Implementing SAT Algorithms in Software

Callum Donovan
1915769
September 30, 2020
Department of Science, Swansea University

Project Dissertation submitted to Swansea University in Partial Fulfilment for the Degree of Master of Science.

Summary

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 dissertation 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[3]. 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.

This project demonstrates the ways in which SAT algorithms can be implemented, but does not add to any existing research in a significant way.

Declarations

Declaration 1

This work has not previously been accepted in substance for any degree and is not being concurrently submitted in candidature for any degree.

Statement 1

This dissertation is the result of my own independent work/investigation, except where otherwise stated. Other sources are acknowledged by giving explicit references. A bibliography is appended.

Statement 2

I hereby give consent for my dissertation, if accepted, to be available for photocopying and for inter-library loan, and for the title and summary to be made available to outside organisations.

Contents

1 Introduction **5**
- 1.1 Motivation . . . 5
- 1.2 Aims and Objectives . . . 6
  - 1.2.1 FREQ1 . . . 6
  - 1.2.2 FREQ2 . . . 6
  - 1.2.3 Non-Functional Requirements . . . 7
  - 1.2.4 NFREQ1 . . . 7

2 Background **7**
- 2.1 P vs NP . . . 7
- 2.2 Boolean Satisfiability Problem . . . 7
- 2.3 Complete vs Incomplete SAT Methods . . . 8
- 2.4 DPLL Algorithm . . . 9
- 2.5 Modern SAT . . . 10

3 Related Work **10**
- 3.1 Current Research . . . 11
  - 3.1.1 CDCL . . . 11
  - 3.1.2 NeuroSAT . . . 11
- 3.2 Popular SAT Solvers . . . 11
  - 3.2.1 Minisat . . . 11
  - 3.2.2 CryptoMiniSat . . . 12
- 3.3 Limitations . . . 12

4 Design **12**
- 4.1 Base Algorithm . . . 12
- 4.2 Algorithm D - SAT by Cyclic DPLL . . . 13
- 4.3 Algorithm L - Utilising Lookahead . . . 13
- 4.4 Algorithm X - Looking Ahead . . . 14

5 Methodology **15**

6 Implementation **15**
- 6.1 Input . . . 15
- 6.2 Output . . . 16
- 6.3 Tool-chain . . . 17
  - 6.3.1 Compiler & Libraries . . . 17
  - 6.3.2 Editor & Environment . . . 18

7 Testing **18**
- 7.1 Software Testing . . . 19
  - 7.1.1 Unit Testing . . . 19
- 7.2 Checking For Correct Solutions . . . 19
- 7.3 Monitoring Performance Metrics . . . 20
  - 7.3.1 UNIX Time Command . . . 20

8 Results **21**

9 Evaluation **23**
9.1 Accuracy 23
9.2 Performance 23
9.3 Methodology 24
9.4 Implementation 24

10 Future Work **24**
10.1 Pre-processing 24
10.2 Parallel Solving 24
10.3 SIMD 25

11 Challenges **25**
11.1 Software Based 25
11.1.1 C++ Learning Curve 25
11.2 Project Based 26
11.2.1 Late Start 26
11.2.2 Coronavirus 26
11.2.3 Maths Heavy Topic 26

12 Conclusion **27**

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

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[3] . 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 one 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 an 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 project, we will just be concentrating on the general concepts and ideas behind the most common instances of SAT. Along with some deep diving when required to ensure we understand what we're trying to achieve.

2.1 P vs NP

Before we progress into the basics of SAT, we can quickly explore the reason behind it being such an important topic within computer science. The P vs NP question is one of the most prevalent in the field of complexity theory. It is one of the seven millennium prize problems put forward by the Clay Mathematics Institute and, if solved, fetches a $1 million prize for the first correct solution. The concept of P vs NP asks whether every problem whose solution can be verified quickly, can also be solved quickly. On the surface sounding simple, but in the details an incredibly hard question that has plagued computer scientists since the 1950s.

The Boolean Satisfiability Problem was the first to be proven NP-Complete, by both Steven Cook and Leonid Levin in 1971 and 1973 respectively[1][2]. It was later included in a well-known paper, *Reducibility Among Combinatorial Problems* by Richard Karp[4], which generated a renewed interest amongst researchers. In this paper, Karp proved how a large set of problems could be reduced to an instance of SAT. Thus meaning each one would be solvable if there was a suitable algorithm.

2.2 Boolean Satisfiability Problem

As stated by Donald Knuth[3], the Boolean Satisfiability Problem is defined as such:

> "Given a Boolean Formula $F(x_1, ..., x_n)$, expressed in so called "conjunctive normal form" as an AND of ORs, can we "satisfy" F by assigning values to its variables in such a way that $F(x_1, ..., x_n) = 1$?"

Given this definition, we can present a formula as detailed above:

$$F(x1, x2, x3) = (x_1 \vee \bar{x_2}) \wedge (x_2 \vee x_3) \wedge (\bar{x_1} \vee \bar{x_3}) \wedge (\bar{x_1} \vee \bar{x_2} \vee x_3)$$

And given this formula, we can satisfy it by setting $x_1x_2x_3 = 001$.

