SAT Solving: A Comparative Analysis of DPLL and CDCL Algorithms

Author: Rohan Hetalkumar Desai

Supervisor: Dr Christopher Hampson

Student ID: K21006729

Course: Computer Science BSc

April 25, 2024

Contents

Abstract

Boolean Satisfiability (SAT) is a fundamental problem in computer science which has applications in many fields. Although SAT is an NP-complete problem, advancements in SAT algorithms have made it possible for many industry problems to be solved quickly. This project is a comparative study of DPLL and CDCL algorithms, seeking to understand which instances of SAT problems each algorithm performs best on. This project provides background on SAT, DPLL, and CDCL. It further provides the design, implementation and evaluation of DPLL and CDCL algorithms. By experimenting with their designs, this project looks for potential optimisations to the DPLL and CDCL algorithms. Furthermore, we implement DPLL with a recursive and iterative structure to evaluate which design is more effective. This study is part of the research for finding efficient SAT-solving algorithms.

Originality Avowal

I verify that I am the sole author of this report, except where explicitly stated to the contrary. I grant the right to King's College London to make paper and electronic copies of the submitted work for purposes of marking, plagiarism detection and archival, and to upload a copy of the work to Turnitin or another trusted plagiarism detection service. I confirm this report does not exceed 25,000 words.

Acknowledgements

I am deeply grateful to my supervisor, Dr Christopher Hampson, for his support, expert guidance, and invaluable feedback throughout the duration of this project.

Chapter 1

Introduction

The Boolean satisfiability (SAT) problem involves determining whether a variable assignment of truth values (true or false) exists that satisfies the Boolean formula. This problem is a difficult challenge as the search space for possible assignments grows exponentially with the number of variables.

SAT is the first problem proven to be non-deterministic polynomial time complete (NP-complete). This was shown by the Cook-Levin theorem [20]. SAT is a decision problem, meaning it outputs "yes" or "no" for every input. A problem D, falls into the NP-complete category if it is in the NP class and also NP-hard. [29]. To be in the NP class means that a solution to problem D can be verified in polynomial time, and to be NP-hard means that every problem in NP can be polynomially reduced to the problem D [29]. By the definition of NP-complete, if the SAT problem can be solved in polynomial time, it implies that P equals NP. Consequently, all problems within NP, can be efficiently solved in polynomial time. This would have significant implications in many fields, such as AI, cryptography and formal verification.

The SAT problem is at the centre of computational problem-solving, presenting challenges and opportunities across many fields. While SAT being NP-complete means it can take exponential time to solve a SAT problem, its application in a range of fields generates a lot of interest in finding efficient algorithms [86]. Instances of SAT can be found in formal verification of software and hardware [35], [32], cryptography [67], [51], AI planning [34] and many more can be found in [47]. A SAT solver is a program that aims to solve the SAT problem. SAT solvers are crucial for solving complex decision problems across these diverse fields. The choice of algorithm directly impacts the effectiveness of solving complex SAT instances [56]. The efficiency of SAT solvers will directly impact the reliability of software and hardware systems, the security of cryptographic algorithms and decision-making processes in AI planning. As a result, decades of research have gone into implementing efficient SAT solving algorithms. One is the Davis-Putnam-Logemann-Loveland (DPLL) algorithm introduced in 1961 by Martin Davis, George Logemann and Donald W. Loveland [22]. The DPLL algorithm is an improvement of the earlier Davis-Putnam algorithm [23]. Before the DPLL algorithm, SAT solving faced scalability issues. The DPLL algorithm was developed for a more systematic and efficient approach to SAT solving [22]. DPLL algorithm has a backtracking mechanism that uses unit propagation and pure literal elimination at each step, overcoming challenges faced by earlier algorithms. The DPLL algorithm has revolutionised SAT-solving, and many modern SAT-solvers use the DPLL mechanisms and share similar structures [57].

A more recently introduced algorithm is the Conflict-Driven Clause Learning (CDCL) algorithm [49]. The CDCL algorithm was developed in the late 20th century [49] due to the ongoing pur-

suit of finding more efficient SAT-solving techniques. CDCL algorithm is a major advancement to SAT-solving; it keeps the foundations of DPLL, containing unit propagation and backtracking, and introduces advanced mechanisms such as conflict analysis and learned clause propagation [49]. Conflict analysis allows CDCL to identify learned clauses, which allows the algorithm to adapt during the solving process. Unlike DPLL, CDCL uses non-chronological backtracking, allowing the algorithm to aggressively reduce the search space.

If CDCL is the descendant of DPLL, why not use it for every SAT problem? The general consensus is that there is no superior SAT solver. Instead, each solver has its own strengths and weaknesses, against different types of SAT problems [81]. In This project, we compare the structure of DPLL and CDCL while also exploring which instances of SAT each algorithm performs better on to improve decision-making when choosing between DPLL and CDCL.

Chapter 2 is focused on reviewing existing literature and providing the historical context of SAT, DPLL and CDCL. It will start by providing definitions of key terms in SAT. It will briefly provide the history of the SAT and showcase its importance, how it is used in multiple domains, and examples of formulas used in some of these domains. It will then delve into the theoretical foundations and components of DPLL and CDCL algorithms and provide examples of how they work. This chapter will lay the groundwork for a comprehensive understanding of the upcoming chapters.

In Chapter 3, we showcase and compare the design of DPLL and CDCL algorithms. This design chapter will start by outlining specific requirements/objectives for this project. It will provide alternative designs as well as justifying design choices. We provide the time and space complexity for functions in the algorithm in big O notation as a theoretical measure of execution time and space required based on the input size. Furthermore, this chapter details key functionalities for an in-depth analysis of both algorithms. In essence, chapter 3 provides a well-defined approach to implementation.

Chapter 4 is the implementation section, which offers a comprehensive insight into the practical aspects of this project. It will showcase the programming code and explain any changes compared to the pseudocode in the design chapter. The programming language, programming libraries and platform used to implement the algorithms will be shown, ensuring transparency in the technical aspects of this project. The provided code will include comments to increase readability and assist others in understanding the implementation.

Chapter 5 is the experiment and evaluation chapter, where we will run experiments and compare the performance of the implemented DPLL and CDCL algorithms. Firstly, it will showcase the experimental setup, performance metrics, and solver configurations. It then presents the observed results and analyses the performance and suitability of the implemented solver configurations on different SAT-solving scenarios. Furthermore, this chapter will detail the testing methodologies used to validate the functionality of DPLL and CDCL implementations; this includes unit testing and integration testing that covers various scenarios and edge cases. Lastly, we assess which of the project's aims and objectives have been met.

In Chapter 6, we will delve into the legal, social, ethical, and professional issues of this project. We will discuss aspects such as its effects on computer security, the environment, the economy, and commercial factors. Finally, we will mention any licensing concerns related to the project.

Chapter 7 will summarise the key findings from this project and explain their implications. Furthermore, we will explain the project's limitations. The chapter will finish with recommendations for future work based on the insights gained from the project and mention other approaches to SAT

solving.

1.1 Project Motivations

SAT-solving algorithms have significant economic and commercial implications [86]. They can drive economic growth by improving efficiency and optimising resource allocation. Several SAT-solving algorithms exist, and selecting the most suitable one can directly influence efficiency and cost. This project will be exploring two SAT-solving algorithms, the Davis-Putnam-Logemann-Loveland (DPLL) algorithm [22] and Conflict-Driven Clause Learning (CDCL) algorithm [49]. SAT has many real-world applications [47]. As there is no one best SAT-solving algorithm for every SAT instance [81], this project aims to provide valuable information on the effectiveness of DPLL and CDCL algorithms. This information can then improve decision-making when selecting between the DPLL or CDCL algorithms for specific applications.

1.2 Scope

There are many approaches to SAT solving; this project focuses on DPLL and CDCL. The scope of the project is to implement, analyse and compare the DPLL and CDCL algorithms. The comparison will consist of how both algorithms approach solving SAT problems and their performance across different SAT formulas to assess the strengths and weaknesses of each algorithm. Additionally, this project will discuss the history of SAT, DPLL, and CDCL, emphasising the importance of SAT and how these two algorithms were developed. This project will include the implementations of DPLL and CDCL from scratch; however, not all code implementations will be shared to focus only on key functions of the algorithms. There are many types of SAT problems; this project focuses on random-3-SAT, planning SAT and pigeonhole SAT problems.

1.3 Aims

The primary aim of this project is to conduct a comparative analysis of the structure and performance of DPLL and CDCL algorithms to decide when to use each algorithm. The secondary aim is to implement the algorithms to compare their performance. The final aim is to experiment with the designs of both algorithms to find potential optimisations.

Chapter 2

Background

2.1 Preliminaries

This section will provide definitions and examples to comprehensively understand the upcoming work.

For other definitions in SAT, DPLL and CDCL, which are not mentioned in this chapter, please see [9].

2.1.1 Boolean Satisfiability Problem

Definition 2.1.1 (Basic Boolean Operators)**.** There are three basic boolean operators: NOT, AND, OR.
The NOT operator, denoted by "¬" or "-", returns the opposite truth value of a boolean variable.
The AND operator, denoted by "∧", returns *true* if both boolean variables are *true*, else it returns $false$.
The OR operator, denoted by "∨", returns *true* if either of the boolean variables is *true*, else it returns $false$.

NOT

x	x'
0	1
1	0

AND

x	y	xy
0	0	0
0	1	0
1	0	0
1	1	1

OR

x	y	x+y
0	0	0
0	1	1
1	0	1
1	1	1

Figure 2.1: Truth table of NOT, AND and OR operations [11].

Definition 2.1.2 (Propositional formula)**.** A propositional formula consists of Boolean literals that are connected using logical operations such as negation, conjunction, disjunction, or implication. [9].

Example 2.1.1 (Propositional formula)**.** *Here are examples of propositional formulas and if they can be satisfied or not:*
$(x \vee y) \vee (\neg x \wedge y)$*, can be satisfied by the assignment* $x = false$ *and* $y = true$*.*
$(x \wedge \neg x)$*, cannot be satisfied as it requires* x *to be true and* $false$ *simultaneously, this is not possible.*

Definition 2.1.3 (Clause)**.** A clause is a disjunction of literals.

Example 2.1.2 (Clause)**.** *All of the following are clauses:* x, $(x \vee y)$, $(x \vee y \vee \neg z)$

Definition 2.1.4 (Conjunctive Normal Form (CNF))**.** A propositional formula is in CNF if it is a single clause or conjunction of clauses.

By this definition, a propositional formula is only satisfied if all its clauses are satisfied. A clause is satisfied if any one of its boolean literals is satisfied.

Example 2.1.3 (CNF)**.** *The following formula is in CNF:*
$(x \vee y \vee \neg z) \wedge (w \vee \neg y \vee z \vee \neg x) \wedge (x) \wedge (y \vee \neg z)$
This CNF formula consists of 4 clauses with lengths varying from 1 to 4 variables. It is satisfiable by many different assignments, one of which is: $x = true$, $w = true$, $y = true$, $z = false$.

Definition 2.1.5 (DIMACS CNF)**.** The DIMACS CNF format serves as a standard file format for representing propositional formulas in CNF. In this format, each variable is denoted by a positive integer, while its negation is represented by the negation of that integer [24]. Each line in the file corresponds to a clause, and the 0 at the end of a line represents the end of the clause.

Example 2.1.4 (DIMACS CNF)**.** *The formula* $(x) \wedge (\neg x \vee y \vee \neg z) \wedge (x \vee \neg y \vee \neg z) \wedge (\neg y \vee z)$ *can be represented in DIMACS CNF as:*

```
1 0
-1 2 -3 0
1 -2 -3 0
-2 3 0
```

0 indicates the end of a clause.

Definition 2.1.6 (Boolean satisfiability problem (SAT))**.** Does there exist an assignment of truth values which satisfies a given propositional formula in CNF.

2.2 Literature Review

2.2.1 Boolean Satisfiability Problem

The boolean satisfiability problem (SAT) holds significance from both theoretical and practical perspectives. It was the first problem proved to be NP-complete, and its importance extends to areas such as formal verification of software and hardware [35, 32], cryptography[67], [72], [51], [50] and AI [64], [34].

Brief History of SAT

The early foundations of SAT were laid by George Boole's work in the mid-1800s, providing a mathematical representation for logical operations using binary variables [14]. In the early 20th century Ernst Schröder's work on symbolic logic enabled the manipulation of logical expressions using algebraic techniques [68]. In the 1960s, Stephan Cook established the concept of NP-completeness and proved that SAT is NP-complete [20]. Initial algorithms used to solve SAT relied on brute-force and backtracking approaches [23]. These algorithms were limited to solving simple SAT problems and could not handle larger more complex SAT problems. In the 1960s, the DPLL algorithm was introduced [22], which provided a more systematic approach to SAT solving and is still used in modern

SAT solvers. In the 1990s João Marques-Silva and Karem Sakallah developed the CDCL algorithm, which uses learnt clauses to guide the search more effectively [49]. Starting in 2002, the annual SAT Competitions [65] is a competitive event for SAT solvers. It showcases and compares the performance of many SAT solvers, driving innovation in the field. There have been many key milestones in the history of SAT, none of which provided a polynomial-time solving algorithm for SAT.

Advancements in computer hardware have had a major impact on SAT solving. The speed of SAT solving algorithms is affected by computer hardware such as the Central Processing Unit (CPU) [16], [70], Graphics Processing Unit (GPU) [70], [21], [58] and Random Access Memory (RAM). Modern computers consist of multiple CPUs with high processing speed. They can execute code faster by parallel processing, reducing solving times for SAT problems [70]. Furthermore, Modern computers consist of GPUs, which, similar to CPUs, can carry out multiple processes simultaneously, helping to reduce the amount of time to run a program [70], [21], [58]. However, there are only a few SAT solvers that use GPUs [58]. Modern computers also have increased RAM, allowing SAT solvers to handle larger SAT instances. This is important for solving real-world problems that may contain a larger number of variables and clauses.

Importance of SAT in Formal Verification

Verifying the correctness or quality of systems is becoming progressively challenging as systems grow larger and become more intricate. [35], [32]. Formal verification is used to ensure the correctness of hardware and software systems through mathematical reasoning [35]. Manual formal verification is possible and relatively quick for simple systems. However as the complexity of the system increases, the manual process becomes more cumbersome and error-prone.

In software verification, properties of the software can be expressed as logic formulas. The formulas are represented as an instance of Satisfiability modulo theories (SMT). A SMT solver consists of a SAT solver [57]. A SMT solver translates the SMT expression into CNF, which then can be solved using a SAT solver [82]. SMT solvers are used to check the satisfiability of these formulas, which would determine whether the software satisfies or violates the specified properties. If the properties are violated (i.e. there is no solution to the formula) that would indicate there is a bug in the software [35]. Figure 2.3 shows the transformation of a function to an instance of SMT.

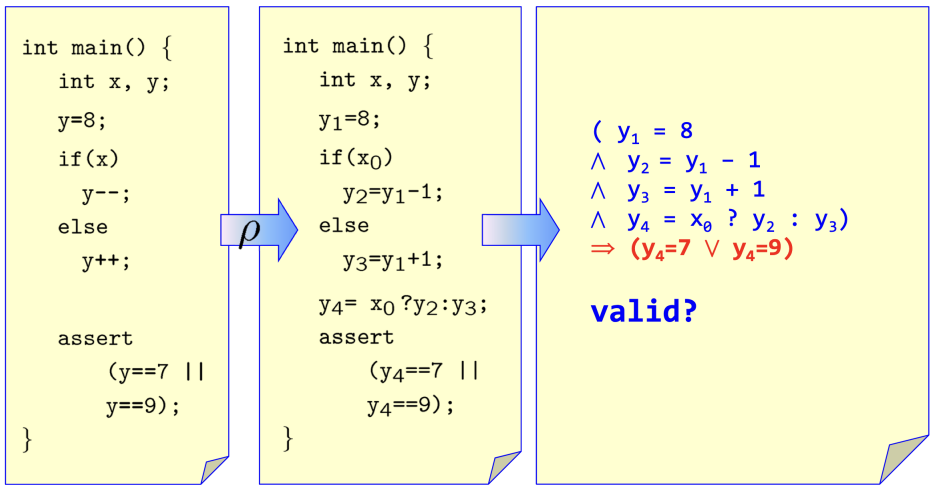

Figure 2.3: Function to Single Static Assignment (SSA) form to Satisfiability modulo theories (SMT) formula [30]

Each function in a software is first converted to Single Static Assignment (SSA) form and then converted into a corresponding SMT instance, which is then converted into a corresponding SAT instance. With multiple functions, there will be several SAT instances to solve. If any of the SAT

instances are unsolvable, that would indicate an error in the respective function.

Similarly to software verification, SAT plays a significant role in hardware verification. Hardware designs are typically described using hardware description languages (HDLs). SAT solvers are used to check the correctness of these designs against specified properties [32]. SAT solvers are also used for Automatic Test Pattern Generation (ATPG), these patterns are designed to detect faults or errors in the hardware [32]. SAT solvers find input combinations that will trigger the detection of faults [32]. This will ensure the reliability and quality of the design.

The cost of a mistake in software and hardware systems can be very expensive [33], [73]. By using SAT solvers in formal verification, risks can be reduced [32] and bugs can be detected earlier in the development process to save time and resources.

An efficient SAT solver would speed up the process of exploring design space and verifying the properties of hardware and software designs, reducing development time. It would also be able to analyse and solve larger instances of SAT, meaning more complex systems can be verified. Overall, SAT plays a major role in software verification, from verifying designs to identifying bugs. SAT has reduced development time and saved costs. Without SAT, verifying complex systems without making any errors becomes a tedious process.

Importance of SAT in Cryptography

SAT is a double-edged sword in cryptography. It can be used to identify weaknesses in algorithms and verify protocols, but it can also be used to exploit cryptographic algorithms.

Using a broken/weak cryptographic algorithm/scheme can be very expensive [78]. Broken/weak cryptography can be exploited to expose sensitive information, modify data and much more[78]. Cryptanalysis is the process of decoding messages without the key. SAT plays a crucial role in identifying the weaknesses of cryptographic algorithms through SAT-based cryptanalysis [51].

Cryptanalysis of hash functions is to find collisions, where two distinct inputs produce the same hash output [51]. SAT solvers can be used to attack hash functions. The hash function operations can be converted into a SAT instance, which is then solved by a SAT solver [51]. In 2005, several standard cryptographic hash functions were broken by attackers [51]. This exposed the vulnerability of hash functions, and new versions of hash functions, such as SHA-3, were created, which are resilient to a SAT-based attack [51], [31].

The NP-completeness of SAT can be leveraged to create reliable and secure cryptography schemes [67]. A SAT-based public key cryptography scheme involves having the public key as a SAT instance, and the private key would be the assignment of values that satisfy the SAT instance [67]. The public key encrypts messages, while the private key decrypts them [67].

$$
\begin{aligned}
\textit{priv} &= (1,1,0,0,1,0,0) \\
\textit{pub} &= (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_4 \vee x_5) \\
&\quad \wedge (x_1 \vee x_6 \vee x_7)
\end{aligned}
$$

Figure 2.4: A SAT-based public and private key[67]

The strength of this SAT-based cryptography scheme is that the search space of SAT grows exponentially with the number of variables. Having a larger number of variables will make the cryptography scheme less vulnerable to attacks such as brute force [67]. It is important to note that the number

of clauses affects the strength of the cryptography scheme; more clauses will increase the difficulty for a SAT-solver to find a solution [67]. The Advanced Encryption Standard (AES) is a trusted algorithm used by governments and many organisations [80]. It has 3 variations, the strongest of which is the AES-256. The AES-256 has a possible key space of 2^{256} [3]. With a SAT-based cryptography scheme, the same key space can be matched by using 256 variables in the private key. It is important to note that the number of clauses in the private key is also a key factor in the strength of the key [67].

Like many cryptography schemes there are attacks to a SAT-based public key cryptography scheme such as the Decoding attack, Oracle attack and Reducing SAT problem attack [67]. However, these attacks can be weakened [67].

An important aspect of cryptography is verifying protocols. Short and simple protocols can be easily verified manually; however, as they increase in size and complexity, manual verification becomes cumbersome. Similar to formal verification of software, SAT is used to verify cryptographic security protocols [72]. The process involves translating protocols into an instance of SMT, which is then converted into an instance of SAT by the SMT solver [72]. The SAT instance is then solved by the SAT solver, and if it can not be solved, then it implies a violation of specified properties [72].

A polynomial time SAT solver would disrupt cryptography. All encryption schemes are vulnerable to P=NP [69], even those that are not SAT based such as RSA. On the positive side, efficient SAT solvers will be able to quickly identify weaknesses and verify the correctness of cryptographic algorithms/protocols, leading to increasing security. Overall, SAT plays a major role in cryptography, from finding weaknesses in algorithms to verifying protocols, which has improved the security of cryptographic algorithms. Without SAT, verifying complex systems without making any errors becomes a tedious process.

Importance of SAT in AI

Many AI problems, such as planning [34] and searching [34] can be encoded as SAT instances. Modern SAT solvers are highly optimised and can handle large instances of SAT problems [52]. Their efficiency and scalability enable AI systems to tackle real-world problems with large search spaces.

Planning is a common task in everyday life, whether that be deciding on which route to take to another city or buying groceries [79]. Planning involves determining a sequence of steps in order to achieve a goal. You might even want to add optimisation metrics such as minimising time taken or minimising cost. With a small number of states and actions, it may be easily possible to find an optimal plan manually; however, as the number of actions and states grows, finding the optimal plan manually becomes infeasible.

SAS+ represents a planning problem using multi-valued state variables [34]. SAS+ planning problem consists of an initial state, goal state, actions, and transitions between states, all of which can be encoded using Boolean variables and logical formulas [34]. Each state in the problem is represented as a set of Boolean variables [34]. The preconditions and effects of actions are translated to logical formulas [34]. The constraints of the problem are also translated to logical formulas [34]. Figure 2.5 shows the translation of a planning problem in Fast Downward's translator format to the DIMACS-CNF formula. Once the problem has been encoded as a SAT instance, a SAT solver can be used to find a satisfying assignment. The satisfying assignment represents an action plan, which is a sequence of actions that goes from the initial state to the goal state [34]. SAT-based planning can handle optimisation metrics, such as minimising the number of actions or costs, by encoding additional objective functions [34]. This makes SAT-based planning more valuable as planning is

usually looking to optimise some metric.

```
1                    begin_state    1                  p cnf 6 15
begin_variable       0              begin_operator     3 0
var1                 end_state      move r0 r1         -5 0
-1                   begin_goal     0                  -4 0
2                    1              1                  ...
Atom at-robby(r0)    0 1            0 0 0 1            5 -6 1 0
Atom at-robby(r1)    end_goal       0                  -3 4 1 0
end_variable                        end_operator       -5 6 0
                                                       -1 1 0
```

Figure 2.5: "The first three listings are the concrete syntax of a planning problem in Fast Downward's translator format. The fourth listing is a DIMACS-CNF formula" [2].

As mentioned previously in Importance of SAT in Formal Verification, SAT can verify software and hardware systems. This also applies to AI software and hardware systems. However this is a similar process, so we will not repeat it here.

AI is reshaping the board game landscape [74]. It provides players with challenging adversaries by adapting to players' strategies and simulating various play styles [74]. Positions of board games such as Chess and Mastermind [1] can be translated to an instance of SAT, allowing for a SAT-based AI for board games [46], [60].

An efficient SAT solver would have major implications in AI. It can save time, money and more by quickly solving large and complex planning problems with optimisation metrics. Furthermore, efficient SAT solvers would enhance AI by allowing them to rapidly search through a large search space, possibly making SAT-based AI undefeatable. Overall, SAT plays a major role in AI, from finding optimal plans to beating humans in boring games. Without SAT, planning can be a very lengthy process, especially if you are looking to optimise a metric.

2.2.2 DPLL

The Davis, Putnam, Logemann, and Loveland (DPLL) is a sound and complete algorithm [22], [23], meaning it can find a solution when one exists. In this subsection, we discuss the history, importance and structure of the DPLL algorithm, as well as show a visual example of how the algorithm works.

Brief History and Importance of DPLL

The DPLL algorithm was introduced by Martin Davis, George Logemann, and Donald W. Loveland, in 1961 [22], following the Davis-Putnam algorithm (DP), which quickly depleted available memory [39]. The DPLL algorithm introduces a chronological backtracking mechanism [22], which allows for a systematic search through the space of possible assignments. It further introduces the notion of unit propagation and pure literal elimination that allows for a more efficient search through the search space [22].

DPLL finds applications beyond SAT solving, extending into areas like automated theorem proving or satisfiability modulo theories (SMT), which uses DPLL to prove mathematical theorems [57].

The DPLL algorithm has evolved the landscape of SAT solving. Most modern-day state-of-the-art SAT solvers are based on different variations of the DPLL algorithm [6].

[1] Mastermind, a game of strategy, challenges players to crack a hidden code. In an experiment, it took the SAT-based AI just 10.51 guesses on average to find the secret, showcasing its power [60].

Structure and Example of the DPLL algorithm

The DPLL algorithm uses three key techniques to guide its search: unit propagation, pure literal elimination and splitting.

Definition 2.2.1 (Unit clause)**.** A unit clause consists of only one literal.
A unit clause can only be satisfied by assigning an appropriate truth value to the literal. Unit clauses leave no choice but to satisfy the literal; otherwise, the clause is unsatisfiable.

Example 2.2.1 (Unit clause)**.** *Here are examples of unit clauses and how to satisfy them:*
$(x \vee y \vee \neg z) \wedge (\neg w) \wedge (x) \wedge (y \vee \neg z)$*, Here* x *and* $\neg w$ *are unit clauses and can be satisfied by assigning* $x = true$ *and* $w = false$*.*

Definition 2.2.2 (Unit propagation)**.** Unit propagation is the process of finding unit clauses and satisfying them until no more unit clauses exist in the formula.

Definition 2.2.3 (Pure literal)**.** A pure literal is a variable present in a propositional formula with only one polarity, either positive or negative.

In CNF, satisfying a pure literal will satisfy all of the clauses containing it.
Note: Propositional formulas do not have to be in CNF for pure literals to exist. We will mention CNF as we look at solving SAT problems in CNF.

Example 2.2.2 (Pure literal)**.** *Here are examples of pure literals and how to satisfy them:*
(y)*, Boolean variable* y *occurs with only positive polarity in the formula, therefore, it is a pure literal and can be satisfied by assigning* $y = true$*.*
$(x \vee y) \wedge (x \vee \neg y)$*, Boolean variable* x *occurs with only positive polarity in the formula, therefore, it is a pure literal and can be satisfied by assigning* $x = true$*. This will satisfy both clauses.*
$(x \vee y) \wedge (\neg x \vee \neg y) \wedge (\neg z \vee y) \wedge (\neg z \vee x)$*, Boolean variable* z *occurs with only negative polarity in the formula, therefore it is a pure literal and can be satisfied by assigning* $z = false$*. This will satisfy the 3rd and 4th clause.*

Definition 2.2.4 (Pure literal elimination)**.** Pure literal elimination is the process of finding pure literals and satisfying them until no more pure literals exist in the formula.

Both unit propagation and pure literal elimination simplify decision-making by making deterministic variable assignments, which can significantly speed up the search process. Splitting, which is also known as branching, is the algorithm's search strategy. It involves assigning a truth value to a variable and then recursively solving the formula. If a contradiction occurs during this process, the algorithm backtracks and negates the truth value of the variable and keeps splitting until a solution is found or all possibilities are exhausted.

Given a formula f in CNF, DPLL(f) returns $true$ if f is satisfiable or $false$ if it is unsatisfiable. Consider the following Boolean formula f in CNF: $(A \vee B \vee C) \wedge (\neg A \vee \neg B \vee C) \wedge (A) \wedge (\neg A \vee \neg B) \wedge (B \vee \neg C \vee E) \wedge (E \vee F) \wedge (\neg B \vee \neg C \vee \neg D \vee E) \wedge (\neg E \vee \neg F) \wedge (\neg E \vee F)$.

