Journal on Satisfiability, Boolean Modeling and Computation 8 (2012) 117-122

Contrasat – A Contrarian SAT Solver

SYSTEM DESCRIPTION

Allen Van Gelder `http://www.cse.ucsc.edu/~avg/`
Computer Science Dept., University of California
Santa Cruz, CA 95064
U.S.A.

Abstract

The SAT solver `Contrasat` is a small variation of the well-established `Minisat` solver. It was entered in the *Minisat hack track* of the 2011 SAT competition, and was judged to be first place in one category. This paper describes the code change and its motivation. The characterization as "contrarian" is explained. Experimental results are summarized.

Keywords: *SAT solver, CDCL, Minisat, volunteer, Contrasat*

Submitted September 2011; revised February 2012; published March 2012

1. Introduction

The satisfiability (SAT) problem has been much studied from the empirical point of view for practical solving, although the problem is *NP*-complete. The *Minisat hack track* has been included the 2009 and 2011 SAT competitions to explore small changes in the well-established `Minisat` solver [2]. For 2011, the base version was 2.2.0 (which label can only be found in `doc/ReleaseNotes-2.2.0.txt` and is absent from the source code, itself). `Contrasat` employs a small code change to `Minisat`, and was entered in the 2011 Minisat hack track. This system description applies to `Contrasat 2.2.0.B`. Several URLs have information relevant to this report:

A	`http://www.cse.ucsc.edu/~avg/Contrasat/`	Contrasat source code and other information;
B	`http://satcompetition.org/2011`	comprehensive information on the 2011 SAT Competition, including source codes of submitted solvers;
C	`http://www.cril.univ-artois.fr/SAT11/`	results from the 2011 SAT competition.

Since the initial report on `Chaff` [4], it has been an accepted dogma for high-performance SAT solving (at least within the CDCL family, which includes `Chaff` and `Minisat`) that so-called "Boolean constraint propagation" (BCP) should be made as fast as possible, since it occupies about 80% of the processing time. ("Boolean constraint propagation" is usually called "unit-clause propagation" in earlier literature.) We call `Contrasat` a "contrarian" solver because it challenges this dogma. The only change to the reference `Minisat` code *slows down* the BCP processing by a significant factor.

After describing the code change, we explain the motivation, then summarize some results from the 2011 SAT competition, and present some additional experiments.

2. Modified Unit-Clause Propagation Procedure

The standard procedure for BCP maintains `uncheckedLits`, set of literals that should be assigned **true** and are waiting to be propagated. BCP is "seeded" by adding an assumed literal (the *decision literal*) to the initially empty `uncheckedLits`. Repeatedly, while the set is not empty and no clause has been falsified, BCP removes a literal p, assigns it to **true**, and checks certain clauses that contain $\overline{p}$ to see if they are now unit clauses, in the sense that the clause has at most one literal that is not falsified by existing assignments, now including p. Whenever such a clause C is found, its unassigned literal, say q, is said to be the *implied literal* of C, and is added to `uncheckedLits`. We call C the *antecedent* of q. Actually, most solvers record that q is true when it is *added* to `uncheckedLits`, rather than when it is removed. This timing difference is further discussed later.

This section describes how `Contrasat` modifies the standard procedure. To follow the details, the reader should be familiar with the general workings of CDCL solvers such as `Chaff` [4] and `Minisat` [2].

The most common data structure for implementing `uncheckedLits` is a first-in, first-out (FIFO) queue. Adding and deleting require only constant time per operation, using an array. As mentioned, the literal is actually assigned when it *enters* `uncheckedLits`, rather than when it is removed, but the order of assigned literals is the same if `uncheckedLits` is a FIFO queue.

Ever since the `Chaff` developers observed that 80% percent of the solver's time is typically spent in BCP [4], it has been an article of faith that BCP should be as fast as possible. Much low-level implementation effort has been directed toward this goal. Most of this work is unpublished and found only in the publicly available source codes.

`Contrasat` adopts the contrarian view that the choice of which antecedent clause to remove from `uncheckedLits` is important, and uses a *minimizing* priority queue to select the heuristically best antecedent. The priority queue is implemented as a *binary heap*, which uses an array. Operations for adding and removing elements take $O(\log k)$ time when the heap contains k elements, so significant efficiency might be sacrificed in managing `uncheckedLits`.[1.]

