# Efficient Partial Order CDCL Using Assertion Level Choice Heuristics*

Anthony Monnet and Roger Villemaire

Université du Québec à Montréal, Montreal, Canada
anthonymonnet@aol.fr
villemaire.roger@uqam.ca

**Abstract.** We previously designed Partial Order Conflict Driven Clause Learning (PO-CDCL), a variation of the satisfiability solving CDCL algorithm with a partial order on decision levels, and showed that it can speed up the solving on problems with a high independence between decision levels. In this paper, we more thoroughly analyze the reasons of the efficiency of PO-CDCL. Of particular importance is that the partial order introduces several candidates for the assertion level. By evaluating different heuristics for this choice, we show that the assertion level selection has an important impact on solving and that a carefully designed heuristic can significantly improve performances on relevant benchmarks.

## 1 Introduction

The SAT problem consists in deciding if a given propositional formula expressed in conjunctive normal form is satisfiable, i.e. if there exists a truth assignment that makes the formula true. Furthermore, a satisfying assignment, or model, has to be returned if the formula is satisfiable. Many decision problems can be encoded using a propositional formula, such that this formula is satisfiable iff the considered problem has a solution.

Conflict-driven clause learning (CDCL) [9] is the algorithm used by the most efficient complete SAT solvers. Unlike basic depth-first search that only undoes the last decision when a conflict is reached, CDCL is able to analyze the reasons for this conflict and to define the assertion level, which is the second to last decision level involved in this conflict. It then backtracks directly to this assertion level, often undoing several decision levels at once, in order to ensure that this conflict will not be encountered again in this branch of the search. It therefore performs a much more efficient pruning of the search space than regular depth-first search, often leading to a significantly faster solving of problems.

CDCL has however the negative side-effect of destroying parts of the current partial assignment not directly related to the conflict. Indeed, by returning straight to the assertion level, it entirely destroys all instantiations in subsequent decision levels. By definition, none of them were directly involved in the conflict,

* We gratefully acknowledge the financial support of the Natural Sciences and Engineering Research Council of Canada on this research.

except for the decision level where the conflict was discovered. In the worst case, these instantiations may even belong to a different connected component of the problem and couldn't possibly be affected by the conflict resolution, even indirectly. CDCL thus can cause the unnecessary deletion of previous parts of the search, which may prevent the detection of some conflicts or the completion of a satisfying assignment, ultimately slowing down the solving process.

This deletion of unrelated parts of the search is caused by the implicit total ordering on successive decisions during the search, and this total order can be relaxed without damaging the correctness, completeness and termination of the algorithm. We therefore designed partial order CDCL (PO-CDCL) [11], a variant of CDCL maintaining a partial order between decision levels, which allows to locally undo less instantiations during a conflict-directed backtrack. In practice, some SAT problems (for instance encodings from the formal verification of superscalar microprocessors [17]) have a relatively sparse dependency between decision levels during solving, and we showed that PO-CDCL significantly decreases the solving time on these instances.

The aim of this paper is twofold. First, we show that the efficiency of PO-CDCL is due to the fact that it dramatically reduces the search efforts needed to reach successive conflicts and hence prune the search space. Secondly, we consider a new parameter of the algorithm introduced by the partial order: unlike in a regular CDCL, the assertion level of a conflict clause is not uniquely defined and can be chosen using various heuristics. We show that this choice has a significant impact on the search, and that heuristics affecting the average amount of instantiations undone by conflicts can further significantly improve the performance of PO-CDCL. Interestingly, the solving of satisfiable problems is improved when this average amount of undone instantiations increases, while unsatisfiability is proved faster when it decreases. Moreover, this quantity is most efficiently controlled indirectly by choosing assertion levels that maximize or minimize the number of additional dependencies they would introduce between decision levels.

The remainder of this paper is organized as follows: Section 2 introduces the PO-CDCL algorithm. Section 3 reviews related methods seeking to reduce the amount of instantiations deleted during a non-chronological backtracking algorithm in CSP and SAT. Finally, section 4 presents experimental results obtained with the implementation of PO-CDCL in a state-of-the-art CDCL solver using various heuristics for the choice of the assertion level. These results are used to analyze the causes of the efficiency of PO-CDCL with various assertion level heuristics on satisfiable and unsatisfiable SAT instances with low level interdependencies.

## 2 PO-CDCL

CDCL [9] is a satisfiability solving algorithm based on the older depth-first search DPLL [3], enhanced with conflict-directed backtracking and clause learning. It successively assigns arbitrary values to variables (it takes decisions) until

either a clause is violated or all variables are assigned. After each decision, an exhaustive round of unit propagation is performed to deduce all possible consequences of the current assignment using this inference rule. A decision level is the set formed by a decision and all the propagations it entails.

Unit clauses are efficiently detected using watched literals [13], a method keeping track of two not instantiated literals in each clause that isn't already satisfied. When a literal $l$ is instantiated, a clause $c$ cannot become unit unless it contains the opposite literal $\neg l$ and this literal is watched in $c$. The algorithm thus only has to check clauses containing $\neg l$ as a watched literal for possible unit propagations.