There are 8 steps to the algorithm:

1. Apply unit propagation until no more unit clauses exist in the formula.

   f contains one unit clause: (A)
   After satisfying the unit clause $f = (\neg B \vee C) \wedge (\neg B) \wedge (B \vee \neg C \vee E) \wedge (E \vee F) \wedge (\neg B \vee \neg C \vee \neg D \vee E) \wedge (\neg E \vee \neg F) \wedge (\neg E \vee F)$
   f contains one unit clause: $(\neg B)$

After satisfying the unit clause $f = (\neg C \vee E) \wedge (E \vee F) \wedge (\neg E \vee \neg F) \wedge (\neg E \vee F)$
f contains no unit clauses, so continue.

2. Apply pure literal elimination until no more pure literals exist in the formula.

   f contains one pure literal: $(\neg C)$
   After satisfying the pure literal $f = (E \vee F) \wedge (\neg E \vee \neg F) \wedge (\neg E \vee F)$
   f contains no pure literals, so continue.

3. Check if the formula is empty; if so, then it indicates that all clauses have been satisfied and return *true*; otherwise, continue.

   f is not empty, so continue.

4. Check if the formula contains any empty clauses; if so, then it means the clause is unsatisfiable and returns $false$; otherwise, continue.

   f contains no empty clauses, so continue.

5. Randomly assign an unassigned variable to *true* and update the formula accordingly.

   Assign variable E to *true*.
   $f = (\neg F) \wedge (F)$

6. Recursively call DPLL with the updated formula; if the recursive call returns *true*, then the formula is satisfiable and returns *true*. Else, go to step 7.

   In this case it will return $false$, as $\neg F$ and F are both unit clauses and cannot be satisfied at the same time.

7. Backtrack by negating the previously assigned variable, updating the formula, and recursively calling DPLL with the updated formula.

   Assign variable E to $false$.
   $f = (F)$

8. If the second recursive call returns *true*, then the formula is satisfiable, and the algorithm returns *true*; otherwise, if both recursive calls fail to find a satisfying assignment, the formula is unsatisfiable, and the algorithm returns $false$.

   In this case it will return *true* as it will assign F to *true* which will make f empty.

In this example we only branch on one variable which is E, however for other SAT problems it is likely the algorithm will have to branch on many variables.

It is important to note that, at most, the algorithm branches on a single variable per call, leading to at most two recursive calls per function call. The DPLL algorithm only backtracks to the previous decision level; this is known as chronological backtracking. Figure 2.6 1 shows the pseudocode representation of the DPLL algorithm.

Given a formula f in CNF with n variables. The DPLL algorithm has a worst-case performance of $O(2^n)$, best-case performance of $O(1)$ (constant) and worst-case space complexity of $O(n)$. This is for the basic algorithm shown in 1; variants of the algorithm will have different worst-case space complexity but the same worst-case performance and best-case performance [25].

Algorithm 1 DPLL

```
Algorithm DPLL
    Input: A set of clauses Φ.
    Output: A truth value indicating whether Φ is satisfiable.

function DPLL(Φ)
    // unit propagation:
    while there is a unit clause {l} in Φ do
        Φ ← unit-propagate(l, Φ);
    // pure literal elimination:
    while there is a literal l that occurs pure in Φ do
        Φ ← pure-literal-assign(l, Φ);
    // stopping conditions:
    if Φ is empty then
        return true;
    if Φ contains an empty clause then
        return false;
    // DPLL procedure:
    l ← choose-literal(Φ);
    return DPLL(Φ ∧ {l}) or DPLL(Φ ∧ {¬l});
```

Figure 2.6: DPLL algorithm pseudocode [25].

Variations of DPLL Algorithm

Over the years, many variants and optimisations of the DPLL algorithm have been implemented to enhance its efficiency. One notable variation is the CDCL algorithm, which includes a mechanism for learning new clauses when reaching conflicts [10]. Some optimisations are decisions heuristic [39], dynamic restart policies [63] and parallelization techniques [27].

The original DPLL algorithm uses data structures such as a list to store the formula and stack for recursive backtracking. Many variations of the DPLL algorithm use different data structures to improve the efficiency of the DPLL algorithm. The paper [28] mentions the head-tail lists to be efficient with the DPLL algorithm. The Head-Tail data structure, as described in [28] and [84], assigns two pointers to each clause: the head and the tail pointer. The head pointer refers to the first literal in the clause, while the tail pointer refers to the last literal [28]. As literals are removed from the clauses as the negation of the literal is satisfied, the head and tail move towards each other. If a satisfied literal is identified, the clause is satisfied [28]. If both the head and tail references point to the same literal, the clause is deemed unit [28]. When no unassigned literal can be identified, the clause is deemed to be either satisfied or unsatisfied, depending on the truth value of the literal indicated by the other pointer. [28]. This data structure can be used to find unit clauses and check if a clause is satisfied or not efficiently. However, maintaining this data structure can be costly when backtracking [28].

The papers [86], [28] and [41] mention watched-literals and to be more specific, two watched-literals to be an efficient data structure for SAT solving. Similar to head-tail lists, two references are associated with each clause. However, they can move in any direction [28]. The advantage of the two watched-literals data structure is that no updates to literal references are necessary during backtracking, saving computational time. Furthermore, there is no need to keep additional references [28]. The disadvantage is that unit or unsatisfied clauses are found after searching all the literals of the clauses [28].

SAT-solvers, which use either watched literals or head-tail lists data structures, don't need to update the formula when an assignment has been made. By not tracking which clauses have been removed or updated, solvers can avoid updating any of the clauses during backtracking, which will save them a lot of time. However, these SAT-solvers do not use pure literal elimination as figuring out which literals are currently pure means losing the substantial efficiency gains from the data structures [83]. For further information about the two watched-literals data structure, please see [28], [86], [41].

Other than using different data structures, there are variations of the DPLL algorithm with a different algorithmic structure, such as the iterative DPLL algorithm. Instead of recursively branching on a variable and recursively backtracking, this variation iterative does these things. We will later mention this variation in the design 4. Another variation can be seen in [41], where the DPLL algorithm does not use pure literal elimination.

Problems with DPLL

The DPLL algorithm will traverse down the left-sided path first and then makes its way down the right-sided path if it does not find a solution. It is very similar to a depth-first search but the difference is DPLL has unit propagation and pure literal elimination that usually diverges the algorithm's path from a depth-first search path. An obvious problem can be seen here. What if the only solution is assigning $false$ to all the variables? This will lead to the DPLL algorithm wasting a lot of time searching paths that will eventually be a dead end. In the worst case, the DPLL algorithm will attempt every single assignment, that would be 2^n assignments, where n is the number of variables. This is why the worst-case time complexity of DPLL is $O(2^n)$. Additionally, if the input formula is unsatisfiable, the DPLL algorithm will, in the worst case, try almost every possible assignment combination. This is not efficient and feasible for very large instances of SAT. For example, with an unsatisfiable formula u with 100 variables and many clauses, there are 2^{100} combinations. Modern computers are able to guess more than 100 billion passwords per second [1]. Let's assume they are able to try 100 billion assignment combinations per second; this would mean, in the worst case, it will take approximately 40,000 years to solve the formula u, and after all the long wait, it will return $false$. This problem occurs due to chronological backtracking.

It is important to note that while the DPLL solver could possibly try all 2^n assignments to find a solution, it usually does much fewer assignments.

2.2.3 CDCL

The Conflict Driven Clause Learning (CDCL) algorithm implements the DPLL mechanisms but can learn new clauses and backtracks non-chronologically. Clause learning and backtracking non-chronologically do not affect the soundness or completeness of the algorithm. Proofs of CDCL soundness and completeness can be found in [48], [85]. In this subsection we discuss the history, importance and structure of CDCL and well as show an example of how the algorithm would work.

Brief History and Importance of CDCL

Back in the early 90s, successfully solving a formula with hundreds of variables was seen as a major achievement [49]. However, SAT solvers have since advanced significantly, becoming highly efficient programs capable of easily solving large instances of SAT formulas [49]. The main driver behind this success is CDCL SAT solving [49]. The development of Conflict-Driven Clause Learning (CDCL) algorithms was motivated by the practical applications of SAT in automated planning, automatic test pattern generation, and timing analysis [49]. CDCL SAT solver's range of practical applications has expanded in recent years [49], and they have been a major success in a number of applications

such as the ones mentioned above as well as formal verification, cryptography and bio-informatics [10].

The CDCL algorithm has given a significant boost to SAT solvers, and it is used in many of the modern SAT solvers, such as MiniSAT, Zchaff SAT, Z3, Glucose and ManySAT [18].

Structure of CDCL

The CDCL algorithm uses three key techniques to guide its search: unit propagation, learnt clauses and backjumping.

Definition 2.2.5 (Decision level)**.** The level at which a boolean variable is assigned a truth value. If a boolean variable x is the first to be assigned a truth value, it is said to be assigned at decision level 1. It is represented as $x@1$ if x is assigned *true* or $\neg x@1$ if x is assigned $false$. A decision and all of its implications will have the same decision level.

Definition 2.2.6 (Backjumping)**.** Going back one or more decision levels.

Definition 2.2.7 (Learnt clause)**.** A clause that is derived from the conflict. The learnt clause is the clause that is added to the original formula.

Unlike DPLL, the CDCL algorithm has a special data structure called an implication graph. This data structure keeps track of the assignments made and their implications and can be used to analyse the conflict and generate a learnt clause, which is added to the original formula. The learnt clauses are used to guide the algorithm's search more effectively in future iterations.

Definition 2.2.8 (Implication)**.** Assignment of a variable causes deduction of truth values for other variables.
There is an implication $x@i \rightarrow y@i$ if the assignment $y@i$ was set by unit-propagation from a clause containing $\neg x \vee y$ [54].

Example 2.2.3 (Implication)**.** *Consider the CNF formula:*
$f = (x \vee \neg y) \wedge (w \vee y \vee \neg z) \wedge (x \vee z)$
Assume the first decision assigns z to true, so $z@1$. There are no unit clauses, so there are no implied assignments. Assume the second decision assigns y to true, so $y@2$. There is a unit clause that consists of literal x, so the implied assignment is $x@2$.

Definition 2.2.9 (Implication graph)**.** An implication graph is a directed acyclic graph that represents the logical dependencies between assigned variables. It has a set of vertices V and a set of directed edges E [19]. Each vertex in V represents a variable assignment at a particular decision level. E.g: $x@2$ represents setting x to $false$ at level 2. Each edge represents an implication.

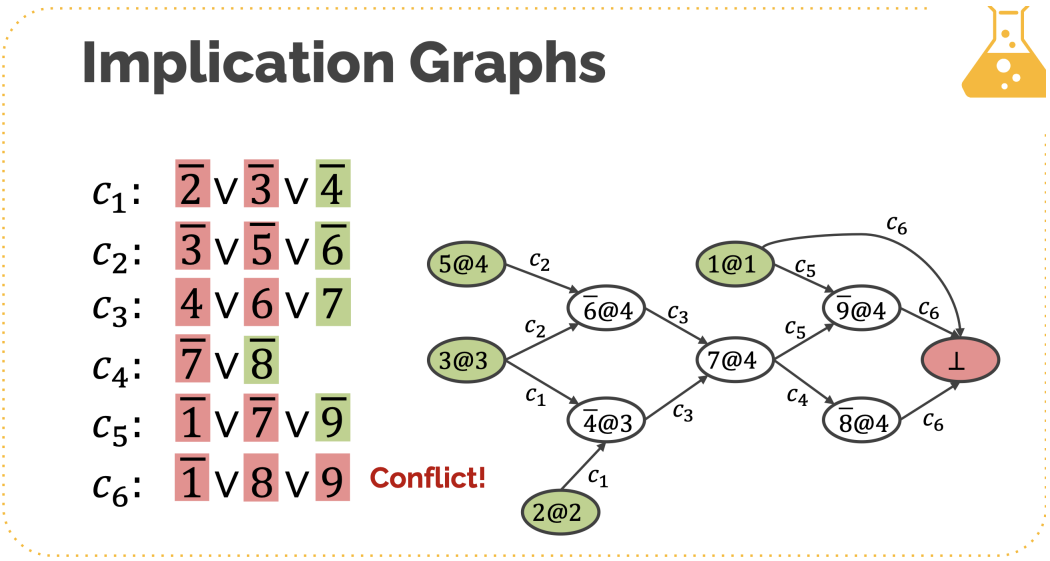

Figure 2.1: Implication graph for a formula comprising of 6 clauses c1-c6 [54]

To analyse the conflict, conflict cuts are created on the implication graph.

Definition 2.2.10 (Conflict cut)**.** A conflict cut in an implication graph is a bipartition of the vertices $V = R \cup C$ such that:
1) Reason side R contains all decisions (source nodes)
2) Conflict side C contains the conflict node (a sink)
3) No edges cross C→R, only R→C [54]

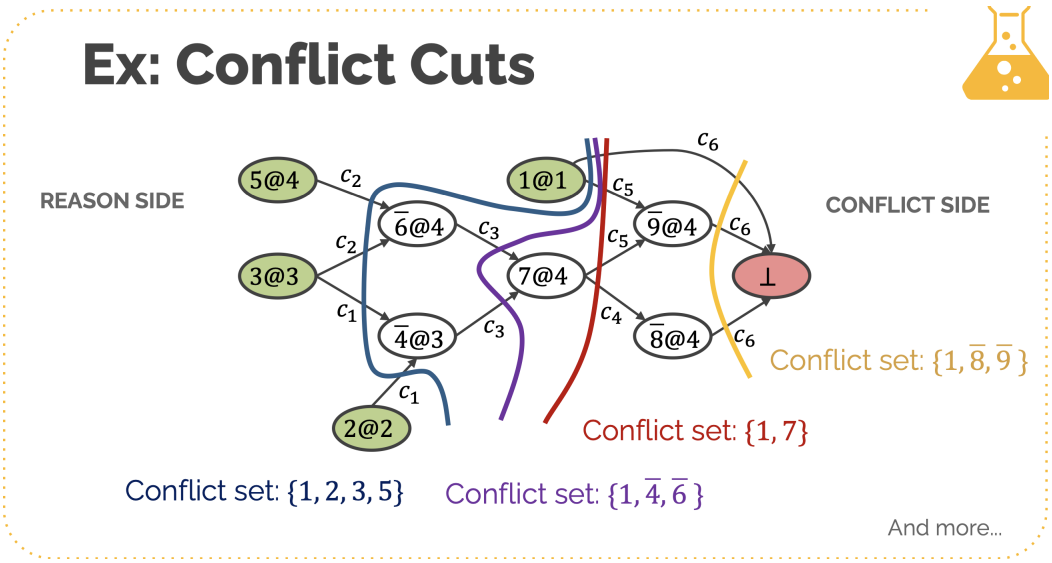

Figure 2.2: Conflict sets and cuts on the implication graph in 2.2.9 [54]

CDCL uses non-chronological backtracking which skips over irrelevant variable assignments, jumping to the identified cause of failure [44].

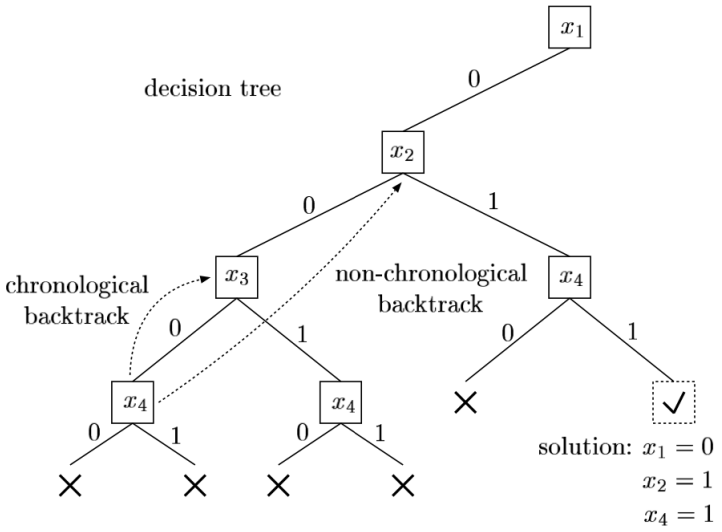

Figure 2.8: Chronological and Non-chronological Backtracking[44].

For example, consider Figure 2.8 which illustrates how chronological and non-chronological backtracking works for a given CNF formula. Following the assignment of value 0 ($false$) to variables $x1$ and $x2$, no possible assignments exist for variables $x3$ and $x4$ that could satisfy the formula. Chronological backtracking jumps to the previous assignment, which in this case is variable $x3$ and wastes time exploring a region of the search space without a solution. On the other hand, non-chronological backtracking has learnt from the conflict and skips variable 3, saving time from exploring a region of the search space without a solution. It is important to note that non-chronological backtracking may not always be possible, and occasionally, CDCL may have to chronological backtrack.

Different CDCL algorithms use different techniques. However, we will be discussing a typical CDCL algorithm, which is shown in algorithm 2.

The main differences to the DPLL algorithm are the call to function `analyse_conflict` each time a conflict is detected and the call to `backjump` function, which allows for non-chronological backtracking (i.e. jumping back 1 decision level or more). Unlike the DPLL algorithm, which uses recursion to explore different branches of the search space, a typical CDCL algorithm does not use recursion. This is to reduce the memory needed to maintain recursive function calls and call stacks. Instead of using recursion, CDCL uses an iterative approach to explore the search space [19]. The CDCL algorithm keeps track of implications using an implication graph, which allows it to analyse conflicts and decide on which decision level to backjump to.

While in the CDCL pseudocode there is no while loop subjecting unit-propagation to continue until no unit-clause appears. This is actually implemented in the unit-propagation function.

Given a formula f in CNF, CDCL(f) returns $true$ if f is satisfiable or $false$ if it is unsatisfiable.
CDCL procedure:

1. Perform unit propagation; if a conflict is reached, then return $false$, else continue.
2. While all variables have not been assigned a value, pick a variable that has not been assigned a truth value and set it to $true$.
3. Perform unit propagation; if a conflict is reached, go to step 4; else, go to step 2.

Algorithm 2 Typical CDCL algorithm

```
function CDCL(φ)
    Inputs φ: Propositional formula
    Output Boolean

    decision-level = 0
    if unit-propagation() = false then                    ▷ there is a conflict
        return false
    end if
    while not all variables have been assigned do
        x = pick-variable()
        x = true
        decision-level += 1
        while unit-propagation() = false do
            if decision-level = 0 then
                return false
            end if
            conflict-clause, backjump-level = analyse-conflict()
            φ.add(conflict-clause)
            backjump(backjump-level)
        end while
    end while
    return true
end function
```

Typical CDCL algorithm pseudocode adapted from [54].

4. If the decision level is 0 then return $false$, else continue.
5. Analyse the conflict and find the conflict clause and backjump level.
6. Add the conflict clause to the original input.
7. Backjump to the backjump level and go to step 3.
8. If the while loop in step 2 ends and does not return $false$, then it means it has found satisfying assignments so return $true$.

Conflict analysis: The `analyse_conflict` function is called when a conflict is detected. This function finds the sequence of variable assignments that caused the conflict using the implication graph and outputs a new clause known as the conflict clause (learnt clause) and the backjumping level (asserting level).

Definition 2.2.11 (Conflict set)**.** A conflict set is a set of assignments which imply a conflict [54]. Given a conflict set, we know at least one literal in the set must be false [54]

Definition 2.2.12 (Conflict clause)**.** A conflict clause represents variable assignments that lead to a conflict. They can be derived from conflict sets, and if no additional computation is done on them, they are the same as learnt clauses.

Clause learning: There are many techniques that can be used to compute the conflict clause; such as 1-UIP (first-UIP) and Rel-sat [75]. The first-UIP method corresponds to the first-UIP cut [75]. It divides the implication graph right before the first UIP (closest UIP to the conflict) is encountered on the path from the conflict node to the current decision variable [75]. It generally produces the shortest conflict clauses [54]. On the other hand, the Rel-sat method corresponds to the last-UIP cut [75]. The last-UIP cut divides the implication graph right before the last UIP; this is the current decision variable at the conflict level [75]. The last-UIP cut creates a conflict set with all of the decision variables in the implication graph. Examples of both clause learning techniques are in the Example of the CDCL algorithm in the play subsection 2.2.3.

Definition 2.2.13 (Unique Implication Point (UIP))**.** Given an implication graph, a vertex l is a unique implication point (UIP) if every path from the most recent decision variable vertex to the conflict vertex traverses through l.

The clause learnt during conflict analysis is added to the input formula, which allows the CDCL solver to avoid similar conflicts in subsequent iterations of the search. This process of conflict-driven learning helps guide the search towards a satisfying assignment and aggressively cuts search branches to improve efficiency.

Backjumping: There are potentially many levels to which the solver can backjump. However, it back jumps to the second-largest (i.e., deepest) decision level in the learnt clause [54]. This is done to make the learnt clause a unit clause after backtracking so that we can make use of the information we just learnt. If the learnt clause consists of a single literal, then we backjump to level 0 [54].

The `backjump` function undoes variable assignments made after the backjumping level. It effectively reverts the decisions that led to the conflict. The function updates data structures such as the implication graph and the assignment list which stores the variables which have been assigned a value.

After backjumping, the solver applies unit propagation and continues from there. The solver continues the search until a satisfying assignment is found or all possible assignments have been explored without success.

Variations of CDCL algorithm

Over the years, many variations and optimisations of the CDCL algorithm have been implemented to enhance its efficiency. Some variants include lazy clause generation [71], portfolio-based parallel CDCL [5] and clause database reduction [36], each with their own pros and cons. Some optimisations include lazy data structures [10], heuristic-based search [55] and parallelisation solving [26].

The Head-Tail and two watched-literals data structures mentioned in 2.2.2, also apply to the CDCL algorithm.

Some CDCL SAT solvers, such as incomplete MaxSAT [37] and GSAT (greedy SAT) [38], sacrifice correctness for speed.

Example of CDCL algorithm in play

Here is an example of how the CDCL algorithm solves a SAT problem.

Consider the following Boolean formula f in DIMACS CNF:
-2 -3 -4 0
-3 -5 -6 0
4 6 7 0
-7 -8 0
-1 -7 -9 0
-1 8 9 0

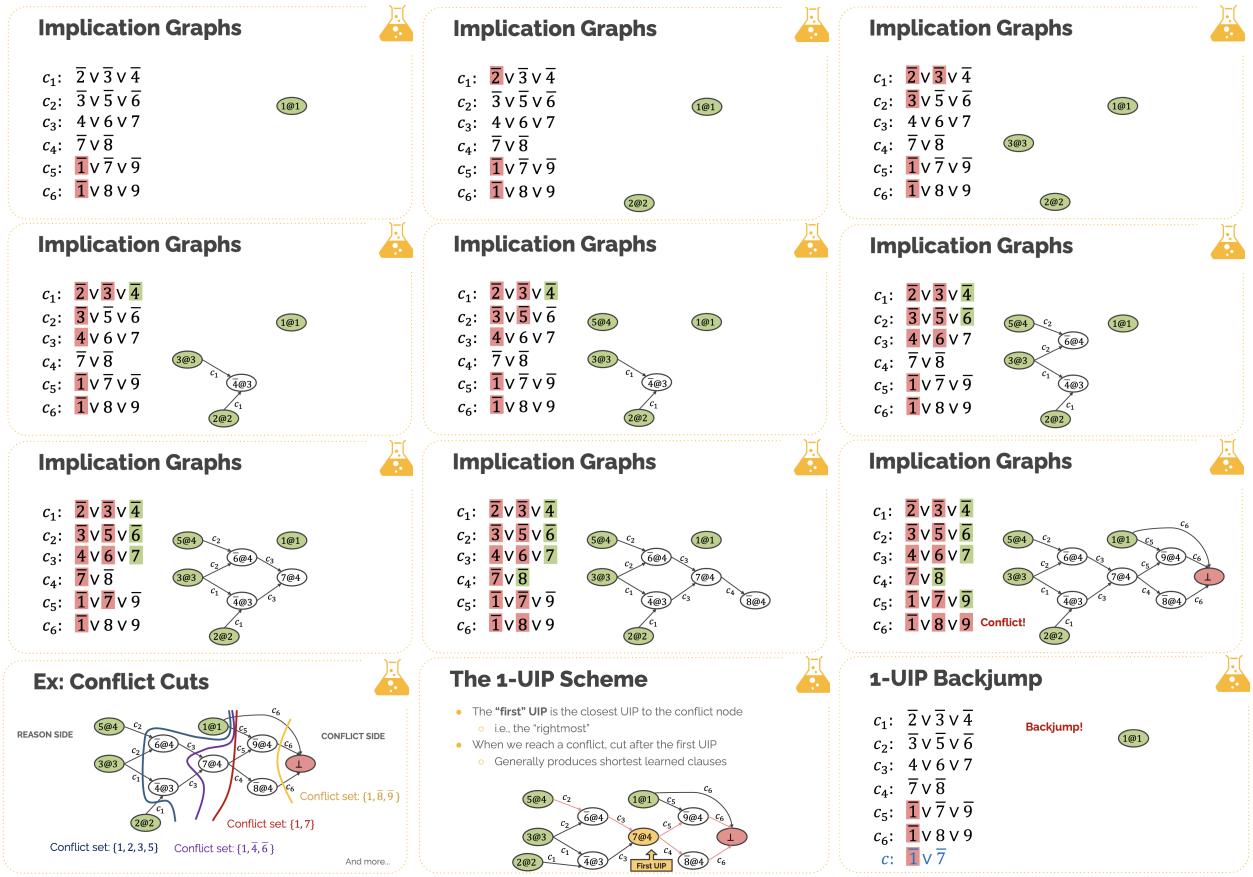

Figure 2.10: Visual representation of CDCL algorithm and implication graph on the formula f [54].

Figure 2.10 consists of 12 images, we will be referencing them from left to right and top to bottom. Each image shows the formula with its updated variable assignments on the left and implication graph on the right except for the image with heading `Ex:Conflict Cuts`, which shows the conflict cuts on the implication graph.

In this example, we pick variables to assign values in numerical order for simplicity; this is known as in-order variable selection.

Image 1: The CDCL solver first checks if there is a conflict after applying unit-propagation. In this scenario, there are no unit clauses and no conflict in the formula, so it increments the decision level, assigns variable 1 to *true* and adds it to the implication graph.

Image 2: The solver applies unit-propagation and checks if there is a conflict. In this scenario, there are no unit clauses and no conflict in the formula, so it increments the decision level, assigns variable 2 to *true* and adds it to the implication graph.

Image 3: The solver applies unit-propagation and checks if there is a conflict. In this scenario, there are no unit clauses and no conflict in the formula, so it increments the decision level, assigns variable 3 to *true* and adds it to the implication graph.

Image 4: The solver applies unit-propagation and checks if there is a conflict. In this scenario, there is no conflict, but there is 1 unit clause in the formula. As $\neg 4$ is the literal in the unit clause, variable 4 is assigned to $false$. The unit-propagation of variable 4 is due to the assignment of variables 2 and 3 to true. Consequently, there is an implication from the vertices of variables 2 and 3 to the vertex of variable 4 in the implication graph. There is no edge from the vertex of variable 1 to the vertex of variable 4, as variable 1 was not in the original clause (clause 1) of the unit clause. It is important to note that the level at which variable 4 is assigned is the same level as variable 3; this is because a new decision level is created when a variable is assigned a variable outside of unit-propagation.

Image 5: The solver applies unit-propagation and checks if there is a conflict. In this scenario, there are no unit clauses and no conflict in the formula, so it increments the decision level, assigns variable 5 to *true* and adds it to the implication graph.

