Verifying Propositional Unsatisfiability: Pitfalls to Avoid

Allen Van Gelder

University of California, Santa Cruz CA 95060, USA,
WWW home page: http://www.cse.ucsc.edu/~avg

Abstract. The importance of producing a certificate of unsatisfiability is increasingly recognized for high performance propositional satisfiability solvers. The leading solvers develop a conflict graph as the basis for deriving (or "learning") new clauses. Extracting a resolution derivation from the conflict graph is theoretically straightforward, but it turns out to have some surprising practical pitfalls (as well as the unsurprising problem that resolution proofs can be extremely long). These pitfalls are exposed, solutions are presented, and analyzed for worse cases. Dramatic improvements on industrial benchmarks are demonstrated.

1 Introduction

With the explosive growth of Sat Modulo Theories (SMT) in the last few years, the focus in propositional SAT solvers is shifting to unsatisfiable formulas, because these are the negated theorems to be proved in many applications. Producing proofs and independently checking them has received limited attention. Two ground-breaking efforts are Goldberg and Novikov [4], who built on BerkMin [3], and Zhang and Malik [9, 10], who built on Chaff [6]. It is important to get our propositional house in order to provide an adequate foundation for the more sophisticated challenge of producing independently checkable proofs for SMT.

The author has argued elsewhere [7] that solvers should be able to produce *easily verifiable* certificates to support claims of unsatisfiability. The gold standard proposed is that the language of certificates should be recognizable in *deterministic log space*, a very low complexity class. Intuitively, an algorithm to recognize a log space language may re-read the input as often as desired, but can only write into working storage consisting of a fixed number of registers, each able to store $O(\log L)$ bits, for inputs of length L.

The rationale for such a stringent requirement is that the buck has to stop somewhere. How are we to trust a "verifier" that is far too complex to be subjected to an automated verification system? And how are we to trust that automated verification system? Eventually, there has to be a verifier that is so elementary that we are satisfied with human inspection.

An explicit resolution proof is one in which each derived clause is stated explicitly, along with the two earlier clauses that were resolved to get the current clause. It is not hard to see that an explicit resolution proof can be recognized in deterministic log space.

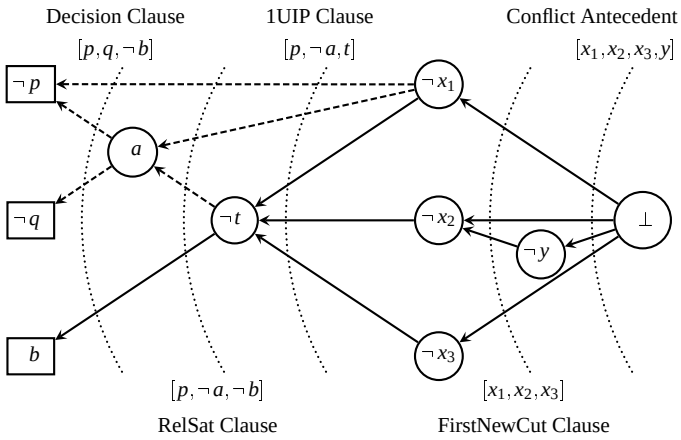

Fig. 1. Conflict graph with several cuts shown.

A detailed specification for an explicit resolution derivation (`%RES`) was used for the "certified unsat" track of the SAT 2005 Competition. Although the results of that track were disappointing, this paper shows how fixing a performance problem in one solver produced orders-of-magnitude improvement, suggesting that the methodology is feasible, after all. Details on current capabilities are available at: `http://www.cse.ucsc.edu/~avg/ProofChecker/`.

2 A Pitfall in Resolution Extraction

Most, if not all, leading SAT solvers use a *conflict graph* data structure to infer *conflict clauses*. Figure 1 illustrates a conflict graph. The notation varies from other papers [11, 2] to better reflect the actual data structures used by the programs. Each graph vertex is associated with a different literal, no complementary literals appear, and the conflict vertex is associated with the constant *false*, denoted by "⊥." The vertex for each implied literal, including *false*, is labeled with an "input" clause, called the *antecedent clause*. This notation agrees more closely with the original presentation . Assumed (guessed) literals, commonly called "decision literals," do not have an antecedent clause.

