MiniSat 2.2 and MiniSat++ 1.1

Niklas Sörensson
Sörensson Research and Development, Göteborg, Sweden.
niklasso@gmail.se

1 Introduction

MiniSat is a SAT solver designed to be easy to use, understand, and modify while still being efficient. Originally inspired by zChaff [10] and Limmat [1], MiniSat features the now commonplace two-literal watcher scheme for BCP, first-UIP conflict clause learning, and the VSIDS variable heuristic (see [5] for a detailed description). Additionally, it has support for incremental SAT solving, and it exists in variations that support user defined Boolean constraints and proof-logging. Since it's inception, the most important improvements have been the heap-based VSIDS implementation, conflict clause minimization [4], and variable elimination based pre-processing [2].

2 MiniSat 2.2

The differences between version 2.2 and 2.1 are minimal, and are not expected to affect performance much. Most changes are adding features (for instance resource controls) or fixing some problems in high memory situations. Some scalability improvements to pre-processing have been made that allows it to be run on some large problems that were previously problematic.

3 MiniSat 2.1

This version is largely an incremental update that brings MiniSat more in line with the current most popular heuristics, but also introduces a number of data structure improvements. Most are rather well known and of lesser academic interest but mentioned in Section 5 for completeness.

Heuristics During the last couple of years it has been made clear that using a more aggressive restart strategy [7] is beneficial overall, in particular if it is used in combination with a polarity heuristic based on caching the last values of variables [12]. MiniSat uses the Luby-sequence [9] for restarts, multiplied by a factor of 100. For polarity caching it stores the last polarity of variables during backtracking.

Blocking Literals It can be observed that when visiting a watched clause during unit propagation, it is most commonly the case that the clause is satisfied in the current context. Detecting this without actually having to read from the clause's memory turns out to be a big win as indicated by [13], [8].

However, these techniques require an extra level of indirection which makes the win less clear cut. Instead, one can pair each clause in the watcher lists with one copy of a literal from the clause, and whenever this literal is true, the corresponding clause can be skipped. This is very similar to the approach used in the implementation of the SAT solver from Barcelogic Tools [11], but differs crucially in the sense that the auxiliary *blocking literal* does not have to be equal to the other watched literal of the clause, and thus there is no extra cost for updating it.

4 MiniSat++ 1.1

This tool is envisioned as a rewrite of MiniSat+ [6], but contains so far only the circuit framework necessary to participate in the AIG track. As an AIG solver it is currently rather simple: the circuit is first simplified with DAG-aware rewriting (inspired by [3], but far less powerful at the moment), then clausified using the improved Tseitin transformation (see [3] for an overview), and finally MiniSat 2.1 is run on the result, including CNF based pre-processing.

5 SAT-Race Hacks

The versions submitted to the SAT-Race contains two data structure improvements designed to improve memory behavior of the solvers: Firstly, binary clauses are treated specially as in MiniSat 1.14 [4]. In combination with blocking literals this is slightly more natural to implement, but on the other hand, there is some overlap in their beneficial effects and the difference thus becomes smaller. Secondly, a specialized memory manager is used for storing clauses. This was introduced to allow 32-bit references to clauses even on 64-bit architectures, but it also gives a small to modest performance benefit on 32-bit architectures depending on the quality of the system's `malloc` implementation.

References

[1] A. Biere. The evolution from limmat to nanosat. In *Technical Report 444, Dept. of Computer Science, ETH Zürich*, 2004.

[2] N. Eén and A. Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proc. of the 8^{th} Int. Conference on Theory and Applications of Satisfiability Testing*, volume 3569 of *LNCS*, 2005.

[3] N. Eén, A. Mishchenko, and N. Sörensson. Applying logic synthesis for speeding up sat. In *SAT*, pages 272–286, 2007.

[4] N. Eén and N. Sörensson. MiniSat v1.13 - A SAT Solver with Conflict-Clause Minimization. *System description for the SAT competition 2005.*

[5] N. Eén and N. Sörensson. An extensible sat solver. In *Proc. of the 6^{th} Int. Conference on Theory and Applications of Satisfiability Testing*, 2003.

[6] N. Eén and N. Sörensson. Translating pseudo-boolean constraints into sat. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 2:1–26, 2006.

[7] J. Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI*, pages 2318–2323, 2007.

[8] H. Jain and E. Clarke. Sat solver descriptions: Cmusat-base and cmusat. *System description for the SAT competition 2007.*

[9] M. Luby, A. Sinclair, and D. Zuckerman. Optimal speedup of las vegas algorithms. In *Israel Symposium on Theory of Computing Systems*, pages 128–133, 1993.

[10] M. W. Moskewizc, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proc. of 12^{th} Int. Conference on Computer Aided Verification*, volume 1855 of *LNCS*, 2001.

[11] R. Nieuwenhuis and A. Oliveras. Barcelogic for smt. `http://www.lsi.upc.edu/~oliveras/bclt-main.html`.

[12] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. In J. ao Marques-Silva and K. A. Sakallah, editors, *SAT*, volume 4501 of *Lecture Notes in Computer Science*, pages 294–299. Springer, 2007.

[13] T. Schubert, M. Lewis, N. Kalinnik, and B. Becker. Miraxt – a multi-threaded sat solver. *System description for the SAT competition 2007.*