To ensure consistent understanding of these concepts before proceeding, we can simplify the notation for each element within SAT. To begin, we have *variables*. These are elements of any convenient set. Variables can be denoted in a few ways, but for this we shall be using Knuth's own notation as stated in his book. Thus, variables will be denoted using numerals $1, 2, 3, ...$ to save having to repeat characters. We can also omit the brackets and operators to make the writing of formulas much faster:

$$R = \{12\bar{3}, 23\bar{4}, 341, 4\bar{1}2, \bar{1}\bar{2}3, \bar{2}\bar{3}4, \bar{3}\bar{4}\bar{1}, \bar{4}1\bar{2}\}$$

Literals are our next notation. They correspond to either a variable or the compliment of that variable. Using our previous example for variables, if 1 is a variable, both 1 and $\bar{1}$ are literals. From this we can easily deduce that if there are n number of variables, there can be $2n$ possible literals. A literal may be considered *pure* if its negation does not appear anywhere in the formula. And thus can be safely set to true or false respectively.

There are multiple instances of the satisfiability problem, so we shorten satisfiability to just SAT. In turn, each instance is typically abbreviated and prefixed to SAT. Examples of this would be 2SAT, 3SAT, and kSAT.

There are more definitions which represent alternative types of clauses that we could encounter. For instance, a clause of length 1 is called a *unit clause*. Clauses of length 2 would be called *binary clause*, length 3 *ternary clauses*, and so on. We can also encounter clauses in which there are no literals, which would be called *empty clauses*. *Empty clauses* are denoted using ϵ and are always unsatisfiable. As you might think, shorter clauses would be easier to satisfy as they contain less literals, but this is not necessarily the case. As we progress further into the design and implementation of this solver, we will learn the importance of covering all bases and making no assumptions of difficulty.

2.3 Complete vs Incomplete SAT Methods

Before we progress onto the details of SAT algorithms, we must understand the two distinct types of solvers; complete and incomplete. Currently we are most interested in the former of the two, as it presents the best usage out of all the problems that are faced within SAT. A complete SAT algorithm is one that finds both satisfiable and unsatisfiable solutions, and returns the corresponding model if satisfiable.

Modern SAT solvers are almost all complete and incorporate the original DPLL algorithm. They are extended using modern techniques to significantly speed up the solving process, and in most cases are capable of solving problems with millions of variables. Examples of some modern complete SAT solvers are Minisat and CryptoMiniSat. As we will detail below, these are two interesting solvers that are designed for particular purposes.

Incomplete solvers are much more rare compared to their complete counterparts. The obvious reason being that they do not return a result that we would be satisfied with, as they do not report if an instance of SAT is unsatisfiable. Aside from this, they do have their uses for specific types of SAT problems, hence why they are still being developed today. An example of an incomplete SAT solver is *WalkSAT*. WalkSAT is based on a local search algorithm which tries to find a satisfying assignment by iteratively improving on them until all constraints are satisfied.

2.4 DPLL Algorithm

The DPLL algorithm is one of the most important milestones in the history of SAT. Itself being a further development of work by Martin Davis and Hilary Putnam in their 1960 paper *A Computing Procedure for Quantification Theory*[5], the DPLL algorithm attempted to take this initial work and refine it.

The DPLL algorithm still forms the basis of most efficient SAT solvers today. Each instance of a SAT solver that utilises the original DPLL algorithm typically adds their own optimisations and improvements to achieve a greater efficiency, something we will expand on in further sections. In simplified terms, the backtracking section works by choosing a literal, assigning a truth value to it, simplifying the formula then checking if the new simplified formula is satisfiable. If this is the case the solver will return that the original formula is satisfiable and return the model. If this is not the case, the solver will backtrack and assign the opposite truth value and run the above checks again. The improvements made to this basic backtracking approach include adding rules to each step for enhanced performance. These rules are:

- Unit Propagation.
- Pure Literal Elimination.

The former capitalises on *unit clauses* to avoid a large section of the search space. By finding a unit clause, the solver will then assign the appropriate truth value to the literal to make it true, making these clauses trivial.

The latter capitalises on variables which only appear in one form throughout the formula, a *pure literal*. If this is the case, clauses in which this pure literal appear can be safely assigned in a way that makes all clauses they are contained in true. Doing this means the solver can safely delete the clauses where this is the case and further reduce the search space.

2.5 Modern SAT

Since the original instances of SAT there has been an astonishing amount of development into faster and more efficient methods. Since the millennium alone SAT solvers have managed to keep up with the quickly advancing hardware, allowing for the processing of SAT problems with sometimes up to millions of clauses. Alongside the state of the art solver have also been attempts to simplify methods used in software, to try and help people just getting into the topic. Solvers such as *Minisat* were designed to be as small and simple as possible, whilst still incorporating advanced techniques for maximum efficiency.

Alongside these advancements has been the annual SAT competition. This competition collates the latest and greatest work from the field and tests it against a plethora of advanced problems to see who comes out on top. For researches, or even hobbyists, being the victor of one of these competitions is an astonishing achievement. But there is more to the competition that just the prize. Multiple categories within the competition have allowed for more experimental ideas to come to fruition. Such examples as *Lingeling, Plingeling,* and *Treengeling* capitalise on the notion of parallelism and attempt to drastically speed up processing times. This is of course easier said than done, and prove that achieving efficient parallelism in SAT is quite a challenging task. As parallelism is becoming a key part of modern processor technology, it is integral to the development of software such as SAT solvers to exploit high core counts. As for SAT, there are a few ways one can go about this.