Image 6: The solver applies unit-propagation and checks if there is a conflict. In this scenario, there is no conflict, but there is 1 unit clause in the formula. As $\neg 6$ is the literal in the unit clause, variable 6 is assigned to $false$. The unit-propagation of variable 6 is due to the assignment of variables 3 and 5 to true. Consequently, there is an implication from the vertices of variables 3 and 5 to the vertex of variable 6 in the implication graph.

Image 7: The assignment of variable 6 to $false$ makes clause 3 a unit clause. As 7 is the literal in the unit clause, variable 7 is assigned to *true*. The unit propagation of variable 7 is due to the assignment of variables 4 and 6 to $false$. Consequently, there is an implication from the vertices of variables 4 and 6 to the vertex of variable 7 in the implication graph.

Image 8: From the assignment of variable 7 to *true*, clause 4 becomes a unit clause. As $\neg 8$ is the literal in the unit clause, variable 8 is assigned to $false$. The unit-propagation of variable 8 is due to the assignment of variable 7 to *true*. Consequently, there is an implication from the vertex of variable 7 to the vertex of variable 8 in the implication graph.

Image 9: From the assignment of variable 7 to $true$, clause 5 becomes a unit clause. As $\neg 9$ is the literal in the unit clause, variable 9 is assigned to $false$. The unit-propagation of variable 9 is due to the assignment of variables 1 and 7 to $true$. Consequently, there is an implication from the vertices of variables 1 and 7 to the vertex of variable 9 in the implication graph. However, from the assignment of variable 9 to $false$, clause 6 becomes an empty clause, and a conflict occurs.

Image 10: The solver tries to analyse the conflict. It creates conflict cuts in the implication graphs to find conflict sets. The first conflict cut is made based on the unsatisfied clause, the first conflict set consists of the negated literals of the unsatisfied clause 1, $\neg 8$, $\neg 9$. The second conflict cut is made on the in-coming edges of the vertices of variables that are in the first conflict cut, so the second conflict set is the in-coming edges of vertices 1, 8 and 9. 1 has no in-coming edge, so it is added instead. The second conflict set is 1, 7. This carries on until it is not possible to go further back in the implication graph. In image 10 there are many conflict sets, so which to choose to generate a conflict clause? we want the learnt clause to become a unit clause right after backjumping so that we are able to apply new knowledge from the learnt clause right away, but we also want it to be informative. We want to find the UIPs and, more specifically, the first-UIP.

Image 11: In this scenario, the solver has found variable 7 to be a UIP, it is also the first UIP as it is the closest UIP to the conflict node. The conflict set, which contains the first UIP, will be used to generate the conflict clause. The learnt clause will be the negation of the literals in the conflict set, which in this scenario is $(\neg 1 \vee \neg 7)$. It is important to note that there are 3 UIP cuts in this implication graph, the 3 UIP cuts are the red, purple and blue lines shown in image 10.

Image 12: After analysing the conflict and finding a conflict clause, we add it to the original clauses, and we backjump to the second highest decision level in the conflict clause, so the conflict clause becomes a unit clause after backjumping. We backjump to variable 1, and it is set to true so that the conflict clause becomes a unit clause. The implication graph, variable assignments, and the clauses are updated according to the backjump variable and level. From here, the solver applies unit-propagation and continues the same procedure; in the end, it will find a satisfying assignment.

Problems with CDCL

Like the DPLL algorithm, the CDCL algorithm needs exponential time in the worst case. This is not feasible for very large instances of SAT. Learnt clauses help the CDCL solver effectively prune the search space and prevent it from revisiting the same conflicts. However, with a complex instance of SAT, the solver may learn a large number of clauses, which can use a lot of memory. And even with sufficient memory, as the number of clauses increases, the time needed to perform unit-propagation becomes impractical, ultimately reducing the solver's performance [40]. Information on deleting learnt clauses can be found in [40].

CDCL algorithm uses an implication graph, which can require a substantial amount of memory for large SAT instances. Updating the implication graph after each variable assignment and backjump can be a time-consuming task with large implication graphs. Furthermore, the number of possible conflict cuts increases with the size of the implication graph so computing the learnt clause will be a slow process with large implication graphs.

It seems that what makes the CDCL algorithm fast at finding a solution is also what slows it down. Inefficient implementations of the implication graph, unit-propagation, backjumping, and analyse conflict functions can be detrimental to the algorithm's speed.

Chapter 3

Design

In this chapter, we present the design of the DPLL and CDCL algorithms and discuss alternative designs. We will showcase the time and space complexity of the functions, which will be related to either n (the number of clauses in the formula), m (the number of literals in a clause), or v (the number of variables in the formula).

3.1 Requirements and Objectives

There are several objectives, requirements, and limitations for this project.

3.1.1 Objectives

- Implement the original DPLL algorithm with recursive and iterative structure.
- Implement a CDCL algorithm that sacrifices correctness for performance.
- Explore strategies enhancing the performance of the solvers through algorithmic optimisations.
- Create visualisations (e.g. tables, graphs) to show experimental results.

3.1.2 Functional Requirements

- All implementations must be able to handle SAT instances in DIMACS CNF format.
- Implement variable selection mechanisms to choose variables from the SAT formula.
- Implement unit propagation and pure literal elimination.
- Implement backtracking mechanisms for DPLL and CDCL.
- Determine the satisfiability of the input SAT instance by either finding a satisfying assignment or proving unsatisfiability.

3.1.3 Non-Functional Requirements

- The programming language we have chosen to implement the DPLL and CDCL algorithm is C++. Compared to other programming languages, C++ is faster at compilation and, most importantly, execution [76].
- The implemented solvers should be deterministic. They should be able to produce the same number of variable assignments needed to find an answer with the same inputs.

3.1.4 Considerations for Fair Comparison

- Both algorithms will have similar functions when possible.
- A diverse set of benchmark SAT instances, including real-world problems, will be used to evaluate their performance.
- Same performance metrics will be used to evaluate the implementations.
- Performance will be measured on the same hardware and software.
- Impact on performance from external will be limited.

3.2 DPLL

3.2.1 Overview

The DPLL algorithm consists of many functions. The class diagram for the DPLL algorithm can be found below. The DPLL algorithm can have different structures, such as recursive or iterative. Both designs aim to achieve the same goal of efficiently determining the satisfiability of a propositional formula.

It is important to note that many functions are designed to take advantage of pass-by-reference in C++. Please refer to [59] for more information on pass-by-reference.

Class Diagram

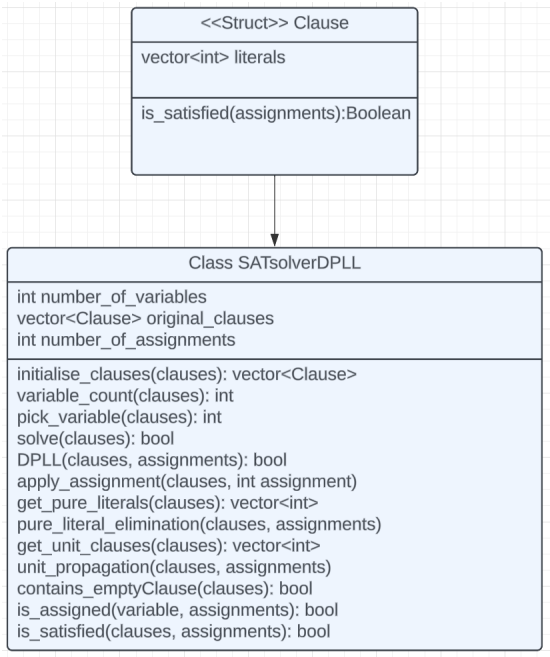

Figure 3.1: Recursive DPLL class diagram

A structure (struct) is a user-defined data type in C++. We use a structure to store all the literals in a clause to access a clause easily and its literals and efficiently check if the clause is satisfied. This structure has been created for simplicity and readability and is not a requirement of the algorithm.

It is important to note that this class diagram consists of additional functions that may not have been used in the original algorithm, such as contains-emptyClause, is-assigned, is-satisfied, etc. These

functions have been added to improve code readability and make testing easier. The iterative DPLL class diagram would consist of one additional function: backtrack.

Recursive DPLL Design

Algorithm 3 DPLL Recursive Design

```
function DPLL(φ, A)
    Inputs φ, Propositional formula. A, List of assignments
    Output Boolean

    unit-propagation(φ, A)
    pure-literal-elimination(φ, A)
    if φ is empty then
        return true
    end if
    if φ contains an empty clause then
        return false
    end if
    x ← pick-variable()
    x ← true
    if DPLL(φ ∧ x, A) = true then      ▷ φ ∧ x represents the updated formula after x is
assigned a value.
        return true
    else
        x ← false
        return DPLL(φ ∧ x, A)
    end if
end function
```

This algorithm design is similar to the one shown in 1. Both algorithms have the same functionalities, but the only difference is that instead of the while loop being in the DPLL function, it has now been moved inside the unit-propagation and pure-literal-elimination functions. This is added for simplicity, as otherwise, the DPLL function would have to check if a unit clause or pure literal exists and then apply unit propagation or pure literal elimination. Furthermore, this slight adjustment improves efficiency by applying more than one unit literal per unit-propagation function call and more than one pure literal per pure-literal-elimination function call. See 9 for the design of the unit-propagation function and 11 for the design of the pure-literal-elimination function for a better understanding of this adjustment.

Advantages of recursive design

- Increase speed: When backtracking occurs, the formula does not need to be explicitly updated, as it automatically reverts to its previous state when returning from a recursive call.
- Easy implementation: Recursive design requires fewer additional functions and fewer data structures, reducing the complexity of the implementation.
- Clarity: Recursion enhances code clarity and minimises the time required for writing and debugging code. [61].

Disadvantages of recursive design

- High memory usage: Recursive calls require stack space, which stores function parameters, local variables (this is different from variables in SAT) and return address. Depending on the function parameters and local variables, the stack may require large amounts of memory.
- Errors: If enough memory is unavailable, recursion can lead to stack overflow errors, especially for large inputs.

The recursive design does not need time to revert the formula and assignments when backtracking. However, it sacrifices memory for this improvement, as an instance of the updated formula and assignments are created for each recursive call. The recursive DPLL algorithm can be designed not to create a copy of the assignments but to update them; however, to reduce extra computation, we have not designed it in such a manner.

Important considerations regarding recursive design

- This design copies the formula for every recursion, which can be too memory-expensive. This design should be used when there is an abundant memory to reduce the risk of a stack overflow error.
- Efficient memory management and optimisations are essential to minimise memory usage and improve scalability.

Iterative DPLL Design

Although DPLL has a natural recursive structure, we can make it iterative by using a stack to store the assignments.

Algorithm 4 DPLL Iterative Design

```
function DPLL(φ,A)    Inputs φ, Propositional formula. A, List of assignments
    Output Boolean

    decision-level ← 0
    unit-propagation(φ,A)
    pure-literal-elimination(φ,A)
    if φ is empty then
        return false
    end if
    while not all variables have been assigned do
        increment decision-level
        x ← pick-variable()
        x ← true
        while unit-propagation(φ,A) = false or pure-literal-elimination(φ,A) = false do
            if decision-level = 0 then
                return false
            end if
            backtrack()
            x ← false
        end while
    end while
end function
```

This design does not use recursive calls; instead, it uses a while loop to search. Like the recursive design, the unit-propagation and pure-literal-elimination functions consist of while loops to iterate over all unit clauses and pure literals in the formula.

Advantages of iterative design

- Efficient memory usage: By not using recursive calls, the iterative design reduces stack usage, mitigating the risk of stack overflow errors.
- Easier Testing: An iterative approach makes it easier to test and verify if the algorithm is working correctly.

Disadvantages of iterative design

- Complexity: Requires addition functions, which can increase complexity.
- Decrease in speed: We must revert the formula for every backtrack.

The iterative design needs to waste time reverting the formula when backtracking, which reduces its efficiency, especially for large formulas. However, it saves memory, as a new formula instance is not needed.

Important considerations regarding iterative design

- Selecting appropriate data structures is essential for an efficient search.
- Optimisations for backtracking are essential to improve speed. Reverting the formula can be a time-consuming process if not managed efficiently.

Other DPLL variations

We have not focused on the DPLL variation with the data structures mentioned in 2.2.2 to compare the original mechanisms of the DPLL algorithm with the mechanisms of the CDCL algorithm without having data structure optimisations.

3.2.2 Variable Selection

Variable selection is an essential process for efficient SAT solving. Variable selection methods range from choosing randomly to using a heuristic [87]. In this project, we will look at two different variable selection methods, but first, we will define a function that will be used in the variable selection function.

Algorithm 5 Is Variable Assigned

```
function IS-VARIABLE-ASSIGNED(V,A)
    Inputs V: Variable. A: List of assignments
    Output Integer

    for assignment in A do
        if assignment = V or assignment = ¬V then
            return true
        end if
    end for
    return false
end function
```

The is-variable-assigned function 5 checks whether a variable has been assigned a value. This is useful so that we do not accidentally assign a variable two values. The time complexity of this algorithm is $O(v)$.

Uninformative Variable Selection

Now that we have defined a function that checks if a variable is assigned, we can build a function for variable selection using this function. We will first look at in-order variable selection.

Algorithm 6 In Order Variable Selection

```
function SELECT-VARIABLE-IN-ORDER(A)
    Inputs A, List of assignments
    Output Integer

    for k ← 1 to v do
        if is-variable-assigned(x, A) = false then
            return x
        end if
    end for
    return 0                                  ▷ All variables have been assigned a value
end function
```

The function 6 checks in the order of variable numbers in which variables have not been assigned a value. The function returns an integer, representing the variable that has not been assigned a value or 0 if all variables have been assigned a value. The time complexity of this algorithm is $O(v^2)$ as, at worst, it does a for loop for every variable, and the function is-variable-assigned(x, A) takes $O(v)$ time at worst case. Similar to this function, we could randomly select a number between 1 and the number of variables and check whether they are assigned a value. In both approaches, no information other than the values of the variables that have been assigned is used, so it is not an informative decision.

Informative Variable Selection

For a more informative approach, we can select the variable which appears the most frequently in the formula. This approach will require searching through the formula and counting the times each variable appears, as shown in 7.

Function 7, has a time complexity of $O(nm)$ as the function has to search though all the literals in every clause. The space complexity is $O(v)$ as we use a map/dictionary to store all the variables and the number of times they appear. This approach is slower than in order variable selection and random variable selection, however with this approach we may be able to satisfy more clauses and or remove many negations of the assigned value of the variable from the clauses, leading to less/smaller clauses.

Another method for variable selection is selecting the most frequent literal. Satisfying the most frequent literal will lead to the most considerable decrease in formula size. Similar to most frequent variable selection, this method can be implemented with both variations. This method, however, isn't used as it can return a negative value. In DPLL and CDCL, we usually assign a variable to true first. Then, if it leads to a conflict, we backtrack and negate the variable. It is important to note

Algorithm 7 Most Frequent Variable Selection

```
function SELECT-MOST-FREQUENT-VARIABLE(φ)
    Inputs φ, Propositional formula
    Output Integer

    variable-count ← map
    most-frequent-variable ← -1
    most-frequent-count ← 0
    for x ← 0 to n do
        for y ← 0 to m do
            variable-count[φ[x][y]] += 1      ▷ φ[x][y] represents the literal in position y of clause in position x of the formula
            if variable-count[φ[x][y]] > most-frequent-count then
                most-frequent-variable ← abs(φ[x][y])          ▷ abs returns the absolute value
                most-frequent-count ← variable-count[φ[x][y]]
            end if
        end for
    end for
    return most-frequent-variable
end function
```

that the ordered map will have double the space complexity as it will now need to store the count of all literals and their negation.

3.2.3 Unit Propagation

The DPLL algorithm spends most of its time propagating literals [43]. Thus, optimising unit-propagation and pure-literal elimination is important for increased efficiency.

We will split the unit-propagation process into two functions. One function finds the unit clauses, and the other applies them until no more unit clauses are found.

Algorithm 8 Get Unit Clauses

```
function GET-UNIT-CLAUSES(φ)
    Inputs φ, Propositional formula.
    Output List of integer

    unit-clauses ← []
    for x ← 0 to φ.size do                    ▷ traverses through all of the clauses in φ
        if φ[x].size = 1 then
            unit-clauses.add(φ[x][0])          ▷ adds the literal to unit-clauses list
        end if
    end for
    return unit-clauses
end function
```

Function 8, iterates through all the clauses, checking if any clause is of size 1; if so, it is added to the unit-clause list. The function has a time of $\theta(n)$ and space complexity of $O(n)$.

Now we can use function 8 to create the unit-propagation function.

Algorithm 9 Unit Propagation

```
function UNIT-PROPAGATION(φ, A)
    Inputs φ, Propositional formula. A, List of assignments

    unit-clauses ← get-unit-clauses(clauses)
    while unit-clauses.size > 0 do
        for x ← 0 to unit-clauses.size do
            literal ← unit-clauses[x]
            if is-assigned(literal) = false then
                φ ← apply-assignment(literal, φ)        ▷ updates φ according to the literal
                A.add(literal)
            end if
        end for
        unit-clauses ← get-unit-clauses(clauses)
    end while
end function
```

The function 9 gets the unit clauses and applies all of them; it does check if the variable has already been assigned or not to make sure that a variable is not assigned two different values. The formula is updated using the apply-assignment function 19, and the literal is added to the assignment list. The function then checks if the updated formula consists of unit clauses; if it does, it applies them; otherwise, the function ends. For the iterative design, the unit-propagation function returns a boolean value, $false$, if a conflict is found; otherwise, $true$. It is important to note that due to this small change in the design of the unit-propagation function, the iterative design won't apply more unit clauses when it finds a conflict; it will also not call the pure literal elimination function as it found a conflict. This will mean that the iterative design is likely to apply fewer assignments than the recursive design. For this variation of the unit-propagation function, modify function 9 by adding an if statement to check if there is a conflict or not after the formula has been updated. This function's time complexity depends on the apply-assignment function 19. The space complexity of this function is $O(n)$ as, at maximum, all of the clauses will be stored in the unit-clause list.

3.2.4 Pure Literal Elimination

Like unit propagation, pure literal elimination is a crucial function. Its performance will directly impact the speed of the algorithm. As it is often called in the DPLL algorithm, we need to ensure we are efficiently searching for pure literals and applying them. First, we will look at searching for pure literals; we iterative through all the literals and count the frequency of each literal every time the function is called. This searching process will have a time complexity of $\theta(nm)$. A potentially faster method to get pure literals is shown in 23.

The function 10 counts the frequency of each literal in the clause and then checks if a variable is of a single polarity.

Now that we have a function to get the pure literals, we will design a function to apply them. This function will have the same structure as the unit-propagation function, as we need to apply pure literals until there are none in the formula.

Algorithm 10 Get Pure Literals

```
function GET-PURE-LITERALS(φ)
    Inputs φ
    Output List of integer

    pure-literals = []                                                    ▷ list
    literal-count ← initialise-to-zero()                                  ▷ map
    for x ← 0 to φ.size do
        for y ← 0 to φ[x].size do
            literal-count[φ[x][y]] += 1
        end for
    end for
    for literal in literal-count do
        if literal-count[literal] > 0 and literal-count[-literal] = 0 then   ▷ check if the variable is of single polarity
            pure-literals.add(literal)
        end if
    end for
    return pure-literals
end function
```

Algorithm 11 Pure Literal Elimination

```
function PURE-LITERAL-ELIMINATION(φ)
    Inputs φ, Propositional formula

    pure-literals ← get-pure-literals(clauses)
    while pure-literals.size > 0 do
        for x ← 0 to pure-literals.size do
            literal ← pure-literals[x]
            if is-assigned(literal) = false then
                φ ← apply-assignment(literal, φ)
                A.add(literal)
            end if
        end for
        pure-literals ← get-pure-literals(clauses)
    end while
end function
```

Function 11, gets the pure literals and applies all of them, it does check if the variable has already been assigned or not to make sure that a variable is not assigned two different values. The formula is updated using the apply-assignment function 19, and the literal is added to the assignment list. The function then checks if the updated formula consists of pure literals; if it does, it applies them; otherwise, the function ends. For the iterative design, the pure-literal-elimination function returns a boolean value. If a conflict is detected, then it does not apply any more pure literals and returns $false$. Otherwise, it returns $true$. For this variation of the pure-literal-elimination function, modify function 11 by adding an if statement to check if there is a conflict or not after the formula has been updated. This function's time complexity depends on the apply-assignment function 19. The space complexity of this function is $O(v)$ as, at maximum, half of all different literals (the same as the number of variables) will be stored in the pure-literal list.

3.2.5 Back Tracking Mechanism

With recursive DPLL, no additional backtracking function is required. However, with iterative DPLL, a backtracking function is needed to return to the previous assignment. The function 12 is for backtracking in iterative DPLL.

The function 12 removes all the assignments up to the literal l; it does not just remove the prevision assignment, as that could have been due to unit propagation or pure literal elimination. Essentially, this function backtracks to the previous decision level. Once all the assignments after l have been removed and l has been removed, the function does not remove any more assignments and reverts to the previous formula by applying the variable assignments in the assignments list to the original input formula.

3.3 CDCL

3.3.1 Overview

Like the DPLL algorithm, the CDCL algorithm consists of many functions; the class diagram for CDCL is shown in Figure 3.2. We believe the updating process of the implication graph takes a substantial amount of time, which is why we will design a CDCL algorithm that sacrifices correctness for speed. Rather than updating the implication graph correctly, we will simply empty it after each backjump. This will lead to incorrect learnt clauses and therefor incorrect back jumping, however we believe this will increase the speed of the solver. We will design the CDCL algorithm such that this is the only functionality that reduces correctness, and other parts of the algorithm will be the same as those of a typical CDCL algorithm. In essence this incomplete algorithm will always correctly identify an unsatisfiability problem. However, it will fail to find solutions for some satisfiable instances.

This approach of sacrificing correctness for speed is not unusual and has been implemented in many SAT solvers such as incomplete MaxSAT [37] and GSAT (greedy SAT) [38].

The variable selection and other shared functions will be designed similarly for a fair comparison. We will not mention the designs of these functions here as they will be very similar, if not the same, to the ones shown in the design of DPLL with small changes.

It is important to note that many functions are designed to take advantage of pass-by-reference in C++. Please refer to [59] for more information on pass-by-reference.

Algorithm 12 Iterative DPLL Backtracking

function BACKTRACK(ϕ, A, D, l)

Inputs ϕ, Propositional formula. A, List of assignments. D, List of decision assignments, l, Last decision literal.

Output Integer

clauses ← original-clauses ▷ original-clauses is the original input formula, it is an attribute to the SAT solver class
for x ← 0 to D.size **do**
assignment ← D[x]
if assignment < 0 **then** ▷ if decision assignment is negated
D.remove(assignment)
end if
if assignment > 0 **then** ▷ if decision assignment is not negated
l ← assignment ▷ l is now the backtracking variable
break ▷ ends the for loop
end if
end for
if D is empty **then** ▷ if there are no more assignments to backtrack to, then we return 0, which indicates all variables have been searched through
return 0
end if
for y ← 0 to A.size **do** ▷ We must remove the assignments made after the backtracking variable
assignment ← A[x]
if assignment ≠ l **then**
A.remove(assignment)
else ▷ Once we reach the backtracking variable, we remove it so that its negation can be added in the DPLL function
A.remove(assignment)
break ▷ ends the for loop
end if
end for
for z ← 0 to A.size **do**
clauses ← apply-assignment(A[z], ϕ) ▷ clauses are updated
end for
return l
end function

Class Diagram

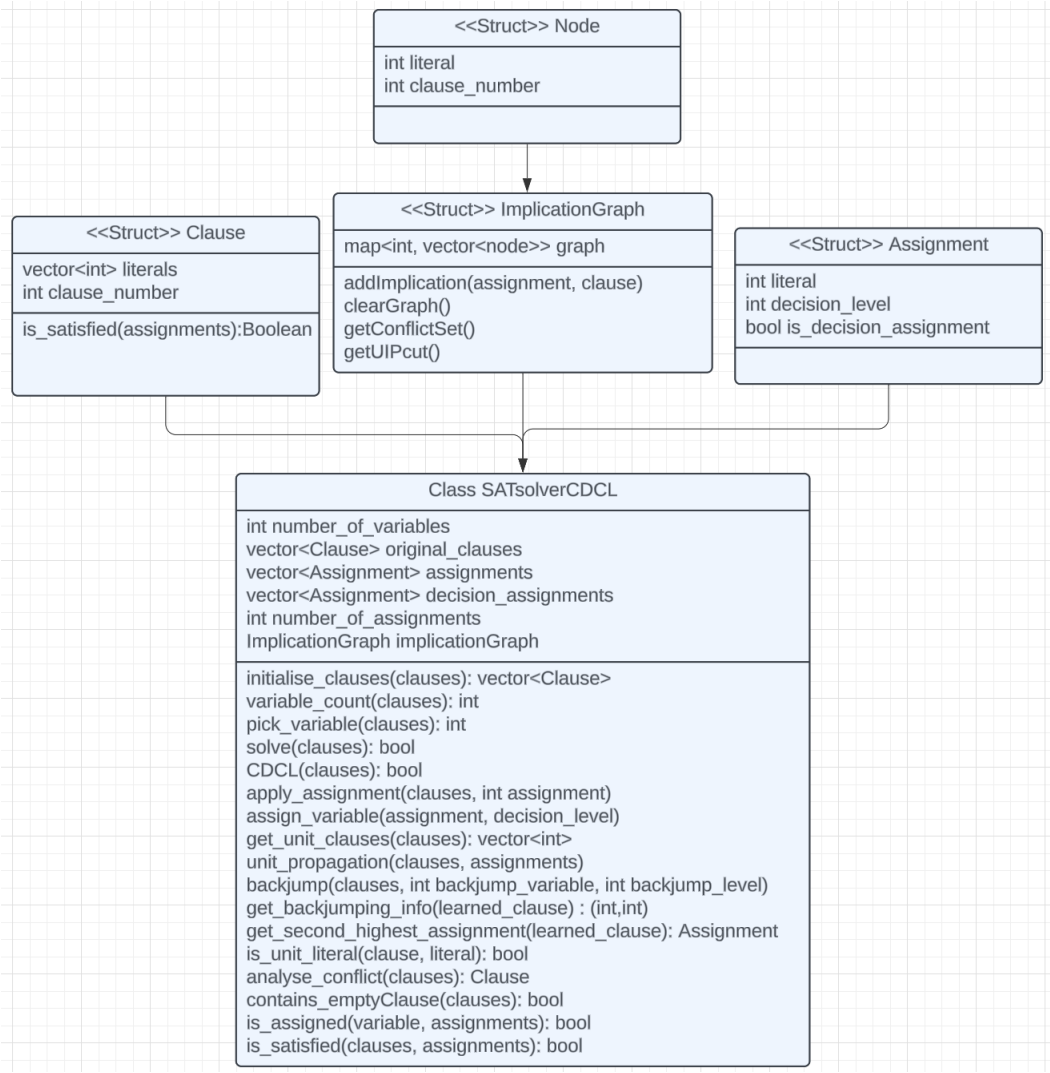

Figure 3.2: CDCL class diagram

We add three new structures: "Assignment," "ImplicationGraph," and "Node." The Assignment and Node structures have been created for simplicity and readability and are not required by the algorithm. However, the ImplicationGraph is a requirement for a CDCL algorithm as it is used for conflict analysis. It can be created within the CDCL class, but for readability and maintainability, we have created a separate structure for it. The Assignment structure stores all the assignments so that we can easily access which variable is assigned which truth value and at what decision level. The Node structure stores the nodes (vertices) of the implication graph. The ImplicationGraph structure represents the implication graph. It can add implications between the nodes, update the graph, create UIP cuts, and get the conflict set, which is then converted into the conflict clause.

