Implementing SAT Algorithms in Software

Callum Donovan
1915769
September 1, 2020
Swansea University

1 Abstract

Modern SAT solvers are able to process large SAT data sets quicker than ever before. The research community have been able to assist in the development of these algorithms since the early 60's, and work is still being done today to improve the performance of these tools. This paper will be exploring the concepts of a SAT solving algorithm, and how we can translate them into working software tools using modern programming languages. The selected algorithm is taken from Donald Knuths epic "The Art of Programming", specifically his newest addition "Facsicle 6, satisfiability".

Algorithm D is a simple implementation of a cyclic DPLL SAT solver that utilises specific data structures and backtracking to achieve modest performance in most use cases. We have implemented this in C++17, and tested it against a small suite of regular SAT problems presented in the DIMACS format.

Contents

2 Introduction

In modern computer science, there exist a large set of computational problems that are deemed to be impossible to compute in a reasonable amount of time. This set of problems, named NP, forms the basis of what a large majority of computer science researchers have attempted to solve. From the initial scientific labelling of the P and NP set of computational problems in the early 20th century[1], there has been significant time and effort put into discovering the fastest methods in which we can compute these problems. One such instance of these problems would be the Boolean Satisfiability Problem.

This problem was first proved to be NP-Complete in 1971 by Stephen Cook and Leonid Levin, Cook being the first to publish his paper in 1971[2]. Levin further improved on Cook's paper by including search problems with the Boolean Satisfiability Problem in his paper published in the USSR in 1973[3]. Since those initial findings, the world of NP-Complete problems has expanded significantly and to this day still proves that there is an ever growing demand for the advantages that improved SAT solving algorithms bring.

Donald Knuth, a well-known computer science researcher from the United States, recently published another volume of his epic work, "The Art of Computer Programming". In this volume, Knuth described a set of algorithms that can be used to solve SAT formulas. Our goal is to explore one of Knuth's algorithms and translate it into a simple piece of software using C++. Thus learning more about the ways in which SAT algorithms can be implemented in software, as well as the methods which can be used to translate it into a program that others can use.

A secondary goal of this project is to explore the concepts that make SAT algorithms efficient when implemented in software. Research that already focuses on this will be analysed and implemented into the outcome of this project, and further possible improvements will be found that could be used within the industry.

2.1 Motivation

The motivation for this project stems from the desire to constantly iterate and improve on existing solutions so that we can do things faster and more efficiently. The area of SAT solving is extremely important as it provides key solutions to areas of science that rely on the ability to process data sets quickly. SAT solvers are not only useful in this area of computer science, but are also used across the field of science in its entirety. SAT solvers can be used to process problems from a multitude of sources, mainly other problems that are reduced from their respective sets to the CNF SAT formula. Although improvements in this field will do more for NP-Complete problems than any other sets, they will still provide benefits to problems that are contained in the NP-Hard set. Examples of NP-Hard problems include protein folding, cryptography, scheduling and many more.

Although there are already existing solutions that are very efficient and more performant than ever before, there is always room for improvement. This project aims to explore these concepts in which existing solutions can be built upon, and new solutions can be created. Donald Knuth has proposed a set of algorithms that should be able to efficiently process SAT formulas when implemented in software. We will be implementing one of these algorithms to study its efficiency when compared to well-known existing solutions. If one of these solutions proceeds to be substantially efficient in processing the aforementioned SAT formulas, there is a potential for great scientific advancement in a wide range of fields, as stated previously.

2.2 Aims

To reduce the above sections into a more concise format that explains the exact outcomes desired from this project, our aims are as follows:

- To research and explore the concepts of SAT solving algorithms, detailing the best possible current methods.
- To analyse a SAT solving algorithm as described by Donald Knuth in his book, ”The Art of Computer Programming”.
- To create a C++ program that implements the aforementioned algorithm effectively.
- To test the C++ program to ensure that it can successfully and correctly process CNF SAT formulas.

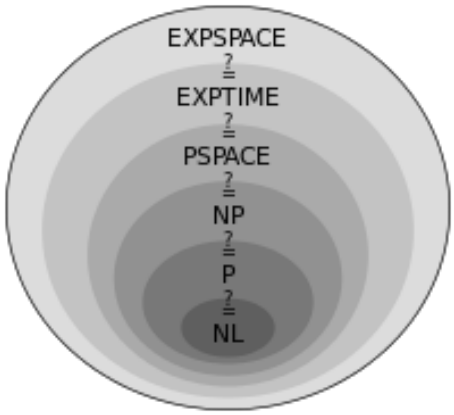
3 Background Research

In order to fully understand the project that we are undertaking, we must review and analyse existing literature. This literature will form the basis of the project and will be a guide as to what to expect. There are multiple different components to consider when talking about both SAT and NP problems, the former of which is contained within the latter. Existing research has already led us to create some of the best SAT solvers that we could have ever dreamed of, but we cannot settle for what we already have because it may seem good enough. Science is all about pushing the boundaries of what we already know, and applying the unknown to see what may come of it.