A conflict occurs when all literals in a clause are false. CDCL then infers a conflict clause $\gamma$, which is a logical consequence of the original formula, is also false under the current assignment and has only one literal instantiated at the current decision level. The second largest decision level represented in $\gamma$ is called the assertion level. The conflict is resolved by undoing all decision levels above the assertion level. $\gamma$ becomes unit, it is propagated and the search continues at the assertion level. The algorithm terminates either when all variables are assigned without causing any conflict (the formula is satisfied by this assignment) or when a conflict occurs at decision level 0 (the formula is unsatisfiable).

The pseudocode of PO-CDCL is given in Alg. 1. It consists in a few modifications of the regular CDCL algorithm. A partial order $\Delta$ keeps track of dependencies between decision levels and is used to determine the assertion level and the levels to delete during conflicts. Dependencies are added during the unit propagation phase detailed in Alg. 2. A level $i$ depends on a level $j$ (noted $j <_\Delta i$) when level $j$ had an influence on unit propagations at level $i$. This can happen in two cases.

First, when a variable $l$ is propagated by a unit clause $c$, this propagation obviously depends of all other literals in $c$. For all literals $l' \in c \setminus \{l\}$ whose decision level is different from the current decision level $\lambda$, the dependency $level(l') <_\Delta \lambda$ is added to $\Delta$. This case is handled by lines 14 and 15 of Alg. 2.

Secondly, when a false watched literal $\neg l$ at level $\lambda$ doesn't need to be replaced in a clause $c$ because $w$, the second watched literal in $c$, is true, the dependency $level(w) <_\Delta \lambda$ is added (line 7 of Alg. 2). Intuitively, this dependency means that the true watched literal $w$ avoided a watched literal replacement at level $\lambda$ and therefore had an impact on the unit propagations at this level. More technically, this dependency ensures that the clause will remain correctly watched by forbidding to uninstantiate $w$ while keeping $\neg l$ instantiated.

With a partial order on decision levels, the backtrack phase only requires to delete levels that depend on the assertion level (and of course the conflict level itself). This deletion (at lines 12 and 13 of Alg. 1) is necessary to keep the consistency of the algorithm by preventing circular dependencies between levels.

Finally, the last modification affects the definition of the assertion level. This level has to be involved in the conflict clause, and the conflict clause must become unit after the backtrack. This implies that no decision level occuring in the conflict clause must be undone by the backtrack, except for the conflict level

**Algorithm 1** PO-CDCL

```
 1: σ ← ∅  /* begin with the empty assignment */
 2: λ = 0  /* λ is the current decision level */
 3: loop
 4:     c ←Propagate /* propagate new instantiations */
 5:     if c ≠ NIL then /* a conflict was found during propagations */
 6:         if λ = 0 then /* conflict at decision level 0 */
 7:             return false /* unsatisfiable problem */
 8:         else
 9:             γ ← Analyze(c) /* infer the conflict clause γ */
10:             candidates ← all Δ-maximal elements in levels(γ) \ {λ}
11:             choose a in candidates /* a is the assertion level */
12:             for l >_Δ a do
13:                 delete level l
14:             λ ← a /* a becomes the current level */
15:             Learn(γ)
16:             PropagateAssertion(γ)
17:     else /* no conflict during propagations */
18:         if all variables are instantiated then
19:             return σ /* σ is a model */
20:         else
21:             λ ← NewLevel
22:             Decide(λ)
```

**Algorithm 2** Propagate

```
 1: Π ←{instantiations not yet propagated}
 2: while {Π ≠ ∅} do
 3:     choose l ∈ Π
 4:     for all clauses c s.t. ¬l is watched in c do
 5:         w ← the second watched literal in c
 6:         if σ(w) = true then
 7:             set level(w) <_Δ λ
 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 ∈ levels(c) \ {λ} do
15:                         set l <_Δ λ
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 NIL /* no conflict occured */
```

$\lambda$. In a total order CDCL, the assertion level is uniquely defined as the second largest decision level in the conflict clause. With a partial order, however, any decision level in the conflict clause can be chosen as the assertion level, provided that no other level involved in the conflict, except $\lambda$, depends on it. In other words, the assertion level can be any maximal element of $<_\Delta$ restricted to the set of conflict clause levels different from $\lambda$ (lines 10 and 11 of Alg. 1).

Similarly to the original CDCL algorithm, PO-CDCL is complete, correct and always terminates [11].

## 3 Related Works

PO-CDCL is conceptually related to some variations of the Conflict-Direct Backjumping (CBJ) algorithm for CSP solving which, similarly to CDCL for SAT, resolves conflicts by computing a nogood (equivalent of the conflict clause) and deleting the entire search progress starting at the culprit variable decision (roughly equivalent of the decision at the conflict level). In the case of CSPs, search progress consists not only of variable assignments, but also of values eliminated from domains of variables.