Portfolio SAT solvers are composed of multiple solvers that are all good at tackling different types of problems. As there is no single solver that is better at all problems than all others, it makes sense to apply each solver to a part of the problem that it is best at. For instance, we can collate some of the most competitive solvers into a single *portfolio*, and feed all of them the same problem. Once one of the solvers returns their result, all the others will also terminate. As you can imagine, this would lead to a large amount of duplicate processing. And whilst this is true, the advantage is that applying the problem to all these solvers will still be competitive, as the range of solvers will increase the chances of finding a solution in the fastest time possible. As with most parallel workloads, the downside to this is that the machine running these solvers will have to posses a large amount of memory. An example of an effective portfolio style solver is *HordeSat*[6], which is capable of almost linear speedup depending on the problem set.

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 Current Research

3.1.1 CDCL

Although it may feel like progress in this field can be quite slow, there is constantly ongoing research into methods in which we can improve the existing methods. Typical DPLL based solvers iteratively improve by including extra features that solve individual problems, such as look-ahead. Other research has suggested altering the traditional DPLL algorithm to be utilised on other problems such as *satisfiability modulo theories*. In this instance, an extension of the DPLL algorithm was created called *DPLL(T)*. Put simply, this version of the algorithm transforms an SMT problem into a SAT formula where atoms are replaced by boolean variables. Modern SAT goes a step further by implementing advanced features such as *clause learning* and *back jumping*, more taking inspiration from the original DPLL algorithm as opposed to iterating on top of it. The resulting algorithms are called *conflict-driven clause learning* and present a massive improvement over traditional DPLL based solver.

3.1.2 NeuroSAT

But aside from these developments, there is still alternative research being done to approach the SAT problem in different ways. One such instance of this is *NeuroSAT*, a SAT solver that utilises neural networks to power its internals. Although proven to not be anywhere near as performant as state-of-the-art SAT solvers, such as the ones mentioned above, it lays the foundation for the potential use of machine learning and AI in the world of SAT. As we have all come to expect in Computer Science, machine learning can sometimes be considered a buzzword for a lot of things that amount to nothing. However, NeuroSAT has been able to solve substantially larger SAT instances than it had been trained on[7]. Thus proving to be quite promising in concept and sparking interest in its potential. Although the concluding remarks of this paper suggest that there is no obvious way forward from this concept, it seems that it could potentially be something that could be learned from as a method of approaching SAT from a different direction.

3.2 Popular SAT Solvers

3.2.1 Minisat

A popular and early solver that has won many competitions[8] is Minisat. Written in C++ and boasting just a mere few hundred lines of code in its slimmest form, 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.

3.2.2 CryptoMiniSat

CryptoMiniSat is an advanced, modern SAT solver created by Mate Soos, Karsten Nohl, and Claude Castelluccia[9]. 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. This solver is the work of massive proportions and contains a extensive list of features to try to make as efficient a solver as possible. Based off of their 2009 conference paper *Extending SAT Solver to Cryptographic Problems*, it contains over 50, 000 lines of code as opposed to Minisat's 1,500. As stated in the projects FAQ, one of its main aims from the offset was to apply SAT solvers to cryptographic problems. However this ended up being just one of the many types of SAT that it excelled at.

3.3 Limitations

It can be quite challenging to come up with potential limitations for state-of-the-art solvers. But, when generally speaking, there is always potential for improvement based upon the advancements in computer hardware. Many SAT solvers nowadays are already more than good enough for industrial purposes. They are something that would have been merely dreamed of when the DPLL algorithm was first created. But as we go ahead into a promising future filled with technological innovations, such as the rise of quantum computing or alternative methods of constructing the transistor, we can be sure that further innovations with SAT will follow.

4 Design

The following section will include analysing and detailing the algorithms presented by Knuth. This will include both *Algorithm D* and *Algorithm L*, both of which are representative of varying complexity DPLL based solvers. We will flesh out the general structure to be translated into code and investigate the required data structures along with how they can be implemented in our chosen language.

4.1 Base Algorithm

The key to all the most successful SAT solvers is the algorithms. As we previously established, the DPLL algorithm is considered one of the firsts and produced good results for the time. As of the last decade, 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:

```
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 AND {L}) or DPLL(C AND {not(L)});
```

Figure 1: Generic pseudocode for DPLL.

4.2 Algorithm D - SAT by Cyclic DPLL

Knuth presents multiple algorithms to us in his book, but for this project are interested in specifically two of them. Algorithm D is the first we shall analyse, which will lay the foundation for Algorithm L.

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 possible 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.

Knuth names this algorithm "*Algorithm D - Satisfiability by Cyclic DPLL*. It is based on the original algorithm presented by Martin Davis, George Logemann, and Donald Loveland[10], extending previous work Davis did with Hilary Putnam[5]. At the time of it's inception, this algorithm was seen as fantastic. But due to limitations of the hardware, their methods of storing and processing data were quite verbose. It involved having to record all the data of the current node onto a magnetic tape before branching. When they wished to backtrack to the previous node, they would restore the data from the tape they had created. But now we have huge stocks of memory available to use whenever necessary, allowing us to write more complex and faster solvers using equally more complex data structures, as we will explore below.