3.1 Computational Complexity

To help with understanding the following topics, we will quickly explain the basic concepts of computational complexity. Computational complexity theory is an area of study that focuses on classifying computational problems according to their difficulty. These classes are then grouped together by relation to form a scale of computational complexity. Computational problems are problems in which will be solved by a computer via an algorithm. The most important classes of which are covered in this paper as a foundation for the project that is to be carried out. A simple representation of the different classes of complexity theory can be shown like in figure 1:

Figure 1: Complexity classes visualised [4]



3.2 P and NP

Before we can start on the main topic of the project we must first understand the terms that will be used throughout the remainder of this section. There exists a major unsolved question in computer science, P versus NP. Initially just part of scientific theory within

the field of computational complexity, it became one of the seven Millennium Prize Problems put forward by the Clay Mathematics Institute in 2000.

As we will investigate below, Stephen Cook and Leonid Levin hardened the existence of the P and NP problem independent of each other in the early 1970's. A simple way to think of the classes P and NP is by their time to find a solution. Problems that reside in the P class can have solutions found within a reasonable amount of time, or "quickly". Whereas problems that reside in NP cannot have their solutions found within a reasonable amount of time. But if given the solution to a problem, we can verify the answer quickly. Put simply, P problem solutions are easy to find, NP problem solutions are easy to check.

3.3 Cook-Levin Theorem

To kickstart our development of understanding SAT solvers and their place in the scientific landscape, we must cast our thoughts back to their initial conception in the early 20th century. Although there had been talk of the concepts of P and NP long before the first concrete scientific evidence of it, it was not until the publishing of the Cook-Levin theorem in 1971 and 1973 that we would truly expand upon them. The Cook-Levin theorem is the combination of two papers published at similar times, albeit in separate countries, that described and proved the same ideas. Stephen Cook published his paper "The Complexity of Theorem Proving Procedures" in the US in 1971[2]. His paper was the first to prove that the Boolean Satisfiability Problem was indeed, NP-Complete. Interest in Cook's theory was further increased by Richard Karp's 1972 paper "Reducibility among Combinatorial Problems"[5]. Karp expanded upon Cook's initial paper by including a further 20 NP complete problems, all of which could be reduced to CNF SAT.

Later, in 1975, the scientific interest into P and NP was further increased from the work of Theodore P. Baker, John Gill and Robert Solovay. Their paper "Relativizations of the P = NP Question" proved that theoretically, solving NP problems in an Oracle machine model requires exponential time to complete[6]. An Oracle machine being an abstract, Turing complete machine that is used to study decision problems in complexity theory.

A similar result to both Baker, Gill, Solovay and Cook was found in the USSR by Leonid Levin in his 1973 paper "Universal Search Problems"[3]. The main difference between the two being the inclusion of search problems by Levin in his paper. Search problems differ from the problems published by Cook and Karp as they required finding the solution of the problem, as opposed to just confirming it's existence.

3.4 CNF SAT

To further understand the concept of SAT solvers and what they aim to achieve, we must understand both SAT and its most common form, CNF SAT. The Boolean Satisfiability Problem, often referred to as SAT, is the problem of deciding whether a formula in

boolean logic is satisfiable. A formula can be deemed satisfiable when at least one interpretation of it leads to the entire formula evaluating as true. We can make this interpretation by assigning true or false values to the logical variables contained within the formula.

When we refer to SAT formulas, we are typically referring to its most common variation CNF-SAT. CNF, or conjunctive normal form, is where the entire formula is a conjunction of clauses. Where each clause is a disjunction of literals. As stated previously, Stephen Cook's "The Complexity of Theorem-Proving Procedures" proves that a Boolean Satisfiability Problem formula, expressed in conjunctive normal form, was indeed the first NP-Complete problem[2]. An example of a SAT formula in conjunctive normal form is as follows:

$$(x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg x \vee y \vee z)$$
$$\wedge(\neg x \vee y \vee \neg z) \wedge (\neg x \vee \neg y \vee z) \wedge (\neg x \vee \neg y \vee \neg z)$$

This particular formula would be called 3-SAT, as each clause contains at most 3 literals. This is one of Karp's 21 NP-Complete problems as mentioned above and is a good starting point for proving that other problems are NP-Complete via polynomial-time reduction. The main advantage of using 3-SAT formulas when concerning SAT solvers, is that we want to be able to put our most challenging formulas in their simplest possible form. Using 3-SAT allows us to analyze and generate better algorithms easier than other formats.

3.5 How Do SAT Solvers Work?

Typically the algorithms that existed to solve SAT problems existed mainly to model worst-case scenarios for research. However, since the turn of the millennium there has been a steady increase in amount of effective and efficient algorithms that can be applied to a large range of industrial use cases. These algorithms focused on performance and scalability, and their development aided in the research of many industry centered problems. One such instance of these applications would be their inclusion in the toolbox of electronic design automation, or EDA.

