15-354 Final Project: SAT Solver

Fan Pu Zeng
fzeng@andrew.cmu.edu

Boolean satisfiability (SAT) solvers have played an important role in software and hardware verification, automatic test pattern generation, planning, scheduling, and solving challenging problems in algebra [1]. This report will introduce my SAT solver, its implementation details, designs and algorithms used, some comparisons between using different heuristics for splitting and choosing of variable assignments in the unforced case, challenges faced, and future directions.

1 SAT Solver Overview

The SAT solver uses the classical Davis–Putnam–Logemann–Loveland (DPLL) algorithm, which is based off backtracking search. In order to speed up the search process, unit propagation (also known as boolean constraint propagation), and the usage of watchlists for ease of backtracking is also implemented. Both of these will be discussed in detail later.

2 Project Files

2.1 Overview

An overview of the project files, and how to run them is given in this section to whet the appetites of the reader.

- `src/sat.py` This is the entry point of the project, and this is also where the DPLL algorithm is implemented in the `SATSolver` class
- `src/loader.py` This contains the `Loader` class that loaders a `.cnf` file into our proprietary CNF format
- `src/lib.py` This contains the definitions for the classes representing various SAT objects. In particular, we have the following:
  - `Variable`: This represents a base variable, i.e x_1.
  - `Var`: This represents a variable in one of two states: either negated (i.e $\neg x_1$), or not negated (i.e x_1). A `Var` is associated with a base `Variable`. We will use the term var in our discussion to make it clear that we are talking about vars that has the potential to be negated, instead of the base variable.
  - `Clause`: This represents a CNF clause. We require at least two vars in each clause due to our implementation of watchlists watching 2 vars. This is an acceptable assumption since if not, we can simply duplicate the same vars that already appear. Duplicated vars are allowed. Example: $(x_1 \vee x_2 \vee \neg x_3)$
  - `SAT`: This contains a list of CNF clauses, which represents our SAT instance. Example: $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$
- `src/assignment.py` This contains the `Assignment` class, that manages all the current variable assignment information, and also has methods related to assignments, like unit propagation and backtracking.

- `small/` This is a directory that contains small hand-crafted test cases that can be easily traced and verified.
- `Makefile` A Makefile to set up the `/dat` directory, and to run the SAT solver on each of them automatically.

2.2 Running the SAT Solver

To run the SAT solver, run `./src/sat.py <filename>`. It also takes in verbosity flags, `-v` and `-vv` depending on the level desired. By default, only the test case being run and the result is given as output. `-v` prints the initial SAT formula, the current decisions being made and whenever backtracking is performed. It also outputs the final satisfying assignment. For instance, try running `./src/sat.py -v dat/sat/uf50-0100.cnf` to see the assignments for one of the test cases. `-vv` outputs almost all information about what the algorithm is currently doing.

As an example, try running `./src/sat.py small/small-sat1.cnf`, which corresponds to the CNF $(x_1 \vee x_2 \vee x_3) \wedge (\neg x_1 \vee \neg x_2 \vee \neg x_3)$, which should output `SATISFIABLE`. For an unsatisfiable instance, try running `./src/sat.py small/small-unsat2.cnf`, which is an unsatisfiable instance corresponding to the following SAT instance:

$$(\neg x_1, x_2, x_4) \wedge (\neg x_2, x_3, x_4) \wedge (x_1, \neg x_3, x_4) \wedge (x_1, \neg x_2, \neg x_4) \wedge$$
$$(x_2, \neg x_3, \neg x_4) \wedge (\neg x_1, x_3, \neg x_4) \wedge (x_1, x_2, x_3) \wedge (\neg x_1, \neg x_2, \neg x_3)$$

This should output `UNSATISFIABLE`.

2.3 Test Files

The `dat` directory should already be setup in the submission directory, but in the event of issues, running `make clean` and `make setup` should restore the testcases directory (if you run into issues, please let me know - but try to avoid restoring it if there are no issues, in the event of uptime issues with the testcase source). The testcases are taken from SATLIB (`https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html`), and are split into two folders - `dat/sat` and `dat/unsat`, for satisfiable and unsatisfiable instances respectively. Each directory contains 1000 SAT instances, each of which contains 50 variable and 218 clauses. SATLIB claims that these instances are generated uniform at random.

To run the SAT solver on all the instances that should be satisfiable, run `make sat`. This will take a while (around 10 minutes). To run unsatisfiable instances, run `make unsat`. This will take even longer. This should also give a lot of confidence to the correctness of the SAT solver's implementation, as it gives the correct answers for both test suites containing 1000 test instances.

3 Algorithm

The two interesting algorithms implemented for SAT solving are DPLL and unit propagation. We will examine both in detail.