4.3 Algorithm L - Utilising Lookahead

Knuth further expands on the previous algorithm by implementing a procedure called *look-ahead*. This procedure attempts to determine the best direction in which to branch by performing *look-ahead* on a set of variables, then evaluating the resulting reduction of the formula. Put simply, the *look-ahead* procedure

attempts to predict the future for the success of the present. As detailed by Knuth, we must switch from the 'lazy data structures' that were utilised in *Algorithm D*, to more 'eager' ones. The reason for this being that one of the major bottlenecks of *Algorithm D* is the detection of forced variable assignments. For *Algorithm D*, these indirect references aren't particularly fast, and by improving upon them we can speed up the algorithm significantly. Of course, this means that we must increase the complexity equally to match. These 'eager' data structures allow for much faster detection of forced values.

In order to achieve this, we must keep track of the binary clauses that occur in the relevant subproblem $F|L$. These clauses are kept in what Knuth calls a 'bimp-table', which is denoted by `BIMP(`l`)`, for every literal l. We only need to store the relevant facts about $(u \vee v)$, and can do so directly in a convenient way. This is done by listing u in `BIMP(`$\bar{v}$`)` and v in `BIMP(`$\bar{u}$`)` respectively. These two tables are stored using a sequential list of length `BSIZE(`l`)`.

This leads us to the next point Knuth raises that influenced his alterations for *Algorithm L*. It assumes that all the clauses that it will be facing will have a length of 3 or less. This is because all instances of the general SAT problem can be converted into 3SAT. Much like the above notation, these clauses are then represented by what Knuth calls 'timp-tables'. We can take the previous 'bimp-table' concept and apply it to these 'timp-tables'. Resulting in the respective notations of `TIMP(`l`)` to represent each literal's sequential list which has the length `TSIZE(`l`)`. The significance of this list allows us to neatly represent ternary clauses in a binary way, and link them together cyclically to represent the original ternary clause. As *Algorithm L* does not generate any ternary clauses, the memory can be safely allocated after the problem has been parsed.

A major difference to the previous algorithms comes in the form of variables having *degrees of truth*. This concept allows the algorithm to deduce the potential consequences of a literal in a particular context via a breadth-first search procedure. This is achievable by using a sequential stack $R_0, R_1, ...,$ to record the names of the literals that have received values.

Knuth also highlights another algorithm inside of *Algorithm L* which he calls *Algorithm X*. This is the meat of the algorithm and is where the *looking ahead* occurs. It is what is called before *Algorithm L* decides which literal to branch on, and is really where we can see some major performance improvements over *Algorithm D*. As Knuth points out, the factor of speedup which we can achieve with this *Algorithm X* depends on the complexity in which we put into it. A simple version of this *lookahead* procedure will provide us with a speedup of three times over *Algorithm D*. But Knuth points out that by spending time to create a much more sophisticated version, speedups of up to ten times more can be achieved. This is of course most prevalent on the larger instances of SAT that we would test against.

4.4 Algorithm X - Looking Ahead

This algorithm is based on a well known paper *Implementing Additional Reasoning into an Efficient Look-Ahead SAT Solver*, by Marijn Heule[11]. His SAT

solver is considered to be one of the most impressive in the field, and utilises multi-level look-ahead techniques to achieve high accuracy branching prediction. In order to find the best way in which to branch, we must consider three points that will discover the potential of a literal's assignment. The first is to find the best candidate of the free variables that are currently available. The second is to let these literals to share their implications. The third is to investigate the immediate effect that would entail our decisions. Using this approach, we can look into the hypothetical future and use that information to feed our branching decision. In some cases this algorithm may even find that a decision will allow us to solve the problem and terminate successfully, something that the other algorithms may have not found after a significant amount of search space.

5 Methodology

Before we begin talking about the implementation, is is necessary to quickly mention the methodology that we shall be using in development. This project is not being shared amongst a large team of engineers. This allows us to choose from a plethora of software methodologies that would typically not be considered. The usual norm is to commit to the AGILE methodology no matter the project. But for us we do not need the potentially cumbersome nature that can come from AGILE. We also need to consider the idea of rapid prototyping and development. For this reason, we have chosen to use the traditional Waterfall methodology. It gives us the general structure for the project, and fits well with our rigid requirements.

6 Implementation

This section concerns the general input and output that our solver will use. As well as the method in which we will attempt to implement it. If there are any tools already available to us, or libraries that may help us, they will also be listed here. We will also be glazing over the tool-chain and related items that are to be used to create the solver. This includes the language and the compiler that will be used. As well as the editor and reasoning behind these choices.

6.1 Input

The input of the program will be a CNF formula presented in the DIMACS file format. This file format is the standard for the SAT competitions and allows for simple parsing of the problem. The format consists of few elements to ensure the problem is presented cleanly, the first of which being comments. Comments can be marked using the character `'c'` followed by a single whitespace character:

```
c This is a comment line
```

Lines prefixed with this will be treated as a comment and thus will not be parsed as part of the problem. After the comments is a line telling the SAT

solver about the problem it is about to ingest, denoted using the character `'p'` followed by a single whitespace character.

```
p cnf 255 829
```

This line will include the type of problem, followed by the number of clauses then variables. Following the problem description line is the actual formula. This is presented by either a positive or negative integer, terminated with a `'0'`.

```
15 16 17 18 19 20 21 0
```

Each line corresponds to a clause formatted in the defined problem type. And each integer represents a unique variable. Using this format we can create consistent problem files which our solver will be able to ingest and process easily. An example of a problem formatted in this format is presented below:

```
c pigeonhole 2 1
c label:unsatisfiable
c label:easy
p cnf 2 3
1 0
2 0
-1 -2 0
```

Figure 2: Example format of a DIMACS file.