Recent papers have observed the connection between conflict graphs and resolution [4, 9, 2, 8]. Of course, given a cut, the antecedent clauses on the conflict side logically imply the conflict clause, which consists of the negations of the reason-side literals adjacent to some vertex on the conflict side (i.e., one or more edges cross the cut to such literals).

The question is how to exploit the structure of the conflict graph to obtain a resolution derivation of the conflict clause. This question is not as simple as it might appear, in view of the fact that the algorithm published by Zhang and Malik, and actually implemented in `zverify_df` (in the `zchaff` distributions), has an exponential worst case. Their algorithm is based on Figure 3 *of their*

```
1.    cl = final_conflicting_clause;
2.    while (!is_empty_clause(cl)) {
3.        lit = choose_literal(cl);
4.        var = variable_of_literal(lit);
5.        ante_cl = antecedent(var);
6.        cl = resolve(cl, ante_cl); }
```

Fig. 2. Pseudocode to generate resolution proof from conflict graph.

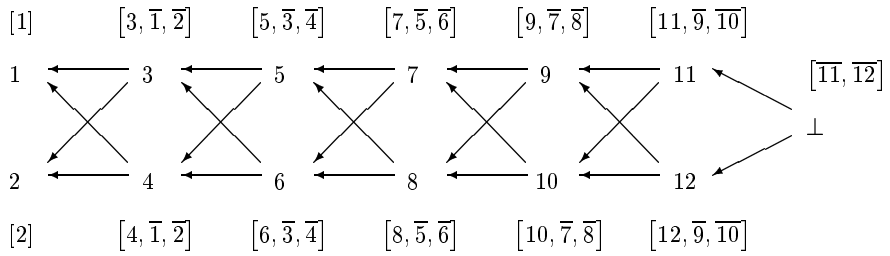

Fig. 3. DAG family ($h = 6$ instance) with exponential worst case for `zverify_df` as published and distributed (through 2006). Antecedent clauses are shown in brackets.

paper [9], the crucial part of which appears in *our* Figure 2. It is important to note on line 1 that `final_conflicting_clause` is *not* a conflict clause. Rather, it denotes the antecedent of $\bot$, the "input" clause that became empty during unit propagation (see "Conflict Antecedent" in Figure 1). The *conflict clause* is the final value of `cl`, and is empty if the solver was correct. An invariant is that `cl` contains the *negations of* some literals in the conflict graph.

Line 3 is implemented two ways in different versions. In one version, the literal chosen is one with minimum *DAG height*, which is the maximum path length to a vertex whose antecedent is a unit clause. In another version, the literal chosen is simply the one with the lowest variable number, essentially an arbitrary choice. The conflict graph family that generates exponential behavior in the size of the conflict graph is the same for both versions. The parameter of this family is h, the DAG height of the *false* vertex. A member with $h = 6$ is shown in Figure 3. The resolution developed by Figure 2 begins as follows, where "(11)" denotes resolution with clashing literal 11:

$$[\overline{11}, \overline{12}]\ (11)\ [11, \overline{9}, \overline{10}]\ (9)\ [9, \overline{7}, \overline{8}]\ (7)\ [7, \overline{5}, \overline{6}]\ (5)\ [5, \overline{3}, \overline{4}]\ (3)\ [3, \overline{1}, \overline{2}]$$
$$(1)\ [1]\ (2)\ [2]\ (4)\ [4, \overline{1}, \overline{2}]\ (1)\ [1]\ (2)\ [2]\ (6)\ [6, \overline{3}, \overline{4}]\ (3)\ [3, \overline{1}, \overline{2}]\ \ldots$$

Literals 1 and 2 will be resolved upon 2^{h-1} times each; literals 3 and 4 will be resolved upon 2^{h-2} times each, etc.; the total number of resolutions is $2\,(2^h - 1)$.

This is a case where theory translates into practice, at least in the case of the arbitrary choice of literal in the version dated 2004.11.15. Table 1 shows

Table 1. Resolution proof length inefficiencies. Sizes are mega-literals. "+?" indicates job was killed when size exceeded stated number.

