Software Reliability
SAT Solver

Abraão Pacheco Dos Santos Peres Mota - CID: 00941232
Giulio Jiang - CID: 00979137
Domenico Marino - CID: 00000000

Department of Computing
Imperial College London
November 4, 2017

1 Algorithm

We propose our implementation of a simple yet functional SAT solver, entirely written in C. The program is completely self-contained and does not depend on any external library other than the standard `libc`.

Our solver is based on the DPLL algorithm, on top of which we have implemented a few optimizations. DPLL has been chosen for being intuitive in its workings, and for its expandability to include additional features and optimizations.

The solver accepts standard DIMACS CNF file formats, and will print UNSAT, or SAT accompanied by a model.

It is interesting to point out that our solver is completely deterministic, as it does not rely on any random number generator.

1.1 Overview of execution

We implement a standard DPLL loop. At each iteration, we pick a variable and assign an arbitrary value to it. Afterwards, we run BCP and PLP, evaluate the formula, and continue with assignments or backtrack depending on the result.

1.2 BCP

We implement BCP as a unit resolution algorithm. Given the formula, the BCP loop automatically assigns a value to all unit clauses found until there is no unit clause left in the formula, or a contradiction is derived.

The algorithm scans through the current working set for variables that have no assignment, to find the clauses that have exactly one unassigned variable. This means that our BCP algorithm will also work on clauses that have more than one variable present, although by construction of our unsatisfied clauses this isn't the case.

Automatic assignments generated by BCP are recorded in the assignments history table in a distinct section from the arbitrary assignments, as they are treated differently when backtracking.

1.3 PLP

`Ab can you do this?`
some details about PLP and why it helps?

1.4 Backtracker

The main loop of the solver does not use recursion, therefore we cannot use the call stack to keep track of the state to which we revert to in case we derive a contradiction. Additionally, we do not create copies of the formula, the variables and the clauses, but also work on the data in-place, which makes direct recursion impractical.

For the above reasons, we implemented a backtracker based on a heap-allocated assignment history tracker, which provides the full history of decisions, and a variable assignment map, which represents the current assignments to all variables.

The joint action of the decision algorithm and the backtracker create a depth-first search behaviour of the assignments space in the order that is initially generated for the variable. See the optimizations on variable decision section for the initial order of the variables.

The backtracker uses information contained in the assignment history levels to undo actions performed by both BCP and PLP and prepare the working context for the next assignment by unassigning all temporary values from the variables. This approach allows us to work on a single copy of clauses and variables when exploring unknown branches.

1.5 Data structures and design

We have implemented the following data structures:

- `arraylist` - A simple heap-allocated array list, used for long-term storage of items that are read frequently but rarely modified
- `linkedlist` - A doubly-linked list, used for data that is frequently inserted or deleted at arbitrary positions in the list
- `arraymap` - An integer-key map specifically designed for the variables assignment map, optimized for fast element access
- `hashset` - A generic hash set implementation, used to check for duplicate items

2 Optimizations

In an attempt to increase the efficiency of our solver, we have implemented the following optimizations.

2.1 Constant time formula evaluation

`Ab, please do this?`

2.2 Clause literals optimization

How we detect duplicate literals in a clause, and discard clauses with p or ¬p

2.3 Variable decision heuristics

`Dom, please do this?`

3 Difficulties and testing

About the bug that led us to cleaning up the clauses during parsing.

About the bug in the unsat tracker during unassignment that did not correctly re-evaluate the formula.

`Dom, please write how you did your additional testing?`

4 The Undefined Behaviour Bug

Our undefined behaviour bug patch introduces a use-after-free in some circumstances during backtracking.

As described in our optimizations section, we use two trackers to remember lists of clauses that are currently unsatisfied and currently false. At each assignment and un-assignment, we update the lists.

It would be a plausible bug to free the element being removed from one of those lists while removing the linkedlist's node. We introduce the bug by freeing a clause when removing it from the list of

false clauses. However, our clause objects are shared among other data structures as well, so they might be reused after this free.

The bug doesn't always manifest itself when running the solver normally. In fact tests 2, 3 and 4 run correctly despite the bug, while test 1 crashes with a segmentation fault. These behaviours could be different using different compilers or operating systems.

Clang's address sanitizer is able to correctly identify the use-after-free.

5 The Functional Bug

Suggestion: we do not detect duplicate literals in a clause, causing potential cases where clauses can be added twice to the unsat list or to the false clauses.