision of level i was set before the decision of level j. In PO-CDCL, we only set a strict partial ordering Δ between decision levels. We will say that i is a dependency for j, or equivalently that j depends on i, and note $i <_\Delta j$ if $(i,j) \in \Delta$. $i \leq_\Delta j$ is the reflexive extension of $<_\Delta$. $i <_\Delta j$ means that decision level i had an influence on propagations at level j. Consequently, level j should be deleted when level i is deleted or modified. Two cases of the PROPAGATE procedure add dependencies between levels (see Alg. 5):

1. At lines 14 and 15, when a unit clause $c = \{l_1, \ldots, l_i\}$ propagates the literal l_i, then this propagation at the current level obviously depends on all other levels occurring in c: $\lambda(l_1), \ldots, \lambda(l_{i-1}) <_\Delta \lambda$ (except when $\lambda(l_j) = \lambda$).

2. At line 7, when $\sigma(w) = \text{true}$, we add the dependency $\lambda(w) <_\Delta \lambda$ if $\lambda(w) \neq \lambda$. Indeed, in this case a clause c is checked because one of its watched literals $\neg l$ is false, but $\neg l$ doesn't need to be replaced because the second watched literal w is true. w is the reason why we

Algorithm 5 PROPAGATE *[PO-CDCL]*

```
1: Π ←{instantiations not yet propagated}
2: while {Π ≠ ∅} do
3:     choose l ∈ Π
4:     for c ∈ C | ¬l is watched in c do
5:         w ← the second watched literal in c
6:         if σ(w) = true then
7:             Δ ← Δ ∪ {(λ(w), λ)}  /* λ depends of λ(w) */
8:         else
9:             Ω ← {l' ∈ c | σ(l') ≠ false} \ {w}
10:            /* Ω is the set of literals that could replace ¬l */
11:            if Ω = ∅ then /* no other literal in c can be watched */
12:                if σ(w) = undef then /* c is unit */
13:                    σ(w) ← true /* w is propagated by c*/
14:                    for l' ∈ c \ {w} | λ(l') ≠ λ do
15:                        Δ ← Δ ∪ {(λ(l'), λ)}  /* λ depends of λ(l') */
16:                    Π ← Π ∪ {w}
17:                else
18:                    return {c}  /* c is a conflict */
19:            else
20:                choose w'∈Ω
21:                ω(c) ← {w, w'}  /* w' is watched instead of ¬l */
22:    Π ← Π \ {l}
23: return ∅  /* no conflict occured */
```

Algorithm 6 ASSERTIONLEVEL(γ, λ) *[PO-CDCL]*

```
Θ ← {λ(l) | l ∈ γ}\{λ}  /* Θ is the set of levels involved in the conflict, except λ*/
Γ ← {i ∈ Θ, ∄j ∈ Θ | i <_Δ j}  /* Γ is the set of maximal elements in Θ */
choose a ∈ Γ
return a
```

can stop watching c for unit propagations, but we have to make sure that w will not be uninstantiated before $\neg l$, else c could become unit without being properly watched. This is impossible with a total order on decision levels but could happen with a partial order.

ASSERTIONLEVEL also has to be modified, as indicated in Alg. 6. Partial order will allow some freedom in the choice of the assertion level. In CDCL, it is uniquely defined as the largest level in the set $\Theta = \{\lambda(l) \mid l \in \gamma\} \setminus \{\lambda\}$ of decision levels involved in the conflict clause, minus the current decision level. In PO-CDCL, due to the partial order, Θ may have several largest elements. Each of these largest elements is eligible as a valid assertion level, so that the assertion level can be arbitrarily picked amongst them.

