Improved Conflict-Clause Minimization Leads to Improved Propositional Proof Traces

Allen Van Gelder

Univ. of California, Santa Cruz, CA 95064
http://www.cse.ucsc.edu/∼avg

Abstract. Recent empirical results show that recursive, or expensive, conflict-clause minimization is quite beneficial on industrial-style propositional satisfiability problems. The details of this procedure appear to be unpublished to date, but may be found in the open-source code of MiniSat 2.0, for example. Biere reports that proof traces are made more complicated when conflict-clause minimization is used because some clauses need to be resolved upon multiple times during the minimization procedure as found in MiniSat 2.0. Biere proposes a proof-trace format in which the set of clause numbers needed for a certain derivation is given, but their order is not specified. This paper presents a new procedure for conflict-clause minimization that is slightly more efficient and, more importantly, discovers a correct order so that each clause used for the derivation is resolved upon only once. This permits the proof trace to specify the order in which to use the clauses, greatly reducing the burden on software that processes the proof trace. The method is validated on the unsatisfiable formulas used for industrial benchmarks in the verified-unsatisfiable track of the SAT 2007 competition.

1   Introduction

Sinz and Biere [6] and later Biere [1] describe and discuss a system of proof traces and checking for Sat solvers based on conflict-driven clause learning, such as `zchaff`, `minisat`, `picosat`, and many others. In many respects, their proposal is simply the union of two earlier ground-breaking proposals: Goldberg and Novikov proposed to output the literals of each derived conflict clause [4], while Zhang and Malik proposed to output the sequence of clause numbers whose linear resolution would create each derived conflict clause [9]. The original motivation for outputting proofs was to provide certificates of correctness that could be checked offline. Biere argues that proofs have other uses in several applications [1]. In these contexts, proofs are viewed as explanations, the main goal is to extract useful information, rather than check correctness. Therefore, the format should be compact and easy to use. Biere argues that the proof trace should contain *both* the conflict clause and the clause numbers needed to derive it.

However, the Sinz and Biere proof-trace format differs in one important respect from other proposals. It produces the *unordered set* of clause numbers that

are sufficient to derive the conflict clause. One reason given is that when conflict-clause minimization is employed, the solver knows which clauses are *necessary* for the derivation, but some of the clauses might be used multiple times and figuring out a resolution order would entail extra work. Other inference methods that might appear in the future also might not be amenable to linear derivations.

This short paper shows that it is feasible to produce an ordered sequence, even when "recursive" conflict-clause minimization is employed. Although our procedure is specific for conflict graphs, the idea may well extend to other settings, where more clauses are available. The procedure given in this paper has been "dropped into" `MiniSat 2.0`, replacing the "recursive" conflict-clause minimization in the distribution (called "expensive" in the `MiniSat` code), and has sped up the program slightly. But this minor speed-up is really a by-product. The main motivation is that our procedure is able to discover a correct sequence for deriving a minimized conflict clause by a linear resolution in which no variable is resolved upon more than once, and consequently no clause is used more than once. Such proofs were dubbed *trivial resolutions* by Beame *et al..* who showed that these proofs are of minimum length among those using only clauses in the conflict graph [2].

After the original and new methods are described, experimental results are presented based on the industrial benchmarks used for the SAT–2007 verification track. See `http://www.cse.ucsc.edu/∼avg/ProofChecker/` for details.

2 Conflict Clauses and Conflict Graphs

Leading SAT solvers use a *conflict graph* data structure to infer *conflict clauses*. Readers unfamiliar with conflict graphs and their relationship to conflict clauses should consult citations in this paragraph. Figure 1 illustrates a conflict graph. Our notation varies from some others [10,2] to reflect the data structures used by `zchaff`, `MiniSat`, `picosat`, etc. Arrows indicate the *reference* direction in the data structures, and "$\perp$" is associated with a clause that became *empty* during unit-clause propagation, as in the original presentation [5].

An *antecedent clause* determines the edges leaving the vertex, and *vice versa*. In Figure 1, the antecedent clause of "$\perp$" is $\left[\overline{e}, \overline{f}, \overline{j}\right]$, and the antecedent clause of "e" is $\left[e, \overline{g}, \overline{k}\right]$.[1] For working through examples, we assume that literals of antecedent clauses are stored in alphabetical order.

