# Implementing SAT Algorithms in Software


Callum Donovan
1915769
September 16, 2020
Swansea University

# Abstract

The Boolean Satisfiability Problem is a fundamental part of Computer Science. First proven to be an NP-Complete problem by both Stephen Cook in 1971[1], and Leonid Levin in 1973[2]. Since then, many more NP-Complete problems have been identified, along with their respective uses in the real world outside of pure computer science. Due to this, there has been a growing need for solvers that can effectively and efficiently process these problems. In the last two decades alone, huge progress has been made to coincide with the advancements in technology. And more SAT solvers have appeared that can be deployed in industries where they are most needed.

This paper explores the basic concept behind SAT solvers, and how they can be implemented in software using modern programming languages and tools. The algorithm that will be implemented is one proposed by Donald Knuth in his book "The Art of Computer Programming". Knuth proposes many algorithms, ranging from the most basic backtracking based algorithm, to an advanced implementation of WalkSAT[!]. We will explore the fundamentals of how to implement these algorithms, implement one of them, then test its performance against a suite of SAT problems.

The result of this paper will be a SAT solver implemented in C++, using the data structures and steps provided by Donald Knuth in his book "The Art of Computer Programming".

# Contents

# 1 Introduction

Satisfiability has presented itself a huge area of research throughout the history of computer science. Being the first proved to be NP-Complete in 1971[1], it has been the go-to for complex computation. And using reduction techniques pioneered in the mid-twentieth century, many other types of complex problems can be converted into SAT and solved using the plethora of solvers freely available. In more recent years, SAT competitions have added an incentive to the development of more advanced solvers, such as Minisat, zChaff, and CaDiCaL. On top of these, parallel solvers have gained popularity and one such solver, called Plingaling, ranked very highly in the SAT Competition 2020. Showing how parallel solvers can be just as good as their counterparts, if not better.

Donald Knuth has taken particular interest in this topic with his recent chapter of "The Art of Computer Programming". He discusses how important SAT solvers are to both industry and research, as well as providing algorithms and sample problems to test them against.

This paper will be an exploration into the world of SAT solvers and how they can provide crucial services to both industry and research. As well as methods in which proposed SAT solver algorithms can be implemented into a robust piece of software. Due to the sheer size of literature and research in this field, it would be unrealistic to think it is possible to cover everything in a comprehensive way. Therefore, this paper focuses on the fundamentals of SAT and the significant research through the years since Cooks paper in 1971[1]. As well as the ideas presented by Donald Knuth for implementation in modern programming languages.

## 1.1 Motivation

- Discuss how SAT solvers can aid industry.
- Discuss how learning about SAT can help other people learn too.

The motivation behind this project stems from the ever growing need for more efficient and faster SAT solvers. But in order to begin researching in this field, one must know the basics and tried methods present today. This project will do just that, in both studying literature that has formed the basis of most of the field, and implementing a basic SAT solver.

One of the most common arguments for the research into SAT solving is its viability for both science and industry. This presents a convenient compatibility between the two. As industry requirements grow, so too does the need for research into better algorithms to satisfy those requirements. An example in which SAT solvers are a key part of industry include Electronic Design Automation (or EDA). We will explore these usages further as we progress and highlight the advantages of fast SAT solvers when applied to these industries.

Another common viability for SAT solvers includes their usage for most complex problems that can be reduced to SAT. Most problems in the real world can be cast

as optimisation problems. This allows us to create a mathematical description of these problems, which in turn can present themselves for solving. The use of a solver in this instance would produce results which would allow for cost-saving or general efficiency improvements.

## 1.2 Aims and Objectives

To preface this project we must present the aims and objectives that we wish to aim for. As explained above, we do not wish to push the boundaries of SAT solvers in this particular project. But instead want to explore the basics and expand on current research. Aiding us in this goal will be the series by Donald Knuth, "The Art of Computer Programming". Knuth has recently taken interest in the concept of satisfiability for "Fascicle 6: Satisfiability" , and presents a large amount of content for us to analyse. In this book, Knuth outlines multiple algorithms that could be used to implement multiple common SAT solvers. These include:

- Algorithm B - Backtracking with watched literals.
- Algorithm D - DPLL.
- Algorithm C - CDCL.
- Algorithm L - DPLL with lookahead.
- Algorithm W - WalkSAT.

