Software Reliability Coursework 1
Implement a SAT solver

Anastasios Andronidis, Andrei Lascu (course assistants)
Cristian Cadar, Alastair Donaldson (lecturers)

1 Introduction

In this coursework, your task is to implement a **SAT solver** in the C programming language, in groups of up to three.[1] Your solver should take as input the path to a file containing a boolean formula in conjunctive normal form (CNF), using DIMACS format. Some information about this format can be found at the following links:

- `https://fairmut3x.wordpress.com/2011/07/29/cnf-conjunctive-normal-form-dimacs-format-explained/`
- `http://www.domagoj-babic.com/uploads/ResearchProjects/Spear/dimacs-cnf.pdf`

Your solver should then compute the satisfiability of the formula, printing either **SAT** or **UNSAT** accordingly. If the formula is satisfiable then a *model* should also be printed, such that the formula evaluates to *True* when the model is used as an assignment to the variables of the formula.

You should provide **two** additional versions of your SAT solver, one patched to introduce an **undefined behaviour** to your tool, the other patched to introduce a **functional error**. You should also provide a test case for each version that triggers the introduced bugs.

1.1 Provided files

You will be provided with the following files:

- `sat.c`: the skeleton file for your SAT solver. In its initial form, it always returns SAT and the same model. The solver takes exactly one argument, which should be a file containing the formula in CNF DIMACS format.
- `satmodelvalidator.py`: a model validator implemented in Python, which takes as arguments a file containing a logical formula in CNF DIMACS format and a list of numbers representing a model. More information is provided in Section 3.
- `Makefile`: you must use this to compile your project via the `make` command. In addition, you can run `make tests` to test your program against our tests. **You should not modify the makefile**.
- `tests.sh`: a shell script that will run your compiled solver against the `.cnf` files inside the `tests` folder. Do not invoke this script directly; use `make tests` instead.
- `tests/`: a folder containing various logical formulas in CNF DIMACS format (`*.cnf` files) and their corresponding expected results (`*.results` files).

1.2 Implementation restrictions

The SAT solver should be implemented explicitly in **C** and **without using any external libraries other than libc**. The solver should be **sequential**, i.e. no concurrency is allowed. It should be possible to compile and run the code on a standard lab machine. You are allowed to add as many source (.c) and header (.h) files as you wish, but only in the root of the folder. You are encouraged to add your own test files in the `tests/` folder to further validate your SAT solver. You should **not** modify the other provided files, since your submission will be tested using the **original** files.

[1]You are *required* to use C so (a) you can experiment with undefined behaviours of the C language that can be detected by sanitizers, and (b) because Part 2 of the coursework will test these solvers and will benefit from having them written in the same language; for instance, Part 2 will involve gathering coverage information.

1.3 Patched versions of your solver introducing bugs

For the two additional SAT versions, you should use `git` and create two branches: (1) **undef** and (2) **funcerr**. We expect to find one extra test in each of these branches called: **undef.cnf** and **funcerr.cnf** respectively, showcasing each problem. Both branches should have **exactly one extra commit** on top of the master branch.

The version of your solver in the **undef** branch should be edited such that for certain inputs an undefined behaviour can be triggered. Undefined behaviours will be covered in the course, but in C these include out-of-bounds memory accesses, division by zero, signed integer overflow and use of dynamically-allocated memory after it has been freed. It is not necessary for your solver to crash due to the undefined behaviour that your patch can trigger; while it *may* crash, it is also possible that the undefined behaviour leads to no observable difference in results with some compilers, or that the undefined behaviour causes the results computed by the solver to change without the solver actually crashing. Those bugs should be detectable by *ASan*, *MSan* or *UBSan* and you should explain which sanitizer you used and describe the bug in your report. To enable the sanitizers, you need to compile your project making use of the `flags` variable: for instance, to add the *ASan* sanitizer, you should use the command `make flags="-fsanitize=address"`. More information about sanitizers can be found here:

- `https://clang.llvm.org/docs/AddressSanitizer.html`
- `https://clang.llvm.org/docs/MemorySanitizer.html`
- `https://clang.llvm.org/docs/UndefinedBehaviorSanitizer.html`

The version of your solver in the **funcerr** branch should be edited such that for certain inputs the solver produces a wrong result, e.g. prints the wrong model for a satisfiable formula, or reports that a satisfiable formula is unsatisfiable (or vice-versa). It should be possible to trigger the functional error *without* triggering undefined behaviour, and the input you provide to demonstrate the functional error should not trigger undefined behaviour.

Extra credit will be given for introducing errors that are *subtle* in the sense that they can only be triggered by particular inputs. Some effort should be made to simplify the buggy examples as much as possible.

2 Deadlines and marking

Dates The deadline for forming groups is **Wednesday, 18 October 2017 at 5pm**. The deadline for completing this coursework is **Wednesday, 8 November 2017 at 7pm**.