In Figure 1, the 1-UIP cut has the associated 1-UIP conflict clause:

$$D_0 = \left[\overline{p}, \overline{j}, \overline{k}, \overline{i}, \overline{m}, \overline{r}, \overline{\ell}, \overline{q}\right] . \tag{1}$$

The literals of D_0 are listed in the order that `MiniSat` stores them. Notice that those within a decision level are not in any particular order among themselves. This clause and Figure 1 are used for running examples in this paper.

---

[1] It is easy to see that the set of antecedent clauses for a particular conflict graph is *renamable Horn*, so there is no loss of generality in assuming all vertices correspond to positive literals.
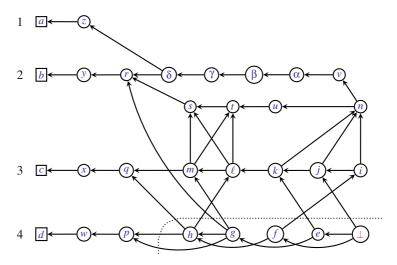
Fig. 1. Conflict graph with 1-UIP cut shown as dotted line. The corresponding conflict clause is $D_0 = [\overline{p}, \overline{j}, \overline{k}, \overline{i}, \overline{m}, \overline{r}, \overline{\ell}, \overline{q}]$. Decision levels and decision literals are on the left.

3 Conflict Clause Minimization in MiniSat 2.0

We now explain the clause minimization procedure in `MiniSat` (and other solvers) using conflict clause D_0 in (1) and Figure 1 as an example. After determining D_0, `MiniSat` 2.0 tests each literal L in D_0, except p, to see if it can be resolved away without (ultimately) adding any new literals to the resulting clause. Only antecedent clauses are considered for such resolutions.[2] The procedure does not actually perform any resolutions. Instead, for each candidate literal L in D_0, a depth-first search rooted at L checks whether all paths leaving L encounter some other vertex in D_0.[3] If this condition holds, L can be removed. The search is aborted as soon as it is determined that some path exists that terminates without meeting D_0. The final conflict clause is called D, and is a subset of D_0.

In the following, the notation "(in)" means the search backtracks from this vertex without exploring further, because this vertex is part of D_0. First, p is bypassed because it is never removable. Next, j is checked, generating the search sequence j, k (in), n, u, t, s, r (in), v, α, β, γ, δ, r (in), z (fail). It is unnecessary to trace the rest of the path from z to a because D_0 has no literals on this decision level, so this must be a failure path. Therefore, j must be kept in D. When the search aborts, all other information found during the search is discarded.

Next, k is checked, generating the search sequence k, ℓ (in), n, ... (the rest is the same as the j search). Then comes i, generating i, j (in), n, ... (the rest is the same as the j search). This repetition shows an inefficiency in the code, but it is not easy to overcome because the depth-first search is coded in

---

[2] In `MiniSat` code, the vector `reason[]` stores what we call the antecedent clause.

[3] Actually, the negation of the vertex is in the clause, but there is no confusion in the simpler wording.

the old-fashioned manner, without recursion, where the programmer manages the vertex stack. This style has no convenient access to *post-order time* for the vertices (also known as *finishing time*).

The minimization procedure continues with m. The sequence is m, q (in), s, r (in), t, s (removable). Thus m can be removed from D_0. Notice that upon the second encounter with s it was remembered that if s is temporarily added to the clause, it can be removed. The same is remembered about t at this point. As long as the overall search is successful, it remembers that all the vertices visited are removable. It is only when the search ultimately fails that the procedure does not know which vertices are removable and discards all new information.

Next, the search from r proceeds to y and b, and fails, so r must be kept in D. Next, ℓ is checked, generating the sequence ℓ, m (in), s (removable), t (removable). So ℓ is also removable. Finally, q is checked and must be kept in D.

In summary, the procedure found first that m can be removed, then that ℓ can be removed. Unfortunately, a resolution derivation that shortens D_0 by first removing m, then removing ℓ, is unnecessarily long and does not fit the pattern of the trivial resolution. Using the orders found in the searches would lead to the following sequence of resolvents (with abuse of notation): $D_1 = D_0 - m + s + t$, $D_2 = D_1 - s$, $D_3 = D_2 - t + s$, $D_4 = D_3 - s$. Finally, D_4 is D_0 with m removed and nothing added.