As discussed in the background of CDCL, a typical CDCL algorithm does not use pure literal elimination, so it is not added to the design. However, we will implement a variation of the CDCL with pure literal elimination to evaluate its impact. The class diagram for this variant will be the same but with get-pure-literals and pure-literal-elimination functions included.

CDCL design

As with the DPLL class diagram, there are additional functions which may not be used in a typical CDCL algorithm. These functions have been added to improve code readability and to make testing

Algorithm 13 CDCL design

```
function CDCL(φ)
    Inputs φ, Propositional formula
    Output Boolean

    if φ is empty then
        return true
    end if
    if unit-propagation(φ, decision-level) = false then
        return false
    end if
    while not all variables have been assigned do
        x = pick-variable()
        x = true
        create new decision level
        while unit-propagation(φ, decision-level) = false do
            if decision level = 0 then
                return false
            end if
            learnt-clause = analyse-conflict(clauses)
            φ.add(learnt-clause)
            backjump-level = get-backjumping-info(learnt-clause)
            backjump(backjump-level)
        end while
    end while
    return true
end function
```

easier. Unlike the DPLL class diagram, we have the list of assignments as an attribute of the class. A new copy of the assignments list is made for every recursive call in recursive DPLL. This is not memory efficient. However, it saves time from updating the assignments list every time it backtracks. As a typical CDCL algorithm has an iterative design, we can leverage it to save memory and update the assignments for every backjump. As we have a memory constraint, we have decided to design the CDCL algorithm and iterative DPLL algorithm in such a manner.

The design of CDCL and iterative DPLL is very similar; however, there are a few observable differences. First, DPLL does not analyse the conflict, whereas CDCL does. From analysing the conflict, the CDCL algorithm finds a learnt clause and adds it to the original formula, whereas, with DPLL, the original formula is constant. Furthermore, CDCL is non-chronological backtracking, so it finds the back jump level and backtracks to that level. In contrast, in iterative DPLL, we backtrack chronologically and set the variable to false.

Other CDCL Designs

Various designs for the CDCL algorithms exist, with some using watched-literals data structure, heuristic search, restarting the search, and pure literal elimination, which can be found in [53], [13], [8], [83].

3.3.2 Unit Propagation

The CDCL algorithm will have a similar unit propagation function to the iterative DPLL algorithm. It will return a boolean value and add an implication to the implication graph by calling the function add-implication 21. The get-unit-clause function will also be modified to return two lists of integers instead of one. One list will consist of the unit clauses, and the other will consist of the clause numbers of the unit clauses. The modifications to both functions will not change their time complexity. However, the space complexity of the get-unit-clause function will increase as we are now storing two lists of integers. This should not be a significant increase in space complexity as, at most, the second list will take $O(n)$ space.

3.3.3 Pure Literal Elimination

Pure literal elimination is typically not implemented into the CDCL algorithm because it may not notably enhance efficiency [83]. While pure literal elimination can simplify the formula, it requires time to identify and eliminate pure literals, and it usually takes longer to search for pure literals than unit clauses. Searching and eliminating pure literals can be done in polynomial time, however if called frequently, it can be computationally expensive. Some CDCL solvers use dynamic variable ordering heuristics to decide which variable to branch on next. These heuristics may not be as efficient with pure literal elimination since pure literal elimination can change the structure of the formula. More about dynamic variable ordering heuristics can be found in [7]. It was found that when adding pure literal elimination in the CDCL solver MiniSat2.2.0, it lead to a major reduction in solver efficiency because the learnt clause management of MiniSat2.2.0 is not suitable for pure literal elimination [83].

Implications in the implication graphs are made due to a deduction of truth values for other variables from assignments. However, pure literal elimination only satisfies clauses and does not make clauses smaller, as the negation of the pure literal does not exist. This makes adding implications more complex. While pure literal elimination can be beneficial in traditional DPLL solvers, its advantages are outweighed by the strengths of techniques in CDCL solvers. This is why pure literal elimination is not commonly used in CDCL solvers.

While pure literal elimination may not be commonly used in CDCL solvers, we will implement it in our CDCL solve to experiment if it can be an optimisation or not.

The get-pure-literals function will be designed the same as shown in 10, and the pure-literal-elimination function will be designed exactly like the one designed for iterative DPLL; same as 11, but it will check if a conflict has occurred after applying the pure literal and returns a boolean value.

While it is easy to find which assignment leads to the elimination of a pure literal, it is impossible to decide the clause number of the pure literal. With unit propagation, we can easily determine the clause number of the implication by taking the clause number of the unit clause; however, there is no pure literal clause, so we cannot get the clause number. The clause number is vital as it is used to find the first-UIP. So, unlike the unit-propagation function for CDCL, we will not add implications when applying pure literals.

3.3.4 Analyse Conflict

As discussed in the background of CDCL, learning clauses is a crucial aspect of the CDCL algorithm. CDCL algorithms use an implication graph to analyse conflicts. Conflict cuts are made on the implication graphs based on UIP (Unique Implication Point) [53]. There can be many UIPs, so which do we choose? We want to learn the most from the learnt clause, so we select the UIP closest to the conflict node, known as 1-UIP (first-UIP) [53]. The last-UIP is the UIP, which is the furthest from the conflict. The last-UIP cut can be computed without an implication graph; only the list of decision assignments is needed. We will be designing an analyse-conflict function which gets the first-UIP cut or the last-UIP cut depending on the implications. More information on UIP can be found in [53], [49], [19].

In the pseudocode for CDCL algorithm 2, the analyse-conflict function returns the learnt clause and back-jump level, however we will split this process up and get the back-jump level separately from the learnt clause, we will further split the analyse conflict into more functions for increased readability of the code. First, we will design the function that will create the UIP cut and return the conflict set of that cut.

The function 14 will create the First or Last UIP cut and return the conflict set of this cut. Its time complexity will depend on which UIP cut it does. The last-UIP will be $O(v)$ as the decision nodes only need to be checked. On the other hand, the first-UIP will be much higher as it requires checking if other literals are part of the cut. Both options have the same space complexity, $O(v)$. It is important to note that this function is part of the implication graph structure, not the CDCL class.

Now that we can generate conflict cuts, we need a function to decide if we do a first-UIP cut or last-UIP cut. The last-UIP cut is computationally faster to generate. However, it is not very informative, whereas the first-UIP cut is computationally slower to generate; however, it is more informative. We will choose which UIP to take depending on the implication graph. If there is more than one path from the current decision variable to the conflict, then we choose the last-UIP. It is possible for a first-UIP to exist if there is more than one path from the current decision variable to the conflict, however, we have not designed it in such a way to save computation time from having to traverse many paths in the implication graph. It is important to note that this design does not affect the correctness of the algorithm. If there is only one path, we take the first-UIP and check that it generates a conflict set with more than one literal; if not, we take the last-UIP. This is a different way to analyse conflict. In a typical CDCL algorithm, the check for if the first-UIP cut generates a conflict set with more than one literal is not done. As we will be clearing the implication graph after each back jump,

Algorithm 14 Get UIP Cut

```
function GET-UIP-CUT(node, D, getLastUIPcut)
    Inputs node: The UIP node. D: List of decision assignments. getLastUIPcut: Boolean
    Output List of integers

    conflict-set ← empty-list
    currentNodeImplications ← graph[node]
    if getLastUIP = true then
        for x ← 0 to D.size do                ▷ Adds all the decision assignments to the conflict-set
            conflict-set.add(D[x])
        end for
        return conflict-set
    else
        for x ← 0 to D.size do
            assignment = D[x]
            for y ← 0 to graph[assignment].size do ▷ for all implications of the decision assignment literal
                for z ← 0 to currentNodeImplications.size do
                    if graph[assignment][x].clause-number = currentNodeImplications[z].clause-number then
                        conflict-set.add(assignment)
                    end if
                end for
            end for
        end for
    end if
    conflict-set.add(node.literal)                ▷ adds the UIP literal
    return conflict-set
end function
```

the first-UIP cut will potentially be different to what it would be if the implication graph is updated correctly after the back jump. This difference in the first-UIP cut will lead to the generation of wrong learnt clauses. To add some correctness to the algorithm, we have decided to take the last-UIP if there is only one literal in the first-UIP cut. The function 15 shows how the first-UIP and last-UIP cuts are chosen.

Algorithm 15 Get Conflict Set

```
function GET-CONFLICT-SET(D)
    Inputs D, List of decision assignments.
    Output List of integers

    decision-literal ← D.end()        ▷ gets the last decision in the decision-assignment (the current decision literal)
    numberOfImplications ← implicationGraph[decision-literal].size        ▷ gets the number of implications of the decision-literal
    node ← empty-node
    if numberOfDecisionAssignmentImplications ≠ 1 then
        return getUIPcut(node, decision-literal, true)        ▷ gets the last-UIP cut
    end if
    while numberOfDecisionAssignmentImplications = 1 do
        node ← traverse-Graph()
    end whileconflict-set ← getUIPcut(node, decision-literal, false)        ▷ gets the First-UIP cut
    if conflict-set.size = 1 then
        return getUIPcut(node, decision-literal, true)        ▷ gets the last-UIP cut
    elsereturn conflict-set
    end if
end function
```

The time complexity of function 15 depends on whether we get the first or last UIP. However, the space complexity is $O(v)$. It is important to note that this function is part of the implication graph structure and not the CDCL class.

The analyse-conflict function can now get the conflict set and convert it into the learnt clause by negating the literals in the conflict set as shown in 16.

Algorithm 16 Analyse Conflict

```
function ANALYSE-CONFLICT(φ)
    Inputs φ, Propositional formula
    Output Clause, the learnt clause

    learnt-clause ← empty-clause
    conflict-set ← implicationGraph.get-conflict-set(decision-assignments)
    for x ← 0 to conflict-set.size do
        learnt-clause.add(-conflict-set[x])
    end for
    return learnt-clause
end function
```

The function 16 has the same time complexity as the function get-conflict-set 15 and has a space complexity of $O(v)$.

3.3.5 Back Jumping Mechanism

We will split the back-jumping process into two functions. One function (get-backjumping-info) will compute the back-jumping literal and the back-jumping level, whereas the other (backjump) will revert the clauses and assignments.

Algorithm 17 Get Backjumping Information

```
function GET-BACKJUMPING-INFO(learnt-clause)
    Inputs learnt-clause
    Output back-jump literal, integer. back-jump level, integer.

    if learnt-clause.size = 1 then
        return (learnt-clause[0], 0)
    else
        assignment ← get-second-highest-assignment(learnt-clause) ▷ gets the assignment with the second highest decision level in the clause
        return (assignment.literal, assignment.level)
    end if
end function
```

The function 17 returns the back jump literal and back jump level. If the learnt-clause is of size one, it returns the literal in the learnt clause and level 0. If learnt-clause is of size greater than 1, then it gets the second highest literal in the learnt clause (i.e. the literal in the learnt clause with the second highest assignment level) so that the highest literal becomes a unit literal after backjumping. This function's time complexity depends on the size of the learnt clause and how quickly it gets the second highest assignment; this can be done in $O(n)$ if implemented correctly. The space complexity of this function is none if the learnt clause is of size 1. Otherwise, it is constant $O(1)$ as we only store one assignment.

Now that we can get the back jump variable and level, we can design a function which reverts the solver to that level. Below is the design for this function.

Algorithm 18 Backjump

```
function BACKJUMP(φ, backjump-level)
    Inputs φ: Propositional formula. backjump-level: integer.

    new-clauses ← original-clauses
    for y ← 0 to assignments.size do
        assignment ← assignments
        if assignment.level > backjump-level then
            assignments.remove(assignment)
        else
            apply-assignment(new-clauses, assignment.literal)
        end if
    end for
    decision-assignments ← update-decision-assignments()
    implicationGraph ← empty
    φ ← new-clauses
end function
```

The function 18 reverts the assignments, decision assignments and clauses to the back jumping level. This would also be done in a typical CDCL algorithm, however what is different is that we do not update the implication graph according to the back jump level, instead we empty the graph. The time complexity of the backjump function depends mainly on the apply-assignment function 19; the updating decision assignments should take $O(v)$ as we will be going through the assignments list and checking if an assignment is a decision assignment. The space complexity of this function would be $\theta(nm)$ as we go back to the original clause and apply the assignments again. This process can be made more time efficient if we keep a new copy of the clauses after each assignment, however this would not be memory efficient.

There are not many variations to the backjump function 18, unless we use different data-structures such as watched-literals, which will have a different reverting procedure.

3.4 Other functions

Apply Assignment

The apply-assignment function 19 traverses through ϕ and checks if it contains the newly assigned literal, if it does then the clause is removed from ϕ as it is satisfied. If the clause contains the negation of the assigned literal, then the negation of the assigned literal is removed from the clause as it cannot be satisfied.

Algorithm 19 Apply Assignment

```
function APPLY-ASSIGNMENT(φ, assignment)
    Inputs φ: Propositional formula. assignment: integer.

    for x ← 0 to φ.size do
        clause ← φ[x]
        for y ← 0 to clause.size do
            if clause[y] = assignment then
                φ.remove(clause)
                break                                   ▷ ends the for loop
            end if
            if clause[y] = -assignment then
                clause.remove(clause[y]) ▷ removes the literal that is the negation of the assignment
            end if
        end for
    end for
end function
```

The function 19 has the time complexity of $O(nm)$ as it can end up checking every literal of every clause. It has no space complexity as it does not store anything.

Initialise Clauses

We initialise clauses by converting them from a vector<vector<int>> (2D vector) to vector<Clause> type. The function below shows this process.

The function 20 is modified for the CDCL algorithm; it also assigns a unique clause number to each clause. The clause numbers are from 0 to n-1. This function has a time complexity of $O(n)$ and space complexity of $O(nm)$.

Algorithm 20 Initialise Clauses

```
function INITIALISE-CLAUSES(φ)
    Input φ: Propositional formula of type 2D vector.
    Output φ: Propositional formula of type vector<Clause>

    clause-list ← empty                                  ▷ empty vector<Clause>
    for x ← 0 to φ.size do
        clause ← empty                                   ▷ empty Clause
        clause.literals ← φ[x]
        clause-list.add(clause)
    end for
end function
```

Add Implication

Add-implication is a function of the implication-graph structure. It adds an implication between a node and other nodes. The function below shows this process.

Algorithm 21 Add Implication

```
function ADD-IMPLICATION(a, clause)
    Input a: Current assignment of type Assignment. clause: the original clause of the unit clause of the type Clause.

    node ← empty                                         ▷ empty Node
    node.literal = a.literal
    node.clause-number = clause.clause-number
    for x ← 0 to clause.size do      ▷ the literals in the original clause which implied the current assignment to be unit literal
        literal ← clause[x]
        if literal ≠ a.literal then
            if node not in graph[-literal] then          ▷ making sure to not add duplicates
                graph[-literal].add(node)                ▷ the negations of the literals in the original clause are giving an implication to the current assignment, as they would need to be unsatisfied for the current assignment to be a unit clause.
            end if
        end if
    end for
end function
```

The function 20 has a time complexity of $O(mv)$ as the for loop has m iterations, and it takes $O(v)$ time to check if a node is not in the implication list of the literal. The space complexity is $O(m)$ as, at most, we add the node m times.

3.5 Data Structures

In this section, we justify the selection of data structures used in the design of the SAT solvers and discuss the custom data structures developed specifically for them. Some data structures address each algorithm's unique needs and requirements, while others improve readability and maintainability.

3.5.1 Justification of Data Structure Choices

The data structures for our SAT solver designs were carefully selected to optimise performance and reduce memory wastage. Below are the key justifications for data structure choices:

- Vector: Vectors have a constant time to access elements and add elements to the front or back [12], which is crucial for efficiently storing and accessing clauses and assignments.
- Unordered-map: Unordered maps have a constant time average complexity for insertion, deletion, and lookup operations [77], ensuring efficient identification of pure literals and conflict analysis.

While the chosen data structures offer significant benefits, they have trade-offs and limitations. Vectors are inefficient when inserting or deleting elements in the middle. It is also inefficient at searching if an element exists [12]. If these operations are used frequently,, this could reduce performance. While unordered maps provide an efficient way to look up elements, they are difficult to iterate through in a specific order as elements are not stored in any particular order.

Alternative data structures, such as linked lists and sets, were considered. Linked lists are less suitable due to their linear access and deletion by index time complexity, which could hinder performance, especially when accessing and deleting elements in a large implication graph. Sets or unordered sets can be used instead of vectors; however, they do not have a constant time to access elements by index, which hinders performance as we often access elements by index.

3.5.2 Custom Data Structures

We used standard C++ data structures and designed our own data structures to personalise their functionality. Here are the data structures we created:

- Clause Structure: We defined a structure for a clause consisting of a vector of integers to store the literals within each clause and a clause number to distinguish different clauses. However, the clause number is not used in DPLL design.
- Assignment Vector/Structure: We defined a structure for Assignment for CDCL design consisting of an integer, which represents the literal; an integer level, which represents the decision level of the assignment; and a boolean, which represents if the assignment is a decision assignment or not. This structure is unnecessary for DPLL design, so we used a vector of integers instead to store the assigned literals.
- Node Structure: We defined a structure for Node for CDCL design consisting of two integers: literal and clause-number. This structure is used to store vertexes in the graph.
- Implication-Graph Structure: We defined a structure for Implication-Graph for CDCL design consisting of an unordered map to represent the graph. It consists of functions that include add-implication, get-UIP-cut, and get-conflict-set. The DPLL design does not require an implication graph.

Chapter 4

Implementation

4.1 Programming Environment

As mentioned in the non-functional requirements, the programming language used to implement the designs is C++ due to its speed [76] and extensive library collection.

We have used the platform Replit [62], an online integrated development environment (IDE), to implement the designs. We chose Replit for its accessibility and integrated version control systems. Replit is a browser-based IDE that allows me to work on the project from different locations with other devices without needing installation or setup. Additionally, the integrated version control system ensured smooth version management and tracking of changes.

We used standard C++ libraries such as algorithm, chrono, fstream, iostream, vector, and unordered-map to streamline various operations, such as time measurement, file handling, and data storage.

The clause, assignment, and node structures have been developed to increase code readability and maintainability. The implication graph structure has been developed to represent the implication graph efficiently. The structure includes specialised functions, which are requirements for the conflict analysis of this particular CDCL algorithm.

4.2 File structure

The CDCL and DPLL files contain some functions that are the same, yet we created distinct files. By maintaining distinct files for each solver variant, clarity and readability are enhanced. This further allows for changes or optimisations to be made to one solver without impacting the other.

```
SAT_Solvers/
|
|— DPLLRecursive.cpp      # C++ source file for recursive DPLL solver
|— DPLLIterative.cpp      # C++ source file for iterative DPLL solver
|— CDCL.cpp         # C++ source file for CDCL solver
|— DPLL_Tests.cpp         # C++ source file for DPLL solver unit, integration and stress tests
|— CDCL_Tests.cpp         # C++ source file for CDCL solver unit, integration and stress tests
|— DIMACS-CNF-input.txt    # Text file containing DIMACS CNF input for SAT instances
|—Readme.md              # Instructions on running the code
```

Figure 4.1: File structure

We have created a separate file to store the DIMACS CNF input. This offers significant benefits over manually entering the input directly into the code. Firstly, this separation enhances code readability and maintainability, as we can easily identify and modify input data without going through the solver implementation details. Furthermore, storing input data in a separate file enhances the

ease of experimentation; we can simply replace the old DIMACS CNF input with the new one for each experiment. Another advantage is the ability to reuse and share the input data across different variations of the algorithms. For example, the input can be used in all solver files without adding the input to each file.

4.3 DPLL

We have implemented two variations of the DPLL algorithm: recursive and iterative. Along with them are the functions each variation requires. Most functions are the same or very similar, with minor changes for a fair evaluation of the variations.

4.3.1 Recursive DPLL

Figure 4.2 is the implementation of algorithm 3.

```
bool DPLLRecursive(vector<Clause> clauses, vector<int> assignments) {
  unit_propagation(clauses, assignments);
  pure_literal_elimination(clauses, assignments);
  if (clauses.size() == 0) { //no more clauses left then the formula is solved
    return true;
  }

  if (contains_emptyClause(clauses)) { //if there is an empty clause then it means
there is a conflict.
    return false;
  }

  int assignment = pickVariableInOrder(assignments); // can be changed to
pickMostFrequentVariable
  if (assignment == 0) {
    return false;
  }
  assignments.push_back(assignment); // not a stack, but a queue.
  vector<Clause> clauses_duplicate = clauses; // need a duplicate clause to check if
the positive assignment can be used to find a satisfying assignment and to not have to
revert the clause after not finding a solution.
  apply_assignment(clauses_duplicate, assignment);
  if (DPLLRecursive(clauses_duplicate, assignments)) {
    return true;
  }
  remove(assignments, assignment);
  assignments.push_back(-assignment); // negation of the assignment is added.
  apply_assignment(clauses, -assignment);
  return DPLLRecursive(clauses, assignments); // no more variation for the assignment
so we return what ever the recursive function returns.
}
```

Figure 4.2: DPLL Recursive Implementation

An additional check has been added to see if the assignment is 0, meaning all variables have been assigned a value. Furthermore, we duplicate the input clauses to traverse down one path. If we cannot find a satisfying assignment, we do not have to revert the formula; we can negate the assignment and use the input clauses to traverse the other path.

4.3.2 Iterative DPLL

Figure 4.3 is the implementation of algorithm 4.

```
bool DPLLIterative(vector<Clause>& clauses, vector<int>& assignments) {
  int decision_level = 0;
  unit_propagation(clauses, assignments);
  pure_literal_elimination(clauses, assignments);
  if (clauses.size() == 0) { //no more clauses left then the formula is solved
    return true;
  }
  vector<int> decision_assignments; // assignments that were not unit propagated or pure literal eliminated.
  if (contains_emptyClause(clauses)) { //if there is an empty clause then it means there is a conflict.
    return false;
  }
  while (assignments.size() != number_of_variables) {
    decision_level++;
    int assignment = pickVariableInOrder(assignments); // can be changed to pickMostFrequentVariable
    assignments.insert(assignments.begin(),assignment); // assignments are stored in a stack format using a vector.
    decision_assignments.insert(decision_assignments.begin(),assignment);
    apply_assignment(clauses, assignment);
    number_of_assignments++;
    while (unit_propagation(clauses, assignments) == false || pure_literal_elimination(clauses, assignments) == false) { // if you do
while unit_propagation(clauses, assignments) == false || pure = pure_literal_elimination(clauses,assignments) == false) it takes a
different path, pure doesnt even get applied if unit_propagation returns false. This is due to programming language logic.
      if (decision_level == 0) {
        return false;
      }
      assignment = backtrack(clauses, assignments, decision_assignments, assignment); // get the assignment to back track to
      if (assignment == 0) { // if there are no more assignements to backtrack to then assignment is 0 and therefor we return false.
        return false;
      }
      if (is_assigned(assignment, assignments)){ // another check, even though it wont be assigned, but just to be sure.
        cout<<"already assigned"<<endl;
        return false;
      }
      assignments.insert(assignments.begin(),-assignment); // add the negation of the assignment.
      decision_assignments.insert(decision_assignments.begin(),-assignment);
      decision_level = decision_assignments.size(); // updates the decision level
      apply_assignment(clauses, -assignment); // update the clauses
    }
  }
  return is_satisfiable(original_clauses, assignments); // double checks that the assignment can satisfy the original clauses.
}
```

Figure 4.3: DPLL Iterative Implementation

An additional variable, "decision-assignments," has been added to track assignments made outside of unit propagation and pure literary elimination. While it is of type vector, it is used like a stack, insert on top and remove from top. We have not used the stack data structure, as we would have to remove each item from the stack to traverse it and then add it back when we are done. This additional variable is utilised in the backtrack function. Furthermore, we access elements by index with the decision-assignments variable, and it is not possible to access elements by index with a stack.

4.3.3 Variable Selection

Figures 4.4, 4.5 and 4.6 are the implementation of algorithms 5, 6 and 7, respectively.

```
bool is_assigned(int variable, vector<int> &assignments) { // checks if a variable is
assigned.
  for (int x = 0; x < assignments.size(); x++){
    if (assignments[x] == variable || assignments[x] == -variable){
      return true;
    }
  }
  return false;
}
```

Figure 4.4: Is Variable Assigned

The is-assigned function checks whether a variable has been assigned a value.

```
int pickVariableInOrder(vector<int>& assignments) { // returns the next unassigned
variable in the order of the variable numbers.
  for (int x = 1; x <= number_of_variables; x++) {
    if (!is_assigned(x, assignments)) {
      return x;
    }
  }
  cout << "No more variables to assign" << endl;
  return 0;
}
```

Figure 4.5: Select Variable In Order

The pickVariableInOrder function returns the next variable in order (1,2,3...) that has not been assigned a value.

```
int pickMostFrequentVariable(vector<Clause> &clauses){ // searches through the clauses
and returns the most frequent variable.
  unordered_map<int,int> variable_count;
  int most_frequent_variable = 0;
  int most_frequent_count = 0;
  for (int x = 0; x < clauses.size(); x++){
    for (int y = 0; y < clauses[x].literals.size(); y++){
      variable_count[abs(clauses[x].literals[y])] += 1;
      if (variable_count[abs(clauses[x].literals[y])] > most_frequent_count){
        most_frequent_variable = abs(clauses[x].literals[y]);
        most_frequent_count = variable_count[abs(clauses[x].literals[y])];
      }
    }
  } // as the clauses are updated, the most_frequent_variable does not need to be
checked if it is assigned or not.
  return most_frequent_variable;
}
```

Figure 4.6: Select Most Frequent Variable

The pickMostFrequentVariable function returns the next variable that appears the most in the clauses (formula).

4.3.4 Unit Propagation

Figures 4.7 and 4.8 are the implementation of algorithms 8 and 9, respectively.

```
vector<int> get_unit_clauses(vector<Clause> clauses) { // returns the unit clauses
  vector<int> unit_literals = {};
  for (int x = 0; x < clauses.size(); x++) { // searches through all the clauses
    if (clauses[x].literals.size() == 1) { // checks if the clause has only 1 literal
      unit_literals.push_back(clauses[x].literals[0]);
    }
  }
  return unit_literals;
}
```

Figure 4.7: Get Unit Clauses

The get-unit-clauses function returns the unit clauses by searching through all of the clauses and checking if they are of size 1.

```
void unit_propagation(vector<Clause>& clauses, vector<int>& assignments) { // checks if a unit clause
exists and if it does then removes it from the clauses, stops when there are no more unit clauses.
  vector<int> unit_clauses = get_unit_clauses(clauses);
  while (unit_clauses.size() > 0) { // unit propagation
    for (int literal : unit_clauses) {
      if (!is_assigned(literal, assignments)) {
        apply_assignment(clauses, literal);
        assignments.insert(assignments.begin(),literal);
      }
    }
    unit_clauses = get_unit_clauses(clauses);
  }
}
```

Figure 4.8: Unit Propagation For Recursive DPLL

The unit-propagation function applies (satisfies) the unit clauses until no more unit clauses exist. See D.1 for unit propagation for iterative DPLL.