Finally, we also modify BACKTRACK (see Alg. 7) since the goal of our method is to undo less instantiations during this phase. CDCL resolves a conflict by undoing all instantiations which decision level is larger than the assertion level a. PO-CDCL performs a similar deletion, except that it only deletes decision levels i such that $a <_\Delta i$ (λ may not depend on a but must obviously be deleted in any case). This deletion ensures the antisymmetry of Δ: if a level i such that $a <_\Delta i$ wasn't deleted, the search returning to level a may produce a propagation of level

Algorithm 7 BACKTRACK(a) *[PO-CDCL]*

$\Lambda \leftarrow$ the set of all decision levels
for $i \in \Lambda \,|\, (a <_\Delta i)$ **or** $(i = \lambda)$ **do**
  for $v \in \mathcal{V} \,|\, \lambda(v) = i$ **do**
    $\sigma(v) \leftarrow$ undef

a depending on level i, so we would have simultaneously $a <_\Delta i$ and $i <_\Delta a$. The antisymmetry of Δ is crucial to ensure that the assertion level of a conflict is well-defined. Indeed, without this property, the set of decision levels involved in a conflict clause may not have any maximal element.

Note that we should also always enforce $\forall i \neq 0,\ 0 <_\Delta i$; else, when backtracking to level 0, it would be possible to make a propagation at top-level which depends on a decision.

4.2 Properties

In this subsection, we will prove some properties of PO-CDCL, including that it is correct, complete and that it terminates. Most other properties we will prove are implicit or obvious properties within the original CDCL, but are less straightforward in the case of a partial order.

Proposition 1. *Δ is antisymmetric.*

Proof. Algorithm 5 only adds dependencies to the current decision level λ. To show the antisymmetry of Δ, it is thus sufficient to prove that no other level depends on λ at the moment it becomes the current level. λ can be a newly created decision level, in which case it has initially no dependency. Else, the search returned to λ because it has been chosen as the assertion level for some conflict. Then BACKTRACK deleted all decision levels which depended on λ. In both cases, no non-empty level depends on λ. □

Corollary 1. *Δ is a strict partial order.*

Definition 1. *A propagation l is valid iff $\forall a \in \alpha(l),\ v(a) = false$.*

Proposition 2. *During a PO-CDCL solving, all propagations remain valid.*

Proof. The only way to make a propagation invalid would be to delete a level to which a literal from its antecedent belongs, without deleting the level of the propagation itself. Dependencies added in Alg. 5 when a propagation occurs ensure that such a case can't happen. □

Definition 2. *A SAT solver is propagation-complete iff when its* PROPAGATE *function stops without having detected a conflict, no more clause is unit.*

Lemma 1. *Whenever* PROPAGATE *terminates without encountering any conflict, the following propreties hold. All clauses not yet satisfied watch two undefined literals. Satisfied clauses may watch true, false, or undefined literals, but each clause watches at most one false literal. If a satisfied clause watches a false literal w_1, the second watched literal w_2 is true, and $\lambda(w_2) \leq_\Delta \lambda(w_1)$.*

Proof. We will prove the lemma by recurrence on conflictless calls to PROPAGATE.

Initialization: Before the initial propagation round of the search, all variables are uninstantiated, so all clauses are unsatisfied and watch two undefined variables.
Assume a clause c whose two watched literals become false. PROPAGATE will eventually

check one on them and try to replace it by a true or undefined literal. If it fails, it means that the clause is already unsatisfiable before any decision was made, so the entire formula is unsatisfiable. If it succeeds, the clause now belongs to the next case.
Now assume a clause c with only one false watched literal w_1. If the second watched literal w_2 is true, then c is true and $\lambda(w_1) = \lambda(w_2) = \lambda_0$ so the property is true. If w_2 is undefined, PROPAGATE will look for a second non-false literal w_3. If there is one, c will watch w_3 instead of w_1. Else, it means that the clause is unit, so w_2 is added to the current assignment and c is then a true clause watched by one true and one false literal of the same level.