Let $C = [p_0, \overline{p_1}, \overline{p_j}, \ldots]$ be the antecedent clause ($\overline{p_2}$ is absent in binary clauses). The `Minisat` implementation is such that p_0 is the implied literal, not yet propagated while this element is in `uncheckedLits`, and p_1 is the literal whose propagation caused C to become a unit clause. (These are known as the *watched literals* of C.) When C is a ternary clause or wider and has *some* previously watched literal, the new `Contrasat` code ensures that $\overline{p_2}$ is the *most recently assigned* previously watched literal. If C is now in `uncheckedLits` or is already an antecedent, all of the literals p_1, p_2, ..., are assigned **true**.

Priority is a two-integer tuple, `(level, trail_pos)` with lexicographic order. The major integer for comparison of priority-queue elements is `level`, which stores the maximum *decision level* found among the non-watched literals, $\overline{p_2}$, ..., $\overline{p_{41}}$, or 0 for binary clauses.[2.] The minor integer, `trail_pos`, stores the order in which p_1 was assigned **true**.[3.]

1. `Minisat`, written in `C++`, already has a `Heap` template, making it easy to define a new `Heap` class based on user-supplied *data type* and *priority function*.
2. The SAT 2011 Competition version only checked $\overline{p_2}$, but is otherwise identical to 2.2.0.B. The cut-off of 40 is heuristic and defined at compile time.
3. Since p_1 is always assigned at the current decision level, either global or local trail position can be used.

Use of the priority queue can change both the assignment order and antecedents, which can greatly influence what clause is learned when a conflict eventually occurs. The rationale for our design is discussed in the next section.

3. Motivation

Suppose that a conflict is discovered after making an assumption, say $x = \mathbf{true}$, during the following unit-clause propagation. CDCL solvers derive (learn) a new clause, say D. Depending on D, varying amounts of backtracking will be "justified." As introduced in `Grasp` [3], backtracking is "justified" for as many decision levels as D continues to have only one unassigned literal. If at least one level of backtracking is justified by this criterion, then D is called an *asserting clause.*

Audemard *et al.* observed that on certain occasions, clauses other than the recorded antecedents could be used to derive a different clause that would justify further backtracking [1]. They found that certain additional clauses, called *inverse arcs*, could be recorded during unit-clause propagation that might later enable an heuristically "better" clause to be learned.

Van Gelder introduced the term *volunteer* to describe any clause C during unit-clause propagation such that all literals had been assigned values, at most one literal had been assigned **true**, but C was *not* the recorded antecedent for its true literal [6]. A volunteer is a possible alternative antecedent.

Inverse arcs are a subset of volunteers. For any antecedent C, the implied literal q is always later in the trail than all other literals in C. For an inverse arc C, the implied literal q is not only earlier in the trail than some other literal $\overline{p}$ in C, it is also at an earlier *decision level* than p. `Contrasat` has the same general goal of enabling further backtracking, but does not detect inverse arcs.

The solver `Precosat`, authored by Armin Biere, won at least one category of the 2009 SAT competition, and placed highly in several categories. The code is open source and contains a procedure that tries to use volunteers. A difficulty with implementing inverse arcs is the possibility of a cycle in the sequence of resolutions. Examples suggest that detection of such cycles is not simple [6]. Therefore, the `Precosat` procedure imposes a restriction on candidate volunteers that ensures that a cycle cannot be produced. As far as we know, these details are unpublished except that the code is open source.

Recall that, for this discussion, C is a volunteer for literal q. That is, at the time q was assigned, C most likely had some other unassigned literals besides q, so some other clause, say A, became the antecedent of q. (It is also possible that A and C implied q at essentially the same time, but due to an accident of data structures, A was noticed before C.) Suppose we now have a conflict clause D, which has already been through *recursive conflict-clause minimization* [5, 7], and we are hoping to remove $\overline{q}$ from it. If the antecedent A could make progress, this would have been detected during recursive conflict-clause minimization. `Precosat` considers C to be a candidate volunteer if all literals in C other than q were implied or assumed to be **false** before q was *propagated.* This ensures that q did not contribute to implying the negation of any literal in C, as observed first by Biere. This in turn ensures that resolving with C cannot cause a cycle in the sequence of resolutions. Again we omit further details, since this is still only motivation.

We can observe the following: If C passes the `Precosat` test for being a candidate volunteer, then q, and any other unassigned literals of C, were in the set `uncheckedLits` simultaneously at some point. Stated another way, *some* order of processing the the unit clauses that were waiting to propagate their implied literals could have resulted in C becoming the recorded antecedent of q. The needed implied literals $p_1, \ldots, p_j$ were either already propagated or ready to be propagated simultaneously with q.