This input will be read line by line by the solvers parser, recording the key information that it needs. The program will then fill all the relevant data structures that it needs and proceed to processing them.

6.2 Output

The output of our program is just as important as the input. After all, we want to know how the solver has gotten to the solution it has ended at. Much like the input, we will be following a semi-standard format that is outlined at each SAT competition. Using this format will allow us to be consistent with how most SAT solvers output their solutions.

The format of the output is somewhat similar to the DIMACS format. In that each line is prefixed with a single character denoting its purpose, followed by a single whitespace character. The main difference here being that there is no particular order in which the lines must appear. There are three distinct types of lines in this format:

- `'c'`: Denotes a comment line in which the program can output any auxiliary information.

- `'s'`: Denotes the line where the solution is presented. This must be either 'satisfiable', 'unsatisfiable', or 'unknown'. Only one instance of this line is allowed.
- `'v'`: Denotes the values of the solution if one is found.

The resulting output should combine all three of these types to create a comprehensive result to the user. This includes the model that the solver has settled on if the solution is satisfiable. This will allow for the checking of the answer to ensure its correctness.

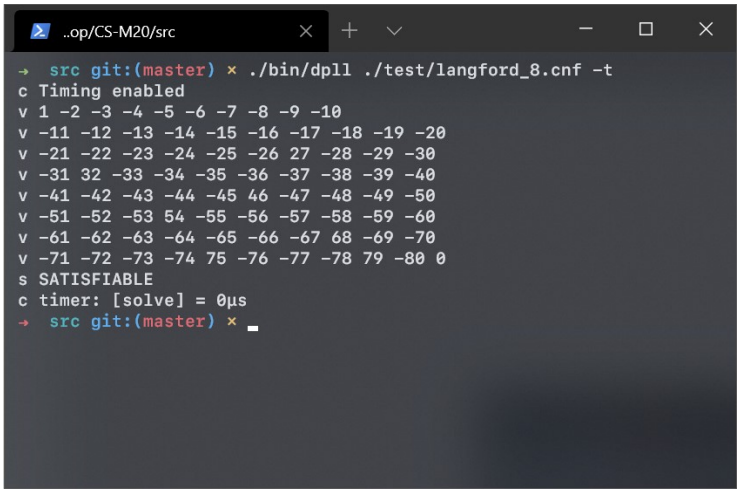

Figure 3: Output format.

6.3 Tool-chain

The following is a quick overview of the tool-chain that will be used for this project. As these aren't the meat of what the project is about, we shall only spend a short time glazing over the available solutions and those we picked.

6.3.1 Compiler & Libraries

There are multiple compilers available for C++ dependant on the platform. Generally for each of the three most common platforms (Windows, Linux, and MacOS) there are reliable and well tested compiler suites available. The GNU Compiler Collection, or GCC, is considered one of the standard compiler suites for C and C++. GCC has implementations for each of the aforementioned platforms via ports. The Windows implementation in particular is included in a few software development environments that are freely available; MinGW, MSYS2, and Cygwin.

As for Linux, the GCC suite can be installed natively typically using the preferred distro's package manager. For this project the code will be compiled on a Debian based system using the GCC suite. Linux provides a simpler method of developing using C / C+ + and streamlines the time from development to testing. Alongside this, the included GDB debugger allows for a simple and lightweight debugging solution. We will also be making use of bash scripts to invoke the software and provide flags to control options, something that can be a little more challenging on a Windows machine.

No development will be carried out on a Macintosh as we simply do not have access to one. Typically these types of software are compiled and run using a Linux system, and thus it would make the most sense to follow the given norm.

6.3.2 Editor & Environment

There are a huge swathe of IDE's and editors available for use and as for which one to use, there isn't a definitive answer. We thought that it would be best to just stick with what we know best, and for this instance that would be Visual Studio Code. Although other IDE's are available that can take a lot of the work out of managing a project. It is not really necessary to use one here. This project will not contain a huge amount of code or files, and general debugging and profiling does not need to be done using bloated IDE's. Visual Studio Code contains a massive inventory of extensions available to give an experience just as rich as an IDE, but without the cumbersome nature of one. As for this project we only need a single extension to get the required development environment:

- C / C++

It is also important to mention any other environment software aside from the previously mentioned. We are going to be using the *time* command to accurately measure the time metrics of the software from start to completion. This command is available through bash and is invoked at the same time as the program itself. An example of the way in which we would use the time command is as such:

```
$ time make bin/dpll
```

Figure 4: Example usage of the UNIX 'time' command.

This command will invoke the 'make' binary to compile our program, called dpll. The output of this command will include both the output of the make command, as well as the time it took formatted to our previous rules:

7 Testing

As with all software, one of the most important parts of the development lifecycle is testing. For this project we must ensure that it not only works without fault, but also returns the correct results that we are looking for. In order to test our

```
real: 4.70s
user: 2.42s
 sys: 0.45s
```

Figure 5: Example output of the UNIX 'time' command.

software we shall generate a plethora of testing criteria to test against. Each of these criteria will have an expected outcome. We will then compare the expected outcome to the actual outcome and record whether they are identical. Although this is a simple technique, it is hard to apply automated testing to our program as it is so niche.