Recurrence: Let's assume the property holds after the n^{th} conflictless call to PROPAGATE.
If the property holds before the $(n+1)^{\text{th}}$ conflictless call, then we can prove it still holds after this call using the same reasoning as for the initialization phase. However, there may be one or more conflictual calls between the n^{th} and $(n+1)^{\text{th}}$ conflictual call. We will now show by another recurrence that after the backtrack following any of these conflictual calls (but before the learnt clause is added to the formula) the recurrence is verified.
Let's assume the property holded after the previous backtrack (or after the last conflictless call in the case of the initialization). When a conflict occurs, then several decision levels, including the current level, are undone. After a backtrack, all clauses are then either in a state verifying the recurrence property, or in a state reached by deinstantiating some literals from such a recurrence state.
Let c be a clause, w_1, w_2 its watched literals and σ, σ' the partial assignments resp. before the conflictual call and after the following backtrack (so $\sigma' \subseteq \sigma$).

- If $\sigma(c) = \text{undef}$, then by recurrence $\sigma(w_1) = \sigma(w_2) = \text{undef}$. Since $\sigma' \subseteq \sigma$, $\sigma'(w_1) = \sigma'(w_2) = \text{undef}$ so the property still holds.
- If $\sigma(c) = \text{true}$ and $\sigma(w_1) = \text{false}$, then by recurrence $\sigma(w_2) = \text{true}$ and $\lambda(w_2) \leq_\Delta \lambda(w_1)$.
  - If $\sigma'(w_2) = \text{true}$, $\sigma'(c) = \text{true}$ and the property still holds regardless of $\sigma'(w_1)$.
  - Else $\sigma'(w_2) = \text{undef}$. Since $\lambda(w_2) \leq_\Delta \lambda(w_1)$, $\sigma'(w_1) = \text{undef}$, so the property holds regardless of $\sigma'(c)$.
- If $\sigma(c) = \sigma'(c) = \text{true}$ and $\sigma(w_1), \sigma(w_2) \neq \text{false}$, then the property holds regardless of $\sigma'(w_1)$ and $\sigma'(w_2)$.
- If $\sigma(c) = \text{true}$, $\sigma(w_1), \sigma(w_2) \neq \text{false}$ and $\sigma'(c) = \text{undef}$, then $\sigma'(w_1) = \sigma'(w_2) = \text{undef}$ so the property holds.

□

Corollary 2. *After a conflictless run of* PROPAGATE*, no clause is false under the current assignment.*

Proposition 3. *PO-CDCL is propagation-complete.*

Proof. According to Lem. 1, after a conflictless run of PROPAGATE, all unsatisfied clauses watch two distinct undefined literals. Hence, none of these clauses is unit (which proves Prop. 3) or false (which proves 2). □

Theorem 1. *PO-CDCL is correct.*

Proof. A SAT solver is correct iff any total assignment it returns is indeed a model of the input formula, i.e. if it satisfies all clauses. A total assignment can only be returned by PO-CDCL after a conflictless run of PROPAGATE. According to Cor. 2, no clause is false under this assignment. As the assignment is total, no clause can be undefined either. So all clauses are satisfied, and the total assignment is a model. □

Theorem 2. *PO-CDCL is complete.*

Proof. A SAT solver is complete iff it never erroneously reports a satisfiable formula as being unsatisfiable. Lemma 3 of [22] proves the completeness of CDCL by showing that the empty clause can be derived by recursively resolving the final conflict clause against the antecedents of its variables. This proof is also valid within CDCL because according to Prop. 2 all propagations are valid, hence all literals of its antecendent are still false, except for the propagation itself. The proof also shows that the resolution is finite, since the process doesn't resolve against the same variable twice. This is also still true in PO-CDCL, because Δ^+ is a partial order (Cor. 1). □