Dynamic Backtracking (DB) [5], in contrast with CBJ, only undoes the culprit variable and restores only eliminated values for which the culprit variable was part of the nogood. This strategy is equivalent to dynamically moving the culprit variable to the end of the search branch before undoing it, provided a limited amount of search information is deleted. It has the advantage of only partially undoing the work made after the culprit variable. Similarly to PO-CDCL, it minimizes the quantity of undone search progress by relaxing the strict total order on variable decisions. The main difference is that DB is defined as a search-only algorithm without any inference; therefore the conflict can always be resolved without undoing any other decision than the culprit variable. Also note that the usual total order is considered during the analysis phase; unlike the assertion level in PO-CDCL, the culprit variable in DB remains thus uniquely defined.

Partial Order Backtracking (POB) [10] similarly only uninstantiates the culprit variable for each conflict and only restores values whose elimination depended on it. The difference is that it initially allows to choose the culprit variable amongst all variables in the nogood, but progressively sets precedence constraints between variables in order to ensure termination. This freedom in the choice of the culprit variable is stronger than the freedom PO-CDCL offers for choosing the assertion level. It however however comes with a strong permanent and increasing constraint on decision heuristics, whereas constraints set by PO-CDCL between decision levels only apply until these decision levels are undone, and hence have no impact on the choice of decision variables.

Tree decompositions methods integrated within CDCL [6,8,4,12] and CBJ [7] solvers also indirectly limit the quantity of unrelated instantiations undone during a backtrack. Decompositions [16] are used to compute recursive separators of the instance, i.e. sets of variables whose instantiation breaks the problem in

several connected components. These methods start the search by instantiating all separator variables, and then completely instantiate a connected component before making any decision in another component. When a conflict occurs in a connected component, the resulting backtrack then can't destroy any part of the search in other components thanks to this constrained ordering. Besides scalability issues which make it very difficult to efficiently compute useful decompositions on large SAT problems [12], tree decompositions only capture the static connectivity of a problem and therefore can't take into account the polarity of instantiations and the many propagations they cause. At any point of the search, the actual connectivity is likely to be much more sparse than predicted by decompositions. Therefore, a conflict in a connected component may actually delete instantiations in another component. PO-CDCL, on the other hand, considers the exact connectivity at any time of the search. It also distinguishes sets of variables that haven't interacted yet in the current search branch even if they belong to the same connected component; it considers actual interactions between already instantiated variables rather than potential interactions between still unassigned variables.

Finally, phase saving [15], in contrast with tree decompositions, is a 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. It thus allows to recover search progress that was lost during a conflict resolution. However, unlike partial order CDCL, this recovery doesn't save the computational cost of repeating the time-consuming propagation phase. Also, phase saving memorizes the polarity of all variables, even if they were actually involved in the conflict. This side effect sometimes decreases solving performance, as reported by the authors [15].

Note that, at the opposite, some strategies have been designed to enhance SAT solving by increasing the quantity of instantiations undone during conflict-directed backtracks [14,2].

## 4 PO-CDCL Analysis and Assertion Level Heuristics

The PO-CDCL algorithm was implemented by introducing a partial order on decision levels in the state-of-the-art CDCL solver Glucose 1.0 [1]. The resulting PO-CDCL solver is named PO-Glucose and its source code is available at `http://www.info2.uqam.ca/~villemaire_r/Recherche/SAT/120619generalized_glucose.tar.gz`. In this implementation, level dependencies are stored in three structures: two directed adjacency lists, representing the relation in both directions, and one boolean matrix. The combination of these structures allows to perform efficiently all operations on the partial relation: somes cases require to check the relation between a precise pair of decision levels, which can be done in constant time using the matrix. At the opposite, the algorithm sometimes requires to list of all levels depending on a given level, in which case using the adjacency list is obviously more efficiently, particularly when there are

many active decision levels with little interdependence. Note that only direct dependencies are stored; transitive dependencies are only needed during conflict resolution on a small subset of variables and it is much more efficient to compute this partial transitive closure when it is required than to enforce and store transitivity during the entire algorithm.

As the size of the matrix grows quadratically with the number of decision levels, our implementation disables it if this number reaches a predefinite threshold. The algorithm then proceeds using only adjacency lists, which is slightly less efficient but significantly better than exhausting primary memory. Theoretically, the size of adjacency lists could also grow quadratically in the case of dense dependencies between decision levels; however, it seems that in practice the number of decision levels tends to decrease when this density grows. The memory requirement of adjacency lists thus remains relatively moderate.

The remaining of this section presents and compares experimental results obtained with this implementation and with the original Glucose solver. We will more particularly focus on the impact of assertion level choice heuristics on the overall behaviour and performance of the algorithm. All tests were run on a 3.16 GHz Intel Core 2 Duo CPU with 3 GB of RAM, running a Ubuntu 11.10 OS, with a time limit of 1 hour for each execution (not including the preprocessing phase, which is identical for all tested variants).