We shall be analysing each of these algorithms and presenting the data structures needed and methods of implementation using C++. Once our implementation is complete, and testing has proved the solver to be working as intended, we will record the performance metrics in a scientific manner to ensure accuracy. These metrics will then be compared to other FOSS solvers that are well tested and commonly used. This will give us a benchmark as to how increasing complexity of the algorithm and program can achieve greater performance. As well as how the programming language chosen can effect the overall performance.

We can easily turn these actions into functional and non-functional requirements. Doing so will allow us to test against the requirements stated and compare the expected result to the actual result. This format matches a more "software engineering" approach, which is desired for this project.

### 1.2.1 FREQ1

The resulting software should be able to process and input CNF formula in the semi-standard DIMACS format.

#### 1.2.2 FREQ2

The resulting software should be able to evaluate the satisfiability of a simple formula from FREQ1, and return the result when found. Simple refers to a formula which most SAT solvers would be able to process in a short amount of time. Result refers to the decision returned from the solver that concludes as ”SATISFIABLE” or ”UNSATISFIABLE”.

#### 1.2.3 Non-Functional Requirements

These requirements differ from functional requirements as they are not crucial to the project's success. These are simply goals that, if time allows, can be implemented for further data.

#### 1.2.4 NFREQ1

The resulting software should be able to process a complex formula within a reasonable amount of time. Reasonable refers to the length of time it takes to return a result when compared to other SAT solvers, i.e. 300 seconds to return ”SATISFIABLE”.

## 2 Background

Before we get into the implementation and software, it is always good to have a solid understanding of the topic at hand. SAT in particular is quite a deep topic, it is easy to spend a long time researching into all the intricacies[!]. For this paper, we will just be concentrating on the general concepts and ideas behind 3SAT problems.

- Maybe include some complex problem sets here?

### 2.1 SAT Fundamentals

- Define the SAT problem with reference.
- Explain how SAT formulas are constructed.
- Explain early attempts to solve SAT formulas.
- Discuss the DPLL algorithm, the first computer implementation of a solver.

### 2.2 3SAT

- Define 3SAT.
- Show a problem that would be considered 3SAT.
- Explain why 3SAT is of particular interest.

### 2.3 Modern SAT

- Talk about how SAT has evolved since the original DPLL.
- Modern SAT competitions.
- Parallel SAT solvers.
- General future SAT solver stuff.

## 3 Related Work

Moving on from the background of SAT, we can look into existing solutions that are freely available. As of 2020, there are plenty of FOSS programs that can be used to solve large sets of problems. This is in part due to the regularity of the SAT competitions that encourage people to build groundbreaking solvers.

### 3.1 Minisat

A popular and early solver that has won many competitions[!] is Minisat. Written in C++ and boasting just a mere few hundred lines of code, Minisat has been proven to be an effective tool for medium sized problem sets. First written in 2003 by Niklas Een[!] and Niklas Sorensson[!], its primary goal was to help developers and researchers get into the field of SAT solving by providing a simple interface and minimal codebase.

- Talk about Minisat.
- Go into detail about offshoots from Minisat, such as CryptoMiniSat.
- Newer CDCL algorithms.
- Look into SAT 2020 to find some interesting new solvers.
- Basically talk about applications then the research currently happening.

### 3.2 CryptoMiniSat

As the name suggests, this solver builds upon the foundations laid by Minisat to create an all in one solution to multiple problems. CryptoMiniSat implements most of the useful features from Minisat 2.0 core, along with PrecoSat and Glucose to try to make a one-stop solution. CryptoMiniSat is still in active development, hosting their code on GitHub for open collaboration.

```
Algorithm DPLL
    Input: A set of clauses C
    Output: A truth value

Function DPLL(C)
    if C is a consistent set of literals then
        return true;
    if C contains an empty clause then
        return false;
    for every unit clause {L} in C do
        C ← unit-propagate(L, C);
    for every literal L that occurs pure in C do
        C ← pure-literal-assign(L, C);
    L ← choose-literal(C);
    return DPLL(C   {L}) or DPLL(C   {not(L)});
```

# 4 Design

- Summary of what we would want from a design.
- Include the algorithm from Donald Knuth (algorithm D).
- Discuss the data structures that we will need to implement based off of DK's ideas.