Theorem 3. *PO-CDCL always terminates.*

Proof. $\forall i \in \mathbb{N}$, let Λ_i and Δ_i be the set of decision levels and the associated partial order after the first i instantiations in the PO-CDCL search ("at time i"). If the search terminates after n instantiations, we will assume that $\forall i > n$, Λ_i and Δ_i represent the state at the end of the search. PO-CDCL as we described it never actually deletes any decision level or dependency, so we can write $\forall i < j \in \mathbb{N}$, $\Lambda_i \subseteq \Lambda_j$ and $\Delta_i \subseteq \Delta_j$. Let us define the (possibly infinite) sets of all decision levels and dependencies during the search: $\Lambda_\infty = \bigcup_{i\in\mathbb{N}} \Lambda_i$, $\Delta_\infty = \bigcup_{i\in\mathbb{N}} \Delta_i^+$. Thanks to the infinite chain of inclusions on $(\Lambda_i)_{i\in\mathbb{N}}$ and $(\Delta_i)_{i\in\mathbb{N}}$, Δ_∞ is a partial order on Λ_∞, and $\forall i \in \mathbb{N}$, $\Delta_\infty \cap (\Lambda_i \times \Lambda_i)$ is a partial order on Λ_i. Let Ψ be any total order extending Δ_i. Similarly, its restriction to $\Lambda_i \times \Lambda_i$ is a total order on Λ_i. We now have a total order on all decision levels which is compatible with the local partial order at any point of the search.

$\forall i \in \mathbb{N}$, $\forall j \in \Lambda_\infty$, let us note $k_i(j)$ the number of variables instantiated at level j at time i (or at the end of the search if it terminated after less than i instantiations).

$$\rho_i(j) = \begin{cases} 0 & \text{if } j = 0 \text{ or } k_i(j) = 0 \\ |\{k \in \Lambda_\infty \setminus \{0\} \mid k <_\Psi j \text{ and } k_i(j) \neq 0\}| + 1 & \text{else} \end{cases}$$

is a function that orders all non-empty decision levels at time i according to Ψ. Finally, let us define

$$f(i) = \sum_{j\in\Lambda_\infty} \frac{k_i(j)}{|\mathcal{V}|^{\rho_i(j)+1}} \ .$$

$f(i)$ is defined, as in Lem. 1 from [22], such that one variable at a decision level j has more weight that the sum of the weight all variables at higher decision levels. As in this lemma, it proves that $f(i)$ is a strictly growing function until the search finishes. Indeed, when some decision levels are uninstantiated, their weight is compensated by the assertion added at assertion level, which is strictly lower than all undone levels.[1] Similarly, the weight of a decision level can decrease when a decision is taken in a formerly empty level with a lower ρ order, but again their weight loss is compensated by the higher weight of this new decision. As $f(i)$ strictly grows as long as the search continues and can only take a finite number of values, the search is finite. □

[1] this proof assumes that for each conflict we set that the conflict level depends of the assertion level, which has been omitted from the presented code but can be added without inconsistency.

Algorithm 8 ANALYZE(ϕ)

/* ϕ is the false clause detected during unit propagation */
/* γ will be the conflict clause produced by conflict analyzis */
$\gamma \leftarrow \phi$
while $\{|\{l \in \gamma \,|\, \lambda(l) = \lambda\}| > 1\}$ **do**
    /* there remains more than one literal of level λ in γ */
    $l \leftarrow \text{LAST}(\gamma, \lambda)$ /* pick the last instantiated literal of level λ in γ */
    $\gamma \leftarrow \gamma \otimes_{var(l)} \alpha(l)$ /* resolution of γ and $\alpha(l)$ on the variable of l */

Definition 3. *A learnt clause is non-redundant if obtained by resolving at least two clauses of the formula. A conflict is non-redundant if its analyzis produces a non-redundant learnt clause.*
A learnt clause is useful if it becomes unit after the backtrack.

Proposition 4. *All clauses learnt during a PO-CDCL are non-redundant and useful.*

Proof. A shown by Alg. 8, if the conflict clause is produced without any resolution, it means that the false clause ϕ only contained one literal of level λ. This implies that before the propagation round responsible for the conflict, either ϕ was already false or it was unsatisfied with only one undefined variable. Both possibilities can be ruled out using the proof of Lem. 1. Hence all clauses learnt during PO-CDCL are non-redundant.
Before the backtrack, γ contains by definition exactly one literal at the conflict level. Since the conflict level is always undone by the backtrack, γ is unit after the backtrack unless another decision level involved in the conflict is undone. The latter case is impossible by definition of the assertion level (see Alg. 6). Therefore γ is useful. □

5 Experimental Results

In order to evaluate the practical efficiency of PO-CDCL, we implemented PO-GLUCOSE[2] as a modification of state-of-the-art solver GLUCOSE 1.0 [1]. Glucose was chosen because it has ranked as one of the most efficient solvers on application benchmarks during the last SAT competitions and races [20] and is based on MINISAT [6] which has also been a regular winner of these competitions.