These SAT solvers are based on developments from well established algorithms and are applied to software libraries ready for production use. Modern SAT solvers have allowed for dramatic improvements to the speed of EDA, and are able to solve problems containing many thousands of clauses[7].

3.5.1 Davis-Putnam-Logemann-Loveland (DPLL)

The Davis-Putnam algorithm formed the basis of what modern SAT solvers still use today. In their 1960 paper "A Computing Procedure for Quantification Theory"[8],

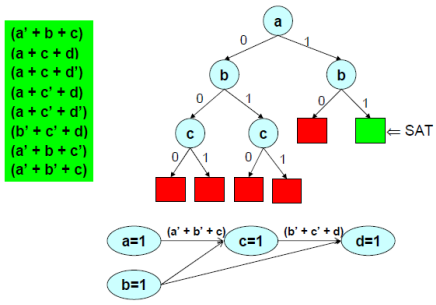
Martin Davis and Hilary Putnam propose an algorithm which includes steps to find the satisfiability of propositional ground instances.

A further refinement using this algorithm was developed in 1962 by Martin Davis, George Logemann and Donald W. Loveland. The DPLL algorithm built upon the original by using backtracking for deciding the satisfiability of propositional logic formulae in conjunctive normal form[9]. As we have established previously, SAT solving has an array of benefits when applied to a multitude of different fields. And the DPLL algorithm is still used in modern SAT solvers today, such as Chaff, GRASP and MiniSAT.

The algorithm works using backtracking. It firsts chooses a literal, assigns a truth value to it, then simplifies the formula. It then recursively checks if the simplified formula is satisfiable. If it is, then we deem the formula satisfiable, but if it is not we go back and choose the opposite truth literal, simplify then loop recursively through it again until it is satisfiable.

Modern algorithms are almost always based upon the fundamental operations of this original algorithm. Many of them are purely extensions that aim to improve performance in specific scenarios or industrial applications. Even though they may have their own names, they are still considered part of the DPLL algorithm group and are deemed as some of the best around.

Figure 2: DPLL final implication graph. [10]



3.5.2 Other Algorithms

Although most modern SAT solving algorithms are based off of the original DPLL algorithm, there are many who developed their own algorithms. These attempt to attack the problem from different angles and employ methods such as local search, or randomization. Local search is based on improving the assignment of variables iteratively until the problem is deemed satisfied. However, local search is considered an incomplete method, as opposed to DPLL which is considered complete. Incomplete algorithms attempt to find a solution the formula, but cannot determine whether the formula is unsatisfiable. An example of a local search based algorithm would be WalkSAT or GSAT. Randomized algorithms that use a heuristics based approach include the PPSZ algo-

rithm. These algorithms have been proven to have excellent runtime when compared to other algorithms when given a 3-SAT or k-SAT problem[11].

3.5.3 Parallel SAT Solving

Since the initial invention of the SAT solving algorithm, there has been leaps and bounds of advancement in computing technology. The most important of which being the addition of multiple cores within a single CPU. SAT solvers would typically run on a single physical processor, which limited the processing capacity to a single instance of the solver. If the performance of the algorithm is of optimal and consistent speed, the time in which a known problem may take to solve will depend on the speed of the single core that is processing it. Take for instance the DPLL algorithm, it uses recursion to decide whether a formula is satisfiable or not. The speed at which this recursion can run is related to the general performance of the single processor it is on. Before multi-core systems were invented, the main goal was to ensure the algorithm could run as fast as possible, using as little resources as possible. CPU's and memory were expensive, and thus these algorithms had to make do with what they had. In modern times, we've been able to overcome the majority of the cost factor in regards to both memory and processing power. This has led to the creation of SAT solvers that take advantage of multi-core systems. These solvers can be grouped into three categories: parallel local search, portfolio and divide-and-conquer. Each take advantage of abundant resources in their own ways.

Portfolio SAT solvers are based on the idea that no SAT solver can be deemed a jack-of-all-trades solution. Each type of SAT solver has both strengths and weaknesses that allow it to be good for one type of formula, but bad for another. This has led to an approach where the same formula is processed using a collection of different SAT solvers on each processing core, hence the name portfolio. This approach can use both different types of SAT solvers, or different configurations of the same SAT solver, to find the best solution in the most optimal time. Once one of the SAT solvers reports that they have found the solution and the formula is deemed satisfiable, the portfolio halts the process and returns the instance which solved it[12].

Divide-and-conquer SAT solvers work in the opposite method to Portfolio SAT solvers. Instead of applying multiple different solvers against a formula, it is split up and given to each processing core to independently figure out. Although this seems like the most logical method of introducing multi-core processing into a problem, it comes with multiple hurdles that have to be offset. The main issue being that each search space will vary dramatically in the amount of time it takes to process. This can lead to multiple problems and load balancing becomes incredibly complex to achieve[12]. A modern approach to combat these problem includes using two types of algorithms to deal with the formula in two stages. First the algorithm which is better at small but hard problems will be used to split the formula into many small formulas equivalent to the original formula. Each smaller formula will then be given to a CDCL algorithm to process independently of the original algorithm. Due to the nature of the smaller formulas still being equivalent to the original, if one of the smaller formulas

evaluates to TRUE, the original formula is deemed satisfiable[13].