We previously noticed [11] that since PO-CDCL is designed to take advantage of the independence between decision levels during solving, it performs best on problems where this independence is relatively high. If we consider the partial order $\Delta$ as a set of ordered pairs of decision levels, such that the first level in each pair depends of the second level of the pair, the cardinality of $\Delta$ can be used as a measure of this independence. Benchmarks from formal verification of superscalar microprocessors [17] are an example of problems with a very sparse relationship between levels, possibly because of the high parallelism in verified models. Therefore, the following experiments were conducted on 6 series of these benchmarks:

- pipe_unsat_1.0 and pipe_unsat_1.1 verify correct specifications of various-sized superscalar microprocessors with two different encoding variants;
- pipe_sat_1.0 and pipe_sat_1.1 represent ten different buggy variants of the size 12 case, again encoded in two different ways;
- pipe_ooo_unsat_1.0 and pipe_ooo_unsat_1.1 are two different encodings verifying the correctness of various-sized superscalar microprocessors handling out-of-order execution of instructions.

Benchmarks verifying correct and buggy specifications are respectively unsatisfiable and satisfiable.

Glucose implements the phase saving strategy mentioned in section 3. We disabled phase saving in PO-Glucose because partial order CDCL was partly designed as an alternative to phase saving. Moreover, preliminary tests indicated that PO-Glucose often performs significantly better with phase saving disabled. To make sure the performance differences we observe are not simply caused by the presence or abscence of phase saving rather than by the partial order, we

| family | #inst | TO | | TO-phase | | PO | | PO-least-undos | | PO-most-undos | | PO-least-deps | | PO-most-deps | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | #to | time | #to | time | #to | time | #to | time | #to | time | #to | time | #to | time |
| pipe_sat_1.0 | 10 | 6 | 25 364 | 0 | 6 887 | 0 | 6 601 | 0 | 7 334 | 0 | 2 042 | 0 | 1 264 | 2 | 10 399 |
| pipe_sat_1.1 | 10 | 1 | 7 258 | 0 | 1 182 | 1 | 3 766 | 1 | 4 010 | 0 | 186 | 0 | 185 | 1 | 3 820 |
| pipe_unsat_1.0 | 13 | 5 | 23 172 | 7 | 25 627 | 5 | 19 456 | 5 | 19 697 | 5 | 19 192 | 5 | 20 338 | 4 | 17 742 |
| pipe_unsat_1.1 | 14 | 5 | 20 706 | 7 | 28 460 | 6 | 23 591 | 6 | 23 130 | 6 | 22 837 | 6 | 23 149 | 6 | 22 198 |
| pipe_ooo_unsat_1.0 | 9 | 2 | 10 757 | 1 | 6 989 | 2 | 11 670 | 3 | 13 321 | 2 | 10 613 | 2 | 10 799 | 2 | 10 420 |
| pipe_ooo_unsat_1.1 | 10 | 1 | 11 457 | 4 | 30 563 | 1 | 12 153 | 2 | 16 185 | 2 | 15 909 | 2 | 16 485 | 1 | 11 592 |
| total | 66 | 20 | 98 714 | 19 | 99 708 | 15 | 77 237 | 17 | 83 677 | 15 | 70 870 | 15 | 72 236 | 16 | 76 196 |

Table 1: Compared performances of Glucose without (*TO*) and with (*TO-phase*) phase saving, PO-Glucose with the default chronological assertion level heuristic (*PO*) and with 4 other heuristics based on the amount of instantiations undone by the backtrack (*PO-least-undos*, *PO-most-undos*) or on the number of level dependencies added (*PO-least-deps*, *PO-most-deps*). For each *series* of benchmarks, containing *#inst* instances, the number of timeouts (*#to*) and the total solving time in seconds (*time*) is given.

| series | #inst | TO | | TO-phase | | PO | | PO-least-undos | | PO-most-undos | | PO-least-deps | | PO-most-deps | |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| | | #to | checks | #to | checks | #to | checks | #to | checks | #to | checks | #to | checks | #to | checks |
| pipe_sat_1.0 | 10 | 6 | 60 962 | 0 | 174 962 | 0 | 74 034 | 0 | 104 876 | 0 | 23 970 | 0 | 12 816 | 2 | 45 065 |
| pipe_sat_1.1 | 10 | 1 | 257 229 | 0 | 38 492 | 1 | 1 438 | 1 | 4 542 | 0 | 1 361 | 0 | 1 123 | 1 | 1 938 |
| pipe_unsat_1.0 | 13 | 5 | 204 171 | 7 | 336 799 | 5 | 34 804 | 5 | 39 208 | 5 | 23 256 | 5 | 52 394 | 4 | 58 161 |
| pipe_unsat_1.1 | 14 | 5 | 95 137 | 7 | 384 249 | 6 | 51 028 | 6 | 35 829 | 6 | 26 370 | 6 | 34 717 | 6 | 11 874 |
| pipe_ooo_unsat_1.0 | 9 | 2 | 124 488 | 1 | 107 063 | 2 | 75 783 | 3 | 48 229 | 2 | 55 363 | 2 | 56 183 | 2 | 51 501 |
| pipe_ooo_unsat_1.1 | 10 | 1 | 141 491 | 4 | 57 129 | 1 | 76 962 | 2 | 31 329 | 2 | 25 691 | 2 | 35 458 | 1 | 66 023 |
| total | 66 | 20 | 740 730 | 19 | 1 098 693 | 15 | 314 050 | 17 | 264 013 | 15 | 156 011 | 15 | 192 692 | 16 | 234 562 |

Table 2: Compared performances of the same Glucose and PO-Glucose variations on the same series of benchmarks. For each series, besides the number of timeouts (*#to*), the total number of clause checks performed (*checks*, given in millions) is listed. When several solvers timed out on the same instance, they were considered as having all needed the same amount of clause checks (the smallest amount amongst timed out solvers).

compared PO-Glucose with the original Glucose, but also with a variant in which phase saving is disabled.