4.3.5 Pure Literal Elimination

Figures 4.9 and 4.10 are the implementation of algorithms 10 and 11, respectively.

```
vector<int> get_pure_literals(vector<Clause> &clauses) { // searches through all the clauses.
  vector<int> pure_literals = {};
  unordered_map<int, int> literal_count = {};
  for (int i = 1; i <= number_of_variables; i++) { // initialise the literal count to 0
    literal_count[i] = 0;
    literal_count[-i] = 0;
  }
  for (int x = 0; x < clauses.size(); x++) { // count the number of each literal in the clauses
    for (int y = 0; y < clauses[x].literals.size(); y++) {
      literal_count[clauses[x].literals[y]]++;
    }
  }
  for (auto pair : literal_count) { // check which literal appears and its negation does not
appear in any clause.
    if (literal_count[pair.first] > 0 && literal_count[-pair.first] == 0) {
      pure_literals.push_back(pair.first);
    }
  }
  return pure_literals;
}
```

Figure 4.9: Get Pure Literals

The get-pure-literals function searches through the clauses and counts the frequency of each literal. It then checks if the variable has a single polarity; if it does, it gets added to the pure literals list.

```
void pure_literal_elimination(vector<Clause> &clauses, vector<int> &assignments) { // checks if
a pure literal exists and if it does then it is applied and stops when there are no more pure
literals.
  vector<int> pure = get_pure_literals_naive(clauses);
  while (pure.size() > 0) { // pure_literal_elimination
    for (int literal : pure) {
      if (!is_assigned(literal, assignments)) { // if the pure literal is not assigned, only
then it is assigned.
        apply_assignment(clauses, literal); // updates the clauses
        assignments.insert(assignments.begin(),literal); // adds pure literal to the assigments.
      }
    }
    pure = get_pure_literals_naive(clauses);
  }
}
```

Figure 4.10: Pure Literal Elimination For Recursive DPLL

The pure-literal-elimination function applies (satisfies) the pure literals until no more pure literals exist. See D.2 for pure literal elimination for iterative DPLL.

4.3.6 Back Tracking Mechanism

Figure 4.11 is the implementation of algorithm 12.

```
// returns the assignment to backtrack to
int backtrack(vector<Clause> &clauses, vector<int> &assignments,vector<int>& decision_assignments, int assignment) {
  clauses = original_clauses;
  for (int y = 0; y < decision_assignments.size(); y++){
    int assign1 = *next(decision_assignments.begin(), y);
    if (assign1 < 0){ // if the previous decision assignment is negative we dont want to go back and negate the
assignment as then we would going in a loop, we want to back track to the assignment that is positive and negate it.
      remove(decision_assignments,assign1);
      y--;
    }
    else if (assign1 > 0){
      assignment = assign1; // once a positive assignment is found we can backtrack to that assignment.
      break;
    }
  }
  if (decision_assignments.size() == 0){ // if there are no more assignments to backtrack to then return 0.
    return 0;
  }
  for (int x = 0; x < assignments.size(); x++) {
    int assign = *next(assignments.begin(), x);
    if (assign != assignment) { // removes all the assignments up to the backtracking assignment.
      remove(assignments,assign);
      x--;
    } else { // we want to remove the backtracking assignment so that we can add its negation later.
      remove(assignments,assignment);
      remove(decision_assignments,assignment);
      break;
    }
  }
  for (int k = 0; k < assignments.size(); k++) {
    int assignment = *next(assignments.begin(), k);
    apply_assignment(clauses, assignment); // clauses are reverted back
  }
  return assignment;
}
```

Figure 4.11: Backtrack

The backtrack function chronologically reverts the clauses, assignments, and decision assignments. It is only used in iterative DPLL.

4.4 CDCL

We have implemented one variation of the CDCL algorithm. The unit propagation and pure literal elimination function for CDCL can be found in appendix D.

```
bool CDCL(vector<Clause> clauses) {
  int decision_level = 0;
  if (clauses.empty()) { //no more clauses left then the formula is solved
    return true;
  }
  if (unit_propagation(clauses, decision_level) == false) { //if there is a conflict.
    return false;
  }
  while (assignments.size() != number_of_variables) { // while not all variables have been
assigned a value.
    int literal = pickVariableInOrder(); //could also pick most frequent variable, literal is true
and both functions return positive values only.
    decision_level++; // new decision level
    Assignment currentAssignment = {literal, decision_level, true};// creates a new assignment and
as it is a decision assignment it sends true as well.
    number_of_assignments++;
    apply_assignment(clauses, literal); // clauses are updated with given assigned literal.
    assignments.push_back(currentAssignment);
    decision_assignments.push_back(currentAssignment);
    while (unit_propagation(clauses, decision_level) == false) {
      if (decision_level == 0) {
        return false;
      }
      Clause learned_clause = analyse_conflict(clauses);
      learned_clause.clause_number = original_clauses.size();
      original_clauses.push_back(learned_clause); // add learned clause to the original clauses.
      pair<int, int> backjump_info = get_backjumping_info(learned_clause);
      int backjump_variable = backjump_info.first;
      int backjump_level = backjump_info.second;
      backjump(clauses, backjump_variable, backjump_level); // revert the clauses and assignments
to back jump level.
      decision_level = backjump_level;
    }
  }
  return true; //satisfying assignments have been found
}
```

Figure 4.12: CDCL Function

Figure 4.12 is the implementation of algorithm 13. The CDCL function attempts to solve the SAT problem.

4.4.1 Analyse Conflict

Figures 4.13, 4.14 and 4.15 are the implementation of algorithms 14, 15 and 16, respectively.

```
vector<int> getUIPcut(Node& node, vector<Assignment>& decision_assignments, bool getLastUIPcut){
  vector<int> conflict_set;
  vector<Node> currentNodeImplications = graph[node.literal]; // implications of the current node,
which is the UIP node if it exists.
  if (getLastUIPcut){ // gets the last UIP cut
    for (int x = 0; x < decision_assignments.size(); x++){
      conflict_set.push_back(decision_assignments[x].literal);
    }
    return conflict_set;
  }
  else{ //gets the first UIP cut
    for (Assignment assign : decision_assignments){ // for all decision assignments
      for (int x = 0; x < graph[assign.literal].size(); x++){ // for all implications of the
decision assignment literal
        for (int y = 0; y < currentNodeImplications.size(); y++){
          if (graph[assign.literal][x].clause_number == currentNodeImplications[y].clause_number){
// if the implication of the assignment is in the same clause as the current node implication.
            if (!is_in_list(conflict_set, assign.literal)){ // no need to add assignments if they
are already in the conflict set.
              conflict_set.push_back(assign.literal);
            }
          }
        }
      }
    }
  }
  conflict_set.push_back(node.literal); // add the UIP literal to the conflict set.
  return conflict_set;
}
```

Figure 4.13: Get UIP Cut

The getUIPcut function creates the first or last UIP cut on the implication graph, depending on the arguments given to it. To ensure code reuseability, implementation to get both cuts has been added to the same function; however, it can be split into two functions.

```
vector<int> getConflictSet(vector<Assignment>& decision_assignments){
  int decision_literal = decision_assignments.back().literal;
  int implications = graph[decision_literal].size(); // implications of the current decision literal.
  Node node;
  if (implications != 1){ // if there is more than 1 or no path to the conlict.
    return getUIPcut(node, decision_assignments, true); // get conflict set with last UIP cut.
  }
  node = graph[decision_literal][0]; //node of the implication from the decision literal.
  int current_literal = decision_literal;
  while (implications == 1) { // while there is only one path to the conflict keep traversing.
    node = graph[current_literal][0];
    implications = graph[node.literal].size();
    current_literal = node.literal;
  }
  vector<int> conflict_set = getUIPcut(node, decision_assignments, false); //get conflict set with the
first UIP cut.
  if (conflict_set.size() == 1){ // if conflict set has only 1 literal from first UIP cut then use Last
UIP cut.
    return getUIPcut(node, decision_assignments, true);
  }
  return conflict_set;
}
```

Figure 4.14: Get Conflict Set

The getConflictSet function uses first-UIP or last-UIP methods depending on the implications and gets the conflict set using the implication graph.

```
Clause analyse_conflict(vector<Clause>& clauses) { //generates the learned clause from the
conflict.
    Clause learned_clause;
    vector<int> conflict_literals = implicationGraph.getConflictSet(decision_assignments);
    for (int y = 0; y < conflict_literals.size(); y++) { // the learned clause is the negation of
the conflict literals.
      learned_clause.literals.push_back(-conflict_literals[y]);
    }
    return learned_clause;
  }
```

Figure 4.15: Analyse Conflict

The analyse-conflict function gets the conflict literals from the implication graph and negates them to create the learnt clause, which is returned.

4.4.2 Back Jumping Mechanism

Figure 4.16 and Figure 4.17 are the implementations of algorithms 17 and 18.

```
pair<int, int> get_backjumping_info(Clause& learned_clause) {
    if (learned_clause.literals.size() == 1) { // if the learned clause is a unit clause then return
the literal and go to decision level 0.
      return make_pair(learned_clause.literals[0], 0);
    }

    Assignment assignment = get_second_highest_assignment(learned_clause); //gets the second highest
assignment in the learned clause.
    return make_pair(assignment.literal, assignment.level);
  }
```

Figure 4.16: Get Backjump Info

The get-backjumping-info function computes the variable and level the solver needs to backjump to.

```
void backjump(vector<Clause> &clauses, int backjump_variable, int backjump_level) { //revert to the
backjump level
    vector<Clause> new_clauses = original_clauses; // copy of the original clauses.
    decision_assignments.clear();
    for (int x = 0; x < assignments.size(); x++) { // for all the assignments in the assignment vector
      Assignment assignment = assignments[x];
      if (assignment.level > backjump_level) { // remove all assignments that are at a level higher
than the backjump level.
        assignments.erase(assignments.begin() + x);
        x--;
      } else { // if the assignment is below the backjump literal level then apply the assignment to
the new clauses
        apply_assignment(new_clauses, assignment.literal);
        if (assignment.is_decision_assignment) {
          decision_assignments.push_back(assignment);
        }
      }
    }
    implicationGraph.clearGraph();
    clauses = new_clauses; // set clauses to new clauses
  }
```

Figure 4.17: Backjump

The backjump function reverts the clauses, assignments and decision assignments according to the backjump variable and level. It also clears the implication graph.

4.5 Other functions

This section shows the implementation of the apply-assignment, initialise-clauses and add-implication functions.

Apply Assignment

Figure 4.18 is the implementation of algorithm 19.

```
void apply_assignment(vector<Clause>& clauses, int assignment){ // applies the assignment to the clauses.
  for (int x = 0; x < clauses.size(); x++){
    for (int y = 0; y < clauses[x].literals.size(); y++){
      if (clauses[x].literals[y] == assignment) { // if the literal is in the clause then remove the clause.
        clauses.erase(clauses.begin() + x);
        --x;
        break;
      }
      else if (clauses[x].literals[y] == -assignment){ // if the negation of the literal is in the clause
then remove the negated literal.
        clauses[x].literals.erase(clauses[x].literals.begin() + y);
        y--;
      }
    }
  }
}
```

Figure 4.18: Apply Assignment

The apply-assignment function updates the clauses according to the assignment.

Initialise Clauses

Figure 4.19 is the implementation of algorithm 20.

```
vector<Clause> initialise_clauses(vector<vector<int>> &clauses) { // vector<vector<int>> is converted into
vector<Clause>. The clause numbers are also added.
  vector<Clause> clause_list = {};
  for (int x = 0; x < clauses.size(); x++) {
    Clause clause;
    clause.literals = clauses[x];
    clause_list.push_back(clause);
  }
  return clause_list;
}
```

Figure 4.19: Initialise Clauses

The initialise-clauses function converts the formula from a 2D vector to a 1D vector of type Clause. This function is the implementation for the DPLL. For the CDCL, add a unique clause number to each clause from 0 to n-1, where n is the number of clauses; this can be done by setting the clause number of each clause to x in the for loop.

Add Implication

Figure 4.20 is the implementation of algorithm 21.

```
void addImplication(Assignment currentAssignment, Clause clause) {
  Node node;
  node.literal = currentAssignment.literal;
  node.clause_number = clause.clause_number;
  for (int literal : clause.literals){ // the literals in the clause which implied the currentAssignment to
be unit, their negations are giving an implication to the currentAssignment, as they would need to be
unsatisfied for the currentAssignment to be a unit clause.
    if (literal != currentAssignment.literal){
      bool add = true;
      for (auto &assignment : graph[-literal]){
        if (assignment.literal == node.literal){ // if the node is already in the implications of the
literal, then we don't need to add it again.
          add = false;
        }
      }
      if (add){
        graph[-literal].push_back(node);
      }
    }
  }
}
```

Figure 4.20: Add Implication

The add-implication algorithm creates an implication in the graph between assigned variables.

Chapter 5

Experimentation and Evaluation

5.1 Experimental Setup

Each experiment was performed on a MacBook Air 2020 with a M1 processor. It has an 8-core CPU with four high-performance cores and four high-efficiency cores and is equipped with 8GB of RAM [45].

We executed the code using the Replit platform. It is important to note that while Replit is a free IDE, it limits your CPU power and memory to 50% of its max capacity. This limit can be removed by buying a Replit subscription, as done in this project. However, this is not required, and other IDEs can be used.

The inputs were stored in a separate file and represented in DIMCAS CNF format, a standard format for encoding SAT problems.

5.2 Performance Metrics

The performance metrics used:

- Runtime: Measure the time the solver takes to find a solution or determine unsatisfiability.
- Memory usage: Peak memory usage by the solver during the solving process.
- Number of assignments: The total number of assignments made to variables during the solving process.
- Correctness: The number of correct answers divided by the number of experiments.

Note that the correct answers mean whether the SAT solver correctly determines whether an SAT problem is satisfactory or not.

5.3 Solver Configurations

We evaluate many configurations of the DPLL and CDCL algorithms. Below is the list of configurations.

- dpll-R-1: Recursive DPLL algorithm with in-order variable selection.
- dpll-R-2: Recursive DPLL algorithm with most frequent variable selection.
- dpll-I-1: Iterative DPLL algorithm with in-order variable selection.

- dpll-I-2: Iterative DPLL algorithm with most frequent variable selection.
- cdcl-1: CDCL algorithm with in-order variable selection.
- cdcl-2: CDCL algorithm with most frequent variable selection.
- cdcl-3: CDCL algorithm with in-order variable selection and pure literal elimination.
- cdcl-4: CDCL algorithm with most frequent variable selection and pure literal elimination.

5.4 Experimental Procedure

We add the problems in DIMCAS CNF, one by one, into a text file. The text file is then read, and the data is stored in a 2D vector of integers. This is then given to the solver, which converts it into a vector of clauses and initialises its attributes. Note that we do not count the time taken to initialise the values. We start the timer when the solver starts its SAT-solving procedure and end the timer when the solver returns an answer.

We limit external processes running on the computer to increase the amount of processing power and memory available. However, we can not restrict the internal processes, so we will run each input three times and take an average for the time taken and memory used. We also limited the time to solve each instance to 600 seconds.

We used the memory consumption shown in Replit for memory usage. We did not use the task manager, as it refreshes every second, and most of the program executions lasted less than one second. It is important to note that when showing memory usage, Replit takes into account the memory usage of the compiled code. We calculated the memory used by the program by looking at the peak memory usage and subtracting it by the memory usage of the compiled code. It is important to note that this method of calculating memory usage can lead to inaccurate results.

5.5 Results

We used random-3-SAT, planning SAT and pigeon-hole SAT problems to run experiments on the solvers.

- The random-3-SAT problems were chosen to analyse the performance of the SAT solvers on problems of different difficulties.
- The planning SAT problems were chosen to analyse the performance of the SAT solvers on real-world problems.
- The pigeon-hole problems were chosen as they are one of the hardest problems for SAT-solvers [17].

All of the SAT problems used in experiments are from SATLIB Benchmark Problems [66] [1].
The figure below is a visualisation of the solver configurations.

[1] *https : //www.cs.ubc.ca/ hoos/SATLIB/benchm.html*

SOLVER CONFIGURATIONS

	PURE LITERAL ELIMINATION	STRUCTURE	IN ORDER VARIABLE SELECTION	MOST FREQUENT VARIABLE SELECTION
dpll-R-1	✓	Recursive	✓	
dpll-R-2	✓	Recursive		✓
dpll-I-1	✓	Iterative	✓	
dpll-I-2	✓	Iterative		✓
cdcl-1		Iterative	✓	
cdcl-2		Iterative		✓
cdcl-3	✓	Iterative	✓	
cdcl-4	✓	Iterative		✓

Figure 5.1: Solver Configurations

5.5.1 Random-3-SAT problems

In this set of SAT problems, we experimented on random-3-SAT problems with size (number of variables): 20, 50, 75, 100, and 125. We selected 15 satisfiable examples for each size and 15 unsatisfiable examples for each size except 20 as it was not included in [66].

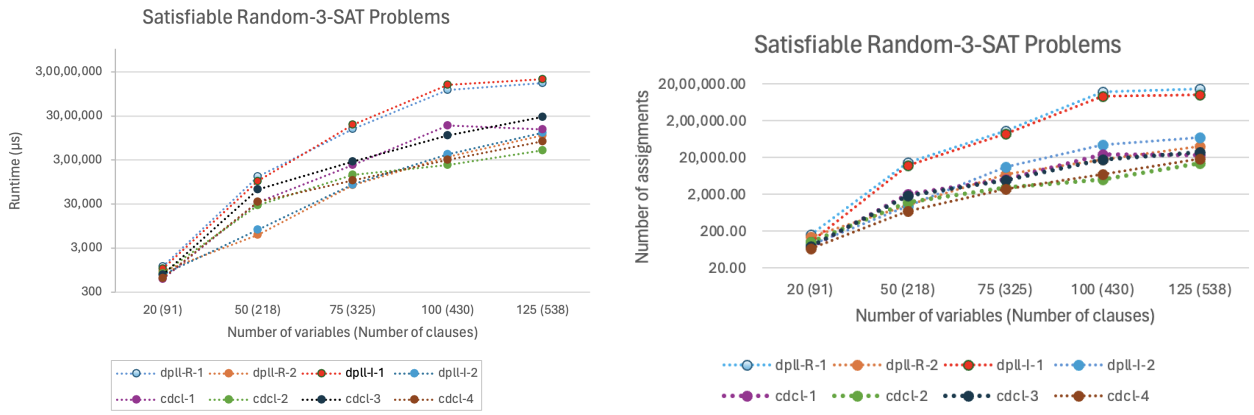

Figure 5.2: Average runtime and average number of assignments of each solver configuration on satisfiable random-3-SAT problems

Figure 5.3: Average runtime and average number of assignments of each solver configuration on unsatisfiable random-3-SAT problems

For the random-3-SAT problems, the table for runtime, number of assignments, correctness, memory usage, standard deviation to mean percentage of runtime, and runtime per assignment of each solver configuration can be found in B.1.

5.5.2 Planning SAT problems

In this set of SAT problems, we experimented on the Planning SAT problems with sizes (number of variables) 48, 116, and 459. There is only 1 example for each size in [66], and all examples are satisfiable.

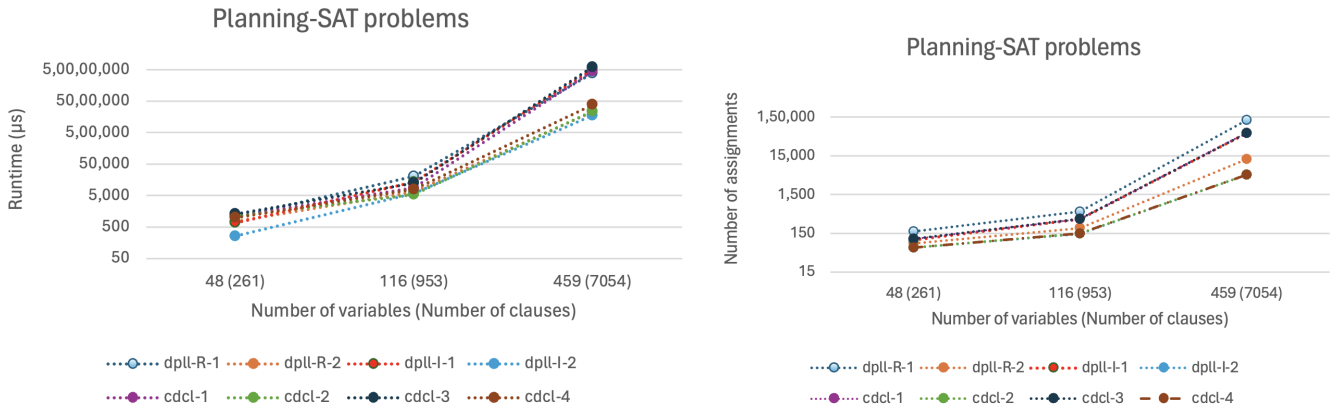

Figure 5.4: Runtime and number of assignments of each solver configuration on planning SAT problems

For the planning SAT problems, the table for runtime, number of assignments, correctness, memory usage, standard deviation to mean percentage of runtime, and runtime per assignment of each solver configuration can be found in B.2.

5.5.3 Pigeon hole problems

In this set of SAT problems, we experimented on the pigeon-hole SAT problems with sizes (number of variables) 42, 57. There is only 1 example for each size in [66], and all examples are unsatisfiable.

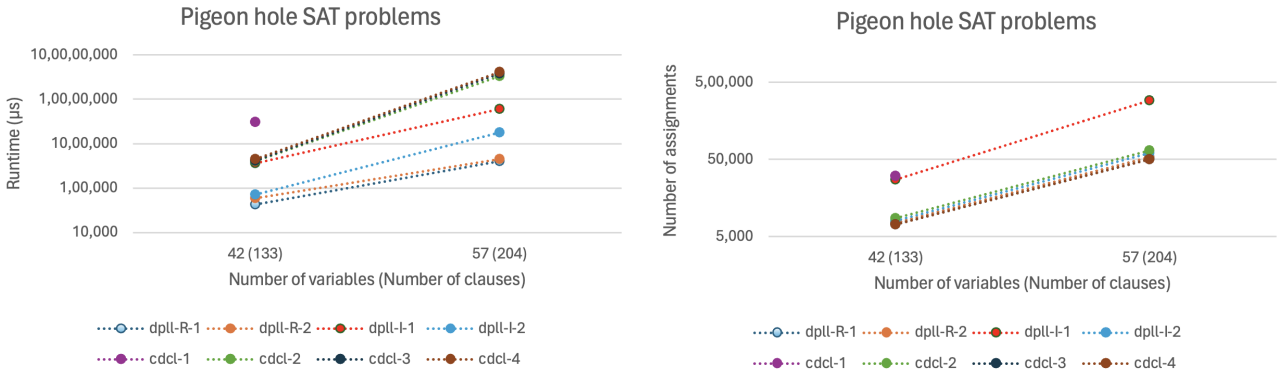

Figure 5.5: Runtime and number of assignments of each solver configuration on pigeon-hole SAT problems

For the pigeon-hole SAT problems, the table for runtime, number of assignments, correctness, memory usage, standard deviation to mean percentage of runtime, and runtime per assignment of each solver configuration can be found in B.3.

5.6 Performance Analysis

In this section, we analyse the performance of the SAT solver configurations on each type of problem. It is important to note that, as explained in 5.4, all average memory usage data is inaccurate.

5.6.1 Random-3-SAT Problems

Satisfiable Random-3-SAT Problems

For the satisfiable random-3-SAT problems, all solvers took around the same average runtime to solve the problems of size 20. However, as the size of the problems increased, the average runtime for the solvers spread out. This can be because the problems of size 20 were very easy to solve and as problems got bigger the solvers needed to search more.

All solvers that used the most frequent variable selection outperformed their counterparts that used in-order variable selection on average runtime and average number of assignments, in some cases by less than 20 times the runtime. The solvers made a more informed choice by selecting the most frequent variable. This leads to significant reductions in the formula for every decision assignment, which leads to faster solutions. This pattern can be seen in all sizes; however, for size 20, the cdcl-2 solver did more assignments on average to solve the problem than cdcl-1, meaning the most frequent variable selection might not always lead to faster solutions for the smaller problems. However, as the size increases, the most frequent variable selection signifies its dominance over in-order variable selection.

All DPLL solvers found the correct answer to all of the problems, whereas the CDCL solvers did not. The correctness of all CDCL solvers decreased as the size of the problems increased. This can be because the number of clauses also increased with the size, limiting the solution space. This suggests that when the search space is big, but the solution space is small, the CDCL solvers will likely not find the correct answer due to their aggressive approach to cutting the search space. When the CDCL solvers could not find solutions, they likely searched more through the search space than when they did find a solution, leading to a higher runtime for those problems. So, potentially, the CDCL solvers are even faster than the results show for satisfiable problems.

As expected, the average memory usage was higher for both recursive DPLL solvers than for both iterative DPLL solvers for all problem sizes. While the CDCL solvers do not have a recursive structure, they use more memory on average than all DPLL solvers. This may be because the implication graph creates multiple instances of the same node to store for each implication. This only worsens as the number of variables increases. Also, the CDCL solvers learn new clauses for every backjump; this will increase memory usage every time a new clause is added to the original formula.

On average, the iterative DPLL solvers did fewer assignments than their counterpart recursive DPLL solvers. This was expected, as explained in 3.2.3 the iterative DPLL solvers do not apply unit propagation or pure literal elimination when a conflict has occurred. However, their average runtime per assignment has increased due to iterative DPLL solvers having to backtrack iteratively. For problems of sizes 75, 100 and 125, the iterative DPLL solvers apply fewer assignments on average than recursive DPLL solvers but have considerable more runtime. This implies that for smaller sizes and when the average number of assignments is small, the iterative DPLL solvers are not significantly affected by having to backtrack iteratively; however, for larger problems, the solver has to spend more time to backtrack.

No solver was the fastest at solving problems of each size; however, for the sizes 100 and 125, cdcl-2

was the fastest. This was due to it learning the wrong clauses and cutting down large branches, which lead to quicker solutions. However, it is important to remember that this same strategy leads to giving the wrong answers.

Pure literal elimination seems to negatively impact the average number of assignments and average runtime for the CDCL solvers. As expected, finding pure literals is a computationally expensive procedure when done frequently. This is shown by the increase in the average runtime per assignment for the CDCL solvers that use pure literal elimination compared to their counterparts that do not.

While the two DPLL solvers with the most frequent variable selection were the fastest at solving problems of sizes 50 and 75, they were overtaken by the two CDCL solvers with the most frequent variable selection for problems of sizes 100 and 125. This suggests the CDCL algorithms have better scalability than the DPLL algorithms. However, if correctness is required, then the CDCL solvers have shown that they are unreliable as they lose correctness as the number of variables and clauses increases. While all of the CDCL solvers have a significantly larger average number of assignments than the DPLL solvers for problems of sizes 50 and above, the CDCL solvers apply notably fewer assignments on average, which keeps their runtime relatively low.

The average standard deviation to mean percentage for runtime gradually decreases for most of the solvers as the problem size increases, suggesting the problems' difficulty was gradually stabilising. However, there is still a significant standard deviation to the mean percentage of $> 20\%$ for every solver on size 125, which is expected as the problems are randomly generated. The problems of size 20 have the biggest standard deviation to mean percentage, suggesting varying difficulty. However, all solvers were able to solve it relatively quickly.

Unsatisfiable Random-3-SAT Problems

For the unsatisfiable random-3-SAT problems, all solvers take longer on average to solve them than for the satisfiable random-3-SAT problems. This is because with satisfiable problems, there are multiple assignments that the solvers can find, and they most likely won't have to search through all of the assignments. However, with unsatisfiable problems, the solvers will need to search more through the search space as there are no satisfying assignments.

The dominance of selecting the most frequent variable over in-order variable selection continues with the unsatisfiable problems, as does the dominance of CDCL solvers on DPLL. However, the dominance of selecting the most frequent variable over in-order variable selection is less for random unsatisfiable problems than for random satisfiable problems.

All solvers found the correct answer, with the CDCL solvers finding the answer quicker due to non-chronological backtracking.

Unlike the random satisfiable problems where there was no single fastest solver for each problem size, the cdcl-2 solver is the fastest at solving the random unsatisfiable problems at each size. This is because it uses the most frequent variable selection and has non-chronic backtracking, which, in turn, reduces, on average, the number of assignments it does, leading to the answer faster.

Pure literal elimination again negatively impacts the runtime of the CDCL solvers. While the CDCL solvers with pure literal elimination do have a lower number of average assignments needed to find an answer, it takes a long time to find the pure literals, which increases the run time and makes them slower than their counterparts without pure literal elimination.