Finally, local search algorithms are the hardest to parallelize. One method used includes creating an instance of a local search algorithm on one processing core. The initial instance will begin processing the formula, and when it decides to restart its search it will produce a configuration that can be passed to a new instance of the algorithm on a new processing core. This configuration will build upon the results from the initial instance to increase the chances of finding a solution. This can happen repeatedly over multiple processing cores until the algorithm produces a result[14].

With the development of even faster processors with more cores becoming accessible to the wider public, we believe that research into parallel algorithms would be the best area to focus on. High performance and efficient processors will become even cheaper in the following years, and the ability to be able to study the behaviour of these algorithms when applied to many processing cores is becoming easier and easier. Even more interestingly, the most recent developments in quantum computing have allowed more research to be put to these phenomenal devices. And there is possibility for the technology of quantum computing to become more common in the coming years, allowing for them to be used for more trivial operations such as SAT solving.

3.6 Existing Solutions

As stated previously, there are multiple SAT solvers publicly available in the form of libraries. They allow anyone to download and use them for any projects or research they are currently working on. Most of these are well-known in the area of SAT solvers and are already proven to be of excellent performance when given large formulas. The following are a collection of some of the most renowned libraries used today in the industry.

Existing libraries typically use the semi standard DIMAC format to ingest formulas. This format consists of three basic rules:

- A comment line. Any line beginning with ”c” will be seen as a comment line.
- A summary line. This line will tell the library about the problem that it is to process. These lines start with a ”p”, followed by the type of problem ”cnf”, followed by the number of variables and clauses. Depending on the parser used to process these files, some may require this line to be first.
- A clause line. This line tells the algorithm about each clause contained in the formula. Each clause must be described using numbers separated with a single space. The final character denoting the end of the clause is always a 0.

This format allows for quick conversion from mathematical format to an ingestible file, and is useful for testing a library with multiple types of formulas. An example of a clause:

$$(x \vee y \vee z) \wedge (x \vee y \vee \neg z) \wedge (x \vee \neg y \vee z)$$

converted into the DIMACS format:

```
c A formula in DIMACS format
p cnf 3 3
1 2 3 0
1 2 -3 0
1 -2 3 0
```

3.6.1 MiniSat

The MiniSAT library is a relatively small and fast SAT solver created by Niklas Eén and Niklas Sörensson in 2003. They initially created this library as a method to get inexperienced people into the world of SAT solving. Its initial version contained a slim 600 lines of code and implemented all of the modern features that were popular at the time. The resulting library was a conflict-driven clause learning state of the art SAT solver. The authors further modified the library to compete in more categories in the 2005 SAT competition. Even though the extension was considered a ”hack” in the eyes of the authors, it proceeded to win silver in a few categories and proved to be formidable in categories where it wasn't expected to perform well.

The final version of MiniSAT is hosted on GitHub for public use, and is freely available for anyone to use and modify to any extent. There are modern forks of MiniSAT that are still competing in SAT solver competitions today, thus proving that a well implemented library based on a modern extension of the original DPLL algorithm can perform just as well as proprietary industrial solutions.

MiniSAT includes a simple C++ interface, using three vocabulary types:

- `Minisat::Var` - Used to represent variables
- `Minisat::Solver` - Provides the core functionality of the solver
- `Minisat::Lit` - Used to represent literals in the formula

These interfaces can be used within a C++ program to create a basic implementation that can be passed clauses. In order to pass the interface clauses to process, we must include one of MiniSAT's utility types: `Minisat::vec<T>`. Using these types we can easily create a C++ script that processes a logical SAT formula:

```
#include <iostream>
#include <minisat/core/Solver.h>

int main() {
    using Minisat::lbool;
    using Minisat::mkLit;
```

```
        // Create solver instance
        Minisat::Solver solver;

        // Create three variables
        auto a = solver.newVar();
        auto b = solver.newVar();
        auto c = solver.newVar();

        // Create three clauses and their included literals
        solver.addClause(mkLit(a), mkLit(b), mkLit(c));
        solver.addClause(mkLit(a), mkLit(b), -mkLit(c));
        solver.addClause(mkLit(a), -mkLit(b), mkLit(c));

        // Begin solving, return the model if satisfiable
        auto sat = solver.solve();
        if (sat) {
            std::clog << "SATISFIABLE\n MODEL FOUND: \n";
            std::clog << "A = " << (solver.modelValue(a) == l_True)
<< '\n';
            std::clog << "B = " << (solver.modelValue(b) == l_True)
<< '\n';
            std::clog << "C = " << (solver.modelValue(c) == l_True)
<< '\n';
        } else {
            std::clog << "UNSATISFIABLE\n";
            return 1;
        }
    }
```

Further work is required in order to write a script that can solve SAT formulas in CNF, but is still somewhat simple from the use of the MiniSAT library and its interface.