This leads to the idea behind `Contrasat`. By implementing `uncheckedLits` as a priority queue, the door is opened to selecting "better quality" antecedents to go with the implied literals, compared to oblivious FIFO order. The door is also opened to selecting literals to be implied in an order that results in an earlier conflict or a "better quality" falsified clause. If good enough heuristics can be found for the *priority* function, improved search will more than pay back for the increased overhead of managing the priority queue.

The following formula fragment and search fragment with variables s–z illustrate the influence of the priority queue. See URL **A** in Section 1 for more details.

$$\begin{array}{llllllll} C_1 = [\overline{s}, \overline{v}, \overline{x}, \overline{y}] & C_2 = [y, \overline{u}, \overline{x}] & C_3 = [z, \overline{v}, \overline{x}] & C_4 = [y, \overline{v}, \overline{z}] \\ C_5 = [u, \overline{v}, \overline{w}] & C_6 = [x, \overline{t}, \overline{u}] & C_7 = [s, \overline{z}] & C_8 = [w, \overline{v}, \overline{y}] \end{array}$$

Assume the search begins: *Level 1*: assume v, end. *Level 2*: assume w, imply(u, C_5), end ("(u, C_5)" means C_5 is the antecedent for u). We next examine level 3 in more detail.

With the traditional FIFO for `uncheckedLits`, the continuation may be: *Level 3*: assume t, enq(x, C_6), deq(x, C_6), imply(x, C_6), enq(y, C_2), enq(z, C_3), deq(y, C_2), imply(y, C_2), enq($\overline{s}$, C_1), deq(z, C_3), enq(s, C_7), deq($\overline{s}$, C_1), deq(**false**, C_7). The first UIP clause derived is $[\overline{x}, \overline{u}, \overline{v}]$. C_8 is an inverse arc, but `precosat` declines to try it, because trail positions of y and w show a loop risk. In this case resolving $[\overline{x}, \overline{u}, \overline{v}]$ with C_8 *would* lead to a loop after further resolution with C_2. Backtrack to level 2 is the final outcome.

Consider instead a priority queue (priority shown after antecedent, when relevant). *Level 3*: assume t, enq(x, C_6), deq(x, C_6), imply(x, C_6), enq(y, C_2, 2, 2), enq(z, C_3, 1, 2), deq(z, C_3), imply(z, C_3), enq(y, C_4, 1, 3), enq(s, C_7, 0, 3), deq(s, C_7), imply(s, C_7), enq($\overline{y}$, C_1, 3, 4), deq(y, C_4), imply(y, C_4), deq(y, C_2) (already implied), deq(**false**, C_1). The first UIP clause derived is $[\overline{x}, \overline{v}]$. Backtrack to level 1 is the outcome. Different examples can allow FIFO to backtrack more than the priority queue.

The actual *priority* function implemented (as of this system description) is quite crude, but seems to have achieved some success. A small amount of work is expended to try to place a recently assigned literal in the third position (subscript 2) of every clause, rather than have that literal be arbitrary. The maximum decision level of the non-watched literals in subscripts 2 through 41 (limited by clause length) becomes the major priority field at the time the clause becomes a unit clause. If one of these literals has a large decision level, this clause is less attractive for far backtracking than another clause whose inspected literals have a smaller decision level.

Note that if the clauses are both in the priority queue, the second literals must have been propagated at the current decision level. If both major priority fields are at the same decision level, then the tie-break on priority is: which second literal was propagated earliest? The intuition behind this choice is that long implication chains are probably unfavorable.

In summary, `Contrasat` tries to achieve the benefits of using volunteers after a conflict without ever really processing volunteers, as such. Instead it tries to choose them as an-

Table 1. Excerpt of results from 2011 SAT Competition, Applications, Satisfiable, with 116 instances. Three "top" solvers and two well known reference solvers are shown.

Solver	number SAT solved	adjusted average cpu seconds	50-th best cpu seconds	number UNSAT solved
`Contrasat-A`	99	418	52	100
`CIR_Minisat`	99	516	47	103
`MPhaseSAT64`	99	579	104	102
...				
`Precosat` (2009)	96	669	117	110
`Minisat` (2.2.0)	95	560	53	99

adjusted average based on 99 best times, 5000 for time-outs.

tecedents to begin with, before the conflict. This approach is supported by theory: Van Gelder has shown that any clause that is a logical consequence of the antecedents and volunteers combined has a resolution derivation in which each literal is resolved upon at most once [6]. Both an antecedent and a volunteer for the same literal are not really necessary.