Small GN03		
benchmark	2004.11.15	with fix
5pipe	153	15
5pipe_1_ooo	986	92
5pipe_5_ooo	2320	94
6pipe	169	97
6pipe_6_ooo	3072+?	486
7pipe	358	319
9vliw_bp_mc	308	58
barrel7	2754	5
barrel8	3072+?	73
barrel9	3072+?	80
c3540	393	78
c5315	162	9
c7552	2650	22

IBM_FV SAT 2005		
benchmark	2004.11.15	with fix
01_SAT_dat.k10	3	0.134
07_SAT_dat.k30	3	2.582
07_SAT_dat.k35	3	2.859
18_SAT_dat.k10	174	0.511
18_SAT_dat.k15	102520+?	14.410
1_11_SAT_dat.k10	13	0.338
1_11_SAT_dat.k15	128	2.077
20_SAT_dat.k10	13	0.208
23_SAT_dat.k10	3	0.062
23_SAT_dat.k15	15259+?	3.737
26_SAT_dat.k10	0.036	0.036
2_14_SAT_dat.k10	70	0.857
2_14_SAT_dat.k15	3220	7.188

the sizes of resolution proofs on some smaller industrial benchmarks, for the original program and a fixed version that uses a better order, which will be described shortly. (Although `zverify_df` does not output the resolution proof, it materializes all the clauses.) The GN03 benchmarks are the smaller ones reported previously [4, 9]. Those labeled "`IBM_FV`" are from the "industrial" category of the SAT 2005 Competition (see `http://www.satcompetition.org/2005` and the URL given in the introduction).

The purpose of showing this data is to show that extracting a resolution proof from the conflict graph has the potential for very bad performance, but it is not intended to criticize the Princeton `zchaff` team in any way, who have been very cooperative. Their paper *did* go through peer review without the problem being noticed.

3 Avoiding the Pitfall via "Trivial Resolution" (TVR)

We now introduce some terminology and notation to study efficient methods of extraction, with worst-case guarantees. We shall use the notation that n is the number of vertices in the conflict graph, m is the number of edges, d is the maximum out-degree of any vertex in the conflict graph, and w is the number of literals in the conflict clause to be derived. Note that the sum of the lengths of the antecedent clauses is m. Thus $(m + w)$ is the combined length of the input and output.

A *linear resolution derivation* is one in which the first clause is an "input" clause, called the *top clause*, and each resolution operation has the previous clause in the derivation as one operand; the second operand may be an "input" clause or an earlier-derived clause of the linear derivation. (We shorten "resolution derivation" to "derivation" when there is no ambiguity.) An *input derivation* is a linear derivation in which the second operand must be an "input" clause.

For our purposes, an "input" clause is one that existed while the conflict graph was being constructed; this includes antecedent clauses of the conflict graph. Note that Figure 2 provides a framework for input derivations using antecedent clauses.

Beame *et al.* [2] define a *trivial resolution derivation* (**TVR**) to be an input derivation with the further restriction that no clashing variable occurs in more than one resolution operation. They show (their Proposition 4) that the conflict clause can be derived by a TVR using the antecedent clauses of the conflict-side vertices of the conflict graph, and using the antecedent of the *false* vertex as the top clause. The successful TVR has exactly n resolution operations, but using a correct order is crucial.

To get a worst-case bound on total derivation size of TVR, measured in number of literals, we note that it is possible for the current resolvent to grow by up to $(d-1)$ literals per resolution for the first n/d steps to a size of $(w + 1 + n\,(d-1)/d)$, even if the final conflict clause is fairly narrow. Thereafter, it can shrink only one literal per resolution, so the sum of the sizes of all resolvents in the derivation is bounded by $(w + n\,(d-1)/2\,d)\,n$, which can be quadratic in the size of the conflict graph.

The trouble with TVR scheme is that the correct order is not readily accessible from the data structure of the conflict graph. To obtain a valid TVR order, the rule for `choose_literal` on line 3 of Figure 2 should be:

Cut-Crossing Rule: Choose a literal all of whose incoming edges originate from a vertex whose literal has already been resolved upon, or from the *false* vertex (reworded from Beame *et al., op cit.*).

Some other data structure is needed to provide or compute an appropriate order.