4 Project Description

Expanding further on the aims listed previously, this section will focus on the exact details of the project that we will be undertaking. To begin, the projects main deliverable will be a working piece of software written in C++. This software should meet the aims of the project that are listed, and should also coincide with the research undertaken. This project sits between both research and practical, but the end result should primarily be the software that is to be created. The following will be a guide as to the exact requirements of the software and what we are expecting to create based on these guidelines. Alongside this, we will list the tools and methods that are to be used whilst creating this software.

4.1 Functional Requirements

The following section describes the expected functional requirements for the project we will be undertaking. There aren't that many functional requirements that are to be expected when undertaking a project like this, as the resulting software is only going to be a relatively simple implementation when compared to larger software projects.

4.1.1 FREQ1

The resulting software should be able to process an input CNF formula in the semi-standard DIMAC format, as described previously.

4.1.2 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".

4.2 Non-Functional Requirements

The following section describes the non-functional requirements for the project. These will be auxillary requirements that aren't necessarily essential for the success of this project. Much like the functional requirements, this project is relatively small compared to most and doesn't need to have many features or requirements in general.

4.2.1 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”.

4.2.2 NFREQ1

The resulting software should be able to process different forms of SAT formula other than traditional CNF SAT.

5 Project Plan

The following section will outline the general plan of the project including methodologies, estimated time plan, tools and risks.

5.1 Methodology

There are a large set of software development methodologies available to follow, all of which have their own place in each type of project. Specifically for this project, it is important to note that the resulting software will not have to adhere to the typical iterative systems that larger software systems do. This project will be a relatively small, proof of concept that is more focused on the methods in which it is implemented software-wise. Due to this, we believe that it is not required to use one of the big go-to methodologies that most software projects will use.

5.1.1 Chosen Methodology

The methodology that we have chosen for this project is the Waterfall methodology. Although at first this may seem like a strange choice for modern software development, we believe that the size and scope of the project does not require the need for a larger, team focused methodology such as SCRUM. We find that the Waterfall method is a tried and trusted method of developing software based on concrete requirements with very small teams, or even solo developers.

5.1.2 Alternative Methodologies

There are many different types of alternative methodologies that are used widely in todays software landscape. Some of which are significantly more popular than others due to a number of factors. One example of which would be the Agile software methodology. Within the Agile methodology, multiple different sub-methodologies have been developed which encompass the basic principles of Agile, whilst adding their own

twists. The basic Agile methodology includes many different techniques to enable a fast, iterative approach to creating software in a team.

Using one sub-methodology as an example, extreme programming looks more towards solving the issues with changing client requirements along the development cycle of a software product. In extreme programming, frequent releases over shorter development cycles allow for the adjustment of requirements to be implemented within the next cycle. This leads to a generally more satisfying outcome for the client, but we believe that it increases the chance of both feature-creep and incomplete software. We find that this methodology does not suit the very basic requirements and development plan of the project we are undertaking, and as such was not chosen to be the methodology that we will follow.

5.2 Time Plan

The time plan for this project will span over the next few months. As a simple form of representing the time plant, a Gantt chart has been created and is displayed on the following page.

