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 a 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 a `assignment_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. ¬`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 these 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 section on variable decision optimization 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 prepares the working context for the next assignment by un-assigning 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 we do not yet know the evaluation of. 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 semantic 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 $\mathtt{p} \vee \neg\ \mathtt{p}$). 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 a heuristic that takes inspiration from the *Dynamic Largest Individual Sum (DLIS)* heuristic, but simplified. *DLIS* selects the variable that satisfies the highest number of clauses and 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 and has the added benefit of being 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 variable from there. This is done to speed up the decision process for inputs with a very large number of variables.

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

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 as a queue (implemented using our linkedlist), 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.

2.6 Unit clause watch

To increase the effectiveness of BCP, we keep track of all the unsatisfied clauses with a single unassigned literal in a separate list. BCP does not need therefore to iterate through all unsatisfied lists and find those that only contain a single unassigned literal. This optimization yields large speed improvements in general as it reduces the underlying complexity of our BCP algorithm.

3 Difficulties and testing

One of the custom tests we used highlighted an edge case our solver wasn't handling correctly - using duplicated literals in a single clause. In any logical clause, having a duplicated literal adds no semantic value, so these can be ignored. However, our parser wasn't explicitly ignoring literals that were already added to our `variable_map` structure. As adding literals updated other state across our solver, this meant we added 2 duplicate clauses to our `unsat` clause list. This meant when we removed that clause from the list, there would still be the duplicate left in the list. This lead to the solver prematurely deciding that a formula was unsatisfiable due to the extra unsatisfied clause.

The test formulae provided are useful for the parsing and initial testing, but are definitely not useful to assess the performance of the solver, mainly due to the fact that performance for a small number of variables or clauses is already very fast without any heuristic or optimisation. To assess our performance we relied on formulae from `SATLIB`[1] - this website provides various sets of randomly generated formulae ranging in sizes. These formulae, plus the ones from `SATCOMP` resulted in a comprehensive set of queries to check for correctness and efficiency.

Alongside this, 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 our `PLP` implementation that resulted in the execution terminating early and reporting the formula to be `UNSAT`. This bug

[1]http://www.cs.ubc.ca/ hoos/SATLIB/benchm.html
[2]http://minisat.se/MiniSat.html

only appeared with reasonably large formulae, which made the bug hard to spot by us. Additionally, due to time constraints, we did not implement watch literals, 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

MergeSort off-by-1

Our undefined behaviour bug patch introduces an array out of bounds access.

As described in the optimizations section, we sort the list of variables to have the ones participating in the most clauses first. We introduce an array-out-of bounds access of 1 element past the end, which is undefined behaviour. The change does not produce any visible result on in our tests (although this could be certainly system dependant), but the Address Sanitizer is easily able to identify the out-of-bounds access.

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 subtle bug as it only expresses itself when the previous assignment is contradictory to the value a pure literal should logically be in the existing unsat clauses - i.e. a purely negated literal *should* be assigned to false in the formula to satisfy it (as PLP attempts to do with our current set of unsat clauses). However, if the existing assignment to this variable is true, then PLP will ignore this and attempt PLP using a negated assignment. This bug doesn't cause an infinite loop due to how we are keeping our assignment history. Further assignments made down the line by PLP have as a parent assignment the original assignment that the variable had, even though it isn't being taken into consideration. These PLP assignments will inevitably cause an unsat formula on the parent assignment because they are ignoring the assignment history.

A concise example of when this happens can be found in the original `test4.cnf` file provided (a copy has been made in `funcerr.cnf`). When the first variable is decided to be `true` by our algorithm, the leftover unsat clause contains a negated literal of the first variable, and because it is our only unsat clause, this is an occurrence of a pure literal. This then instantly triggers the bug as PLP assigns that variable to `false`.