As for ensuring the program returns correct results, the best way that we will be able to check is to use well known and reputable solvers to generate results and compare the two, or to verify the results using a verification library. This can also be added to the testing criteria and formatted in a table if preferred. The following will be the details of the testing strategy alongside the results.

7.1 Software Testing

There are a vast array of testing tools freely available to use for most software projects. These have been created from the need for automated testing due to programs changing rapidly. In this case, we do not need to deploy any advanced testing techniques or tools. As we have quite rigid requirements and a small codebase, keeping track of changes is somewhat trivial. Much like discussed in the initial sections of this paper, this is similar to the reasoning behind our choice of methodology.

7.1.1 Unit Testing

Unit testing is one of the most common types developers can use. This type of testing is where individual components or sections of the program are individually tested against a given criteria. Although there are multiple ways in which to achieve this, we do not need to overcomplicate this by writing hundreds of edge case tests. More importantly for this project, we only need to ensure that the program both compiles successfully, and produces the correct result. Unit testing may not be exactly translatable to our program as there isn't so much individual components. But we can use the idea instead to test individual files.

7.2 Checking For Correct Solutions

Progressing from the program testing, we must also ensure the solver is returning correct results. This can also lead us to any bugs that may be present within the program. Using common SAT problems presented in the DIMACS format

we can compare the results from our program to results from another solver. For statistical reasons we should ensure that each problem set is ran at least 10 times on each solver. To further back up the testing results we could run the same problems against a third solver. This would allow for comparison between each solver as a fall-back. In this instance, the project that we found in the previous sections also included scripts to verify the results of each problem. We will be using this to ensure each result and its returned model are correct.

7.3 Monitoring Performance Metrics

Now that we are sure that we have an implementation that is working as intended, we can begin measuring its performance. We will be doing this against a set of SAT problems that increase in difficulty. In order to get results that are viable, we will be testing against each difficulty level (easy, medium, and hard) at least 10 times. We must also consider the fact that our solver will not always be able to solve a problem in what we would consider a 'reasonable' amount of time. Because of this, the test set script will implement a timeout that we can set depending on the difficulty. We feel the best way in which this can be utilised is by using a standard timeout of 60 seconds for each difficulty. Then depending on which problem files take longer than 60 seconds, and thus terminate automatically, will be noted down and further tested with increasing timeout periods. If the solver cannot produce a result after being given a much larger timeout period, we can deem this as a result in which the program cannot solve the problem instance that we have given it in a 'reasonable' amount of time, and thus not meeting *FREQ1*.

7.3.1 UNIX Time Command

On top of the above *timeout* functionality, we can also use the standard UNIX *time* command. This command is very simple to use and allows us to measure the time each individual problem takes. The *time* command returns three different results that correspond to three parameters; real, user, and sys. We previously showed in the *Editor & Environment* section that this command will be formatted to show each output below the other. We can then pipe this output to a file to permanently record the results. To automate this, the following command will be issued:

```
$ for ((n=0;n<10;n++)); do time ./script/test.sh -blook -dhard
            -t60s |& tee results_hard${n}.txt; done
```

Figure 6: Bash script to automate test runs.

Using this command will invoke our script to test the solver against a set of problems. By using the bash for loop, we can tell it to do this as many times as we wish, and using the redirect functionality we write the results to a new file. This neatly inserts the data we need including information about the given set,

which binary we are using, the name of each individual problem, the time each problem takes, and whether the solver times out on a given problem.

```
Testing binary bin/look
make: 'bin/look' is up to date.
Testing label:satisfiable, label:easy:
test/empty.cnf
real    0m0.010s
user    0m0.000s
sys     0m0.000s
✔

test/factoring_4.cnf
real    0m0.010s
user    0m0.000s
sys     0m0.016s
✔

test/factoring_6.cnf
real    0m0.010s
user    0m0.000s
sys     0m0.000s
✔
```

Figure 7: Example output to file after testing script.

Real corresponds to the 'wall clock' time that the program took. This refers to the actual time it took from invocation to completion whilst the CPU handles other processes. The 'user' time corresponds to the time the CPU spent on the program in User mode. Calls made in Kernel mode will not be reflected in this time. The 'sys' time corresponds to the time the CPU spent in Kernel mode when executing the program. These three metrics give us a good idea as to what the CPU spends the most time on when executing our program.

8 Results

After we are confident that the solver created from *Algorithm L* is working as intended, we can commit to measuring its performance against other solvers of a similar nature. For this, we modified an already included bash script to print out not just the output from the solver, but also the time each individual problem took. Like stated previously, this was done simply with the standard *time* command. Then by executing the script, we get the name of each problem, along with the real, user, and system time, as well as whether the solver managed to solve it within the timeout period we assign. Although the output format of this batch test could be much improved, it serves a simple purpose and means we do not have to individually time each problem instance.