Our implementation does not explicitely store the entire partial order Δ; instead, we only keep track of all direct dependencies between decision levels. The algorithm only requires to find all levels depending directly or indirectly of candidate assertion levels during the ASSERTIONLEVEL procedure, which can be easily done by a few recursive traversals of the dependency tree from these levels. Maintaining the full transitive relation Δ would require a time-expensive enforcement of transitivity after each new propagation, which is much less efficient according to our preliminary tests.

For the choice of the assertion level, we kept in our experiments the basic CDCL strategy by choosing amongst candidate assertion levels the latest created one. We don't modify restarts (they still undo all instantiations except top level assertions), nor their frequency.

GLUCOSE uses phase saving by default. As we partly designed PO-CDCL as an alternative to phase saving, we disabled it in our implementation PO-GLUCOSE. Moreover, preliminary

[2]Source code of PO-GLUCOSE is available at `http://www.info2.uqam.ca/~villemaire_r/Recherche/SAT/120210partial_order_glucose.tar.gz`

Table 1: Compared performances of Glucose without phase saving (*TO*), Glucose with phase saving (*TO-phase*) and PO-Glucose (*PO*) on the set of 300 application benchmarks from the SAT 2011 competition. The first line shows the total solving time for each implementation (*tot.*), counted in days, hours and minutes. Each instance was given a time limit of one hour, the number of instances that couldn't be solved within that limit is indicated in column *#to*. The second line gives the total number of clause checks needed for solving all instances (checks are counted in billions). Limit was set to 100 billions of checks for each instance, the number of unsolved instances is again given in column *#to*.

	TO		TO-phase		PO	
	#to	tot.	#to	tot.	#to	tot.
time (d:hh:mm)	122	6d02h05m	111	5d15h47m	144	7d03h23m
clause checks (Bn)	113	13 911	103	12 869	127	15 200

experiments indicated us that enabling phase saving in PO-Glucose almost always caused a signification degradation of performances. In order to make sure that performance differences were not solely caused by disabling phase saving, PO-Glucose was compared with the original Glucose implementation including phase saving, but also with a slight variant where phase saving was disabled. Experiments were conducted on a 3.16 GHz Intel Core 2 Duo CPU with 3 GB of RAM, running a Ubuntu 11.10 OS.

Our tests confirm that in practice the PO-CDCL algorithm is able to save instantiations compared to regular CDCL during the solving of any non-trivial benchmark, although the average number of instantiations saved per conflict varies a lot amongst benchmarks (from less than one to several thousands).

In order to test the behaviour of PO-Glucose on a wide range of SAT benchmarks, we ran it on the set of 300 application benchmarks from the SAT 2011 Competition. Results are summarized in Table 1. They clearly show that in general PO-Glucose tends to degradate solving performances compared to Glucose, no matter if phase saving is enabled or not. If we compare PO-Glucose with Glucose with phase saving (the best performing of both Glucose variants), only 26 of the 300 instances are solved faster by PO-Glucose, while 153 are to the contrary solved slower than by Glucose. Glucose is globally slightly less efficient when phase saving is disabled, but even then it still clearly outperforms PO-Glucose.