## 4.1 Algorithm

The key to all the most successful SAT solvers are the algorithms. The first SAT solver to be implemented for a computer used the DPLL algorithm. As we previously established, the DPLL algorithm is considered one of the firsts and produced good results for the time. As of 2020, there have been a plethora of improvements to the original DPLL algorithm that has allowed it to keep up with the increasing complexity of problems available. The general pseudocode for the original DPLL algorithm can be written as such:

For our project, the algorithm that we will be interested in is a general DPLL algorithm. As previously discussed, this is just one of a few algorithms given from Donald Knuth. Although it is relatively simple, it is a good way to explore the basic idea of software SAT solvers. Forming a foundation for further improvements, much like those explained in algorithm L. Donald Knuth describes the algorithm in a series of steps, also outlining the data structures that we need to implement it as he has described.

- Discuss algorithms presented by DK.
- Discuss which algorithm we are planning to implement.
- Talk about the ones we aren't going to anyway?

# 5 Implementation

- Discuss the general idea of the program, including inputs (DIMACS) and outputs (SAT / NON-SAT).
- Discuss the process in which we are going to be solving these problems.
- Talk about our tool-chain and related crap.
- Talk about the attempts at implementing it.
- Show the final implementation.

## 5.1 Input

- The input will be a CNF formula in the DIMACS file format.

## 5.2 Processing

- Discuss how the processing works.
- This is the real meat that basically explains the algorithm.
- I feel it would be best to at least discuss the DPLL algorithm.
- Use content from TAOCP and online.

## 5.3 Output

- The output will be ”SATISFIABLE” or ”NOT SATISFIABLE”.
- Could also output the best solution.
- Explain the output standard for SAT competitions.
- SAT Competition 2002 output rules (see above comment).

## 5.4 Data Structures

- Discuss why we’re going to have to implement specific data types.
- Provide examples on the types of structures to be used.

### 5.5 Tool-chain

- Discuss the compiler and standard libraries
- Discuss the editor and debugger used.
- Discuss the OS & related things (such as shell).

### 5.6 Pseudocode & Code

- Investigate general pseudocode of generic DPLL algorithm (Wiki).
- Create some pseudocode from algorithm.
- Create some actual code to show?

## 6 Testing

- Summarize what we want from the testing phase.
- Discuss the problem sets that we are going to be throwing at this.
- Discuss program testing first, even though this is not really what we mean by ”testing”.
- Record some testing metrics, such as time to solve, memory, CPU cycles, etc.
- Describe any regions that could potentially be improved.

### 6.1 Software Testing

- Discuss methods of software testing (Unit, manual, etc...)
- Outline a testing table for manual testing.
- Discuss testing against the functional requirements stated previously.

### 6.2 Checking For Correct Solutions

- Add results from test folder
- Use two other solvers and give them the same problem
- Check at least 10 times to ensure consistent results

### 6.3 Monitoring Initial Performance Metrics

- Measure the performance of this SAT solver (algorithm we picked?)
- Measure the performance of other SAT solvers.
- Compare the results and deduce why the performance differences exist.
- Explain the use of the `'time'` command with special format.

### 6.4 Improving Performance

- Explore performance implications of algorithm.
- Talk about DK's performance predictions from TAOCP.
- Talk about how we could potentially improve the performance of our solver.
- Maybe talk about things like parallelism?

## 7 Evaluation

- Discuss our findings statistically.
- Discuss how implemented code could be improved (summarise).
- Talk about the methodology we used.
- Talk about the risks encountered, and the time frame in which we carried this out.
- Discuss the things that could potentially be improved if we did the project again.
- Discuss the pandemic.

## 8 Conclusion

- Just make a short summary of every single thing that was typed out above!
- Don't forget to basically copy and paste this to the abstract.

# References

[1] S. Cook, "The complexity of theorem proving procedures," Proceedings of the third annual ACM symposium on Theory of Computing, pp. 151–158, 1971.

[2] L. Levin, "Universal search problems," Problems of Information Transmission, pp. 115–116, 1973.