After testing each instance at least 10 times, we can analyse the results and come to a conclusion on the efficacy of our solver. First off, we can quickly look at the results from the *easy* problem set. As you would expect, a solver of this caliber should be able to breeze by these simple problem instances. However, it is still worth checking that this is true as there could be an edge case to which the solver may struggle with. After collating and analysing the results for this set

we find that our solver manages to breeze through each of the problems. Taking a negligible amount of time to process each one, which can easily be attributed to the timing and logging functionality having to print to the `stdout`. All of the problems in this set returned the same values and any deviance here was in the range of being negligible:

```
real: 0.011s
user: 0.00s
sys: 0.016s
```

Figure 8: Most common result from 'easy' difficulty.

Second, we can look into the results of the *medium* problem set. Although this set may present itself to be more challenging to a simpler implementation of a SAT solver, it should not stress ours. But, as stated above, it is still worth gathering more data than necessary as it could provide valuable information. However, upon collation and analysis of the results, we can see that much like the 'easy' problem set, our solver breezed through each problem. With most being consistent across the board with the previous results. The upper bound for this set was found to be with the problem *langford_prime_10*, with the following results:

```
real: 0.652s
user: 0.609s
sys: 0.016s
```

Figure 9: Upper bound result from 'medium' difficulty.

Finally, we move onto the *hard* problem set. This set presents the most challenging formulas to our solver, and as expected there are some which take it to the limits. What we can see from the *hard* problem set is quite interesting. There are consistent timeouts across all 10 tests that show exactly the ones that our solver struggles with:

- Waerden-5-5-177.
- bf-2670-001.
- Cook-9-10.
- Fsnark-51.
- Fsnark-99.

- SSA2670-141.

This is still consistent when given extra time to figure these out, and as such means that we cannot satisfy our original *NFREQ1* with this solver for this specific problem set. This is of course perfectly fine, as these examples do present something very challenging to the solver.

9 Evaluation

The following will be an objective evaluation of the execution and results of this project after it had been completed. Each section will contain criticisms and ideas that could assist in future projects that may concern the same topic. It will also document things that went well, and will conclude with ideas for future work for this field that we thought may be of interest.

9.1 Accuracy

One of the most important things to get right when creating this solver was to ensure the accuracy of the results. This is, after all, the whole point of the solver in the first place. This can naturally be confirmed from our testing phase after the implementation. By using alternative FOSS solvers we can clearly see that the implementation of Knuth's *Algorithm L* works just as intended. And as we will get into in the next section, performance is more than adequate. Of course, to ensure that this implementation would be perfect across the whole range of potential problems it could face would take much longer. We would have to collate a huge amount of problems and perhaps even test against the data sets that would be given at the SAT competition itself. But for the intention of our project, and most importantly for the requirements we stated at the very beginning, this is more than enough for our needs.

9.2 Performance

The second most important metric was the performance. Like stated above, we can draw from the requirements that we crafted at the very start of the project. They proposed that the solver only has to be 'good enough', in the sense that it can solve most easy or medium problems within a reasonable amount of time. 'Reasonable' is based off of the metrics of similar solvers in terms of complexity. Of course, if we put this implementation against something like *CryptoMiniSat* there is no chance that we would be able to match its performance. What we could propose instead is taking the implementation of *Algorithm L*, and expanding on it by implementing experimental features, or taking inspiration from another solver such as *CryptoMiniSat* and implementing their ideas. This would of course take another project, and potentially much more time as the level of complexity increases.

9.3 Methodology

The methodology used for this project turned out to be perfectly suitable, as initially thought. Like stated at the start, the project goals and requirements were very rigid in theory, and in practice stayed consistent to that fact. We found no need to add extra features, especially considering the task at hand was challenging enough. And if any features were deemed interesting to add, that would be something for another project, as mentioned above. One thing to consider would be the possibility of starting another project with another person, in which case the methodology we used may not be as suitable and would have to be discussed in more detail.

9.4 Implementation

As for the implementation, we were fortunate to find that someone had already implemented all the algorithms from Knuth's booklet. As these project files were hosted on GitHub[12] and the license given was the *Unlicense License*, we could freely attain and modify this code for our own usage. This saved a huge amount of time, and after investigation, was found to have been implemented in the exact way that Knuth had proposed. Potential improvements were made to this existing codebase by bringing everything up to date with modern standards, and double checking that everything was working as intended. Of course, the resulting program that we were interested in was only that of *Algorithm L*. And as such, is the only included binary for this project. The other binaries were used for testing purposes. As they are general implementations of each 'style' of common algorithms, they proved to be a bonus when analysing the potential of each against *Algorithm L*.

10 Future Work

10.1 Pre-processing

One simple method that could greatly assist our solver is using a pre-processor. This will take the problem set and analyse it, and return a much simplified problem. This concept is applied across modern solvers already, and is considered a vital part of the SAT solving system.

10.2 Parallel Solving

A more interesting concept is to capitalise on modern processor architectures, and the cheap availability of multi-core CPU's. It is commonplace to find consumer CPU's with at least 4 cores and hyper-threading. By taking advantage of this we can try to divide up the problem set and apply a solver to each smaller instance running on an individual thread. As usual with parallelism, this would not necessarily lead to a linear performance increase. It would require some

research and engineering to determine the best way forward in order to achieve the desired results.