This counterperformance is partly due to the cost of maintaining and handling dependencies during solving. As we pointed it out, unit propagation is one of the most frequent operation performed during SAT solving and is often responsible for the largest part of the solving time. For all propagations, PO-CDCL requires to ensure that the current decision levels depends on the decision levels of all variables in the antecedent clause. This task is relatively lightweight, but as it occurs very frequently it results in a sensibly slower solving: on average, PO-Glucose performs about 30% less clause checks than Glucose in the same time, and in some extreme cases this decrease can reach 75%. Our implementation of PO-CDCL thus starts with a handicap over the regular CDCL and has to drastically reduce the solving trace in order to outperform it in terms of solving time.

Also, PO-CDCL actually follows a longer search path than CDCL on many instances, despite our original intuition. For instance, amongst the 153 instances on which PO-Glucose takes more time than Glucose, it also performs more clause checks on 143 of them. Since the CDCL algorithm is very sensitive to variations, the partial order may have negative side-effects on some aspects of the algorithm, for instance on the dynamic VSIDS heuristic used to choose decision

Table 2: Compared solving time of Glucose without phase saving (*TO*), Glucose with phase saving (*TO-phase*) and PO-Glucose (*PO*) on some example instances. For each instance, *direct dep.* gives the average direct dependency density $\delta(\Delta_{\text{dir}})$, a lower bound of the actual density $\delta(\Delta)$, during the execution of PO-Glucose. AProVE07-03, homer14.shuffled, post-c32s-gcdm16-23 and k2fix_gr_rcs_w9.shuffled are taken from the application benchmarks of the SAT 2011 competition. 7pipe_k and 12pipe_bug4 are two microprocessor formal verification benchmarks taken respectively from the pipe_unsat_1.0 and pipe_sat_1.0 series.

	TO	TO-phase	PO	direct dep.
AProVE07-03	6m24s	7m16s	16m57s	69.13%
homer14.shuffled	7m51s	10m51s	25m14s	39.47%
post-c32s-gcdm16-23	1m18s	1m20s	3m41s	33.36%
k2fix_gr_rcs_w9.shuffled	>1h00m00s	30m20s	9m13s	4.51%
7pipe_k	23m36s	>1h00m00s	3m07s	3.92%
12pipe_bug4	>1h00m00s	18m49s	4m11s	2.07%

variables. We think the issue is that on many instances the advantages gained from using a partial order are outweighted by these drawbacks.

The principle of PO-CDCL being to take advantage of some independence between decision levels, the obvious question is whether this is a frequent phenomenon in SAT solving. During the solving of a problem, if decision levels often depend on all or most previously created levels, PO-CDCL will behave very similarly to CDCL. In that case the overhead of PO-CDCL obviously comes with little benefit. The independence between decision levels can be measured by the density of the partial order Δ.

At any point of the search, let l be the current number of decision levels (not including level 0). We will define the cardinality of Δ as $|\Delta| = |\{(i,j), i <_\Delta j\}|$, i.e. the number of dependencies between decision levels. The maximal cardinality for l decision levels is $|\Delta|_{\max}(l) = (l-1)(l-2)$; it is reached iff Δ is a total order on the l levels. The current density of Δ is then defined by $\delta(\Delta) = \frac{|\Delta|}{|\Delta|_{\max}(l)}$. A low density (near 0) means that there are very few dependencies between decision levels compared to the maximum possible number of dependencies given the current number of decision levels. Conversely, a value of $\delta(\Delta)$ approaching 1 denotes a high amount of dependencies and means that Δ is close to defining a total order on decision levels. Considering the previous discussion, we expect PO-Glucose to perform better on instances with a low average value of $\delta(\Delta)$ during its execution.

Table 2 shows this average value on some example instances, or more exactly a lower bound of it: the average value of $\delta(\Delta_{\text{dir}}) = \frac{|\Delta|_{\text{dir}}}{|\Delta|_{\max}(l)}$ where Δ_{dir} is the set of direct dependencies between decision levels. These examples seem to validate our intuition that PO-Glucose has more chances to ameliorate performances on instances with low level dependencies. Instances that PO-Glucose solves significantly faster than both Glucose variants often have only around 5% or less of the maximum possible direct dependencies. On the contrary, PO-Glucose tends to generally degradate the solving performance on instances having an average direct density of 30% or more. Partial order CDCL thus has indeed more chances to be efficient on instances where decision levels interact moderately with each other.

