Implementing SAT Algorithms in Software

Callum Donovan
1915769
September 1, 2020
Swansea University

1 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, and Leonid Levin in 1973[1]. 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

2 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[!], 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.

2.1 Motivation

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

2.2 Aims and Objectives

- To make a SAT solver using instructions from "TAOCP"
- Make sure the solver can solve generic problem sets in DIMACS.
- Make sure the solver can do this in a decent amount of time.
- Basically FREQ1 and FREQ2.

3 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?

3.1 SAT Fundimentals

- Define the SAT problem with reference.
- Explain how SAT formulas are constructed.
- Explan early attempts to solve SAT formulas.

3.2 3SAT

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

3.3 Modern SAT

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

4 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.

4.1 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.

4.2 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 collaborative

5 Design

- Summary of what we would want from a design.
- Include the algorithm from Donald Knuth.
- Discuss the data structures that we will need to implement.

6 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 toolchain and related crap.
- Talk about the attempts at implementing it.
- Show the final implementation.

6.1 Psuedocode & Code

- Investigate general psuedocode of generic DPLL algorithm
- Create some pseudocode from algorithm.

6.2 Interesting features in C++

- Talk about how C++ can help when creating a program.
- Talk about how the standard template library can be used to reduce program complexity.
- Talk about the use of vectors as a handy data type over C style types.

6.3 Implementing Data Structures

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

7 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.

7.1 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

7.2 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.

7.3 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?

8 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.

9 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] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," Communications of the ACM, vol. 5, no. 7, pp. 394–397, 1962.