In PO-Glucose, the partial order management causes a significant calculation overhead during solving. Indeed, each propagation requires to check and possibly add several level dependencies. As a result, given the same execution time on the same instance, PO-Glucose performs on average about 40% less clause checks (i.e. the number of executions of the innermost **for** loop at lines 4 to 21 of Alg. 2) than Glucose. We think this overhead can't be significantly reduced unless we find some lazy strategy to manage dependencies. Therefore, besides the CPU time used to solve each instance, we also report the number of clauses checked for possible propagations during solving. This quantity gives some insight about which proportion of the PO-Glucose solving time is spent in the search itself and to what extent this time is due to dependency management.

### 4.1 Analyzing efficiency of PO-CDCL

In this subsection, we will consider the default version of PO-Glucose as described in [11] with a choice of the assertion level similar to its definition in a total order CDCL: amongst all candidate assertion levels, the most recently created one is picked. This default version is named *PO* in all tables and figures of this paper. Results of Glucose with and without phase saving are labelled as *TO-phase* and *TO* respectively, *TO-phase* being the default Glucose setting.

As expected, when a conflict occurs during a CDCL solving, there is in practice often a non-negligible quantity of instantiations between the assertion level and the conflict level. Therefore, on our formal verification instances, PO-Glucose locally saves on average 15% of instantiations that would be deleted by a regular CDCL algorithm (they are located in decision levels instantiated after the assertion level but not depending on it). If we consider an entire solving trace, it however deletes on average approximately the same number of instantiations per conflict than the original Glucose, as shown in Table 3. The efficiency of PO-Glucose is thus not obtained by accumulating instantiations faster than with a total order; saved instantiations are likely to be deleted later. However, we will show that although instantiations are only saved temporarily, they can have a significant impact on the overall search.

Tables 1 and 2 show respectively the total time and clause checks needed to solve each benchmark family with this chronological heuristic, compared with performances of the two total order variants. Both versions of Glucose have very contrasted results: the default version with phase saving clearly outperforms the version without phase saving on both satisfiable series, but conversely the version without phase saving performs better on 3 of the 4 unsatisfiable families.

When comparing solving time for each series separately, the performance of PO-Glucose is generally close to the best performing Glucose version and significantly better than the other (except on pipe_ooo_unsat_1.0, where it requires a little more time than the slowest Glucose variant). It also never causes more than one additional timeout than the best performing Glucose version. Thanks to this more balanced behaviour, it significantly outperforms