With the unsatisfiable random problems, the iterative DPLL solvers have a much more similar time to the recursive DPLL solvers. Iterative DPLL solvers again have a lower average number of assignments; however, iterative backtracking increases their runtime, making them have a higher average runtime per assignment.

The recursive DPLL solvers have a major increase in memory usage for random unsatisfiable problems. This could be because they are deeply recursing. The average number of assignments has increased as well, which may have had an impact on the memory usage of the solvers. The CDCL solvers continue to use the most amount of memory, possibly due to learning a large number of clauses, and the iterative DPLL solvers continue to use the least amount of memory.

The average runtime per assignment for most of the solvers continues to increase as the size increases. This is especially the case for the CDCL solvers where the average runtime per assignment has a big increase as size increases. This is because as there are more variables, the implication graph is likely to be bigger, and therefore, it will take longer to add an implication and search through the graph.

Like the random satisfiable problems, the random unsatisfiable problems vary in difficulty, as can be seen by the high average standard deviation to mean percentage for the runtime. This, however, decreases as the sizes increase, indicating that the difficulty of each problem varies less.

Planning SAT Problems

It is important to note that only 1 example of each problem size was experimented on, so the results may not generalise well.

With the random SAT problems, we can see that in some cases, the cdcl-1 and cdcl-3 solvers, which select variables in order, outperform dpll-R-2 and dpll-I-2, which select the most frequent variable. The planning SAT problems showcase the dominance of selecting the most frequent variable over selecting variables in order, where all solvers who selected the most frequent variable had a lower runtime than those who selected variables in order.

All of the solvers find the correct answers. This may be the case as only 1 example of each problem size was experimented on, and this may not be an accurate representation of the correctness of the CDCL solvers on planning SAT problems.

Pure literal elimination has once again not improved the runtime of the CDCL solvers. Although it seems that pure literal elimination simplifies the formula, this does not help as it increases the number of assignments and does not lead to quicker solution finding. As expected, the average runtime of CDCL solvers with pure literal elimination is again higher than that of CDCL solvers without pure literal elimination.

All of the solvers take significantly longer to solve the problem of size 459 than 116. This is because of the increased number of clauses. Many, if not all, functions of all the solvers have a time complexity related to the number of clauses and/or the number of literals. It seems the solvers, especially the iterative DPLL solvers, do not scale well to problems with a large number of clauses.

All solvers solved the planning problems of size 48 and 116 in under 600 assignments, suggesting that the problems were very easy and might not accurately represent more complex planning prob-

lems of the same size.

The Memory usage for the problem of size 459 is significantly higher than that for the other problem sizes for all solvers. This is because of the increased number of assignments and because of the major increase in the number of clauses. The recursive DPLL algorithms need to create a copy of the updated clause for every recursion, and this can take a lot of memory when the clauses do not simplify a lot. dpll-R-2 uses much less memory than dpll-R-1 as it is able to simplify the clause more for every decision assignment, which leads to small clauses and, therefore, less memory used when recursing.

The dpll-I-2 solver, which is around the middle of the park for random problems, is the fastest for planning problems. Its average runtime per assignment is still higher than recursive DPLL solvers. However, it applies significantly fewer assignments, which reduces the runtime.

The CDCL solvers are slower, potentially due to learning the wrong clauses and cutting out branches. It is important to note that we have not updated the implication graph correctly; while this does save time, it may end up costing the solver more time by potentially skipping over the closest solutions.

Pigeon-hole SAT Problems

It is important to note that only 1 example of each problem size was experimented on, so the results may not generalise well.

All instances of pigeon-hole problems are unsatisfiable. However, unlike the random unsatisfiable problems where the CDCL solvers were the fastest, The CDCL solvers seem to struggle with the pigeon-hole problems, with cdcl-1 reaching the time limit for the problem of size 57.

CDCL-1, which does not use pure literal elimination or the most frequent variable selection, was the slowest out of all of the solvers. This was because pigeon-hole problems are designed in such a way that they need to be simplified quite a lot before reaching a conflict. Without pure literal elimination, the simplification can take a very long time, and as we have seen with problem size 57, cdcl-1 struggles to simplify the formula as it does not use pure literal elimination or the most frequent variable selection.

cdcl-2, which uses the most frequent variable selection but not pure literal elimination, is the slowest of all solvers, which found a solution in the time limit given for size 57. However, cdcl-3 and cdcl-4, which use pure literal elimination, are faster. This indicates that for pigeon-hole problems, pure-literal elimination is more effective at simplifying the formula than most frequent variable selection. This is backed by the structure of pigeon-hole problems, which includes clauses of varying sizes, however mostly of size 2, and since there are only 204 clauses for 57 variables, lots of literals are likely to be pure at the start or become pure while solving.

The variable selection methods had no difference in the number of assignments for recursive DPLL solvers and CDCL solvers with pure literal elimination. It seems that there may have been the same amount of each variables. However, there was a difference in the number of assignments for the iterative DPLL solvers and CDCL solvers, which did not use pure-literal elimination. This may be because the solvers stop applying unit clauses and pure literal when a conflict is found, and it seems when selecting the most frequent variable, a conflict is found earlier. This, however, is not the case for cdcl-3 and cdcl-4, maybe due to the small sample size. As the number of assignments is the same for dpll-R-1 and dpll-R-2, we can see that the select most frequent variable method takes a longer time. This was expected as it has a higher time complexity than the select variable in order method. This

can also be seen for cdcl-3 and cdcl-4, where the difference is around 40 microseconds per assignment.

All solvers correctly identified that the pigeon-hole problems were unsatisfiable. This was expected as all pigeon-hole problems are unsatisfiable.

The CDCL solvers use the most memory for pigeon-hole problems, and this is due to the implication graph and learnt clauses.

The DPLL solvers are significantly faster than the CDCL solvers, even though they apply around the same number of assignments, if not more. The runtime per assignment for CDCL solvers is much higher due to them having to iteratively backtrack and maintain an implication graph. The implication graph, which helps the CDCL solver to non-chronologically backtrack and save time by skipping over assignments, ultimately costs the CDCL solver a lot of money by taking a long time to add implications and create UIP cuts.

As with random and planning problems, the CDCL solvers, who use pure literal elimination, have a bigger runtime per assignment as expected.

The memory usage pattern follows from the random problems, with CDCL solvers using the most memory, followed by recursive DPLL and iterative DPLL solvers using the least.

As expected, the runtime per assignment increases with the problem size; however, there is a 10x increase from the problem size 42 to 57 for the CDCL solvers. This suggests that CDCL solvers do not scale well for pigeon-hole problems and that optimisations to iteratively backtracking and implication graphs are required.

5.7 Testing

To verify the correctness of the implemented algorithms, we have created unit tests and integration tests that cover various scenarios and edge cases. We have also created a stress test to test if the programs can handle large SAT problems.

There were 10 unit tests, 1 stress test and 1 integration test for the recursive DPLL solver. There were 11 unit tests, 1 stress test and 1 integration test for the iterative DPLL solver. There were 9 unit tests, 1 stress test and 1 integration test for the CDCL solver. Each unit and integration test consists of one or more test cases. The tests for the DPLL and CDCL implementation can be found in appendix C. As can be observed, all implementations passed all test cases.

5.8 Evaluation

This section will evaluate the aims 1.3, requirements, and objectives 3.1 set earlier in this project.

5.8.1 Aims

The aims of this project are to:

- **The primary aim of this project is to conduct a comparative analysis of the structure and performance of DPLL and CDCL algorithms.** We showcase and compare the

structure of DPLL and CDCL in 2.2.2 and 2.2.3. We analyse the performance of the DPLL and CDCL algorithms in 5.6. This aim has been achieved.

- **The secondary aim is to implement the algorithms.** Implementation of the DPLL algorithms are shown in 4.3.1 and 4.3.2. Implementation of the CDCL algorithm is shown in 4.4. This aim has been achieved.

- **The final aim is to experiment with the designs of both algorithms to find potential optimisations.** We designed 3.2.1 and implemented 4.3.2 an iterative DPLL algorithm, and we sacrificed correctness for speed for the CDCL algorithms 4.4. This aim has been achieved.

5.8.2 Requirements And Objectives

Objectives

The objectives are to:

- **Implement the original DPLL algorithm with recursive and iterative structure.** Implementation of the recursive DPLL algorithm is shown in 4.3.1, and implementation of the iterative DPLL algorithm is shown in 4.3.2. This objective has been achieved.

- **Implement a CDCL algorithm that sacrifices correctness for performance.** Implementation of the CDCL algorithm that sacrifices correctness for performance is shown in 4.4. This objective has been achieved.

- **Explore strategies enhancing the performance of the solvers through algorithmic optimisations.** Optimisations created: most-frequent-variable selection 4.3.3, pure-literal elimination D.2 for CDCL solvers. This objective has been achieved.

- **Create visualizations (e.g. tables, graphs) to show experimental results.** We created tables and graphs to show experimental results 5.5, B. This objective has been achieved.

Functional Requirements

The functional requirements state that the system must:

- **All implementations must be able to handle SAT instances in DIMACS CNF format.** In the experiments and testing, all SAT instances that were run on the implementations were in DIMACS CNF format. This requirement has been fulfilled.

- **Implement variable selection mechanisms to choose variables from the SAT formula.** We have implemented 2 variable selection mechanisms, which are shown in 4.3.3. This requirement has been fulfilled.

- **Implement unit propagation and pure literal elimination.** We have implemented unit propagation and pure literal elimination, which is shown in 4.3.4 and 4.3.5. This requirement has been fulfilled.

- **Implement backtracking mechanisms for DPLL and CDCL** We have implemented backtracking mechanisms for DPLL and CDCL, which are shown in 4.3.6 and 4.4.2. This requirement has been fulfilled.

- **Determine the satisfiability of the input SAT instance by either finding a satisfying assignment or proving unsatisfiability.** The functions shown in 4.3.1, 4.3.2, and 4.4 determine the satisfiability of the input SAT instance. This requirement has been fulfilled.

Non-Functional Requirements

The non-functional requirements state that:

- **The implemented solvers should be deterministic. They should be able to produce the same number of variable assignments needed to find an answer with the same inputs.** When performing experiments and testing, each input was executed several times, and each time, the same number of variable assignments were made. This requirement has been fulfilled.

Considerations for Fair Comparison

The considerations for fair comparisons state that:

- **Both algorithms will have similar functions when possible.** Many functions such as select-variable-inorder, select-most-frequent-variable, apply-assignment, get-unit-clauses, and get-pure-literals are the same for both algorithms, and the other functions are similar with small changes to fit the algorithm requirements. This consideration has been fulfilled.
- **A diverse set of benchmark SAT instances, including real-world problems, will be used to evaluate their performance.** We evaluated the solvers' performance using random 3-SAT problems, real-world planning SAT problems, and pigeon-hole SAT problems. This consideration has been fulfilled.
- **Same performance metrics will be used to evaluate the implementations.** the performance metrics used to evaluate all of the solvers are runtime, memory usage, number of assignments, correctness and runtime per assignment. This consideration has been fulfilled.
- **Performance will be measured on the same hardware and software.** The performance of each solver was measured on only the hardware and software mentioned in 5.1. This consideration has been fulfilled.
- **Impact on performance from external will be limited.** We made sure to not run other applications or updates in the background while experimenting. This consideration has been fulfilled.

5.9 Evaluation Conclusion

All the defined aims, objectives and requirements have been met.

Chapter 6

Legal And Ethical Considerations

This section assesses the possible legal, social, ethical, and professional implications of this project and explains how this project follows the British Computing Society's (BCS) Code of Conduct [15] [1].

6.1 Legal Issues

Since this project does not utilize personal data, there is no risk of exposing or exploiting personal information, making it compliant with the General Data Protection Regulation (GDPR).

This project has no legal or licensing concerns. The development of the DPLL and CDCL algorithms involved the use of existing algorithms and code libraries. The algorithms are not licensed and can be used freely for any purpose. We have referenced the places where we found the algorithms and have not claimed their work. This project uses standard C++ libraries, which are under the Apache license [42]. The Apache license grants the user a patent license from each contributor to "make, have made, use, offer to sell, sell, import, and otherwise transfer the Work." [4].

6.2 Social Issues

We have designed and documented this project with usability for all users in mind. We have also been transparent in our decision-making. For accessibility, the designs of each algorithm are shown so that they can be implemented in any language, and clear documentation is provided for a better understanding of the code.

SAT-solving algorithms can have a direct impact on public well-being by optimising resource allocation in critical sectors such as healthcare, transportation, and infrastructure planning. By efficiently solving complex problems, SAT solvers can improve public services, reduce wait times, and enhance overall quality of life.

SAT-solving algorithms have significant economic and commercial implications [86], particularly in industries reliant on optimisation and decision-making processes. SAT solvers can drive economic growth by improving efficiency and optimising resource allocation. In this project, we showcase the CDCL algorithm, which is more efficient at solving planning problems than the DPLL algorithms.

[1] https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf

6.3 Ethical Issues

In this project, we identified potential biases in evaluating DPLL and CDCL algorithms. We aimed to conduct fair and unbiased experiments, using diverse datasets and methodologies to ensure the integrity and reliability of our findings.

SAT problems have various applications, some of which could potentially be used for malicious purposes, such as to attack cryptographic algorithms. Such consequences are not direct outcomes of the work conducted in this project.

It is important to consider the environmental impact of SAT solvers. SAT solving often involves solving computationally intensive problems that require significant amounts of electricity, which contributes to carbon emissions. The choice of SAT-solving algorithm can impact the environment. Using a less efficient algorithm is likely to consume more electricity. This project has limited the time for a SAT solver to solve a problem to 600 seconds to limit electricity consumption. This project also suggests which algorithms to use for certain SAT-solving problems.

Ensuring the trustworthiness and reliability of SAT-solving algorithms is important, particularly in safety-critical applications. This project has done rigorous testing to verify the correctness of each solver.

Finally, this project did not involve the use of humans or animals.

6.4 British Computing Society's Code of Conduct

The entire project has been conducted in accordance with the British Computing Code of Conduct [15] [2].

The findings presented in this project accurately represent the observed results without misrepresentation, in accordance with the code of conduct 3.e. Any instances where the results may not be accurate have been explicitly acknowledged.

[2]https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf

Chapter 7

Conclusion and Future Work

This final chapter of the project highlights the key findings, limitations and potential areas for future work.

7.1 Conclusion

This project evaluated the performance of the DPLL and CDCL algorithms with and without optimisations. The key findings of this project are:

- No single solver was the fastest at solving each problem. Each solver has its strengths and weaknesses against different types of problems. The DPLL solvers were faster than the CDCL solvers for the pigeon-hole problems; however, they did take more assignments to find a solution. If the average runtime of the CDCL solvers can be reduced, this may make them faster than the DPLL solvers. However, memory usage does need to be considered as the CDCL solvers used, on average, more memory than the DPLL solvers for the random and pigeon-hole problems.
- The CDCL solvers are faster than the DPLL solvers at solving planning and random SAT problems; however, as we know, the implemented CDCL solvers are incomplete and may give incorrect answers for satisfiable problems.
- For most cases, the most frequent variable selection is a far better method than selecting the variables in order. With satisfiable random problems of size 20, there were some examples where the most frequent variable selection did more assignments and took more time; however, as the problem size increased, the solvers with the most frequent variable selection were significantly faster than the solvers with in-order variable selection.
- Pure literal elimination hindered the speed of the CDCL solvers for random and planning problems. However, it became more useful for pigeon-hole problems because it further simplifies the formulas. This goes to show that there are situations where Pure literal elimination can provide efficiency.
- The iterative DPLL solvers did fewer assignments on average than the recursive DPLL solvers for random and planning problems; however, it was still slower than the recursive DPLL solvers due to its iterative backtracking. Although, iterative backtracking uses less memory for all of the problems.
- Ineffective memory management is potentially feasible. Even though the implication graphs of CDCL algorithms create new instances of nodes for each implication. The memory usage is relatively low compared to the RAM capacity of the equipment used.

7.2 Limitations

There are several factors that should be considered when interpreting the results of the experiments.

- The benchmark SAT instances used may not fully represent the diversity of real-world problem domains.
- The selected benchmark SAT instances may contain biases that accidentally favour one solver over the other, potentially skewing the results.
- Limited Samples: There was only one example for each pigeonhole and planning problem size. As a result, the generalisability of the performance of the algorithms on these specific problem types may be limited.
- Clause Sizes: For each variable size of the problem, there was only one clause size. Results may differ if different clause sizes were considered.
- Solver Configurations: There are possible multiple configurations for both algorithms. However, we focused on a few of them.
- Single Hardware and Software Environment: The experiments were conducted on a single hardware and software platform. While this ensures consistency in the experimental setup, it may limit the generalisability of our findings to different computing environments.
- The chosen evaluation metrics may not fully capture the performance of the solvers.

7.3 Future Work

7.3.1 Algorithmic Optimisations

In this subsection, we discuss some algorithmic optimisations for both DPLL and CDCL algorithms.

Select Most Frequent Variable 2nd Variation

We have seen that the most frequent variable selection strategy can provide significant efficiencies. However, the function designed and implemented for the most frequent variable selection has a bigger time complexity than selecting the variables in order.

Even though the function select-most-frequent-variable 7 has a polynomial time complexity, it needs optimisation for when the formula is very large. Searching through the formula every time we look to assign a variable outside of unit propagation and pure literal elimination can be inefficient. Instead, we can use a data structure such as a map that keeps count of each variable's frequency and updates it every time we assign a variable or backtrack. Algorithm 22 is a more time-efficient way of finding the most frequent variable.

The function 22 requires an ordered map, ordered by the most frequent variable. The map will need to be initialised when the solver first receives the formula. This function has a time complexity of $O(v^2)$, which is an improvement from $O(nm)$ and has no space complexity as nothing gets stored. The disadvantage to this variation is that for CDCL, this variation is not suitable as new clauses are added, and the map would need to be updated after each learned clause. However, this variation is suitable for DPLL as no new clauses are learnt.

Algorithm 22 Most Frequent Variable Selection (2nd variation)

function SELECT-MOST-FREQUENT-VARIABLE(A, M)
Inputs A: List of assignments. M: an ordered map with variables as key and its frequency as value.
Output Integer

for variable in M **do** ▷ traverses through the ordered Map checking if the variable is assigned or not
if is-variable-assigned(variable,A) = $false$ **then**
return variable
end if
end for
return -1 ▷ All variables have been assigned a value
end function

Get Pure Literals 2nd Variation

We have seen that getting pure literals can be a long process, especially for formulas with a large number of clauses. As the DPLL algorithms use pure-literal-elimination frequently, it is important to have an efficient way to find pure literals.

The approach shown in 10 has a time complexity of $O(nm)$. A better approach would be to use a data structure such as a map and store the count of each different literal at the start, and when a variable is assigned a value, set the literal count to 0 and the negation of the literal count to 0 as well. However, the trouble comes when backtracking; we will need to update the map by running through all the clauses and checking if the literal and its negation exist in the backtrack updated formula. However, we can stop when we find a single positive literal and single negative literal, as the counts do not matter; what matters is whether a variable occurs with only one polarity or not.

Algorithm 23 Get Pure Literals (2nd Variation)

function GET-PURE-LITERALS
Output List of integer

pure-literals = [] ▷ list
for literal in literal-count **do** ▷ in this design, the map literal-count is an attribute to the SAT solver class.
if literal-count[literal] $>$ 0 and literal-count[-literal] = 0 **then** ▷ check if the variable is of single polarity
pure-literals.add(literal)
end if
end for
return pure-literals
end function

Function 23, has a time complexity of $\theta(v)$. This is faster than 10; however, the literal-count map will need to be updated when a variable is assigned. The updating process is quick as it only needs to set the count of the literal and its negation to 0. It will also need to be updated when backtracking, this process is much slower and can take $O(nm)$ time at worst case making it the same as 10. However, on average, it is likely to be much faster as we only need to search for the backtracking literal and its negation in the reverted clauses; once they are found, the search can be ended as pure literal

elimination is not possible if both polarities exist. We do not care how many of each literal and its negation exists in the clauses; we just care if there is single or double polarity.

Improving CDCL correctness

We have seen that the implemented incorrect CDCL solver can solve random-3-SAT and planning problems quicker than the other solvers. However, we did see that as the problem sizes increased, the accuracy of the CDCL solver significantly decreased. A potential future work can be to improve the accuracy of the CDCL solver while maintaining or increasing its speed.

Scarifice Memory

We have sacrificed correctness for speed, however potentially we can sacrifice memory for speed as well. We have seen that all the solvers use relatively low memory compared to the total RAM available. A recursive CDCL algorithm could savetime having to revert the formula when backtracking, however it would need to be implemented in such a way that it can backtrack non-chronologically.

Data structures

Updating the formula every time an assignment has been made and when backtracking is a computationally expensive procedure, as well as finding unit clauses. Both of these can be done efficiently by using two watched-literals data structure. As mentioned in 2.2.2, when using two watched literals, the formula does not need to be updated when assigning a value to a variable and backtracking. This would provide significant improvement for iterative SAT solvers. Two watched-literals data structure is able to identify unit clauses without having to traverse through the formula, this will improve efficiently especially for when solving large formulas.

7.3.2 Analysis Future Work

Here are some future work suggestions for a better analysis of the algorithms:

- Other benchmark SAT problems, such as the ones used in SAT competitions and other real-world SAT problems, can be used to analyse the performance of each solver.
- More samples can be used to experiment on the solvers to solidify the findings.
- To better assess the solver's scalability, bigger problems can be used to experiment.
- Problems with the same number of variables but a different number of clauses can be used to experiment on the solvers to potentially find new patterns.

Bibliography

[1] *A computer can guess more than 100,000,000,000 passwords per second. Still think yours is secure?* Accessed: 2024-03-15. URL: https://theconversation.com/a-computer-can-guess-more-than-100-000-000-000-passwords-per-second-still-think-yours-is-secure-144418.

[2] Mohammad Abdulaziz and Friedrich Kurz. "Formally verified SAT-based AI planning". In: *Proceedings of the AAAI Conference on Artificial Intelligence*. Vol. 37. 12. 2023, pp. 14665–14673.

[3] *AES-256 Encryption Guide for IT Leaders*. Accessed: 2024-03-12. URL: https://www.kiteworks.com/secure-file-sharing/aes-256-encryption-guide-for-it-leaders/#:~:text=AES%2D256%20encryption%20offers%20unparalleled,brute%2Dforce%20attacks%20virtually%20impossible..

[4] *Apache License*. Accessed: 2024-03-28. URL: https://en.wikipedia.org/wiki/Apache_License#Apache_License_2.0.

[5] Roberto Asín, Juan Olate, and Leo Ferres. "Cache performance study of portfolio-based parallel CDCL SAT solvers". In: *arXiv preprint arXiv:1309.3187* (2013).

[6] Franz Baader and Andrei Voronkov. "Abstract DPLL and Abstract DPLL Modulo Theories". In: *Logic for Programming, Artificial Intelligence, and Reasoning*. Springer. 2005, pp. 36–50.

[7] Alex Bertels et al. *The Automated Design of Variable Selection Heuristics for CDCL Solvers To Target Problem Classes.* Tech. rep. Sandia National Lab.(SNL-NM), Albuquerque, NM (United States), 2016.

[8] Armin Biere and Andreas Fröhlich. "Evaluating CDCL variable scoring schemes". In: *Theory and Applications of Satisfiability Testing–SAT 2015: 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings 18*. Springer. 2015, pp. 405–422.

[9] Armin Biere, Marijn Heule, and Hans van Maaren. *Handbook of satisfiability*. Vol. 185. IOS press, 2009.

[10] Armin Biere et al. "Conflict-driven clause learning sat solvers". In: *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications* (2009), pp. 131–153.

[11] *Boolean Logic*. Accessed: 2024-03-08. URL: https://introcs.cs.princeton.edu/java/71boolean/.

[12] *C++ Vector*. Accessed: 2024-03-26. URL: https://www.scaler.com/topics/cpp-vector-insert/#.

[13] Shaowei Cai et al. "Better decision heuristics in CDCL through local search and target phases". In: *Journal of Artificial Intelligence Research* 74 (2022), pp. 1515–1563.

[14] Alonzo Church. "George Boole. An investigation of the laws of thought, on which are founded the mathematical theories of logic and probabilities. Dover Publications, Inc., New York1951, 11+ 424 pp." In: *The Journal of Symbolic Logic* 16.3 (1951), pp. 224–225.

[15] *CODE OF CONDUCT FOR BCS MEMBERS*. Accessed: 2024-04-13. URL: https://www.bcs.org/media/2211/bcs-code-of-conduct.pdf.

[16] Gabriel Coimbra, Lucas Bragança, and José Augusto M Nacif. "Investigating the use of Modern Heterogeneous CPU-FPGA Architectures on the 3-SAT Problem". In: (2018).

[17] *Combinatorial Problem Solving (CPS)*. Accessed: 2024-04-12. URL: https://www.cs.upc.edu/~erodri/webpage/cps/lab/sat/php/lab-pbs.pdf.

[18] *Conflict Driven Clause Learning (CDCL)*. Accessed: 2024-03-20. URL: https://www.geeksforgeeks.org/conflict-driven-clause-learning-cdcl/.

[19] *Conflict-driven clause learning (CDCL) SAT solvers*. Accessed: 2024-03-12. URL: https://users.aalto.fi/~tjunttil/2020-DP-AUT/notes-sat/cdcl.html.

[20] Stephen A Cook. "The complexity of theorem-proving procedures (1971)". In: (2021).

[21] CS Costa. *Parallelization of sat algorithms on gpus*. Tech. rep. Technical report, INESC-ID, Technical University of Lisbon, 2013.

[22] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Communications of the ACM* 5.7 (1962), pp. 394–397.

[23] Martin Davis and Hilary Putnam. "A computing procedure for quantification theory". In: *Journal of the ACM (JACM)* 7.3 (1960), pp. 201–215.

[24] *DIMACS CNF*. Accessed: 2024-03-16. URL: https://jix.github.io/varisat/manual/0.2.0/formats/dimacs.html#:~:text=The%20DIMACS%20CNF%20format%20is,a%20negation%20of%20a%20variable..

[25] *DPLL algorithm*. Accessed: 2024-03-10. URL: https://en.wikipedia.org/wiki/DPLL_algorithm.

[26] James Edwards and Uzi Vishkin. "Study of fine-grained nested parallelism in CDCL SAT solvers". In: *ACM Transactions on Parallel Computing* 8.3 (2021), pp. 1–18.

[27] Yulik Feldman, Nachum Dershowitz, and Ziyad Hanna. "Parallel multithreaded satisfiability solver: Design and implementation". In: *Electronic Notes in Theoretical Computer Science* 128.3 (2005), pp. 75–90.

[28] Matt Fredrikson and Ruben Martins. "Lecture Notes on SAT Solving Techniques". In: (2018).

[29] Oded Goldreich. *P, NP, and NP-Completeness: The basics of computational complexity*. Cambridge University Press, 2010.

[30] Orna Grumberg. *Introduction to Software Verification*. Accessed: 2024-03-06. URL: http://i-cav.org/cavlinks/wp-content/uploads/2019/07/2-1-2018-lectures-11-cbmc-sat.pdf.

[31] Jian Guo et al. "Exploring SAT for Cryptanalysis:(Quantum) Collision Attacks against 6-Round SHA-3 (Full Version)". In: *Cryptology ePrint Archive* (2022).

[32] Aarti Gupta, Malay K Ganai, and Chao Wang. "SAT-based verification methods and applications in hardware verification". In: *International School on Formal Methods for the Design of Computer, Communication and Software Systems*. Springer, 2006, pp. 108–143.

[33] *How much could software errors be costing your company?* Accessed: 2024-03-06. URL: https://raygun.com/blog/cost-of-software-errors/.

[34] Ruoyun Huang, Yixin Chen, and Weixiong Zhang. "SAS+ planning as satisfiability". In: *Journal of Artificial Intelligence Research* 43 (2012), pp. 293–328.