Marking The final mark will be comprised of: an automated testing phase, where your tool will be run on a set of tests (60% of the marks); the patches for unidentified behaviour and functional errors (10% of the marks); and a report detailing your implementation (30% of the marks).

For the automated testing, the scores will vary from **-3600** to **1200** marks, normalised based on final results. The tests used to evaluate your solvers come from **SATCOMP**, a competition which aims to find the state-of-the-art SAT solvers. These tests will have at most 200,000 variables and 800,000 clauses. A sample of the kind of tests your solvers will be used against are available at the following link:

`http://multicore.doc.ic.ac.uk/SoftwareReliability/SRCoursework1Tests_20.tar`

Each test in the automated test suite will be scored as follows:

- Correctly classified a test as UNSAT: +2 marks
- Correctly classified a test as SAT and provided a correct model: +2 marks
- Correctly classified a test as SAT, but given model is incorrect: +1 mark
- Incorrectly classified a test as SAT: -3 marks
- Incorrectly classified a test as UNSAT: -6 marks
- Other (e.g. crashed, timed out): 0 marks

3 Expected input and output

Your implementation is expected to take exactly one argument as input. The argument should be a path to a file containing a formula in CNF DIMACS format. Your implementation should check the satisfiability of the given formula and print the result to standard output. In case the formula is satisfiable, you should also print a model for the formula. Note that there may be multiple models that can satisfy a given formula. If that is the case, your implementation may print any such model. Furthermore, it is acceptable for multiple runs of the solver on the same input formula to lead to different models being printed. If this is the case, mention it in the report.

A model is represented by a list of integers, corresponding to the variables in the logical formula. A positive integer means that the corresponding variable should be *True*, while a negative integer indicates a *False* value.

3.1 Example

Consider the file `formula.cnf`, located within the provided `tests` folder, containing the following:

```
p cnf 3 2
1 2 0
-1 3 0
```

This corresponds to the logical formula:

$$(A \lor B) \land (\neg A \lor C)$$

The formula is satisfiable, and a valid model is:

$$A = \bot ; B = C = \top$$

In this case, if your implementation succeeds in deducing that the formula is satisfiable and computes this particular model, it should produce (each printed line should end in a new line):

```
$ ./sat formula.cnf
SAT
-1 2 3
```

The model is not unique, and your tool may produce any of the following models:

- 1 2 3
- 1 -2 3
- -1 2 -3
- -1 2 3

We also provide a model validator, in `satmodelvalidator.py`. The tool takes a file containing a logical formula in CNF DIMACS format and a model, as presented above, within quotes. It will then check whether the given model is a valid model for the given formula. Using the above example:

```
$ python satmodelvalidator.py formula.cnf "-1 2 3"
MODEL OK
```

If an incorrect model is given, you will be notified of the first instance of a clause which evaluates to *False*.

4 Expected deliverables

- **code.zip**, containing all your .c, .h files and the `.git` folder with the appropriate branches (one adding an undefined behaviour bug to your code, and the other adding a functional error) and tests to showcase those bugs.
- **report.pdf**, a maximum 4-page report, using 11pt Arial font and minimum 2cm margins, describing: your implemented algorithm; the two patches, the bugs they introduce and features of the inputs you have presented that trigger these bugs; any notable design choices you have made, including details of optimizations you have implemented; notable details of efforts you made to test or verify your tool.

4.1 Optimisations

You are encouraged to implement optimisations in your solver so that it scales better when applied to challenging formulas. You will gain credit for this in two ways. First, the tests on which we evaluate your solver come from a real SAT solving competition (SATCOMP) and will include hard instances that will require optimisations to be handled within feasible time bounds. Second, you can describe any optimisations you have implemented in your report, adding academic weight to it.

The lecture notes provide details on a variety of optimisations. If you are interested and want a particularly high mark then we encourage you to look at the SAT research literature, as well as existing open-source SAT solvers and any other materials that you can find online, as a source for further optimisation ideas. However, your implemented solver must be your own work (i.e. you may not copy-and-paste code from other projects), and you must understand it sufficiently well that you are able to explain how it works (which must be included in the report).

4.2 Report contents

For the report, you are free to include information you consider important regarding your implementation. To get started, here are some pointers:

- If you chose a specific algorithm from the literature, explain why you chose that particular algorithm in favour of others. If you instead implemented your own approach, explain what advantages it offers over existing algorithms you have studied. In both cases, a brief description of the implementation itself should be provided (such as the main algorithm is contained in this file or function, or the model synthesis is performed at this point).
- Any specific optimisations you have implemented, if they were for performance reasons or to handle certain types of inputs.
- Specific difficulties you have encountered during the implementation, be it coding-related, functional or to do with certain corner cases.
- Specific efforts you made to rigorously test or verify your implementation.

One thing the report should show is that you have good understanding of the code you submitted and the design choices you have made. A person with some experience with SAT solving should be able to understand how your tool works from reading your report.

4.3 Solver sharing

Note that your solver, including source code, might be shared with the other groups in Coursework 2.