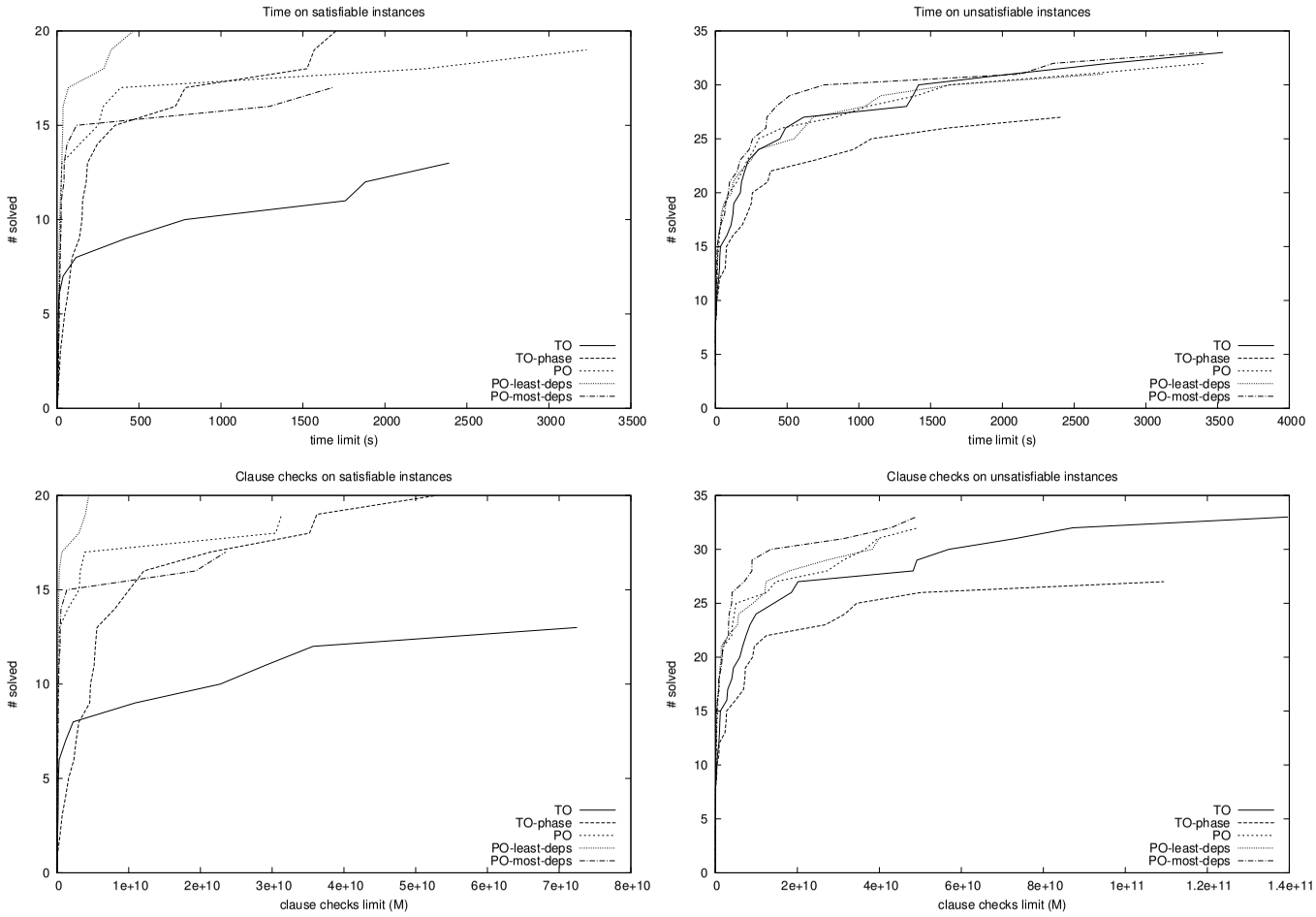


Fig. 1: Four cactus plots comparing the performances of two variants of Glucose, without (*TO*) and with *(TO-phase*) phase saving, with the default PO-CDCL (*PO*) and PO-CDCL with two dependency-counting heuristics (*PO-least-deps, PO-most-deps*). Left plots compare the algorithms on the 20 satisfiable instances, right plots on the 46 unsatisfiable instances. x-axis measures the solving time on top plots and the clause checks performed on bottom plots.

both Glucose with and without phase saving when considering the total solving time on all benchmarks, and manages to solve 4 to 5 more instances in the given time limit.

The cactus plots of Fig. 1 give a better view of the performances on individual instances. Top figures show how many satisfiable and unsatisfiable instances respectively can be solved within a given time limit. The top left figure indicates that PO-Glucose manages to solve many instances very quickly (13 out of 20 are solved in less than 30 seconds each). When the time limit increases, it is eventually beaten by the default setup of Glucose which is able to solve the 3 most difficult instances in a little less than 30 minutes while PO-Glucose needs more time and fails to solve one of them within one hour. It however easily outperforms Glucose without phase saving.

On unsatisfiable instances (top right figure), PO-Glucose considerably outperforms the default version of Glucose with phase saving enabled, no matter what time limit is considered. Within one hour, it solves 5 more instances than default Glucose. Glucose without phase saving is however more efficient and slightly outperforms PO-Glucose on high time limits, although the latter manages to solve more instances in 500 seconds or less.

Since about 40% of the solving time is an overhead due to handling level dependencies, the performance of PO-Glucose is even better when the number of clause checks is considered. Bottom plots of Fig. 1 indicate that PO-Glucose significantly outperforms both versions of Glucose on most satisfiable and unsatisfiable instances. This observation is confirmed by Table 2, which shows that PO-Glucose often requires dramatically less clause checks to solve the same amount of instances than Glucose. Overall, PO-Glucose solves more instances than both Glucose implementations with twice to thrice less clause checks.

This efficiency can be explained by the effect of saved instantiations on the search. Table 4 shows the average amount of clause checks necessary to reach a conflict for various Glucose and PO-Glucose versions. This quantity is almost always dramatically lowered by PO-Glucose, no matter what assertion heuristic is used. The instantiations saved by partial order backtracks, even if they are eventually deleted, seem to be often relevant and help reaching conflicts much faster. As each conflict prunes a part of the search space, partial order thus apparently dramatically improves this pruning, which obviously should help in proving unsatisfiability faster, but in also guiding the search in satisfiable instances towards branches of the search space containing models.

### 4.2 Assertion level heuristics

The default chronological assertion level choice used in the previous subsection was designed to remain as close as possible to the original CDCL algorithm and evaluate the efficiency gain that can be obtained solely by removing less instantiations during backtracks, without further modifying the search. However, we will show that this choice can significantly modify the way the search space is explored, and that particular heuristics can be used to further improve performances of PO-Glucose.