[35] Franjo Ivančić et al. "Efficient SAT-based bounded model checking for software verification". In: *Theoretical Computer Science* 404.3 (2008), pp. 256–274.

[36] Sima Jamali and David Mitchell. "Simplifying CDCL clause database reduction". In: *Theory and Applications of Satisfiability Testing–SAT 2019: 22nd International Conference, SAT 2019, Lisbon, Portugal, July 9–12, 2019, Proceedings 22*. Springer. 2019, pp. 183–192.

[37] Saurabh Joshi et al. "Approximation strategies for incomplete MaxSAT". In: *Principles and Practice of Constraint Programming: 24th International Conference, CP 2018, Lille, France, August 27-31, 2018, Proceedings 24*. Springer. 2018, pp. 219–228.

[38] Henry A Kautz, Ashish Sabharwal, and Bart Selman. "Incomplete Algorithms." In: *Handbook of satisfiability* 185 (2009), pp. 185–203.

[39] Raihan H Kibria and You Li. "Optimizing the initialization of dynamic decision heuristics in DPLL SAT solvers using genetic programming". In: *European Conference on Genetic Programming*. Springer. 2006, pp. 331–340.

[40] Tom Krüger, Jan-Hendrik Lorenz, and Florian Wörz. "Too much information: Why CDCL solvers need to forget learned clauses". In: *Plos one* 17.8 (2022), e0272967.

[41] *Lecture 3: Algorithms for SAT*. Accessed: 2024-03-20. URL: `https://www.cis.upenn.edu/~cis1890/files/Lecture3.pdf`.

[42] *LICENSES*. Accessed: 2024-03-28. URL: `https://www.apache.org/licenses/`.

[43] *Logic Project, Part II*. Accessed: 2024-03-21. URL: `http://www.lsv.fr/~baelde/1920/projlog/sat.html`.

[44] Inês Lynce and Joao Marques-Silva. "The effect of nogood recording in DPLL-CBJ SAT algorithms". In: *International ERCIM Workshop on Constraint Solving and Constraint Logic Programming*. Springer. 2002, pp. 144–158.

[45] *MacBook Air (M1, 2020) Technical Specifications*. Accessed: 2024-03-28. URL: `https://support.apple.com/en-gb/111883`.

[46] Marko Malikovic. "SAT-based Analysis of the Legality of Chess Endgame Positions". In: *Central European Conference on Information and Intelligent Systems*. Faculty of Organization and Informatics Varazdin. 2014, p. 264.

[47] Joao Marques-Silva. "Practical applications of boolean satisfiability". In: *2008 9th International Workshop on Discrete Event Systems*. IEEE. 2008, pp. 74–80.

[48] Joao Marques-Silva. "Search algorithms for satisfiability problems in combinational switching circuits". PhD thesis. University of Michigan, 1995.

[49] Joao Marques-Silva, Inês Lynce, and Sharad Malik. "Conflict-driven clause learning SAT solvers". In: *Handbook of satisfiability*. IOS press, 2021, pp. 133–182.

[50] Fabio Massacci et al. "Using Walk-SAT and Rel-SAT for cryptographic key search". In: *IJCAI*. Vol. 99. 1999, pp. 290–295.

[51] Ilya Mironov and Lintao Zhang. "Applications of SAT solvers to cryptanalysis of hash functions". In: *Theory and Applications of Satisfiability Testing-SAT 2006: 9th International Conference, Seattle, WA, USA, August 12-15, 2006. Proceedings 9*. Springer. 2006, pp. 102–115.

[52] *Modern SAT solvers: fast, neat and underused (part 1 of N)*. Accessed: 2024-03-12. URL: `https://codingnest.com/modern-sat-solvers-fast-neat-underused-part-1-of-n/`.

[53] *Modern SAT Solving*. Accessed: 2024-03-21. URL: `https://sat.inesc-id.pt/~ines/cp07.pdf`.

[54] *Modern Techniques in SAT Solving*. Accessed: 2024-03-12. URL: `https://www.cis.upenn.edu/~cis1890/files/Lecture4.pdf`.

[55] Hidetomo Nabeshima and Katsumi Inoue. "Coverage-based clause reduction heuristics for cdcl solvers". In: *Theory and Applications of Satisfiability Testing–SAT 2017: 20th International Conference, Melbourne, VIC, Australia, August 28–September 1, 2017, Proceedings 20*. Springer. 2017, pp. 136–144.

[56] Alexander Nadel. "Understanding and improving a modern SAT solver". PhD thesis. Tel Aviv University, 2009.

[57] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. "Solving SAT and SAT modulo theories: From an abstract Davis–Putnam–Logemann–Loveland procedure to DPLL (T)". In: *Journal of the ACM (JACM)* 53.6 (2006), pp. 937–977.

[58] Muhammad Osama, Anton Wijs, and Armin Biere. "SAT solving with GPU accelerated inprocessing". In: *International Conference on Tools and Algorithms for the Construction and Analysis of Systems*. Springer. 2021, pp. 133–151.

[59] *Pass by reference (C++ only)*. Accessed: 2024-03-21. URL: https://www.ibm.com/docs/en/zos/2.4.0?topic=calls-pass-by-reference-c-only.

[60] *Playing Hard Mastermind Games with a SAT-based AI*. Accessed: 2024-03-13. URL: https://bohlender.pro/blog/playing-hard-mastermind-games-with-a-sat-based-ai/.

[61] *Recursion: The Pros and Cons*. Accessed: 2024-03-20. URL: https://medium.com/@williambdale/recursion-the-pros-and-cons-76d32d75973a.

[62] *Replit*. Accessed: 2024-03-26. URL: https://replit.com/~.

[63] *Restart of DPLL()*. Accessed: 2024-03-15. URL: http://homepage.divms.uiowa.edu/~hzhang/c188/notes/ch04c-GSAT.pdf.

[64] Jussi Rintanen. "SAT in AI: high performance search methods with applications". In: (2014).

[65] *SAT Competitions*. Accessed: 2024-03-06. URL: http://www.satcompetition.org.

[66] *SATLIB - Benchmark Problems*. Accessed: 2024-03-28. URL: https://www.cs.ubc.ca/~hoos/SATLIB/benchm.html.

[67] Sebastian E Schmittner. "A SAT-based Public Key Cryptography Scheme". In: *Cryptology ePrint Archive* (2015).

[68] Ernst Schröder. *On the formal elements of the absolute algebra*. LED Edizioni Universitarie, 2012.

[69] Alan L Selman. "Complexity issues in cryptography". In: *Computational complexity theory (Atlanta, GA, 1988)* 38 (1989), pp. 92–107.

[70] Ali Asgar Sohanghpurwala, Mohamed W Hassan, and Peter Athanas. "Hardware accelerated SAT solvers—A survey". In: *Journal of Parallel and Distributed Computing* 106 (2017), pp. 170–184.

[71] Peter J Stuckey. "Lazy clause generation: Combining the power of SAT and CP (and MIP?) solving". In: *International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) Techniques in Constraint Programming*. Springer. 2010, pp. 5–9.

[72] Sabina Szymoniak et al. "Sat and smt-based verification of security protocols including time aspects". In: *Sensors* 21.9 (2021), p. 3055.

[73] *The Cost of a Mistake in a Hardware Project*. Accessed: 2024-03-06. URL: https://maddevs.io/blog/cost-of-errors-in-hardware-projects/.

[74] *The Impact of Technology on Board Games: Apps, AI, AR, and Beyond*. Accessed: 2024-03-13. URL: https://joyful-games.com/blogs/card-and-board-games-101/impact-of-technology-on-board-games-apps-ai-ar-and-beyond.

[75] Richard Tichy and Thomas Glase. "Clause learning in sat". In: *University of Potsdam* (2006).

[76] *Top 10 Fastest Programming Languages In 2024*. Accessed: 2024-03-17. URL: https://medium.com/@mohinisaxena/top-10-fastest-programming-languages-in-2024-dee7748e8ecb.

[77] *unordered-map C++*. Accessed: 2024-03-28. URL: https://www.scaler.com/topics/unordered_map-cpp/.

[78] *Use of a Broken or Risky Cryptographic Algorithm*. Accessed: 2024-03-12. URL: https://cwe.mitre.org/data/definitions/327.html.

[79] *What is an example of planning and scheduling in everyday life?* Accessed: 2024-03-13. URL: https://www.qrg.northwestern.edu/projects/vss/docs/Remote-agent/2-planning-scheduling-example.html.

[80] *What Is Data Encryption: Types, Algorithms, Techniques and Methods*. Accessed: 2024-03-12. URL: https://www.simplilearn.com/data-encryption-methods-article.

[81] Lin Xu et al. "SATzilla: portfolio-based algorithm selection for SAT". In: *Journal of artificial intelligence research* 32 (2008), pp. 565–606.

[82] Dennis Yurichev. "SAT/SMT by example". In: *vol* 3 (2020), pp. 1–616.

[83] Aolong Zha, Miyuki Koshimura, and Hiroshi Fujita. "Introducing Pure Literal Elimination into CDCL Algorithm". In: 2016, p. 08.

[84] Hantao Zhang. "SATO: An efficient propositional prover". In: *Automated Deduction—CADE-14: 14th International Conference on Automated Deduction Townsville, North Queensland, Australia, July 13–17, 1997 Proceedings 14*. Springer. 1997, pp. 272–275.

[85] Lintao Zhang. *Searching for truth: techniques for satisfiability of boolean formulas*. Princeton University, 2003.

[86] Lintao Zhang and Sharad Malik. "The quest for efficient boolean satisfiability solvers". In: *Computer Aided Verification: 14th International Conference, CAV 2002 Copenhagen, Denmark, July 27–31, 2002 Proceedings 14*. Springer. 2002, pp. 17–36.

[87] Romanelli Zuim, Jose T Sousa, and Claudionor N Coelho. "A Decision Heuristic for DPLL SAT Solving based on Cube Subtraction". In: *Journal IET Computers & Digital Techniques* (2007).

Appendices

Appendix A

User Manual

This Appendix explains how to execute the code.

Load the folder containing the files onto a SDE supporting C++. Insert the DIMCAS-CNF formula you want to run the solvers on into the file named "DIMCAS-CNF-input.txt". It must be in DIMCAS-CNF and not DIMCAS-CNF-file format. Make sure to comment out the main function of the test files to avoid duplicate function errors. When running the tests for each solver, make sure to comment out the main function and any functions not defined in a class or struct for that solver to avoid duplicate function errors.

Appendix B

Experiment Results

This Appendix showcases the data collected from experimenting with the solvers on random-3-SAT problems (satisfiable and unsatisfiable), planning SAT problems and pigeon-hole SAT problems. It is important to note that, as explained in 5.4, all average memory usage data is inaccurate.

B.1 Results For Random-3-SAT Problems

The runtime, number of assignments, memory usage, standard deviation to mean percentage for runtime and runtime per assignment are all averages taken over 15 samples for each size of satisfiable and unsatisfiable random-3-SAT problems.

B.1.1 Satisfiable Random-3-SAT Problems

Satisfiable Random-3SAT Average Runtime						
	Size	20 (91)	50 (218)	75 (325)	100 (430)	125 (538)
Configurations						
dpll-R-1		1150.4	127191	1535870	11786984	16815120
dpll-R-2		894	6048	80382	358493	1100283
dpll-I-1		1029	100283	1893899	15384927	20482738
dpll-I-2		748	7937	83947	405796	1258645
cdcl-1		604	33028	230955	1839474	1503940
cdcl-2		803	28390	139340	230480	503950
cdcl-3		756	65074	280384	1083849	2839484
cdcl-4		634	33949	102948	302840	803984

Satisfiable Random-3SAT Correctness						
	Size	20 (91)	50 (218)	75 (325)	100 (430)	125 (538)
Configurations						
dpll-R-1		100%	100%	100%	100%	100%
dpll-R-2		100%	100%	100%	100%	100%
dpll-I-1		100%	100%	100%	100%	100%
dpll-I-2		100%	100%	100%	100%	100%
cdcl-1		80%	73%	66%	66%	60%
cdcl-2		80%	66%	73%	47%	40%
cdcl-3		86%	66%	60%	53%	47%
cdcl-4		80%	73%	73%	66%	60%

Satisfiable Random-3SAT Average standard deviation to mean percentage						
	Size	20 (91)	50 (218)	75 (325)	100 (430)	125 (538)
Configurations						
dpll-R-1		35.205146	50.258273	44.20602	23.5141407	20.4546266
dpll-R-2		33.7807606	66.1706349	62.5687343	57.1721066	27.79494
dpll-I-1		34.0136054	50.5509408	54.3050606	26.191109	28.9432692
dpll-I-2		39.171123	38.4024191	58.8156813	52.991651	28.2715937
cdcl-1		41.3907285	31.0191353	44.5290208	32.8272104	33.7069963
cdcl-2		62.6400996	53.867559	57.6926941	56.5697674	59.405695
cdcl-3		40.3439153	43.7809263	35.8023282	28.4159509	21.4353383
cdcl-4		48.5804416	11.8678017	33.9744337	36.245212	38.5271349

Satisfiable Random-3SAT Average Number of assignments						
	Size	20 (91)	50 (218)	75 (325)	100 (430)	125 (538)
Configurations						
dpll-R-1		160	14578	107843	1207496	1480036
dpll-R-2		142	998	7095	17937	40384
dpll-I-1		103	12479	90383	938394	1029384
dpll-I-2		86	993	11338	44394	69394
cdcl-1		74	2034	5220	25039	23049
cdcl-2		101	1220	3038	5038	13948
cdcl-3		79	1868	4938	18039	28304
cdcl-4		68	693	2804	7038	18394

Satisfiable Random-3SAT Average memory usage						
	Size	20 (91)	50 (218)	75 (325)	100 (430)	125 (538)
Configurations						
dpll-R-1		2.5	3.5	5.6	8.1	9.1
dpll-R-2		2.1	3.1	4.4	6.9	7.5
dpll-I-1		1.9	2.4	2.9	3.7	4.4
dpll-I-2		1.7	2.2	2.8	3.5	4.1
cdcl-1		2.3	4.1	6.7	8.9	10.3
cdcl-2		2.3	3.5	5.2	7.5	9.1
cdcl-3		2.4	4	6.7	8.9	11.2
cdcl-4		2.2	3.5	6.2	8.6	9.5

Satisfiable Random-3SAT Average runtime per assignment						
	Size	20 (91)	50 (218)	75 (325)	100 (430)	125 (538)
Configurations						
dpll-R-1		7.19	8.72485938	14.2417218	9.76150977	11.3612912
dpll-R-2		6.29577465	6.06012024	11.3293869	19.9862296	27.245518
dpll-I-1		9.99029126	8.03614072	20.9541507	16.3949546	19.8980536
dpll-I-2		8.69767442	7.99295065	7.40403951	9.14078479	18.1376632
cdcl-1		8.16216216	16.2379548	44.2442529	73.4643556	65.2496855
cdcl-2		7.95049505	23.2704918	45.8657011	45.7483128	36.130628
cdcl-3		9.56962025	34.8361884	56.7808829	60.0836521	100.320944
cdcl-4		9.32352941	48.988456	36.7146933	43.0292697	43.7090356

Figure B-1: Average runtime, average number of assignments, correctness, average memory usage, average standard deviation to mean percentage for runtime and average runtime per assignment for all solver configurations on satisfiable random-3-SAT problems.

B.1.2 Unsatisfiable Random-3-SAT Problems

Unsatisfiable Random-3SAT Average runtime					
	Size	50 (218)	75 (325)	100 (430)	125 (538)
Configurations					
dpll-R-1		584058	4035870	19238374	30492485
dpll-R-2		60384	483840	3139043	6100283
dpll-I-1		450383	3895830	20398173	33594849
dpll-I-2		59680	502832	5893820	6683840
cdcl-1		48590	304847	2738405	4038385
cdcl-2		26859	183746	837494	1693985
cdcl-3		55389	293049	1838483	5938375
cdcl-4		44595	204857	1204837	2994385

Unsatisfiable Random-3SAT Average number of assignments					
	Size	50 (218)	75 (325)	100 (430)	125 (538)
Configurations					
dpll-R-1		50390	304856	1693945	2103893
dpll-R-2		9378	40948	193830	250685
dpll-I-1		50485	203840	1393850	2039475
dpll-I-2		7394	36495	203849	289480
cdcl-1		3840	15395	60496	103940
cdcl-2		2039	7338	28395	40294
cdcl-3		2903	8729	26930	60389
cdcl-4		2594	5893	15293	33948

Unsatisfiable Random-3SAT Correctness					
	Size	50 (218)	75 (325)	100 (430)	125 (538)
Configurations					
dpll-R-1		100%	100%	100%	100%
dpll-R-2		100%	100%	100%	100%
dpll-I-1		100%	100%	100%	100%
dpll-I-2		100%	100%	100%	100%
cdcl-1		100%	100%	100%	100%
cdcl-2		100%	100%	100%	100%
cdcl-3		100%	100%	100%	100%
cdcl-4		100%	100%	100%	100%

Unsatisfiable Random-3SAT Average memory usage					
	Size	50 (218)	75 (325)	100 (430)	125 (538)
Configurations					
dpll-R-1		3.8	6	9.2	12.1
dpll-R-2		3.3	4.5	7.4	10.7
dpll-I-1		2.9	3.6	4.1	5.1
dpll-I-2		2.5	3.2	4.1	4.6
cdcl-1		4.5	8.2	10.1	12.2
cdcl-2		4	6.7	8.9	10.4
cdcl-3		4.5	7.2	10.9	11.3
cdcl-4		3.8	6.8	9	10.6

Unsatisfiable Random-3SAT Average standard deviation to mean percentage					
	Size	50 (218)	75 (325)	100 (430)	125 (538)
Configurations					
dpll-R-1		12.050858	24.375017	15.8505287	12.1434248
dpll-R-2		33.7572867	22.3927331	13.9971004	18.095013
dpll-I-1		40.810599	31.0456052	24.6555611	30.9394098
dpll-I-2		17.2469839	20.6709199	22.1394104	29.0065142
cdcl-1		29.8147767	23.1555502	22.051121	12.1158087
cdcl-2		40.7126103	27.7382909	36.401216	41.0178957
cdcl-3		56.0363971	35.4838269	32.8997331	26.8397499
cdcl-4		59.3900662	53.1331612	40.1661802	30.9711009

Unsatisfiable Random-3SAT Average runtime per assignment					
	Size	50 (218)	75 (325)	100 (430)	125 (538)
Configurations					
dpll-R-1		11.590752	13.238611	11.357142	14.493363
dpll-R-2		6.4388996	11.815962	16.194825	24.334456
dpll-I-1		8.9211251	19.112196	14.63441	16.472302
dpll-I-2		8.0714093	13.778107	28.912676	23.089125
cdcl-1		12.653646	19.801689	45.265885	38.85304
cdcl-2		13.172634	25.040338	29.494418	42.040626
cdcl-3		19.079917	33.571887	68.268957	98.335376
cdcl-4		17.191596	34.762769	78.783561	88.205049

Figure B-2: Average runtime, average number of assignments, correctness, average memory usage, average standard deviation to mean percentage for runtime and average runtime per assignment for all solver configurations on unsatisfiable random-3-SAT problems.

B.2 Results For Planning SAT Problems

The data for planning SAT problems is taken over a single sample for each size.

	Planning SAT problems Runtime		
	48 (261)	116 (953)	459 (7054)
dpll-R-1	1029	20384	38605090
dpll-R-2	744	6614	2299672
dpll-I-1	682	13398	51794321
dpll-I-2	257	5770	1777934
cdcl-1	1119	8830	45029374
cdcl-2	1097	5457	2381243
cdcl-3	1300	12364	60764585
cdcl-4	999	7937	3852756

	Planning SAT problems Number of assignments		
	48 (261)	116 (953)	459 (7054)
dpll-R-1	164	537	125940
dpll-R-2	81	204	12097
dpll-I-1	99	347	57046
dpll-I-2	63	145	4867
cdcl-1	107	352	57666
cdcl-2	64	147	4898
cdcl-3	107	352	57666
cdcl-4	64	147	4898

	Planning SAT problems Correctness		
	48 (261)	116 (953)	459 (7054)
dpll-R-1	100%	100%	100%
dpll-R-2	100%	100%	100%
dpll-I-1	100%	100%	100%
dpll-I-2	100%	100%	100%
cdcl-1	100%	100%	100%
cdcl-2	100%	100%	100%
cdcl-3	100%	100%	100%
cdcl-4	100%	100%	100%

	Planning SAT problems Memory usage		
	Size	48 (261)	116 (953)	459 (7054)
Configurations				
dpll-R-1		2.5	4.5	23.6
dpll-R-2		2.3	4.2	12.5
dpll-I-1		2	2.6	10.3
dpll-I-2		1.8	2.1	5.1
cdcl-1		2	2.2	13.7
cdcl-2		1.9	2.1	13.6
cdcl-3		2.1	2.2	13.6
cdcl-4		2	2.1	13.6

	Planning SAT problems Runtime per assignment		
	48 (261)	116 (953)	459 (7054)
dpll-R-1	6.27439024	37.9590317	306.535572
dpll-R-2	9.18518519	32.4215686	190.10267
dpll-I-1	6.88888889	38.610951	907.939575
dpll-I-2	4.07936508	39.7931034	365.303883
cdcl-1	10.4579439	25.0852273	780.865224
cdcl-2	17.140625	37.122449	486.166394
cdcl-3	12.1495327	35.125	1053.73331
cdcl-4	15.609375	53.9931973	786.597795

Figure B-3: Runtime, number of assignments, correctness, memory usage, and runtime per assignment for all solver configurations on planning SAT problems.

B.3 Results For Pigeon-hole SAT Problems

The data for pigeon-hole SAT problems is taken over a single sample for each size.

	Pigen hole SAT problems Runtime	
	42 (133)	57 (204)
dpll-R-1	43031	401497
dpll-R-2	58678	442891
dpll-I-1	358475	5982083
dpll-I-2	71937	1770135
cdcl-1	3043737	Time limit reached
cdcl-2	405838	33029380
cdcl-3	415294	38082938
cdcl-4	450920	40293495

	Pigen hole SAT problems Number of assignments	
	42 (133)	57 (204)
dpll-R-1	7556	52975
dpll-R-2	7556	52975
dpll-I-1	27326	289212
dpll-I-2	7977	60221
cdcl-1	30571	Time limit reached
cdcl-2	8696	65260
cdcl-3	7196	50455
cdcl-4	7196	50455

	Pigen hole SAT problems Correctness	
	42 (133)	57 (204)
dpll-R-1	100%	100%
dpll-R-2	100%	100%
dpll-I-1	100%	100%
dpll-I-2	100%	100%
cdcl-1	100%	Time limit reached
cdcl-2	100%	100%
cdcl-3	100%	100%
cdcl-4	100%	100%

Pigen hole SAT problems Memory usage			
	Size	42 (133)	57 (204)
Configurations			
dpll-R-1		3.4	8
dpll-R-2		3.5	7.9
dpll-I-1		2.8	5.8
dpll-I-2		2.3	5.2
cdcl-1		3.7	Time limit reached
cdcl-2		3	11.2
cdcl-3		3.5	11
cdcl-4		2.9	10.8

Pigen hole SAT problems Runtime per assignment		
	42 (133)	57 (204)
dpll-R-1	5.69494442	7.57899009
dpll-R-2	7.76574907	8.36037754
dpll-I-1	13.1184586	20.684076
dpll-I-2	9.0180519	29.3939822
cdcl-1	99.5628864	Time limit reached
cdcl-2	46.6695032	506.119828
cdcl-3	57.7117843	754.790169
cdcl-4	62.6625903	798.602616

Figure B-4: Runtime, number of assignments, correctness, memory usage, and runtime per assignment for all solver configurations on pigeon-hole SAT problems.

Appendix C

Testing

This Appendix showcases the unit, integration and stress tests used to verify the correctness of the implemented solvers: DPLL recursive, DPLL iterative and CDCL.