Although most SAT instances we thoroughly examined have little independence during the search, we identified at the opposite some benchmark series where all instances share a low dependency level, resulting in most cases in significant solving speedups. For instance,

Table 3: Solving performances of total order Glucose without (TO) and with (TO-phase) phase saving and PO-Glucose (PO) on two benchmark families of formal verification of microprocessors. All tests were run with a time limit of 1 hour. For each test the necessary amounts of time (in seconds) and of clause checks (in millions of checks) is given, and the best performance amongst the three solvers is printed in bold. Average direct density of Δ is respectively 1.5% on pipe_sat_1.0 and 5% on pipe_unsat_1.0. Some instances of pipe_unsat_1.0 have been ommited: 2pipe_k, which in solved in less then 1s and 1M clause checks by all solvers, and 10pipe_k to 14pipe_k, which all 3 solvers are unable to solve within the time limit.

		time (s)			checked clauses (M)		
		TO	TO-phase	PO	TO	TO-phase	PO
pipe_sat_1.0	bug1	>3 600	15	**9**	>68 766	427	**20**
	bug2	>3 600	722	**17**	>30 290	12 122	**117**
	bug3	1 875	**178**	2 246	22 776	**5 344**	30 485
	bug4	>3 600	1 702	**251**	>42 762	52 933	**3 236**
	bug5	115	34	**25**	2 261	1 181	**265**
	bug6	1 750	354	**138**	35 695	10 056	**1 525**
	bug7	>3 600	783	**389**	>21 687	21 393	**3 902**
	bug8	>3 600	**1 569**	3 230	>84 337	35 203	**31 314**
	bug9	>3 600	**5**	13	>66 840	**73**	82
	bug10	**8**	1 525	282	**145**	36 226	3 089
	total	>25 348	6 887	**6 601**	>375 562	174 962	**74 034**
pipe_unsat_1.0	3pipe_k	**0**	2	1	**13**	82	15
	4pipe_k	**6**	22	15	**217**	876	385
	5pipe_k	**13**	68	37	**520**	2 604	911
	6pipe_k	23	77	**9**	847	2 709	**173**
	7pipe_k	1 416	4 727	**187**	56 905	228 327	**3 977**
	8pipe_k	3 538	4 059	**1 058**	139 673	94 965	**27 288**
	9pipe_k	174	258	**150**	5 948	7 187	**2 006**
	total	5 171	9 212	**1 456**	204 123	336 750	**34 756**

table 3 shows detailed statistics obtained on two benchmark sets from formal verification of microprocessors [21]. These benchmarks have particularly low dependency between decision levels, as shown in the caption of Table 3 and on a couple of examples in Table 2, and PO-Glucose significantly outperforms both versions of Glucose on most instances. Moreover, the management of dependency structures is particularly time-expensive on these instances. Thus the performance of PO-Glucose is even more significant when purely algorithmic indicators are considered, such as the total number of checked clauses: speedups up to one or even two orders of magnitude are common. This means that on these instances partial ordering CDCL consistently manages to explore the search space much more efficiently than the regular CDCL algorithm. Moreover, one family contains satisfiable benchmarks and the other unsatisfiable benchmarks. Thus PO-CDCL can be efficient not only for reaching quickly a model of the instance, but also for pruning the search space.

6 Conclusion

In this paper, we addressed the issue of information loss in CDCL algorithms during conflict-directed backtracks. We designed a variation of CDCL that defines a partial order on decision levels, and showed this order allows to undo less instantiations during backtracks, while keeping all essential properties of the algorithm. Finally, we implemented our algorithm in a state-of-the-art SAT solver and evaluated its efficiency. We noticed that PO-CDCL performs particularly well on benchmarks where the partial order as a low average density during the search. Moreover, some series of benchmarks are characterized by a consistently low density and can be solved significantly faster by PO-CDCL.