| family | #inst | TO | TO-phase | PO | PO-least-undos | PO-most-undos | PO-least-deps | PO-most-deps |
|---|---|---|---|---|---|---|---|---|
| pipe_sat_1.0 | 10 | 1 226 | 2 053 | 1 751 | 1 698 | 3 680 | 4 602 | 1 972 |
| pipe_sat_1.1 | 10 | 1 099 | 1 726 | 1 957 | 1 599 | 3 834 | 4 983 | 1 691 |
| pipe_unsat_1.0 | 13 | 885 | 968 | 1 050 | 1 046 | 1 241 | 1 244 | 970 |
| pipe_unsat_1.1 | 14 | 1 124 | 1 054 | 1 096 | 1 066 | 1 284 | 1 316 | 974 |
| pipe_ooo_unsat_1.0 | 9 | 648 | 679 | 660 | 648 | 685 | 693 | 624 |
| pipe_ooo_unsat_1.1 | 10 | 784 | 610 | 749 | 718 | 781 | 755 | 741 |
| average | 11 | 972 | 1 172 | 1 204 | 1 129 | 1 867 | 2 185 | 1 151 |

Table 3: Comparison of the average number of instantiations undone at each backtrack by various solvers on some benchmark series. Solvers and benchmarks tested are the same as in Table 1.

| series | #inst | TO | TO-phase | PO | PO-least-undos | PO-most-undos | PO-least-deps | PO-most-deps |
|---|---|---|---|---|---|---|---|---|
| pipe_sat_1.0 | 10 | 5 164 | 276 | 18 | 23 | 13 | 9 | 37 |
| pipe_sat_1.1 | 10 | 2 999 | 16 | 11 | 17 | 6 | 10 | 25 |
| pipe_unsat_1.0 | 13 | 1 214 | 1 499 | 21 | 28 | 18 | 10 | 39 |
| pipe_unsat_1.1 | 14 | 203 | 1 271 | 19 | 21 | 13 | 9 | 19 |
| pipe_ooo_unsat_1.0 | 9 | 99 | 13 | 7 | 8 | 5 | 4 | 7 |
| pipe_ooo_unsat_1.1 | 10 | 25 | 1 283 | 9 | 9 | 6 | 6 | 8 |
| average | 11 | 1 546 | 805 | 14 | 19 | 11 | 8 | 23 |

Table 4: Comparison of the average number of clause checks (in millions) needed to reach a conflict by various solvers on some benchmark series. Solvers and benchmarks tested are the same as in Table 1. Note that the correlation between solving performances and the number of clause checks per conflict can be confirmed by comparing both total order versions *TO* and *TO-phase*: the best performing version on a benchmark series is always the one with the least checks per conflict.

Tests with the chronological assertion level choice showed that in 31% of the conflicts, there are several candidate assertion levels, and when it happens there are on average about 10 distinct candidate levels. The strategy used to choose the assertion level thus can potentially have a significant impact on the entire search. Since the primary goal of PO-CDCL is to save instantiations during backtracks, a straightforward local heuristic (named *PO-less-undos* in tables) consists in picking the candidate assertion level that will undo the least instantiations, i.e. that minimizes the quantity of variables located in decision levels depending on the candidate assertion level. However, according to Table 3, this strategy almost doesn't modify the average number of undos per conflict. Consequently, performances obtained with this heuristic are relatively close to results of the default chronological heuristic, as shown in Tables 1 and 2. This seems to indicate that the chronological heuristic already often picks assertion levels causing few uninstantiations.

Surprisingly, the opposite heuristic of picking the assertion level that will cause the most deletions (*PO-most-undos*) is much more interesting. Its performances on unsatisfiable instances are very close to performances of the chronological heuristic. However, as shown in Tables 1 and 2, it dramatically reduces the time and clause checks needed to solve satisfiable instances. pipe_sat_1.0 is solved about 3 times faster and with 7 times less clause checks than the best performing GLUCOSE version. pipe_sat_1.1 is solved more than 6 times faster and with 28 times less clause checks.

On these satisfiable series, as indicated by Table 3, the *most undos* heuristic deletes about twice more instantiations than default PO-GLUCOSE and both total order GLUCOSE implementations. The performance of this heuristic is likely due to this large amount of deletions, coupled to the frequent conflicts caused by partial order CDCL. Our intuition was that keeping as many instantiations as possible would help building a model of the instance faster, but apparently undoing as many instantiations as possible is more useful. It indeed certainly allows to skip unsatisfiable parts of the search space more quickly and to explore more various parts of this space.

Heuristics based on counting instantiations to be undone during the conflict have the drawback to be highly local, and consequently they generally don't reach their goal globally. Indeed, the choice of the assertion level doesn't only affect the current backtrack: dependencies are added between this level and all other levels involved in the conflict. These additional dependencies increase the likelihood for the chosen assertion level to be deleted in future conflicts. If the conflict clause involves $n$ decision levels (not including the conflict level), the assertion level will have to depend on all other $n-1$ levels, but some of these dependencies may already exist. Intuitively, picking the candidate assertion level which will entail the least new dependencies should tend to globally lower the average quantity of instantiations undone during a conflict. Conversely, we expect the opposite heuristic to delete more instantiations per conflict.

For some unexplained reason, it is exactly the opposite that happens. The *least dependencies* strategy causes even more uninstantiations than the *most un-*

*dos* heuristic, causing a slight increase of solving time on unsatisfiable instances, but a further improvement of performances on satisfiable instances. Figure 1 shows that with this heuristic 17 of the 20 satisfiable instances are solved within 70 seconds, the 3 remaining instances being solved in less than 500 seconds each. In contrast, 11 instances require more than 100 seconds and 3 more than 1 500 seconds with the best performing GLUCOSE version.

