Software Reliability
SAT Solver

Abraão Pacheco Dos Santos Peres Mota - CID: 00941232
Giulio Jiang - CID: 00979137
Domenico Marino - CID: 00979083

Department of Computing
Imperial College London
November 7, 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

PLP acts as an optimisation for our solver in order to more quickly determine assignments for a subset of variables in our formula which appear as *pure literals*. A pure literal may appear in several clauses in the formula, but all the occurrences of this literal are either in its normal, positive form (e.g p), or they occur in the negated form of the literal (i.e. $\neg p$). Whenever a pure literal is detected, then we can immediately determine its assignment in the formula to satisfy it - if we have a pure positive literal, setting it to `true` is the only way to satisfy all these occurrences. On the other hand, for pure negated literals, these are set to `false`.

We implemented this by adding two different counters to our variable struct, one counting the occurrences of a positive literal in all the current clauses, while the other counter was for the occurrences of a negated literal. For a variable to occur as a pure positive literal, the matching counter should be greater than 0 and the negated literal counter at 0. The opposite is true for pure negated literals. This implementation needed to track changes to the working set of clauses, so whenever a clause was added or removed from our list of unsatisfied clauses, we looped over all the literals within it and updated the variable state counters accordingly. Doing this state management on assignment allowed us to discern pure variables very quickly, at the small overhead cost of managing this state.

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

We have designed our solver to work around the formula with 2 auxiliary data structures - `false_clauses` is a linked list of the working set of clauses we currently evaluate to be `false`, and `unsat` is a linked list of the clauses that are currently unsatisfied. These 2 structures allow us to focus on working with only the currently important clauses in question, and make it easy for us to evaluate the current state of the formula. Specifically, we know that the formula is false simply by checking the size of our `false_clauses` list to be greater than 0. Passing that check, if there are no clauses in the `unsat` list, we know it is true, otherwise, the entire formula is still unsatisfiable. These 2 data structures allow us to evaluate the overall formula in constant time by querying their size, at the slight cost of keeping them updated as we assign and revert variable mappings.

2.2 Clause literals optimization

We have further optimized clauses at the parsing level - we have ignored literals within a clause that occur more than once (in their specific literal form). In some test cases this behaviour was tested, and raised some bugs in our implementation. A literal occurring more than once in a formula can be ignored at the parse level as it does not add further information to the clause. Additionally, we have also looked out for literals occurring with their negated counterparts in the same clause (for example, a clause of `p` $\vee$ `¬ p`). These clauses become trivially true by logical evaluation, so we discard them from our formula altogether.

2.3 Variable decision heuristics

Our initial variable decision algorithm was very simple, as it involved selecting the first unassigned variable, ordered by the appearance order in the `CNF`, this was very simple to implement, but started showing its limits very early, for example with one hundred variables, the execution time

was already in the order of minutes.

To improve our decision logic, we decided to adopt an heuristic that takes inspiration from *Dynamic Largest Individual Sum (DLIS)* heuristic, but simplified; *DLIS* selects the variable that satisfies the highest number of clauses, a potential drawback of *DLIS* is that the variables have to be sorted after each decision point, and this could potentially be expensive from a performance point of view. Our approach instead selects the unsatisfied variable that appears in the most clauses, and sorts the variables only once, at the beginning of the evaluation, this is done to ensure good performance, moreover, it is simple to implement and easy to debug. As a smaller optimisation, our algorithm remembers what was the last assigned variable and starts searching for the closest unassigned one from there, this is done to speed up the decision for huge numbers of variables.

Using a wide variety of diverse `CNFs` we managed to obtain an average speed-up of 7x to 10x, for some test cases instead this resulted in unquantifiable speed-up, as one of our test `CNFs` (225 variables, dozens of thousands clauses) timed out after 1 hour with the initial approach, and completed in 16 minutes with the refined one.

2.4 Conflicts database

When taking decisions we keep track of the dependency graph between assignment. At a clause that evaluates to false, we can therefore follow the graph to find the root nodes containing the primary assignments, and generate a conflict clause that will be added to the conflicts database and evaluated together with the rest of the formula.

The conflict database has a fixed maximum size, which is necessary because of the complexity of our algorithm as the number of clauses increases. It is stored in a queue, and older conflict clauses are removed when the maximum size is reached.

2.5 Reverse BCP

We traverse known clauses in reverse order. In this way the newest conflict clauses are analyzed first, allowing us to derive falsity faster in the average case.

3 Difficulties and testing

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

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

The test formulae provided are useful for the parsing and initial testing, but are definitely not useful to assess the performance of the solver, this is due to the fact that performance for small number of variables or clauses is already very low without any heuristic or optimisation.

To assess our performance we relied on formulae from `SATLIB`[1], the website provides various sets of randomly generated formulae ranging from medium to large. These formulae, plus the ones from `SATCOMP` resulted in a comprehensive set of queries to check for correctness and efficiency.

Joining these formulae, we have a script that executes `MiniSat 2`[2] on each formula and stores the result for a comparison with our solver. This script allowed us to trace a bug related to `PLP` resulting in the execution terminating early and reporting the formula to be `UNSAT`; This bug only appeared with reasonably large formulae, which made the bug hard to spot by us.

Furthermore we did not implement Watch Literals in time, making our BCP algorithm quadratic in complexity as the number of clauses grows. Given enough time, we would implement Watch Literals as the next feature.

4 The Undefined Behaviour Bug

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

---

[1]http://www.cs.ubc.ca/ hoos/SATLIB/benchm.html

[2]http://minisat.se/MiniSat.html

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

PLP current assignment ignore

For our functional bug, we have chosen to look at the case where in PLP we willfully ignore whether the current variable has been assigned to already or not. Because our implementation of PLP iterates over all the variables in our formula looking for pure literals, if one of these variables is a pure literal and has a pre-existing assignment, then PLP will attempt to overwrite its value. This is a particularly subtle bug as it only expresses itself when the previous assignment is contradictory to the value a pure literal should logically be in a formula - i.e. a purely negated literal *should* be assigned to false in the formula to satisfy it (as PLP attempts to do). However, if the existing assignment to this variable is true, then PLP will ignore this and attempt