We are currently exploring some avenues to further ameliorate performances on instances that we already identified as relevant to partial order CDCL. For instance, the choice of the assertion level was set rather arbitrary in the experiments presented above, but using more relevant strategies to choose this level can lead to even better performances on the formal verification instances on which we focussed in this paper.

References

[1] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence*, pages 399–404, 2009.

[2] Ateet Bhalla, Inês Lynce, José T. de Sousa, and João Marques-Silva. Heuristic-based backtracking relaxation for propositional satisfiability. *Journal of Automated Reasoning*, 35(1–3):3–24, October 2005.

[3] Per Bjesse, James H. Kukula, Robert F. Damiano, Ted Stanion, and Yunshan Zhu. Guiding SAT diagnosis with tree decompositions. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing – 6th International Conference, SAT 2003*, volume 2919 of *Lecture Notes in Computer Science*, pages 315–329. Springer, 2004.

[4] Christian Bliek. Generalizing partial order and dynamic backtracking. In *AAAI/IAAI '98 Proceedings*, pages 319–325. AAAI Press / The MIT Press, 1998.

[5] Vijay Durairaj and Priyank Kalla. Exploiting hypergraph partitioning for efficient boolean satisfiability. In *Ninth IEEE International High-Level Design Validation and Test Workshop, 2004*, pages 141–146. IEEE Computer Society, 2004.

[6] 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*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[7] Matthew L. Ginsberg. Dynamic backtracking. *Journal of Artificial Intelligence Research*, 1:25–46, August 1993.

[8] Matthew L. Ginsberg and David McAllester. GSAT and dynamic backtracking. In Alan Borning, editor, *PPCP'94 Proceedings*, volume 874 of *Lecture Notes in Computer Science*, pages 243–265. Springer, 1994.

[9] Jinbo Huang and Adnan Darwiche. A structure-based variable ordering heuristic for SAT. In Georg Gottlob and Toby Walsh, editors, *IJCAI-03, Proceedings of the Eighteenth International Joint Conference on Artificial Intelligence*, pages 1167–1172. Morgan Kaufmann, 2003.

[10] Philippe Jégou and Cyril Terrioux. Hybrid backtracking bounded by tree-decomposition of constraint networks. *Artificial Intelligence*, 146(1):43–75, 2003.

[11] Wei Li and Peter van Beek. Guiding real-world SAT solving with dynamic hypergraph separator decomposition. In *16th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2004)*, pages 542–548. IEEE Computer Society, 2004.

[12] João P. Marques-Silva, Ines Lynce, and Sharad Malik. Conflict-driven clause learning SAT solvers. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, chapter 4, pages 131–153. IOS Press, 2009.

[13] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999.

[14] David A. McAllester. Partial order backtracking. Research note, Artificial Intelligence Laboratory, MIT, 1993.

[15] Anthony Monnet and Roger Villemaire. Scalable formula decomposition for propositional satisfiability. In *C^3S^2E '10 Proceedings*, pages 43–52. ACM, 2010.

[16] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: engineering an efficient SAT solver. In *Proceedings of the 38^{th} Design Automation Conference (DAC 2001)*, pages 530–535. ACM Press, 2001.

[17] Alexander Nadel and Vadim Ryvchin. Assignment stack shrinking. 6175:375–381, 2010.

[18] Knot Pipatsrisawat and Adnan Darwiche. A lightweight component caching scheme for satisfiability solvers. In João P. Marques-Silva and Karem A. Sakallah, editors, *Theory and Applications of Satisfiability Testing - SAT 2007, 10^{th} International Conference*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[19] Neil Robertson and Paul D. Seymour. Graph minors. II. Algorithmic aspects of tree-width. *Journal of Algorithms*, 7(3):309–322, September 1986.

[20] The international SAT Competitions web page. `http://www.satcompetition.org`.

[21] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, February 2003.

[22] Lintao Zhang. *Searching for Truth: Techniques for Satisfiability of Boolean Formulas*. PhD thesis, Princeton University, June 2003.