3.1 DPLL

Algorithm 1: DPLL algorithm (implemented in `sat.py`)

```
Result: Either SAT or UNSAT
while not all variables assigned do
    if unit propagation returns conflict then
        return UNSAT
    end
    x ← choose splitting var
    val ← choose initial assignment for x
    create new decision level with x=val
    while unit propagation returns conflict do
        if decision level == 0 then
            return UNSAT
        end
        backtrack and set splitting variable for previous decision level to be the negation of
         the original choice
    end
end
return SAT
```

The DPLL algorithm works as follows: at every step, it chooses a variable to assign, and also chooses what value to first try assigning it. Whenever it makes such a voluntary assignment (i.e not forced), a new decision level is created [2]. A decision level contains all the current assignments, and the variable that created the decision level. In our code, this is handled by the `Assignment` class, and the `assignment_stack` keeps track of the decision levels.

Once a new decision level is created, we perform unit propagation (elaborated next). Unit propagation forces assignments, so there is no need to create new decision levels. If unit propagation results in conflicts (i.e a clause that is unsatisfiable for sure given the current assignments), we need to backtrack. Backtracking involves returning to the previous decision level, and forcing the assignment of the variable that caused the conflicting decision level to the negation of its previous assignment. If we ever run out of decision levels, it means that all possible choices of assignments results in conflicts, and therefore the SAT instance is unsatisfiable. By using decision levels (and also watchlists, elaborated later), we can write our code in an iterative instead of recursive manner, which is significantly faster as it avoids the overhead of moving around stack frames.

In the DPLL algorithm, we allow the user to specify their own heuristics for both choosing the splitting variable, and the initial first assignment to use. This can be done by modifying `choose_splitting_var` and `choose_assn` in `heuristics.py` respectively. The file is well-documented and contains information on the information that is passed in that can be used to write a useful heuristic.

By default, if either function throws `NotImplementedError`, the SAT solver will choose the first unassigned variable, and default to trying to assign it to true first. The sample implementation given in `heuristics.py` uses a randomized strategy for both `choose_splitting_var` and `choose_assn`, and is meant to demonstrate how the arguments can be used.

3.2 Unit Propagation

The idea behind unit propagation is that when we have all other vars in a clause evaluating to false except one, then the final one must be set such that it evaluates to true. This is a forced

assignment. This allows for great speedups as the search space can be drastically reduced.

Instead of naively inspecting every clause during unit propagation which is costly, we can instead only examine clauses that are actually affected by assignments [3]. In particular, we only care about clauses that contains vars whose corresponding variables are assigned a value that makes it evaluate to false. For instance, in the clause $(x_1 \vee x_2 \vee x_3)$ we can continue to sleep peacefully x_1 was assigned to true, but we definitely will be concerned if x_1 was assigned to be false.

To this end, we introduce the idea of watched literals introduced by Moskewicz et al [4]. Each clause contains a watchlist, which contains two variables that it is currently watching. It is initialized to be the first two variables. The requirements are that we must always have at least one of the variables that it is watching be non-false (i.e either true or unknown). Whenever this is violated, the clause must find another variable to watch. If this is not possible, then unit propagation returns `CONFLICT`.

Another benefit of using watched literals is that during backtracking, our constraints can only be relaxed (i.e our variables can be unassigned but never assigned), and therefore we do not need to update them.

Algorithm 2: Unit Propagation Algorithm (implemented in `assignment.py`)

```
Result: Either OK or CONFLICT
propagation_queue: whenever we assign x=TRUE, we add ¬x to this queue
while propagation_queue is not empty do
    x ← pop from head of propagation_queue
    for each clause C watching x do
        y ← other var watched by C
        if y == TRUE then
            continue
        else
            try to make C watch a non-False var instead of x
            if no such var exists then
                if y == UNKNOWN then
                    assign y =TRUE
                else
                    return CONFLICT
                end
            end
        end
    end
end
return OK
```

In the unit propagation algorithm given above, we go through each of the vars that is now false as a result of previous assignments, and look at all the clauses watching them, and try to maintain the invariant that each clause is watching at least one non-False variable. If this is not possible and the other variable y being watched is an unknown, we then know we can force it to be true. If y is already assigned, then we have a conflict, and backtracking is inevitable.