10.3 SIMD

Another interesting technique that could be applied to SAT solving is the use of SIMD instructions for data level parallelism. It has been shown in other applications that utilising SIMD instructions can quickly lead to almost linear performance improvements corresponding to each bit width of data processed. The most common SIMD instruction set is *SSE* (Streaming SIMD Extensions). Introduced in 1999 by Intel, it is now commonplace is all modern consumer x86 processors. If there was a way to truly implement SIMD into SAT solvers then there could be a massive potential for speedups in a range of applications. Even if the SIMD instructions were implemented in auxiliary areas of the solver itself, it is worth the time to investigate the potential.

11 Challenges

As with any project there will always be some challenges faced that skew the planned timeline. The following details the problems that specifically effected this project and what could potentially be done to avoid facing these in the future.

11.1 Software Based

Implementing software will always come with its own set of challenges. Many projects have failed in the past for trivial reasons associated with the software section not being fully understood. With this project, the main issue that we faced was with the programming language itself, as detailed below.

11.1.1 C++ Learning Curve

The programming language that was used for this project was deemed suitable based on the style of program to be created. SAT solvers rely on heavy and intricate data structures to achieve what they need in the fastest time possible. Using C++ seemed like a logical solution to this problem, as complex data structures can be created with ease. However, this does require some in-depth knowledge of the language and its features first. C++ is notorious for being a language with a steep learning curve, and given the time to learn it, ends up being one of the most powerful. The problem for this project was that prior in-depth knowledge of the language was not known. This was thought of in the initial design of the project, but combined with the issues of a maths heavy topic, proved to be more challenging that initially expected.

11.2 Project Based

As stated above, most projects will face some form of setback or turmoil within their timescale. This project was no different, and out of all the different forms of risks and setbacks that could have been encountered, one in particular stood out. However, it would not be the only one that we could blame for this project not turning out as successful as initially hoped. There were a few personal reasons which ended up causing much more damage to the time plan than initially thought. And on top of this, over-ambition ended up playing a much bigger role than anticipated. Because of this we feel it is obvious to deem this project more within the realm of failure rather than success. Time was not used optimally and factors out of our control proceeded to exacerbate that fact. The following is a short detailing of the factors involved.

11.2.1 Late Start

One of the first challenges that had effected this project was the late start. This was in part due to both personal obligations that could not be avoided. As well as in part due to the Coronavirus pandemic. Although this did not effect the overall project progressing, it did push the established time-frame beyond what had been originally planned. The domino effect of moving beyond the allowed time buffer meant that each stage of the project had to be pushed back. Thus in turn creeping the expected deadline much further back than anticipated.

11.2.2 Coronavirus

The second major issues encountered was something completely out of our control. The Coronavirus pandemic of 2020 provided a massive slew of issues for all academia. Specifically for this project it meant the general plan for the project, as well as the time plan, were both interrupted. The general lock-down for the United Kingdom resulted in confinement to the home. The availability of face-to-face meetings proved to be unfavourable, and potential disruptions in the home delayed planned work. On top of all of this, there were instances in which the virus ended up being closer than desired. In these cases, the stress of every day life sometimes meant that delays were inevitable.

11.2.3 Maths Heavy Topic

Another major issue encountered for this project was the lack of initial general mathematical background . This topic is a rather maths heavy topic, with complex algorithms that can confuse most people who do not have strong mathematical skills. This lead to this project being delayed in certain phases where more time had to be spent to understand the maths involved. Although in the end this turned out to be good in the sense of challenging, the over-ambition to assume the ability to understand of all relevant material turned against us. As such, in future projects there should be the advance knowledge of the required mathematical background before starting.

12 Conclusion

Concluding this project, there is little that we have gained in the grand scheme of SAT. Although we looked through the history and investigated a few interesting SAT solvers, the actual implementation ended up being trivial due to already existing material. There were no improvements that could have been added within the given time-frame that would have made a difference. On top of this, the project itself was hit with a large variety of problems which lead to too many time delays and preventing any addition of unplanned features. But we have seen that there are promising ideas that can be further worked on. And although academia has been marred by the unforeseen global pandemic, there remains a constant hum of research into improving existing solutions for solving SAT.

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.

[3] D. Knuth, *The Art of Computer Programming*. 2015.

[4] R. Karp, "Reducibility amond combinatorial problems," *Complexity of Computer Computations*, pp. 85–103, 1972.

[5] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, p. 201–215, July 1960.

[6] T. Balyo, P. Sanders, and C. Sinz, "Hordesat: A massively parallel portfolio sat solver," 2015.

[7] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a sat solver from single-bit supervision," 2019.

[8] N. Eén and N. Sörensson, "An extensible sat-solver," in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), (Berlin, Heidelberg), pp. 502–518, Springer Berlin Heidelberg, 2004.

[9] M. Soos, K. Nohl, and C. Castelluccia, "Extending sat solvers to cryptographic problems," in *Theory and Applications of Satisfiability Testing - SAT 2009* (O. Kullmann, ed.), (Berlin, Heidelberg), pp. 244–257, Springer Berlin Heidelberg, 2009.

[10] 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.

[11] M. Heule, M. Dufour, J. van Zwieten, and H. van Maaren, "March_eq: Implementing additional reasoning into an efficient look-ahead sat solver," in *Theory and Applications of Satisfiability Testing* (H. H. Hoos and D. G. Mitchell, eds.), (Berlin, Heidelberg), pp. 345–359, Springer Berlin Heidelberg, 2005.

[12] A. Windsor, "sat." `https://github.com/aaw`, 2020.