CS 7240 Project Report: CDCL SAT Solver Implementation

William Schultz

May 3, 2022

1 Introduction

Satisfiability (SAT) solvers have become powerful tools for solving hard, generic constraint satisfaction problems. Even though the SAT problem is known to be fundamentally hard (NP-complete), these tools are now effective at solving large, nontrivial real world problem instances and are applied widely in hardware and software verification, program analysis, electronic design automation, etc. The goal of this project was to implement a SAT solver based on relatively state of the art techniques, which are mostly based on *conflict driven clause learning* (CDCL) [3, 2, 7], an extension of the foundational DPLL algorithm [6]. The goal was to implement the solver and compare its performance to other state of the art solvers, and to understand the underlying algorithms in more depth.

2 Preliminaries

The satisfiability problem is defined as follows. Given a boolean formula f in conjunctive normal form (CNF), determine if there exists an assignment of values to the boolean variables of f such that f evaluates to true under this assignment. We want to return such a satisfying assignment, or determine that no such assignment exists. For example, the following CNF formula

$$(x_1 \vee x_2) \wedge (\neg x_3 \vee \neg x_1)$$

is satisfiable with an assignment $\{x_1 = 1, x_2 = 0, x_3 = 0\}$. The standard convention is to view CNF formulas as sets of clauses, where a clause is simply a set of *literals* i.e. a variable or its negation. Thus, the above formula would be represented as the set

$$\{\{x_1, x_2\}, \{\neg x_3, \neg x_1\}\}$$

Determining satisfiability of an arbitrary boolean formula in CNF is NP-complete [4].

3 DPLL

A basic approach to solving satisfiability is to view it as a search problem over possible assignments to variables. This is the basic idea behind the Davis–Putnam–Logemann–Loveland (DPLL) algorithm [5]. The DPLL algorithm essentially performs a depth first, brute first search over the tree of possible assignments, performing basic formula simplifications under as it extends partial assignments in its search. In particular, standard DPLL employs the *unit propagation* rule. A clause in a CNF formula is a *unit clause* if it contains exactly one literal. For a CNF that contains a unit clause c, it must be that in any possible satisfying assignment, c must be true, so we must set the variable for the literal in c accordingly to make c true. For example, consider the repeated application of unit propagation to the following formula:

$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$
$$\{\{b\}, \{\neg b, \neg c\}, \{c, \neg d\}\}$$
$$\{\{\neg c\}, \{c, \neg d\}\}$$
$$\{\{\neg c\}, \{c, \neg d\}\}$$
$$\{\{\neg d\}\}$$
$$\{\{\neg d\}\}$$
$$\{\} \quad \text{(SAT)}$$

In this case, repeated application of unit propagation leads us to the empty formula, which, interpreted as an empty conjunction, is trivially satisfied. We also recover the satisfying assignment from this sequence of unit propagation applications i.e. $\{b = 1, c = 0, d = 0\}$.

DPLL performs a depth first search in the tree of possible assignments, backtracking when it encounters a *conflict*, which is defined as occurring when some clause has all of its literals set to *false* in a current partial assignment. To illustrate the work done by a run of DPLL on a given formula, we can show its *termination tree*, which essentially shows the parts of the search tree that it explored during its run. For example, for the following CNF formula (with clauses labeled):

$$\begin{aligned}
c_1 &\quad \{a, b\} \\
c_2 &\quad \{b, c\} \\
c_3 &\quad \{\neg a, \neg x, y\} \\
c_4 &\quad \{\neg a, x, z\} \\
c_5 &\quad \{\neg a, \neg y, z\} \\
c_6 &\quad \{\neg a, x, \neg z\} \\
c_7 &\quad \{\neg a, \neg y, \neg z\}
\end{aligned}$$

we can see the paths taken by DPLL before it eventually finds a satisfying assignment where $\{a = 0, b = 1\}$.