On the other hand, the *most dependencies* heuristic performs poorly on satisfiable instances but very well on unsatisfiable instances. Figure 1 shows that it is by far the best tested solver in terms of checked clauses and that it even steadily outperforms the best GLUCOSE version on all time limits.

This performance is explained by a sensible decrease of the average number of undone instantiations per conflict compared to other PO-Glucose implementations, as shown in Table 3. In the case of unsatisfiable instances, undoing less instantiations seems to help focussing the search on the currently active part of the search space. Favorizing successive conflicts in related parts of the search space results in a more efficient pruning and ultimately requires less conflicts to prove unsatisfiability: regular GLUCOSE with and without phase saving need on average about 7 and 4,6 millions of conflicts respectively for solving unsatisfiable benchmarks. This number drops to between 2 and 2,7 millions of conflicts for previous PO-GLUCOSE variants, and down to 1,75 million with the *most dependencies* heuristic.

These dependencies-oriented heuristics and their contrasted efficiency suggest that on SAT problems with low decision level interdependencies, satisfiability solving can be significantly improved by using totally different strategies depending on the actual satisfiability of the instance: if a model exists, it can be found easier if backtracks undo many instantiations, which helps exploring the search space more dynamically. In the unsatisfiable case, backtracks should at the opposite undo less instantiations to help focus the search on the currently active search space and prove unsatisfiability with less conflicts. Moreover, both types of strategies can be carried out by an appropriate choice of assertion levels in a partial order CDCL search.

Satisfiability of instances with sparse level dependencies can thus be very efficiently checked with PO-CDCL if the answer is known or speculated prior to solving. We think it should be possible to design a more balanced intermediate strategy that would perform significantly better than total order CDCL regardless of the instance satisfiability.

## 5 Conclusion

In this paper, we further analyzed the partial order CDCL algorithm and its behaviour on instances with sparse dependencies between decision levels. We showed that the instantiations saved by the less destructive backtrack of PO-CDCL often allow to discover conflicts dramatically faster, which helps to prune the search space more efficiently. This behaviour explains the good solving performances observed on tested instances. Moreover, we noticed the significant

impact of the assertion level choice on the search and designed several heuristics for this choice. According to our observations, opposite strategies are relevant depending on whether the solved instance is or isn't satisfiable. A satisfying model of the problem can be found faster if the backtrack generally undoes large parts of the assignment, allowing quicker moves in the search space. Conversely, undoing a smaller average quantity of instantiations helps the solver to focus on the currently active part of the search space and leads faster to a proof of unsatisfiability. Finally, we showed that trying to locally control the amount of instantiations undone by each individual backtrack is not the most efficient method; heuristics that choose the assertion level according to the amount of level dependencies it introduces have a stronger influence on the average quantity of assignment deletions.

## References


1. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: IJCAI 2009. pp. 399–404
2. Bhalla, A., Lynce, I., de Sousa, J.T., Marques-Silva, J.: Heuristic-based backtracking relaxation for propositional satisfiability. Journal of Automated Reasoning 35(1–3), 3–24 (2005)
3. Davis, M., Logemann, G., Loveland, D.W.: A machine program for theorem-proving. Communications of the ACM 5(7), 394–397 (1962)
4. Durairaj, V., Kalla, P.: Exploiting hypergraph partitioning for efficient boolean satisfiability. In: HLDVT 2004. pp. 141–146
5. Ginsberg, M.L.: Dynamic backtracking. Journal of Artificial Intelligence Research 1, 25–46 (1993)
6. Huang, J., Darwiche, A.: A structure-based variable ordering heuristic for SAT. In: IJCAI-03. pp. 1167–1172
7. Jégou, P., Terrioux, C.: Hybrid backtracking bounded by tree-decomposition of constraint networks. Artificial Intelligence 146(1), 43–75 (2003)
8. Li, W., van Beek, P.: Guiding real-world SAT solving with dynamic hypergraph separator decomposition. In: ICTAI 2004. pp. 542–548
9. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Transactions on Computers 48(5), 506–521 (1999)
10. McAllester, D.A.: Partial order backtracking. Research note, MIT (1993)
11. Monnet, A., Villemaire, R.: CDCL with less destructive backtracking through partial ordering. In: PAAR 2012. pp. 124–138
12. Monnet, A., Villemaire, R.: Scalable formula decomposition for propositional satisfiability. In: $C^3S^2E$ 2010. pp. 43–52
13. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: DAC 2001. pp. 530–535
14. Nadel, A., Ryvchin, V.: Assignment stack shrinking. In: SAT 2010. pp. 375–381
15. Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: SAT 2007. pp. 294–299
16. Robertson, N., Seymour, P.D.: Graph minors. II. Algorithmic aspects of tree-width. Journal of Algorithms 7(3), 309–322 (1986)
17. Velev, M.N., Bryant, R.E.: Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. Journal of Symbolic Computation 35(2), 73–106 (2003)