MiniSat v1.13 – A SAT Solver with Conflict-Clause Minimization

Niklas Sörensson, Niklas Een

Chalmers University of Technology, Sweden
{nik,een}@cs.chalmers.se

Abstract. In this poster we summarize the features of the MiniSat version entering the SAT Competition 2005. The main new feature is a resolution based conflict clause minimization technique based on self-subsuming resolution. Experiments show that on industrial examples, it is not unusual for more than 30% of the literals in a conflict clause to be redundant. Removing these literals reduces memory consumption and produce stronger clauses which may propagate under fewer decisions in the DPLL search procedure.

We also want to raise attention to the particular version of VSIDS implemented in MiniSat, which we believe is a consistent improvement over the original VSIDS decision heuristic of the same magnitude as many of the recently proposed alternatives [GY02,Ry03].

Introduction

MiniSat is a minimalistic implementation of a Chaff-like SAT solver based on the two-literal watch scheme for fast BCP [MZ01] and clause learning by conflict analysis [MS99]. It entered the SAT 2005 competition, both as a stand-alone solver, and as a back-end to the preprocessor SatELite [EB05]. A number of small improvements has been made in MiniSat with respect to the original Chaff. Two important features are the incremental SAT interface, and the support for user defined boolean constrains [ES03]. Although none of these improvements are relevant for the SAT competition, where only non-incremental SAT problems in CNF are used, they are important when using MiniSat as an integrated part of a bigger system. The pure SAT-speed improvements, relevant to the SAT competition, are listed in the below section.

MiniSat v1.13

1. ***Variable order.*** The decision heuristic of MiniSat is an improved VSIDS order, where variable activities are decayed 5% after *each* conflict. The original VSIDS decays variables 50% after each 1000 conflicts or so. Benchmarks have shown that the improved scheme responds more quickly to changes in the productive set of branch-variables than the original VSIDS, and avoids branching on out-dated variables. Important to the heuristic is increasing the activity of all variables occuring in *some* clause used in the conflict analysis, not just variables of the final conflict clause (as in the first version of MiniSat). To keep the variables sorted on activity at all times, a heap is used in the current version of the solver.

2. ***Binary clauses.*** Binary clauses are implemented by storing the literal to be propagated directly in the watcher list. In our experiments, though not conclusive, this scheme outperformed a version storing all binary clauses in a separate set of vectors on the side. In such a scheme, it is natural to propagate all binary clauses either before or after the bigger clauses gets propagated by the BCP. This affects the implication graph and seems to produce worse conflict clauses.

3. ***Clause deletion.*** MiniSat aggressively deletes learned clauses based on an activity heuristic similar to the one for variables. The limit on how many learned clauses are allowed is increased after each restart. Keeping the number of clauses low seems to be particular important for some small but hard problems. The actual activity heuristic currently used is admittedly a weak point of MiniSat.

4. ***Conflict clause minimization.*** This is the main improvement (due to Niklas Sörensson) in version 1.13 over version 1.12, and the topic of the next section.

Conflict clause minimization

Definition: Let C and C' be clauses, $\otimes_x$ the resolution operator on variable x. If $C \otimes_x C' \subseteq C$ then C is said to be ***self-subsumed* by** C' **w.r.t.** x.

In effect, C' is used to remove x (or $\overline{x}$) from C by the fact that C is subsumed by $C \otimes_x C'$. A particularly useful and simple place to apply self-subsumption is in the conflict clause generation. The following 5-line algorithm can easily be added to any clause recording SAT solver:

```
strengthenCC(Clause C)              – C is the conflict clause
  for each p ∈ C do
      if (reason(p̄)\{p} ⊆ C)
          mark p
  remove all marked literals in C
```

By *reason*(p) we denote the clause that became unit and propagated $p =$True.

For every literal p of the newly generated conflict-clause C, the algorithm tries to self-subsume C by the reason clause for p. If successful, p can be removed, corresponding to replacing C with $C \otimes_p reason(p)$.

The procedure fails if the reason clause for p contains at least one literal q not part of the conflict clause C. However, by following the reason graph (implication graph) backwards for q, we might find that q became true as a consequence of assigning only literals present in the conflict clause, in which case q can be ignored. If this is true for all literals not in C, p can still be removed.

Another way to view this is that any literal in a clause that is implied to be False by assuming a subset of the other literals in that clause to be False, might be removed. We are using the reason graph produced by the BCP to extract some information of this kind; thus making more use of the information we already spent some effort on deriving. For more details, please refer to the source code of MiniSat v1.13.

References

[MS99] J.P. Marques-Silva, K.A. Sakallah. **"GRASP: A Search Algorithm for Propositional Satisfiability"** in *IEEE Transactions on Computers*, vol 48, 1999.

[MZ01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, S. Malik **"Chaff: Engineering an Efficient SAT Solver"** in *DAC 2001*.

[ES03] N. Eén, N. Sörensson. **"An Extensible SAT-solver"** in *SAT 2003*, pp. 502-508.

[EB05] N. Eén, A. Biere. **"Effective Preprocessing in SAT Through Variable and Clause Elimination"** in *SAT 2005*.

[GY02] E. Goldberg, Y. Novikov. **"BerkMin: A Fast and Robust SAT Solver"** in *Design Automation and Test in Europe*, IEEE CNF 2002.

[Ry03] L. Ryan. **"Efficient Algorithms for Clause-Learning SAT Solvers"**, M.Sc. Thesis, Simon Fraser Univ. 2003.