5.2.1 Research + Discovery

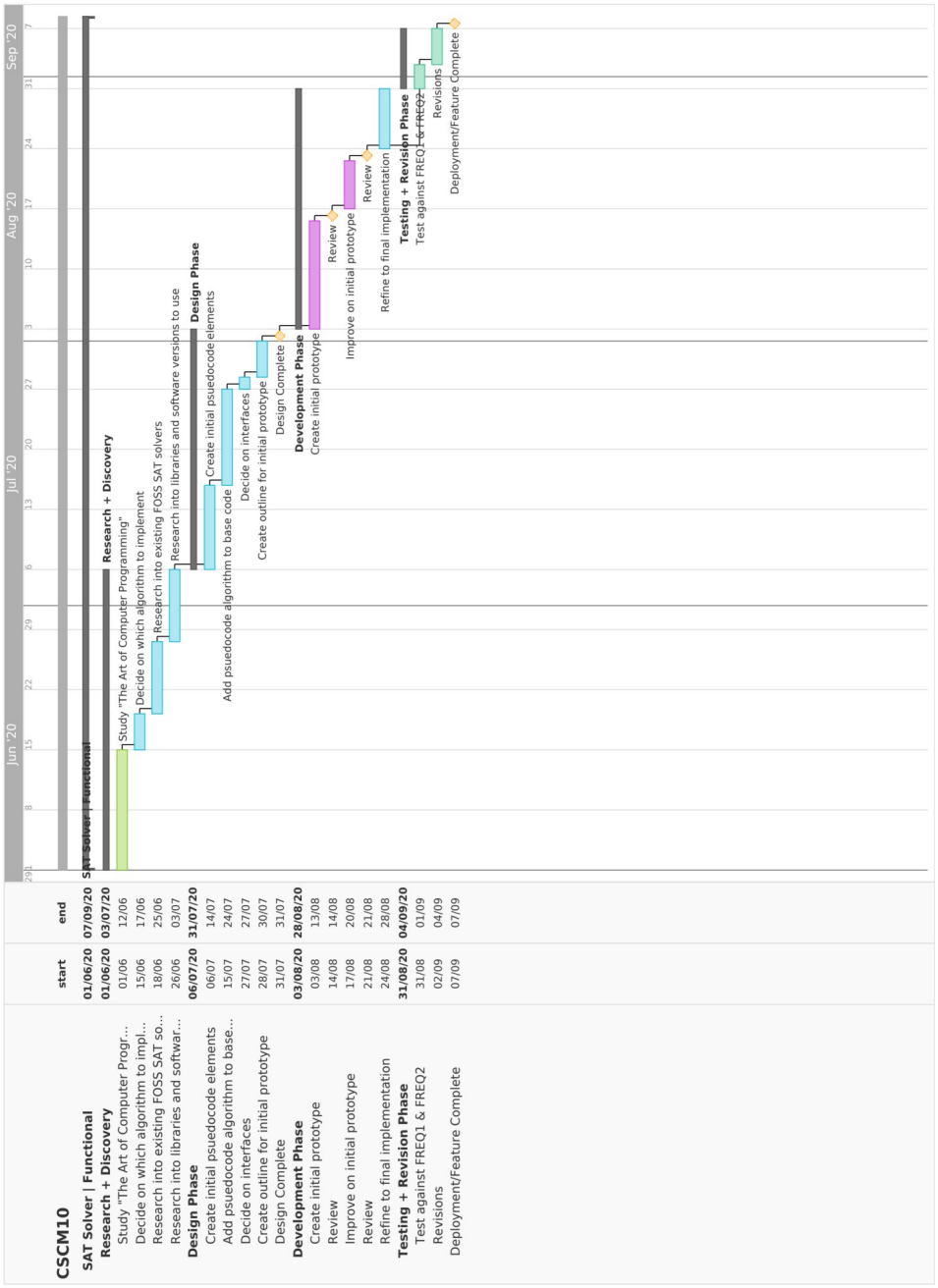
This is the initial development phase of the time plan. It encompasses the

- **Study "The Art of Programming"** - In this task, we will be investigating the Donald Knuth book "The Art of Programming". This will be used to form the basis of our understanding of the algorithms that he has created. One of which will be used for this project and turned into the C++ program.
- **Decide on which algorithm to implement** - Here will will be taking a few days to finalize the algorithm that we want to implement the program we are creating for this project.
- **Research into existing FOSS SAT solvers** - At this point we will find and evaluate the existing solutions that are free and open source. This is an important step to understand the generic layout of most implementations of SAT solvers in software, and should guide us in a general direction that we want to go.
- **Research into existing libraries and software versions to use** - This sounds similar to the previous task, but is instead a further research into the potential libraries and software that we will use to build the final program. An example of this would be the difference between choosing C++ 14 or C++ 17.

5.2.2 Design Phase

In this phase, we will be developing the initial design elements for our program. This will include developing basic psuedocode as a basis for further development using the

Figure 3: Time plan as formatted as a Gantt chart



programming language of our choice. This phase is important to follow, as it will save lots of time and effort that otherwise might be wasted in the development phases by dealing with the layout of the code.

- **Create initial psuedocode elements** - Here we will be creating the general structure of the program via the use of psuedocode. It will allow the boilerplate code for the final program to be added in quickly without much change.
- **Add psuedocode algorithm to base code** - This will expand on the previous task by adding the algorithm in psuedocode to the basic structure we created. This will allow us to fine tune the final overall layout of the program before we being implementing it in real code.
- **Decide on interfaces** - This step will decide what interfaces we will be needing to make public for the software to function as we intend it to. Much like already existing libraries that include interfaces to interact with when implementing them into a program.
- **Create outline for initial prototype** - Finally, we will go into further detail using the initial structure and interface decisions to create a final general outline that we will be adhering to when creating the software using the chosen programming language.
- **Design complete** - Design work is finalized and collated to be used for initial development phase.

5.2.3 Development Phase

At this phase we will be starting the initial implementation of the program based on the material and choices we made in the design phase. This is where the program will begin to get its initial functionality and working prototypes will be made.

- **Create initial prototype** - This is the first step in the development phase and will lay the groundwork for the program using the initial designs. Here the boilerplate code and interfaces will be created.
- **Review** - A small review step to ensure that all the design guidelines were followed
- **Improve on initial prototype** - Improvements are made the boilerplate and steps are made towards implementing the initial part of the algorithm into actual code. As most of the structure has already been created in the design phase, this should be relatively simple.
- **Review** - Another small review to ensure the designs were followed as intended.
- **Refine to final implementation** - Here we will iteratively refine to the final version of the program to ensure it meets the initial requirements of the project. This will be the final development step before we move to full testing.