4. Some Experimental Data

`Contrasat` participated in the *Application* category of the 2011 SAT Competition. Phase 2 (the final) had 300 benchmarks and a time-out of 5000 CPU seconds for single-thread solvers. In the "Satisfiable" subcategory, four solvers (all single-thread as it happens) solved 99 instances to share the lead. Three of these were `Minisat` hacks. `Contrasat` was awarded first place in this subcategory, based on least total CPU time. Notably, the reference solver, `Minisat` 2.2.0, solved only 95 satisfiable instances, while five hacks (not all shown in tables) solved between 95 and 99, so apparently the hacks were accomplishing something. One might wonder about the statistical significance of this outcome, but that question goes beyond the scope of this system description. An excerpt from the 2011 SAT Competition results is shown in Table 1.

Recall that `Contrasat` checks the decision levels of non-watched literals to determine priority, with a cut-off for long clauses. We evaluated cut-offs of 1 (the competition version), 5, 10, 20, and 40. The data were not available before the competition, or we would have entered a different cut-off.

The experiments were done on a 48-core processor with 2.0 GHz clock and 188 GB memory, shared with other users. This platform was found to be 2.25 times slower than that used for the 2011 SAT Competition, so their 5000-second time limit translates to about 3 hours on our platform.

We tested on 106 "Application" instances in the SAT11 directory, which were new for SAT 2011 (Section 1, URL **C**), and were solved by *some* `Minisat` hack or the reference `Minisat` during the competition (a sort of "virtual best hack"); 55 were unsatisfiable, 51 were satisfiable. Unsatisfiable and satisfiable instances are summarized separately in Table 2. It is evident that unsatisfiable and satisfiable instances behave quite differently.

On unsatisfiable instances, `Contrasat` with cut-offs of 20 and 40 produced fewer conflicts and shorter conflict clauses, and it ran faster, compared to the reference `Minisat`. Propagations took *less* time, on average, although the propagation procedure does extra

Table 2. `Contrasat` with various cut-offs for checking decision levels of literals. `Minisat` is shown as reference. Time-out was 3 CPU hours.

	UNSAT, average over 55 instances						SAT, average over 51 instances					
	`Contrasat` cut-off					`Mini-`	`Contrasat` cut-off					`Mini-`
Statistic	1	5	10	20	40	`sat`	1	5	10	20	40	`sat`
number solved	49	50	48	50	50	50	44	44	43	44	43	45
cpu seconds	1339	1253	1287	1055	1150	1234	2006	2068	2349	1973	1948	1764
mega-props./sec.	0.952	1.019	1.020	1.059	1.034	0.983	0.701	0.788	0.727	0.717	0.656	0.664
mega-conflicts	246	198	194	186	199	239						
conflict cl. length	57	55	56	56	54	60						

work. Conflict data for satisfiable instances is omitted because no refutation exists for these instances and the numbers fluctuate widely. See URL **A** (Section 1) for additional data.

5. Conclusion

We described `Contrasat`, a `Minisat` hack that does the "unthinkable": it spends extra time in unit-clause propagation. With cut-offs of 20 and 40 it outperforms the reference `Minisat` on unsatisfiable instances, which was the design goal. However, this conclusion is tentative without extensive testing on a wide range of benchmarks, which is beyond the scope of this system description. Although it won the satisfiable application subcategory in the 2011 SAT Competition, this victory is something of an accident because it won on speed, not by solving more instances. `Minisat` beat `Contrasat` on a different suite of 51 satisfiable application benchmarks.

References

[1] G. Audemard, L. Bordeaux, Y. Hamadi, S. Jabbour, and L. Saïs. A generalized framework for conflict analysis. In *Proc. SAT 2008, LNCS 4996*, Cambodia, Springer, 2008.

[2] N. Eén and N. Sörensson. An Extensible SAT-solver. In Giunchiglia, E. and Tacchella, A., editors, *SAT 2003, Selected Revised Papers, LNCS 2919*, pages 502–518, Sta. Margherita Ligure, Italy, Springer, 2004.

[3] J. P. Marques-Silva and K. A. Sakallah. GRASP–a search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, **48**:506–521, 1999.

[4] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an Efficient SAT Solver. In *Proc. 38th DAC*, pages 530–535, 2001.

[5] N. Sörensson and A. Biere. Minimizing learned clauses. In *Proc. SAT, LNCS 5584*, pages 237–243, Swansea, Wales, Springer, 2009.

[6] A. Van Gelder. Generalized conflict-clause strengthening for satisfiability solvers. In *Proc. SAT, LNCS 6695*, pages 329–342, Ann Arbor, MI, USA, Springer, 2011.

[7] A. Van Gelder. Improved conflict-clause minimization leads to improved propositional proof traces. In *Proc. SAT, LNCS 5584*, pages 141–146, Swansea, Wales, Springer, 2009.