4 Conflict Driven Clause Learning (CDCL)

DPLL is a relatively naive algorithm, in the sense that it is fairly close to naive, brute force depth first search. An improvement to this framework that started being used in SAT solvers in the 1990s and 2000s is known as *conflict driven clause learning* (CDCL). The technique is based around an idea of *learning from conflicts* and *non-chronological backtracking*. That is, when we encounter a conflict in a branch of the search tree (i.e. a clause is falsified), rather than simply backtracking naively, we try to learn more about what variable settings caused this conflict, in an effort to avoid making similar mistakes again in other portions of the search tree. In addition, we backtrack *non-chronologically*. That is, rather than backtracking the previous level of the search tree, we may jump back many levels, avoiding variables that were potentially irrelevant to the conflict we encountered.

5 SAT Solver Implementation

A large portion of this project consisted of working on an implementation of CDCL based SAT solver. The current version consists of approximately 1500 lines of C++, and the source code is found here: `https://github.com/will62794/mysat`. This code also includes an implementation of basic DPLL with unit propagation, in an effort to compare this approach with even a simple, non-optimized of CDCL.

The DPLL implementation also allows for capturing a basic version of the termination tree, as a means to both debug the implementation and also to understand the work done by basic DPLL with unit propagation.

A large part of the initial goal was to ensure correctness of the implemtnation before trying to benchmark its performance. Correctness testing was largely ensured by testing on randomly generated CNF formulas of varying sizes and with varying number of variables and clauses. In addition, the optimized, CDCL implementation was compared against a naive, brute force implementation to test conformance between the two implementations. This approach doesn't scale to large formulas, but works well for formulas with 10s of variables, since even a completely brute force solution can feasibly solve these instances.

6 Evaluation

Our initial evaluation of solver performance consisted of testing a small set of graph coloring benchmarks using both our DPLL and CDCL implementation, and also comparing to MiniSAT version 2.2, which is not a state of the art solver today but includes most of the basic modern techniques used by modern state of the art solvers, and so is very performant relative to naive solver implementations. Our initial benchmarking results are shown in Figure **??**, where the x-axis displays time to solve a benchmark, or timeout within a fixed budget of 25 seconds, on a log scale. This small benchmark set includes a variety of graph coloring problem instances, with around 50-100 variables.

References

[1] MiniSat SAT Solver. `http://minisat.se/`.

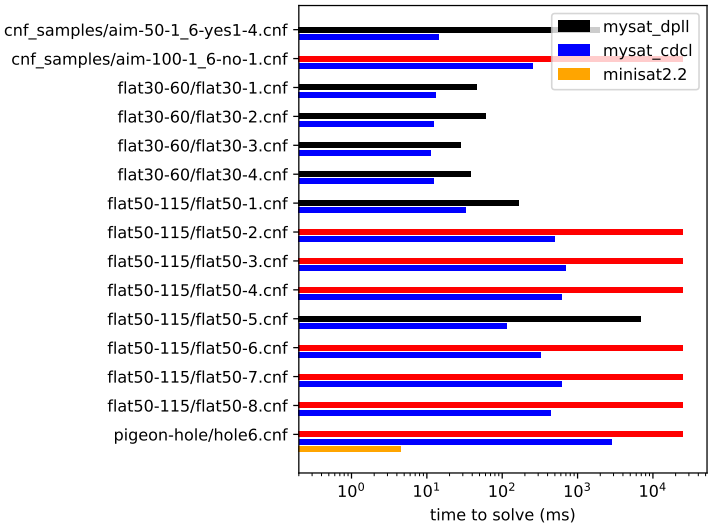

Figure 1: Preliminary comparison of my SAT solver implementation for both CDCL and DPLL variants against the MiniSAT 2.2 solver [1].
??

[2] Conflict-Driven Clause Learning SAT Solvers. https://www.cs.princeton.edu/~zkincaid/courses/fall18/readings/SATHandbook-CDCL.pdf, 2008.

[3] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD, 2009.

[4] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, page 151–158, New York, NY, USA, 1971. Association for Computing Machinery.

[5] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem-proving. *Commun. ACM*, 5(7):394–397, jul 1962.

[6] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *J. ACM*, 7(3):201–215, jul 1960.

[7] J.P. Marques Silva and K.A. Sakallah. GRASP-A new search algorithm for satisfiability. In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996.