5.2.4 Testing + Revision Phase

This is the final stage in our development plan and one of the most important. Here we will put the program through its paces to ensure that it meets the original functional requirements. If time permitted in the development phase, the non-functional requirements will also be tested.

- **Test against FREQ1 & FREQ2** - This task will test the program to its full potential against the original criteria as described by **FREQ1** and **FREQ2**.
- **Revisions** - Any problems or issues found from the full testing will be ironed out and tested again to ensure the program is working as intended.

At this point, the project should be completed and any relevant material should be collated and added to the submission for review.

5.3 Tools

The tool-set that is required for this project is somewhat minimal. Each tool will serve a very particular purpose and no unnecessary tools will be used for the sake of including more. They are as follows:

5.3.1 Code Editor

There are a whole plethora of code editors and IDE's that can be used for this project, but in order to keep the project simple and portable, we will be using the Visual Studio Code editor. This editor is lightweight and platform agnostic, allowing for it to be present on any system that we wish to use. Alongside this, it has a huge extension library that allows for development in almost any language a simple task.

The bare minimum extensions that will be used are as follows:

- C / C++
- C++ Intellisense
- C / C++ Compile Run

These extensions will allow for development on any platform, and the last extension will allow for compilation, running and debug right from the editor itself. This will allow for much faster prototyping and debugging when developing.

5.3.2 Compiler

A compiler is the next important step to consider, especially for this project. Each compiler will vary the types of native libraries that are available when developing, and the

platform that we are developing on will also guide the choice of compiler. For this project, we have chosen to use the MinGW compiler for Windows. It works out of the box without any configuration and allows for development in any environment with Windows. It has been proven to be one of the best compilers available and has excellent support and documentation, something very valuable when developing prototype software.

5.4 Risk Analysis

As this project is quite self-contained and does not require much to get going, the levels of risk are relatively low. We can still find some risks that may pose a threat to the project progressing if we were to encounter them and how we could mitigate them. These are displayed in the risk analysis table below.

- L → Likelyhood
- S → Severity
- D → Discoverability

Table 1: A risk table showing weighting and mitigation.

ID	Title	L	S	D	Total	Mitigation
1	Bad Time Management	3	8	7	168	Follow the provided Gantt chart carefully and ensure tasks are completed to schedule.
2	Hardware Failure	3	9	2	54	Ensure work is backed up to the cloud and is saved regularly.
3	Software Failure	3	10	3	90	Ensure libraries and software used is of the most stable version if possible.

6 Project Overview

Before we can begin the design and implementation of our project, we must gather a general overview of the work to be done. Following the general timeline of the Gantt chart we created in the previous sections, our first step is to research into both existing SAT solvers and how they work. Secondly, we must study the book that we will be taking the algorithm from, Donald Knuths ”The Art of Programming”, which we will refer to as ”TAOP” from now on to simplify things. In our research phase, we will be able to study how some common SAT solvers are implemented, including data structures that they use, and general programming constructs that enable the solver to be used effectively and efficiently.

7 Research Phase

We begin our project following the guidence of the aforementioned Gantt chart, with the research phase. The following will be a look into Donald Knuths ”TAOP” and the observations we take from the initial chapters. It is not completely necessary to read the entire book, as what we’re interested in the most is the algorithms and data-structures that are proposed.

7.1 TAOP

Donald Knuth’s epic, ”The Art of Programming”, contains an in-depth look into the problem of satisfiability. One of the most important topics in computer science, it encompasses some of the hardest problems that plague researchers and engineers alike. To preface our project’s design phase, it is important that we have a sound understanding of the basic ideas and methodologies within the field of satisfiability.

This is aided by the analysis of TAOP, and using the ideas proposed by Donald Knuth, alongside the way in which he easily explains the concepts, allows for good head start. Starting with the first chapters, Donald Knuth outlines the basic ideas of satisfiability.

7.1.1 Satisfiability

In the words of Donald Knuth, satisfiability is ”INSERT DK QUOTE”.

7.1.2 Example Problems

7.2 Algorithms

Following on from the previous section, we can now begin analysis of the algorithms that are presented to us by Donald Knuth. Conveniently for us, Donald Knuth has included the performance metrics and expected efficiency of all the proposed algorithms. This makes choosing which one we wish to implement a little easier, but we must still remember not to stretch outside of the project scope. The following is a short description of three simple algorithms described in TAOP.

7.2.1 Algorithm A

Algorithm A is the simplest of the three that we have chosen. This algoritm is a no-frills backtracking solver, using linked lists and no advanced data structures. Named by Donald Knuth as an enhanced backtracking algorithm with watched literals. Detailed on pages 28 to 29, this algorithm uses

7.2.2 Algorithm B

Algorithm B is an improved implementation of algorithm A, using a slightly more advanced data structure as well as implementing efficiency improvements. This algorithm works in a similar method to algorithm A. But instead introduces the concept of watched literals. This concept provides healthy improvements to the existing code, without a large increase in complexity to the general codebase.