The saving grace is that the input for `zverify_df` is set up by the companion solver, such as `zchaff`, which already *has* such a data structure. The solver creates a sequence of "implied" literals in the chronological order in which they entered the conflict graph. When this sequence is available, `zchaff` uses the *reverse* of this order to generate the trace of an input derivation for `zverify_df` to verify [9]. It is not difficult to see that reverse chronological order satisfies the cut-crossing rule. (A theoretical nitpick is that the sequence might include numerous implied literals that are not in the conflict graph, but have to be looked at anyway, so the *time* is not strictly bounded by the size parameters of the conflict graph.)

As it happens, `zchaff` communicates the successful order to `zverify_df` in its encoding of the "resolve-trace" for all conflict clauses with a *positive* "decision level." Unfortunately, `zchaff` (through 2006) treats decision level zero differently, does not actually create an empty conflict clause, and so the crucial order for decision level zero is *not* in the encoding of the resolve-trace. Instead the final conflict graph itself is encoded in the output. Thus `zverify_df` had an opportunity to go astray.

The "fix" applied by this author to deliver the rightmost column of Table 1 (files available from the author; meanwhile, the "diffs" are available at the URL mentioned in the introduction) involved modifying `zchaff` to follow through at

decision level zero, mainly using the procedures and data structures already in the program. The fixed program creates an empty conflict clause and encodes its resolution order in the resolve-trace, using the same protocol as for the nonempty conflict clauses at positive decision levels. Then `zverify_df` was modified slightly to expect this additional line.

4 Conclusion

A longer paper at the URL in the introduction has additional details, experimental results, and discusses another extraction strategy named Pseudo-Unit Propagation (PUP). Both the TVR and PUP methods guarantee that total derivation size polynomial in the size of the conflict graph, but both have nonlinear worst cases, which are incomparable. Experimental data on industrial benchmarks (not presented in detail) shows that PUP derivations are 60% longer than TVR on average and are longer on about 74% of the benchmarks tested.

The significance of this data and take-home message is: The PUP strategy is a second, milder, pitfall to avoid. A program that generates resolutions "on-line" during unit clause propagation will do essentially the same resolutions as a PUP after-the-fact system. Both theoretical and empirical analyses suggest that this produces more verbose resolution derivations (aside from the on-line resolutions that turn out to be unneeded), compared to the after-the-fact TVR method.

References

1. Baase, S., Van Gelder, A.: Computer Algorithms: Introduction to Design and Analysis. 3rd edn. Addison-Wesley (2000)
2. Beame, P., Kautz, H., Sabharwal, A.: Towards understanding and harnessing the potential of clause learning. J. Artificial Intelligence Research **22** (2004) 319–351
3. Goldberg, E., Novikov, Y.: Berkmin: a fast and robust sat-solver. In: Proc. Design, Automation and Test in Europe. (2002) 142–149
4. Goldberg, E., Novikov, Y.: Verification of proofs of unsatisfiability for CNF formulas. In: Proc. Design, Automation and Test in Europe. (2003) 886–891
5. Marques-Silva, J.P., Sakallah, K.A.: GRASP–a search algorithm for propositional satisfiability. IEEE Transactions on Computers **48** (1999) 506–521
6. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: 39th Design Automation Conference. (2001)
7. Van Gelder, A.: Decision procedures should be able to produce (easily) checkable proofs. In: CP02 Workshop on Constraints in Formal Verification, Ithaca. (2002)
8. Van Gelder, A.: Pool resolution and its relation to regular resolution and DPLL with clause learning. In: Proc. LPAR, LNAI 3835, Montego Bay. (2005) 580–594
9. Zhang, L., Malik, S.: Validating sat solvers using an independent resolution-based checker: Practical implementations and other applications. In: Proc. Design, Automation and Test in Europe. (2003)
10. Zhang, L., Malik, S.: Extracting small unsatisfiable cores from unsatisfiable boolean formula. In: Proc. Theory and Applications of Satisfiability Testing. (2003)
11. Zhang, L., Madigan, C., Moskewicz, M., Malik, S.: Efficient conflict driven learning in a boolean satisfiability solver. In: ICCAD. (2001)