But now it gets worse. To remove ℓ, it is necessary to re-introduce m and remove it all over again: $D_5 = D_4 - \ell + m + s + t$, $D_6 = D_5 - m$, $D_7 = D_6 - s$, $D_8 = D_7 - t + s$, $D_9 = D_8 - s$. The final minimized clause is $D = D_9$. Possibly, this example can be extended to construct an exponential worst case [7].

This example explains why Sinz and Biere advocated that the trace should simply specify that D was *somehow* derived from the antecedents of $\bot$, e, f, g, h, m, s, t, and ℓ, without specifying a sequence.

4   New Minimization Procedure

Our new procedure for minimization uses the modern recursive version of depth-first search (DFS) that provides access to the post-order times of vertices (also called finishing time) [3]. The DFS can be visualized as someone moving around the graph and able to do tasks when they arrive at a vertex either for the first time or upon backtracking. Initially vertices are marked as *in* or *out* of D_0. Upon reaching a vertex L at post-order time, enough information has been gathered to categorize it as one of the following: *keep*: L remains in D; *removable*: L is not in D, but is in D_0 or some intermediate resolvent; *poison*: L must not enter any intermediate (or final) resolvent. A decision literal is *keep* if it is in D_0, otherwise *poison*.

The post-order rules for non-decision literals are straightforward: (1) If L is in D_0 and some successor is poison, *keep*; (2) if L is not in D_0 and some successor is poison, *poison*; (3) if all successors of L are keep or removable, L is *removable*.

The crucial idea is this: at the post-order time for L, if L is found to be *removable*, then it is pushed on a *stack*. (Some removables may be unneeded because they cannot be reached by a path of removables; they can be removed

easily by post-processing.) Correctness easily follows using the fact that the top-to-bottom order of the needed removables is a topological order, as required for a trivial resolution to reduce D_0 to D [7].

A DFS is rooted at each L in D_0, but now information is stored for both failing and successful subsearches, so nothing is discarded and repeated.

Let us trace this procedure on the same example. The notation "$\downarrow$" means the vertex is backtracked to. As before, p is bypassed. A DFS is rooted at j, generating the search sequence j, k, ℓ, m, q, x, c (poison), $\downarrow x$ (poison), $\downarrow q$ (keep), $\downarrow m$, s, r, y, b (poison), $\downarrow y$ (poison), $\downarrow r$ (keep), $\downarrow s$ (removable, push(s)), $\downarrow m$, t (removable, push(t)), $\downarrow m$ (removable, push(m)), $\downarrow \ell$ (removable, push(ℓ)), $\downarrow k$, n, u (removable, push(u)), $\downarrow n$, v, α, β, γ, δ, z (poison), $\downarrow \delta$, $\downarrow \gamma$, $\downarrow \beta$, $\downarrow \alpha$, $\downarrow v$, $\downarrow n$ (all six poison), $\downarrow k$ (keep), $\downarrow j$ (keep).

A second DFS is rooted at i, but both its successors have been visited, so i is immediately categorized as *keep*.

The top-to-bottom order of the *removable* stack is: u, ℓ, m, t, s. Since D_0 has neither u nor $\overline{u}$, resolution with the antecedent of u is not needed. The trivial resolution to reduce D_0 to D uses the antecedents of ℓ, m, t, s, in that order.

5   Experimental Results

We ran several configurations of `MiniSat` 2.0 on 17 industrial benchmarks from the verified-unsatisfiable track of the SAT 2007 competition. At the URL in Section 1, see `minisat-comparison.pdf` and `cert-poster-sat07.pdf` for additional data and benchmark information; space constraints prevent including them here. CPU times are based on Intel XEON 2.00GHz, 4 GB memory.

The modified `MiniSat` consists of "dropping in" the new recursive conflict-clause minimization procedure presented in this paper (see `MiniSat2ccmin.tar`). This change did not affect the computational results, not even the order of the literals within any clauses, as was verified by observing that all printed counter values (in the 100's per run, including lengths and numbers of conflict clauses at each restart point) were identical for both versions.