7.2.3 Algorithm D

Algorithm D is the most performant of the three we have chosen, and uses better data structures as well as more efficiency improvements. It is a typical DPLL style cyclic backtracking algorithm, and can tackle more complex problems.

7.3 Existing FOSS Solutions

There already exist a plethora of existing solutions, typically created to compete in SAT solver competitions.

7.3.1 Minisat

Minisat is a very popular implementation of a CDCL solver, written in as little lines as possible. It can tackle complex problems and remains a good piece of software for learning about advanced SAT solvers.

7.3.2 LearnSAT

LearnSAT is a special implementation of a CDCL solver, focused on describing and teaching how a SAT solver works.

7.4 Libraries and Software

An important thing to consider when undertaking a project such as this, is whether there already exists any solutions. For instance, as TAOP was released in 2015, it is perfectly rational to assume another has already implemented these algorithms.

8 Design Phase

8.1 Initial Psuedocode

8.1.1 Algorithm D

8.1.2 Data Structures

In order to implement one of the algorithms presented by Donald Knuth, we must also implement the data structures that he presents alongside them. These data structures are fundamental to the performance of these algorithms, without them the manipulation of the data that is reqiured would not be as efficient, or in some cases possible. The following is a run-down of the structures that are to be implemented, as presented from the TAOP book.

8.1.3 Compilers

There are multiple compilers available that all achieve the same result, albiet in slightly different ways. For C++, there are three main options used by developers: GCC / G++, MSVC, and Clang. For this project, we had to consider the ease of development on the hardware available. As we will be developing on a Windows machine, alongside the availability of Windows Subsystem for Linux, GCC seemed the obvious choice.

GCC allows compilation on both Windows and Linux, agnostic to the type of editor that will be used. In some situations, it might be easier to boot up an instance of Vim from Linux in the terminal. But in others it may be useful to have the power of a full IDE, such as JetBrains Clion. By using GCC, moving from one machine to another is also a breeze. Thus reducing the development time for this project by streamlining the development toolchain.

Although we could use the Microsoft Build Tools and the included LLDB debugger, which does sport some performance improvements over GDB, the modern MinGW suite has improved significantly over the last few iterations. But to retain the consistency available across all platforms, we shall stick to using GCC / G++ and the bundled GDB debugger. Specifically for Windows, the compiler and debugger are provided by the MinGW toolchain. The specific versions used for this project are listed below:

- MinGW $\rightarrow$ 5.3
- CMake $\rightarrow$ 3.17.3
- GDB $\rightarrow$ 9.2

8.1.4 C++17

The whole point of this project is to demonstrate the implementation of the SAT solver in C++, thus it is obvious that this is what we shall choose. As for which version of C++, there is not much debate as to the need for a specific version. So we shall settle on C++17 as it's new enough to have general improvements, but old enough to be well tested and reliable for our needs.

8.2 Initial Design

The initial designs for the project are based off of the above sections. Using the data structures and psuedocode that we generated, a general idea of how the program may be implemented can be easily obtained.

8.3 Completed Design

9 Development Phase

9.1 Initial Prototype

9.2 Review

9.3 Further Development

9.4 Final Implementation

10 Testing Phase

10.1 Debugging & Testing

10.2 Testing Criteria

10.3 Results

11 Revisions Phase

12 Suggested Improvements

13 Conclusion

References

[1] R. E. Ladner, "On the structure of polynomial time reducibility," *J. ACM*, vol. 22, p. 155–171, Jan. 1975.

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

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

[4] Qef, "A representation of the relation among complexity classes, which are subsets of each other.," 2007.

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

[6] T. Baker, J. Gill, and R. Solovay, "Relativizations of the $\mathcal{P} =? \mathcal{NP}$ question," *SIAM Journal on Computing*, vol. 4, pp. 431–442, 1975.

[7] O. Ohrimenko, P. J. Stuckey, and M. Codish, "Propagation = lazy clause generation," in *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, LNCS*, Springer-Verlag, 2007.

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

[9] 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.

[10] Tamkin04iut, "Dpll final implication graph," 2012.

[11] R. Paturi, P. Pudlák, M. E. Saks, and F. Zane, "An improved exponential-time algorithm for k-sat," *J. ACM*, vol. 52, p. 337–364, May 2005.

[12] T. Balyo and C. Sinz, "Parallel satisfiability," *Handbook of Parallel Constraint Reasoning*, pp. 3–29, 2018.

[13] M. J. H. Heule, O. Kullmann, S. Wieringa, and A. Biere, "Cube and conquer: Guiding cdcl sat solvers by lookaheads," in *Hardware and Software: Verification and Testing* (K. Eder, J. Lourenço, and O. Shehory, eds.), (Berlin, Heidelberg), pp. 50–65, Springer Berlin Heidelberg, 2012.

[14] A. Arbelaez and Y. Hamadi, "Improving parallel local search for sat," in *Learning and Intelligent Optimization* (C. A. C. Coello, ed.), (Berlin, Heidelberg), pp. 46–60, Springer Berlin Heidelberg, 2011.