C.1 DPLL Recusive Tests

```
void test_contains_emptyClause() {
    vector<Clause> clauses = { {{1}}, {{}} };
    SATsolverDPLLRecursive solver;
    assert(solver.contains_emptyClause(clauses) == true);

    vector<Clause> clauses2 = { {{1}}, {{1, 2}}, {{-2}} };
    assert(solver.contains_emptyClause(clauses2) == false);
}
```

```
void test_get_pure_literals() {
    SATsolverDPLLRecursive solver;
    solver.number_of_variables = 4;
    vector<Clause> clauses = { {{1, 2}}, {{3}}, {{-2, -3, 4}}, {{-2, -4}} };
    vector<int> pure_literals = solver.get_pure_literals(clauses);
    assert(pure_literals.size() == 1); // Expecting 1 pure literal
    assert(pure_literals[0] == 1); // Expecting pure literal 1

    clauses = { {{1, 2}}, {{-1, 3}}, {{-2, -3, 4}}, {{-2, -4}} };
    pure_literals = solver.get_pure_literals(clauses);
    assert(pure_literals.size() == 0); // Expecting 0 pure literal

    clauses = { {{1, 2}}, {{3}}, {{-2, 3}}, {{-2, -4}} };
    pure_literals = solver.get_pure_literals(clauses);
    assert(pure_literals.size() == 3); // Expecting 3 pure literal {1,3,-4}, order
doesnt matter.
}
```

```
void test_get_unit_clauses() {
    SATsolverDPLLRecursive solver;
    vector<Clause> clauses = { {{1,3}}, {{2}}, {{-2, 3}}, {{-3, 4}} };
    vector<int> unit_clauses = solver.get_unit_clauses(clauses);
    assert(unit_clauses.size() == 1); // Expecting one unit clause
    assert(unit_clauses[0] == 2); // Expecting unit literal 2

    clauses = { {{1,3}}, {{-1, 2}}, {{-2, 3}}, {{-3, 4}} };
    unit_clauses = solver.get_unit_clauses(clauses);
    assert(unit_clauses.size() == 0); // Expecting no unit clause

    clauses = { {{1,3}}, {{-1}}, {{-2}}, {{-3, 4}} };
    unit_clauses = solver.get_unit_clauses(clauses);
    assert(unit_clauses.size() == 2); // Expecting 2 unit clause
    assert(unit_clauses[0] == -1);
    assert(unit_clauses[1] == -2);
}
```

```
void test_is_assigned() {
    SATsolverDPLLRecursive solver;
    vector<int> assignments = {1, -2, 3};
    assert(solver.is_assigned(1, assignments) == true);
    assert(solver.is_assigned(-1, assignments) == true); // this is true because
variable 1 is assigned to true
    assert(solver.is_assigned(4, assignments) == false);
}
```

```
void test_isSatisfied() {
    vector<Clause> clauses = { {{1, 2, 3}}, {{-1, 2, -3}} };
    vector<int> assignments = {1, 2};
    SATsolverDPLLRecursive solver;
    assert(solver.is_satisfiable(clauses, assignments) == true); // the clauses are
satisfiable by the assignments.

    assignments = {1, -2};
    assert(solver.is_satisfiable(clauses, assignments) == false); // the clauses are
unsatisfiable by the assignments.

    clauses = { {{1, 2, 3}}, {{-1, 2, -3}}, {{1, -
}
```

```
int main(){
  test_isSatisfied();
  test_contains_emptyClause();
  test_get_unit_clauses();
  test_unit_propagation();
  test_get_pure_literals();
  test_pure_literal_elimination();
  test_is_assigned();
  test_solve();
  test_variable_count();
  test_pickVariableInOrder();
  test_pickMostFrequentVariable();
  stress_test();
}
```

```
void test_pickVariableInOrder() {
    SATsolverDPLLRecursive solver;
    solver.number_of_variables = 4;
    vector<int> assignments = {1, -2, 3};

    // Test case where there are unassigned variables remaining
    int variable = solver.pickVariableInOrder(assignments);
    assert(variable == 4); // Expecting the next unassigned variable, which is 4

    assignments = {1, -2, -4};
    variable = solver.pickVariableInOrder(assignments);
    assert(variable == 3); // Expecting the next unassigned variable, which is 4

    // Test case where there are no more unassigned variables
    assignments.push_back(3);
    variable = solver.pickVariableInOrder(assignments);
    assert(variable == 0); // Expecting 0 indicating no more variables to assign
}
```

```
void test_pickMostFrequentVariable() {
    SATsolverDPLLRecursive solver;
    vector<Clause> clauses = { {{1, -2, 3}}, {{1, -2, 3}}, {{-1, -2, -3}}, {{-3}}};
    int variable = solver.pickMostFrequentVariable(clauses);
    assert(variable == 3); // Expecting variable 3 as most frequent

    // Test case where there are no more unassigned variables
    clauses = {{}};
    variable = solver.pickMostFrequentVariable(clauses);
    assert(variable == 0); // Expecting 0 indicating no more variables to assign

}
```

```
void test_pure_literal_elimination() {
    SATsolverDPLLRecursive solver;
    vector<Clause> clauses = { {{1, 2}}, {{-1, 3}}, {{-2, 3}}, {{-1,2}} };
    vector<int> assignments = {};

    // Perform pure literal elimination
    solver.pure_literal_elimination(clauses, assignments);
    assert(clauses.size() == 0); // 3 gets applied, then 2, which leaves no clauses
unsatisfied
}
```

```
void test_solve() { // Test for solve function (integration test)
    SATsolverDPLLRecursive solver;
    vector<vector<int>> clauses = { {1, 2}, {-1, 3}, {-2, 3} };
    bool result = solver.solve(clauses);
    assert(result == true); // satisfiable

    clauses = {{1}, {-1}};
    result = solver.solve(clauses);
    assert(result == false); // unsatisfiable

    clauses = {{1, 2, 3, 4},{-1, -2, -3, -4},{1, 2, -3, -4},{-1, -2, 3, 4}};
    result = solver.solve(clauses);
    assert(result == true); // satisfiable

    clauses = {
      {1, 2, 3, 4,},
      {-1, -2, -3, -4},
      {1, 2, -3, -4},
      {-1, -2, 3, 4},
      {1, -2, 3, -4},
      {1, -2, -3, -4},
      {-1, -2, -3, 4},
      {2},
      {-2},
      {-1, 2, 3, -4},
      {1, 2, -3, -4},
    };
    result = solver.solve(clauses);
    assert(result == false); // unsatisfiable

    clauses = {{1, 2, 3, 4, 5},{-1, -2, -3, -4, -5},{1, 2, -3, -4, 5},{-1, 2, 3, 4, 5},{1, -2, 3, -4, 5}};
    result = solver.solve(clauses);
    assert(result == true); // satisfiable
}
```

```
    void stress_test() {
        SATsolverDPLLRecursive solver;
        // Generate a large SAT formula
        vector<vector<int>> large_formula;
        const int num_variables = 100;
        const int num_clauses = 5000;
        srand(time(NULL));
        for (int i = 0; i < num_clauses; ++i) {
            vector<int> clause;
            for (int j = 0; j < 3; ++j) { // Each clause contains 3 literals, can be
    changed
                int literal = (rand() % num_variables) + 1; // Random variable between 1
    and num_variables
                if (rand() % 2 == 0) literal = -literal; // Randomly negate literal
                clause.push_back(literal);
            }
            large_formula.push_back(clause);
        }

        bool result = solver.solve(large_formula);
        // Assert that the solver returns an answer
        assert(result == true || result == false);
    }
```

```
  void test_unit_propagation() {
    SATsolverDPLLRecursive solver;
    vector<Clause> clauses = { {{1}}, {{-1, 3}}, {{-2, 3}} };
    vector<int> assignments = {};
    solver.unit_propagation(clauses, assignments);
    assert(clauses.size() == 0); // 1 gets applied then 3 is unit clause and gets applied

    clauses = { {{1}}, {{-1, 3}}, {{-2, 3}}, {{1}} };
    assignments = {};
    solver.unit_propagation(clauses, assignments);
    assert(clauses.size() == 0); // to check if 2 of the same unit clauses cause a
  problem

  }
```

```
void test_variable_count() {
    SATsolverDPLLRecursive solver;
    vector<vector<int>> clauses = { {1, 2, 3}, {-1, -2, -3}, {1, 2, -3, -4} };
    int count = solver.variable_count(clauses);
    assert(count == 4); // Expecting 4 variables

    clauses = { {1, 2, 3, 4}, {-1, -2, -3, 5}, {1, 2, -3, -4}, {6}};
    count = solver.variable_count(clauses);
    assert(count == 6); // Expecting 6 variables

    clauses = {{}};
    count = solver.variable_count(clauses);
    assert(count == 0); // Expecting 0 variables
}
```

Figure C-1: Unit, integration and stress tests for DPLL recursive implementation and the main function to run all tests.

C.2 DPLL Iterative Tests

```
void test_contains_emptyClause() {
    vector<Clause> clauses = { {{1}}, {{}} };
    SATsolverDPLLIterative solver;
    assert(solver.contains_emptyClause(clauses) == true);

    vector<Clause> clauses2 = { {{1}}, {{1, 2}}, {{-2}} };
    assert(solver.contains_emptyClause(clauses2) == false);
}
```

```
void test_get_pure_literals() {
    SATsolverDPLLIterative solver;
    solver.number_of_variables = 4;
    vector<Clause> clauses = { {{1, 2}}, {{3}}, {{-2, -3, 4}}, {{-2, -4}} };
    vector<int> pure_literals = solver.get_pure_literals(clauses);
    assert(pure_literals.size() == 1); // Expecting 1 pure literal
    assert(pure_literals[0] == 1); // Expecting pure literal 1

    clauses = { {{1, 2}}, {{-1, 3}}, {{-2, -3, 4}}, {{-2, -4}} };
    pure_literals = solver.get_pure_literals(clauses);
    assert(pure_literals.size() == 0); // Expecting 0 pure literal

    clauses = { {{1, 2}}, {{3}}, {{-2, 3}}, {{-2, -4}} };
    pure_literals = solver.get_pure_literals(clauses);
    assert(pure_literals.size() == 3); // Expecting 3 pure literal {1,3,-4}, order doesnt
matter.
}
```

```
void test_get_unit_clauses() {
    SATsolverDPLLIterative solver;
    vector<Clause> clauses = { {{1,3}}, {{2}}, {{-2, 3}}, {{-3, 4}} };
    vector<int> unit_clauses = solver.get_unit_clauses(clauses);
    assert(unit_clauses.size() == 1); // Expecting one unit clause
    assert(unit_clauses[0] == 2); // Expecting unit literal 2

    clauses = { {{1,3}}, {{-1, 2}}, {{-2, 3}}, {{-3, 4}} };
    unit_clauses = solver.get_unit_clauses(clauses);
    assert(unit_clauses.size() == 0); // Expecting no unit clause

    clauses = { {{1,3}}, {{-1}}, {{-2}}, {{-3, 4}} };
    unit_clauses = solver.get_unit_clauses(clauses);
    assert(unit_clauses.size() == 2); // Expecting 2 unit clause
    assert(unit_clauses[0] == -1);
    assert(unit_clauses[1] == -2);
}
```

```
void test_is_assigned() {
    SATsolverDPLLIterative solver;
    vector<int> assignments = {1, -2, 3};
    assert(solver.is_assigned(1, assignments) == true);
    assert(solver.is_assigned(-1, assignments) == true); // this is true because variable
1 is assigned to true
    assert(solver.is_assigned(4, assignments) == false);
}
```

```
int main(){
  test_isSatisfied();
  test_contains_emptyClause();
  test_get_unit_clauses();
  test_unit_propagation();
  test_get_pure_literals();
  test_pure_literal_elimination();
  test_is_assigned();
  test_solve();
  test_variable_count();
  test_pickVariableInOrder();
  test_pickMostFrequentVariable();
  test_backtrack();
  stress_test();
}
```

```
void test_isSatisfied() {
    vector<Clause> clauses = { {{1, 2, 3}}, {{-1, 2, -3}} };
    vector<int> assignments = {1, 2};
    SATsolverDPLLIterative solver;
    assert(solver.is_satisfiable(clauses, assignments) == true); // the clauses
are satisfiable by the assignments.

    assignments = {1, -2};
    assert(solver.is_satisfiable(clauses, assignments) == false); // the
clauses are unsatisfiable by the assignments.
}
```

```
void test_pickVariableInOrder() {
    SATsolverDPLLIterative solver;
    solver.number_of_variables = 4;
    vector<int> assignments = {1, -2, 3};

    // Test case where there are unassigned variables remaining
    int variable = solver.pickVariableInOrder(assignments);
    assert(variable == 4); // Expecting the next unassigned variable, which is 4

    assignments = {1, -2, -4};
    variable = solver.pickVariableInOrder(assignments);
    assert(variable == 3); // Expecting the next unassigned variable, which is 3

    assignments = {1, -2, -4, -6};

    // Test case where there are no more unassigned variables
    assignments.push_back(3);
    variable = solver.pickVariableInOrder(assignments);
    assert(variable == 0); // Expecting 0 indicating no more variables to assign
}
```

```
void test_pickMostFrequentVariable() {
    SATsolverDPLLIterative solver;
    vector<Clause> clauses = { {{1, -2, 3}}, {{1, -2, 3}}, {{-1, -2, -3}}, {{-3}}};
    int variable = solver.pickMostFrequentVariable(clauses);
    assert(variable == 3); // Expecting variable 3 as most frequent

    // Test case where there are no more unassigned variables
    clauses = {{}};
    variable = solver.pickMostFrequentVariable(clauses);
    assert(variable == 0); // Expecting 0 indicating no more variables to assign

}
```

```
void test_pure_literal_elimination() {
    SATsolverDPLLIterative solver;
    vector<Clause> clauses = { {{1, 2}}, {{-1, 3}}, {{-2, 3}}, {{-1,2}} };
    vector<int> assignments = {};

    // Perform pure literal elimination
    assert(solver.pure_literal_elimination(clauses, assignments) == true); // no conflict
in the clauses
    assert(clauses.size() == 0); // 3 gets applied, then 2, which leaves no clauses
unsatisfied
}
```

```
void test_solve() { // Test for solve function (integration test)
    SATsolverDPLLIterative solver;
    vector<vector<int>> clauses = { {1, 2}, {-1, 3}, {-2, 3} };
    bool result = solver.solve(clauses);
    assert(result == true); // satisfiable

    clauses = {{1}, {-1}};
    result = solver.solve(clauses);
    assert(result == false); // unsatisfiable

    clauses = {{1, 2, 3, 4},{-1, -2, -3, -4},{1, 2, -3, -4},{-1, -2, 3, 4}};
    result = solver.solve(clauses);
    assert(result == true); // satisfiable

    clauses = {
      {1, 2, 3, 4,},
      {-1, -2, -3, -4},
      {1, 2, -3, -4},
      {-1, -2, 3, 4},
      {1, -2, 3, -4},
      {1, -2, -3, -4},
      {-1, -2, -3, 4},
      {2},
      {-2},
      {-1, 2, 3, -4},
      {1, 2, -3, -4},
    };
    result = solver.solve(clauses);
    assert(result == false); // unsatisfiable

    clauses = {{1, 2, 3, 4, 5},{-1, -2, -3, -4, -5},{1, 2, -3, -4, 5},{-1, 2, 3, 4, 5},{1, -2, 3, -4, 5}};
    result = solver.solve(clauses);
    assert(result == true); // satisfiable
}
```

```
void stress_test() {
    SATsolverDPLLIterative solver;
    // Generate a large SAT formula
    vector<vector<int>> large_formula;
    const int num_variables = 100;
    const int num_clauses = 500;
    srand(time(NULL));
    for (int i = 0; i < num_clauses; ++i) {
        vector<int> clause;
        for (int j = 0; j < 3; ++j) { // Each clause contains 3 literals, can be changed
            int literal = (rand() % num_variables) + 1; // Random variable between 1 and num_variables
            clause.push_back(literal);
            if (rand() % 2 == 0) literal = -literal; // Randomly negate literal
            clause.push_back(literal);
        }
        large_formula.push_back(clause);
    }

    bool result = solver.solve(large_formula);
    // Assert that the solver returns an answer
    assert(result == true || result == false);
}
```

```
void test_unit_propagation() {
    SATsolverDPLLIterative solver;
    vector<Clause> clauses = { {{1}}, {{-1, 3}}, {{-2, 3}} };
    vector<int> assignments = {};
    assert(solver.unit_propagation(clauses, assignments) == true); // no conflict in clause
    assert(clauses.size() == 0); // 1 gets applied then 3 is unit clause and gets applied

    clauses = { {{1}}, {{-1, 3}}, {{-2, 3}}, {{-3}} };
    assignments = {};
    assert(solver.unit_propagation(clauses, assignments) == false); // There is a conflict
in clause after applying unit propagation
}
```

```
    void test_variable_count() {
        SATsolverDPLLIterative solver;
        vector<vector<int>> clauses = { {1, 2, 3}, {-1, -2, -3}, {1, 2, -3, -4} };
        int count = solver.variable_count(clauses);
        assert(count == 4); // Expecting 4 variables

        clauses = { {1, 2, 3, 4}, {-1, -2, -3, 5}, {1, 2, -3, -4}, {6}};
        count = solver.variable_count(clauses);
        assert(count == 6); // Expecting 6 variables

        clauses = {{}};
        count = solver.variable_count(clauses);
        assert(count == 0); // Expecting 0 variables
    }
void test_backtrack() {
    SATsolverDPLLIterative solver;
    vector<Clause> original_clauses = {{{1, 2, 3}},    {{1, 2, -3}}, {{1, -2, 3}},
  {{-1, -2, -3}}, {{-1, 2, 3}}, {{-1, 2, 3}},
  {{-1, -2, 3}},  {{-1, 2, -3}}};
  solver.original_clauses = original_clauses;
  vector<int> assignments = {1, -2, 3};
  vector<int> decision_assignments = {3, -2, 1}; // type vector but maintained as a stack
  vector<Clause> clauses = original_clauses;
  solver.apply_assignment(clauses, 1);
  solver.apply_assignment(clauses, -2);
  solver.apply_assignment(clauses, 3);
  int backtrack_assignment = solver.backtrack(clauses, assignments,
decision_assignments, 3);
  assert(backtrack_assignment == 3);

  assignments = {1, -2, -3};
  decision_assignments = {-3, -2, 1};
  clauses = original_clauses;
  solver.apply_assignment(clauses, 1);
  solver.apply_assignment(clauses, -2);
  solver.apply_assignment(clauses, -3);
  backtrack_assignment = solver.backtrack(clauses, assignments, decision_assignments, 3);
  assert(backtrack_assignment == 1);
}
```

Figure C-1: Unit, integration and stress tests for DPLL iterative implementation and the main function to run all tests.

C.3 CDCL Tests

```
void test_contains_emptyClause() {
    vector<Clause> clauses = { {{1}}, {{}} };
    SATsolverCDCL solver;
    assert(solver.contains_emptyClause(clauses) == true);

    vector<Clause> clauses2 = { {{1}}, {{1, 2}}, {{-2}} };
    assert(solver.contains_emptyClause(clauses2) == false);
}
```

```
void test_get_pure_literals() {
    SATsolverCDCL solver;
    solver.number_of_variables = 4;
    vector<Clause> clauses = { {{1, 2}}, {{3}}, {{-2, -3, 4}}, {{-2, -4}} };
    vector<int> pure_literals = solver.get_pure_literals(clauses);
    assert(pure_literals.size() == 1); // Expecting 1 pure literal
    assert(pure_literals[0] == 1); // Expecting pure literal 1

    clauses = { {{1, 2}}, {{-1, 3}}, {{-2, -3, 4}}, {{-2, -4}} };
    pure_literals = solver.get_pure_literals(clauses);
    assert(pure_literals.size() == 0); // Expecting 0 pure literal

    clauses = { {{1, 2}}, {{3}}, {{-2, 3}}, {{-2, -4}} };
    pure_literals = solver.get_pure_literals(clauses);
    assert(pure_literals.size() == 3); // Expecting 3 pure literal {1,3,-4}, order
doesnt matter.
}
```

```
void test_get_unit_clauses() {
    SATsolverCDCL solver;
    vector<Clause> clauses = { {{1,3}}, {{2}}, {{-2, 3}}, {{-3, 4}} };
    vector<int> unit_clauses = solver.get_unit_clauses(clauses).first;
    assert(unit_clauses.size() == 1); // Expecting one unit clause
    assert(unit_clauses[0] == 2); // Expecting unit literal 2

    clauses = { {{1,3}}, {{-1, 2}}, {{-2, 3}}, {{-3, 4}} };
    unit_clauses = solver.get_unit_clauses(clauses).first;
    assert(unit_clauses.size() == 0); // Expecting no unit clause

    clauses = { {{1,3}}, {{-1}}, {{-2}}, {{-3, 4}} };
    unit_clauses = solver.get_unit_clauses(clauses).first;
    assert(unit_clauses.size() == 2); // Expecting 2 unit clause
    assert(unit_clauses[0] == -1);
    assert(unit_clauses[1] == -2);
}
```

```
void test_is_assigned() {
    SATsolverCDCL solver;
    solver.assignments = {{1,1}, {-2,2}, {3,2}};

    assert(solver.is_assigned(1) == true);
    assert(solver.is_assigned(-1) == true); // this is true because variable 1 is
assigned to true
    assert(solver.is_assigned(4) == false);
}
```

```
void test_isSatisfied() {
    vector<Clause> clauses = { {{1, 2, 3}}, {{-1, 2, -3}} };
    vector<Assignment> assignments = {{1,1}, {2,2}}; // 1 is assigned at level 1 and
2 is assigned at level 2.
    SATsolverCDCL solver;
    solver.assignments = assignments;
    assert(solver.is_satisfiable(clauses) == true); // the clauses are satisfiable
by the assignments.

    solver.assignments = {{1,1}, {-2,2}};
    assert(solver.is_satisfiable(clauses) == false); // the clauses are
unsatisfiable by the assignments.
}
```

```
int main(){
  test_isSatisfied();
  test_contains_emptyClause();
  test_get_unit_clauses();
  test_unit_propagation();
  test_get_pure_literals();
  test_pure_literal_elimination();
  test_is_assigned();
  test_solve();
  test_variable_count();
  test_implication_graph();
  stress_test();
}
```

```
void test_pure_literal_elimination() {
    SATsolverCDCL solver;
    vector<Clause> clauses = { {{1, 2}}, {{-1, 3}}, {{-2, 3}}, {{-1,2}} };

    // Perform pure literal elimination
    assert(solver.pure_literal_elimination(clauses, 0) == true); // no conflict in
clause
    assert(clauses.size() == 0); // 3 gets applied, then 2, which leaves no
clauses unsatisfied
}
```

```
void test_solve() { // Test for solve function (integration test)
    SATsolverCDCL solver;
    vector<vector<int>> clauses = { {1, 2}, {-1, 3}, {-2, 3} };
    bool result = solver.solve(clauses);
    assert(result == true); // satisfiable

    clauses = {{1}, {-1}};
    result = solver.solve(clauses);
    assert(result == false); // unsatisfiable

    clauses = {{1, 2, 3, 4},{-1, -2, -3, -4},{1, 2, -3, -4}, {-1, -2, 3, 4}};
    result = solver.solve(clauses);
    assert(result == true); // satisfiable

    clauses = {
      {1, 2, 3, 4,},
      {-1, -2, -3, -4},
      {1, 2, -3, -4},
      {-1, -2, 3, 4},
      {1, -2, 3, -4},
      {1, -2, -3, -4},
      {-1, -2, -3, 4},
      {2},
      {-2},
      {-1, 2, 3, -4},
      {1, 2, -3, -4},
    };
    result = solver.solve(clauses);
    assert(result == false); // unsatisfiable

    clauses = {{1, 2, 3, 4, 5},{-1, -2, -3, -4, -5},{1, 2, -3, -4, 5},{-1, 2, 3, 4, 5},{1, -2, 3, -4, 5}};
    result = solver.solve(clauses);
    assert(result == true); // satisfiable

    clauses = {{-2,-3,-4}, {-3,-5,-6}, {4,6,7}, {-7,-8}, {-1,-7,-9},{-1,8,9}};
    result = solver.solve(clauses);
    assert(result == true); // satisfiable
}
```

```
void test_implication_graph(){
  ImplicationGraph graph;
  vector<int> assignments = {-2,-3,1}; // assignments are -2, -3, 1
  Assignment assignment = {assignments[2],1,false};
  Clause clause = {{1,2,3}}; // this clause becomes a unit clause with assignments
-2 and -3 so there will be an implication from -2 and -3 to 1.
  graph.addImplication(assignment, clause);
  assert(graph.graph.size() == 2);
  assert(graph.graph[-2].size() == 1);
  assert(graph.graph[-3].size() == 1);
  assert(graph.graph[-2][0].literal == 1);
  assert(graph.graph[-3][0].literal == 1);
}
```

```
void stress_test() {
    SATsolverCDCL solver;
    // Generate a large SAT formula
    vector<vector<int>> large_formula;
    const int num_variables = 100;
    const int num_clauses = 5000;
    srand(time(NULL));
    for (int i = 0; i < num_clauses; ++i) {
        vector<int> clause;
        for (int j = 0; j < 3; ++j) { // Each clause contains 3 literals, can be
changed
            int literal = (rand() % num_variables) + 1; // Random variable between
1 and num_variables
            if (rand() % 2 == 0) literal = -literal; // Randomly negate literal
            clause.push_back(literal);
        }
        large_formula.push_back(clause);
    }
    bool result = solver.solve(large_formula);
    // Assert that the solver returns an answer
    assert(result == true || result == false);
}
```

```
void test_unit_propagation() {
  SATsolverCDCL solver;
  vector<Clause> clauses = { {{1},0}, {{-1, 3},1}, {{-2, 3},2} };
  solver.original_clauses = clauses;
  assert(solver.unit_propagation(clauses,0) == true); // no conflict in clause
  assert(clauses.size() == 0); // 1 gets applied then 3 is unit clause and gets
applied

  clauses = { {{1},0}, {{-3},1}, {{-1, 3},2}, {{-2, 3},3} };
  solver.original_clauses = clauses;
  solver.assignments = {};
  assert(solver.unit_propagation(clauses,0) == false); // There is a conflict in
clause after applying unit propagation

}
```

```
void test_variable_count() {
    SATsolverCDCL solver;
    vector<vector<int>> clauses = { {1, 2, 3}, {-1, -2, -3}, {1, 2, -3, -4} };
    int count = solver.variable_count(clauses);
    assert(count == 4); // Expecting 4 variables

    clauses = { {1, 2, 3, 4}, {-1, -2, -3, 5}, {1, 2, -3, -4}, {6}};
    count = solver.variable_count(clauses);
    assert(count == 6); // Expecting 6 variables

    clauses = {{}};
    count = solver.variable_count(clauses);
    assert(count == 0); // Expecting 0 variables
}
```

Figure C-1: Unit, integration and stress tests for DPLL iterative implementation and the main function to run all tests.

Appendix D

Implementation

This is Appendix we showcase the implementation of functions which are similar to the functions shown in the implementation chapter 4

D.1 Unit-propagation

This section shows the implementation of the unit-propagation function for DPLL iterative and CDCL and the implementation of get-unit-clauses for CDCL.

```
bool unit_propagation(vector<Clause> &clauses, vector<int> &assignments) { // checks if a unit clause
exists and if it does then removes it from the clauses, stops when there are no more unit clauses.
    vector<int> unit_clauses = get_unit_clauses(clauses);
    while (unit_clauses.size() > 0) {
      for (int literal : unit_clauses) {
        if (!is_assigned(literal, assignments)) {
          apply_assignment(clauses, literal);
          assignments.insert(assignments.begin(),literal);
          number_of_assignments++;
        }
        if (contains_emptyClause(clauses)) { // beacuse of this iterative DPLL does less assignments,
it stops when a contradiction is found and does not continue to apply more assignments.
          // Found a contradiction
          return false;
        }
      }
      unit_clauses = get_unit_clauses(clauses);
    }
    return true;
  }
```

Figure D-1: Implementation of the unit-propagation function for DPLL iterative.

```
 pair<vector<int>, vector<int>> get_unit_clauses(vector<Clause>& clauses){ // gets the unit
clauses and their clause numbers.
    vector<int> unit_clauses;
    vector<int> unit_clause_numbers;
    for (int x = 0; x < clauses.size(); x++){
      if (clauses[x].literals.size() == 1){
        unit_clauses.push_back(clauses[x].literals[0]);
        unit_clause_numbers.push_back(clauses[x].clause_number);
      }
    }
    return make_pair(unit_clauses,unit_clause_numbers);
  }
```

Figure D-2: Implementation of the get-unit-clauses function for CDCL

```
bool unit_propagation(vector<Clause> &clauses, int decision_level) {
  pair<vector<int>,vector<int>> unit_clause_and_numbers = get_unit_clauses(clauses);
  vector<int> unit_clauses = unit_clause_and_numbers.first;
  vector<int> unit_clause_numbers = unit_clause_and_numbers.second;
  while (unit_clauses.size() > 0){ // while there are unit clauses apply them.
    for (int x = 0; x < unit_clauses.size(); x++){
      int literal = unit_clauses[x];
      if (!is_assigned(literal)){ //checks if the literal has not alreay been assigned.
        Assignment new_assignment = {literal, decision_level, false};
        Clause original_clause = original_clauses[unit_clause_numbers[x]]; //gets the original
clause of the unit clause.
        implicationGraph.addImplication(new_assignment, original_clause);
        assignments.push_back(new_assignment);
        apply_assignment(clauses, literal); // update the clauses with the new assignment.
        number_of_assignments++;
      }
      if (contains_emptyClause(clauses)) { // found a contradiction
          return false;
      }
    }
    unit_clause_and_numbers = get_unit_clauses(clauses);
    unit_clauses = unit_clause_and_numbers.first;
    unit_clause_numbers = unit_clause_and_numbers.second;
  }
  return true;
}
```

Figure D-3: Implementation of the unit-propagation function for CDCL

D.2 Pure-literal-elimination

This section shows the implementation of the pure-literal-elimination function for DPLL iterative and CDCL.

```
bool pure_literal_elimination(vector<Clause> &clauses, vector<int> &assignments) { // checks if a
pure literal exists and if it does then removes it from the clauses, stops when there are no more pure
literals.
  vector<int> pure = get_pure_literals(clauses);
  while (pure.size() > 0) { // pure_literal_elimination
    for (int literal : pure) {
      if (!is_assigned(literal, assignments)) {
        apply_assignment(clauses, literal);
        assignments.insert(assignments.begin(),literal);
        number_of_assignments++;
      }
      if (contains_emptyClause(clauses)) {
        // Found a contradiction
        return false;
      }
    }
    pure = get_pure_literals(clauses);
  }
  return true;
}
```

Figure D-4: Implementation of the pure-literal-elimination function for DPLL iterative.

```
bool pure_literal_elimination(vector<Clause> &clauses, int decision_level) {
  vector<int> pure = get_pure_literals(clauses);
  while (pure.size() > 0) { // pure_literal_elimination
    for (int i = 0; i < pure.size(); i++) {
      int literal = pure[i];
      if (!is_assigned(literal)) {
        Assignment new_assignment = {literal, decision_level, false};
        apply_assignment(clauses, literal);
        assignments.push_back(new_assignment);
        number_of_assignments++;
      }
      if (contains_emptyClause(clauses)) {
          // Found a contradiction
          return false;
      }
    }
    pure = get_pure_literals(clauses);
  }
  return true;
}
```

Figure D-5: Implementation of the pure-literal-elimination function for CDCL