Table 1 shows that, on 16 benchmarks (the 17-th was solved by unit propagation), the modification saved time 9 times, and lost time 7 times. However, profiling the `analyze()` percent, the modification lowered this percent 9 times and raised it 2 times. Most changes were only 1 percent of the total time, but the larger changes all favored the modified version and were 2, 3, and 13 percent of the total time. This is the most specific data we could get, because the conflict-clause minimization code is in-line in the original `analyze()`, which also computes the conflict clause, the backtrack level, and other related data. As expected, since no changes were made except in `analyze()` and its subroutines, the larger differences in CPU time are almost entirely attributable to the differences in `analyze()` percent.

This data provides evidence that our improved procedure for "recursive" conflict-clause minimization achieves its primary purpose, which is to enable a proof trace to show a correct order of resolutions to achieve a trivial resolution derivation of the minimized conflict clause, in the sense of Beame *et al.* [2]. Moreover,

Table 1. `MiniSat` 2.0 original and modified CPU times and fractions

benchmark name	CPU orig	CPU mod	CPU mod–orig	percentage mod–orig	**analyze** pct orig	**analyze** pct mod
eq.atree.braun.7.unsat	3.39	3.41	0.02	0.59	28	27
eq.atree.braun.8.unsat	38.34	37.77	-0.57	-1.49	22	22
eq.atree.braun.9.unsat	174.71	180.87	6.16	3.46	16	16
AProVE07-21	1369.26	1362.23	-7.03	-0.50	11	11
AProVE07-02	4204.60	4227.81	23.21	0.55	21	20
AProVE07-22	397.68	395.76	-1.92	-0.47	17	16
AProVE07-20	753.64	764.27	10.63	1.40	13	14
AProVE07-15	609.09	611.34	2.25	0.37	21	21
IBM_FV_2004··30···k15	1394.55	1357.30	-37.25	-2.70	15	13
itox_vc965	0.25	0.25	0	0	0	0
dated-5-11-u	726.91	738.13	11.22	1.53	8	9
dated-5-15-u	5031.67	4999.84	-31.83	-0.62	20	18
total-5-11-u	84.71	83.81	-0.90	-1.06	14	13
total-5-13-u	206.64	204.36	-2.28	-1.10	13	12
dated-10-15-u	97.12	84.04	-13.08	-14.43	32	19
dspam_dump_vc973	18202.99	17754.12	-448.87	-2.50	38	35
manol-pipe-c10nidw_s	919.38	919.49	0.11	0.01	10	10
TOTAL(17)	34214.88	33724.8	-490.08	-1.44	26	24

this is achieved without increasing the computation time; indeed, modest decreases were achieved.

References

1. Biere, A.: Picosat Essentials. J. Satisfiability 4, 75–97 (2008)
2. Beame, P., Kautz, H., Sabharwal, A.: Towards Understanding and Harnessing the Potential of Clause Learning. J.A.I.R. 22, 319–351 (2004)
3. Baase, S., Van Gelder, A.: Computer Algorithms: Introduction to Design and Analysis, 3rd edn. Addison-Wesley, Reading (2000)
4. Goldberg, E., Novikov, Y.: Verification of Proofs of Unsatisfiability for CNF Formulas. In: Proc. Design, Automation and Test in Europe, pp. 886–891 (2003)
5. Marques-Silva, J.P., Sakallah, K.A.: GRASP–A Search Algorithm for Propositional Satisfiability. IEEE Transactions on Computers 48, 506–521 (1999)
6. Sinz, C., Biere, A.: Extended Resolution Proofs for Conjoining BDDs. In: Grigoriev, D., Harrison, J., Hirsch, E.A. (eds.) CSR 2006. LNCS, vol. 3967, pp. 600–611. Springer, Heidelberg (2006)
7. Van Gelder, A.: Verifying Propositional Unsatisfiability: Pitfalls to Avoid. In: Marques-Silva, J., Sakallah, K.A. (eds.) SAT 2007. LNCS, vol. 4501, pp. 328–333. Springer, Heidelberg (2007)
8. Zhang, L., Malik, S.: Extracting Small Unsatisfiable Cores from Unsatisfiable Boolean Formula. In: SAT 2003, Sta. Marguerita, It. (2003)
9. Zhang, L., Malik, S.: Validating SAT Solvers Using an Independent Resolution-Based Checker. In: Proc. Design, Automation and Test in Europe (2003)
10. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In: ICCAD (November 2001)