4 Experiments with Heuristics

The SAT solving framework provides flexibility for changing the heuristics used easily. I ran some tests to see how well they performed relative to each other, with three different strategies:

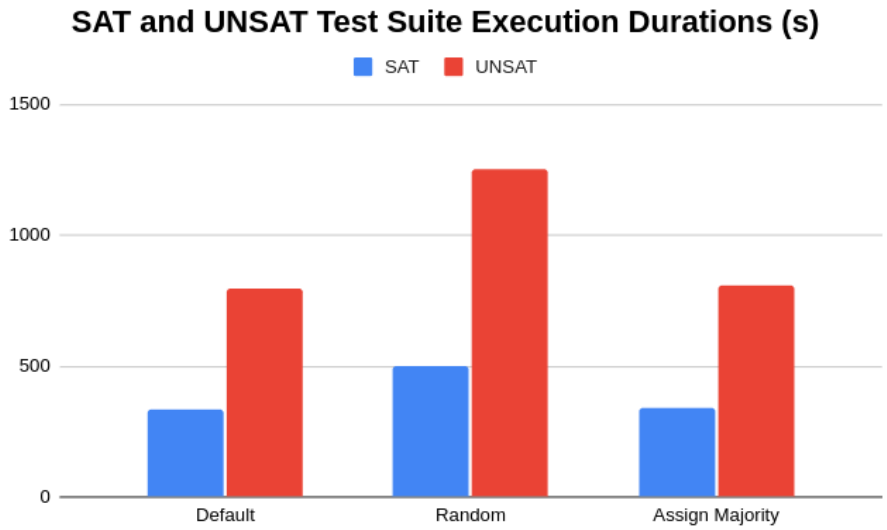

Figure 1: Chart of results

1. Default strategy: use first unassigned variable, and always assign True first
2. Randomized strategy: use any unassigned variable, and use random initial truth values
3. Majority strategy: use first unassigned variable, and assign it the value that will satisfy the majority of the clauses that it appears in

The results are given in the table:

Duration to run complete test suite		
Heuristic	**SAT**	**UNSAT**
Default	337.15	795.01
Random	497.92	1255.64
Majority	338.17	810.15

Figure 1 shows the results graphically. It is surprising that the random strategy produces much worse results than the default strategy and the majority strategy. It is also surprising that the default naive strategy actually performs the best in this case, even though intuitively the majority strategy should perform better. A possible explanation for this behavior is that the majority strategy is a greedy strategy, and therefore random SAT instances can be resistant to greedy strategies.

5 Challenges Faced

Writing the SAT sovler was a fun adventure. Maintaining the watched literals for each clause correctly was much trickier than I expected, and I had to write a few debugging routines and put assertions to diagnose and fix a few bugs (see `check_invariants` in `sat.py`. I also faced an insidious bug where I was modifying the watchlist of a clause while iterating through it, resulting in expected behavior. Debugging was also challenging because the SAT solver always performed correctly on my small hand-crafted SAT instances, and only failed on the much larger testcases from SATLIB which is far more difficult to trace.

It also took me a while to convince myself of the correctness of the algorithm from the interplay between the propagation queue, the watched literals, and the decision levels, although it seems completely obvious to me now.

6 Future Directions

There are many other promising optimizations that can be added to improve the speed of this SAT solver.

One possibility is to implement conflict-driven clause learning (CDCL)introduced by Marques et al [5], which is an extension of DPLL where it remembers conflicts that occurred previously and uses that to learn new clauses. This helps to prune the search space. We can also consider extending CDCL with random restarts, which has shown good results in practice [6]. Random restarts has been shown to allow CDCL to learn about persistently troublesome conflicting variables earlier, and therefore converges to a solution faster.

Another direction is to improve our heuristics for choosing variables and their values. MOMS (Maximum Occurence in clauses of Minimum Size) is a heuristic where we prioritize assigning variables that occurs the highest number of times in short clauses. Bohm's heuristic chooses the variable that appears the most in unsatisfied clauses. The VSIDS (Variable State Independent Decaying Sum) heuristic assigns each variable a weight, which is decayed at each time step. The weight of a variable is increased whenever it is involved in a conflicting clause. The heuristic then selects the variable with the highest score.

Other directions include using conflict-directed backjumping [7], which allows the solver to go more than one level up the decision level given certain conditions. Backjumping is a general technique used to speed up backtracking algorithms.

References

[1] Frank Van Harmelen, Vladimir Lifschitz, and Bruce Porter. *Handbook of knowledge representation*. Elsevier, 2010.

[2] Jediah Katz. Algorithms for SAT.

[3] Swords. Basics of SAT Solving Algorithms, Dec 2008.

[4] M.w. Moskewicz, C.f. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: engineering an efficient sat solver. *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*.

[5] *Handbook of satisfiability*. IOS Press, 2009.

[6] Carla Gomes, Bart Selman, and Henry Kautz. Boosting combinatorial search through randomization. *Proceedings of the National Conference on Artificial Intelligence*, 03 2003.

[7] Richard M. Stallman and Gerald J. Sussman. Forward reasoning and dependency-directed backtracking in a system for computer-aided circuit analysis. *Artificial Intelligence*, 9(2):135–196, 1977.