KING'S College LONDON

University of London

A DPLL-Based SAT Solver Tuned For Imbalanced Instances

BSc Computer Science

Final Project Report

Author: Vitesh Dav Soni

Supervisor: Hana Chockler

Student ID: 1706891

April 22, 2020

Abstract

In this report, I focus on the CDCL algorithm and assess SAT solving techniques with the aim to develop an efficient SAT solver for imbalanced instances. I present a formula to quantify exactly how balanced a particular instance is and use it to test and tune the solver. I review various heuristics by comparing their efficiency on benchmarks and explain which heuristics were implemented in the accompanied SAT solver and why.

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.

Vitesh Dav Soni

April 22, 2020

Acknowledgements

I would like to express my sincere gratitude to my supervisor Dr. Hana Chockler for providing invaluable guidance, comments and suggestions throughout the course of the project.

Contents

Chapter 1

Introduction and Motivation

Ever since the Boolean satisfiability problem (SAT) was proven to be NP-complete it has been regarded as one of the most important problems in theoretical computer science. SAT is a major topic in many areas, including cryptography, verification and artificial intelligence [1]. Hence, research into SAT is very crucial for the future of computer science and its many practical applications.

SAT (in conjunctive normal form) is a problem which involves a conjunction of clauses to be satisfied, where each clause contains a disjunction of one or more literals. A clause is satisfied if at least one literal in the clause is assigned to true. An instance is declared satisfiable if every clause in the formula is satisfied.

As SAT is NP-complete, progression into the Boolean satisfiability problem will eventually either allow us to solve hard problems in polynomial time, or allow us to realise that problems in the NP complexity class are different from problems in the P complexity class. Although a polynomial time algorithm for the SAT problem currently does not exist, many techniques have been developed to reduce the search space and allow SAT solvers to run reasonably quickly. Heuristics such as Variable State Independent Decaying Sum (VSIDS) [2] and the Jeroslow-Wang method [3] have been vital for modern SAT solvers. New techniques such as clause learning, restart policies, deletion policies and the 2 watched literals data structure [4] have also been realised allowing modern SAT solvers to achieve far more than their predecessors.

The two most common complete algorithms implemented in SAT solvers are the Davis-Putnam-Logemann-Loveland (DPLL) algorithm and, in modern SAT solvers, the Conflict Driven Clause Learning (CDCL) algorithm [1]. DPLL algorithms "use a divide-and-conquer strategy: they

split a formula into two subproblems by fixing the a value of some literal, then they recursively process the arising formulas." [5]

There are three key methods which are used in DPLL-based SAT solvers to reduce the search space and time consumption:

- Unit propagation
- Pure literal elimination
- Backtracking

The program that is accompanied with this report will also employ a DPLL-based algorithm, and therefore the methods highlighted above were considered very carefully during the project.

To tackle a SAT problem, it is sometimes advantageous to have various SAT solvers tuned for different types of instances and have them run in parallel. For instance, a SAT solver can be tuned for satisfiable instances, balanced instances or even instances derived from a particular problem like the Graph Colouring problem [6]. The objective of this project is to develop a SAT solver designed to tackle imbalanced instances, that is, instances where most literals appear more often in one polarity. The program will allow users to interact with the SAT solver to solve randomly generated instances as well as any existing instances.

Chapter 2

Background

This chapter summarises key ideas from various research papers about SAT solvers and discusses how they were considered during the development of the accompanied program, known as PolarSAT. It also specifies, in detail, what balanced instances are and how they can be quantified.

2.1 Davis–Putnam–Logemann–Loveland Algorithm

Algorithm 1 Typical DPLL Algorithm

```
1:  procedure DPLL(assignment)
2:      unitPropagation()
3:      pureLiteralElimination()
4:      if conflict() then
5:          return false
6:      end if
7:      if satisfied() then
8:          return true
9:      end if
10:     if unassignedVariables() then
11:         v ← decideNextVar()
12:     else
13:         return false
14:     end if
15:     return DPLL(assignment ∪ {v}) or DPLL(assignment ∪ {−v})
16: end procedure
```

DPLL is a recursive algorithm which forms the foundation for many complete SAT solvers. It involves very smart techniques which are used to prune large search spaces. It operates by first carrying out unit propagation and pure literal elimination to simplify the formula and assign implied variables. It then branches on a variable and assigns it as either true or false. After

the formula is simplified based on the current partial assignment, if the formula is satisfied the instance is declared to be satisfiable. Otherwise, variables are recursively added to the assignment. If a conflict occurs and a clause cannot be satisfied it backtracks to the previous branching variable and assigns the opposite truth value to it. It then continues as before and recursively checks if the formula is satisfied. If all possible branches are exhausted and the formula is not satisfied, the formula is declared unsatisfiable [5].

A DPLL algorithm can be visualised using a binary tree. DPLL at its core is essentially a simple depth-first search algorithm which only prunes the search space whenever a conflict is encountered. It does not learn from past assignments like CDCL does.

A SAT solver, in the worst case scenario, may have to try every possible assignment combination in order to find a solution to a particular instance. The number of possible assignments grows exponentially based on the number of variables in the instance. Other factors such as the clause lengths and the number of clauses included in the instance may also affect the difficulty. Because of the nature of SAT problems and their exponential growth, more advanced techniques need to be added to DPLL SAT solvers for them to have a chance at performing efficiently. Simply applying unit propagation and pure literal elimination does not work well for very large or complex instances.

2.2 Conflict Driven Clause Learning Algorithm

CDCL is an algorithm which adheres to the core principles of DPLL. However, contrary to DPLL, a typical CDCL algorithm does not use recursion. It instead loops whilst there are unassigned variables. The other major differences between them are that CDCL uses [1]:

- The notion of decision levels
- Clause learning and conflict analysis
- Implication graphs
- Non-chronological backtracking
- Restart policies
- Deletion policies

Algorithm 2 Typical CDCL Algorithm

```
1: procedure CDCL
2:     assignment ← {}
3:     propagate()
4:     if conflict() then
5:         return false
6:     end if
7:     level ← 0
8:     while unassignedVariables(assignment) do
9:         level ← level + 1
10:        assignment ← assignment ∪ {decideNextVar()}
11:        propagate()
12:        while conflict() do
13:            {backtrackLevel, clause} ← analyseConflict()
14:            formula ← formula ∪ {clause}
15:            if backtrackLevel < 0 then
16:                return false
17:            else
18:                backtrack(backtrackLevel)
19:                level ← backtrackLevel
20:            end if
21:        end while
22:    end while
23:    return true
24: end procedure
```

In CDCL, each branching variable is assigned at a different decision level, from decision level 0 upwards. More specifically, the decision level is increased by one after every branching assignment. Implied variables, on the other hand, are always assigned at the current decision level. The notion of decision levels is very useful because they allow CDCL SAT solvers to backtrack to the most appropriate decision. Also, if a conflict occurs at decision level 0 the formula can immediately be declared unsatisfiable; a conflict at decision level 0 suggests that no decision made by the SAT solver led to the conflict and so nothing can be undone to resolve it.

Furthermore, every implied literal assignment in CDCL is also characterised by its antecedent clause. That is, the clause which implied its assignment through unit propagation. Using decision levels and antecedent clauses we are able to construct a directed acyclic graph known as an implication graph. Implication graphs are a visual representation of the current trail of assignments. Each node in the graph represents the assignment of a single literal and each edge represents the clause which implied the assignment [1]. They are useful for SAT problems in general because they can be used to prove that an instance is unsatisfiable.

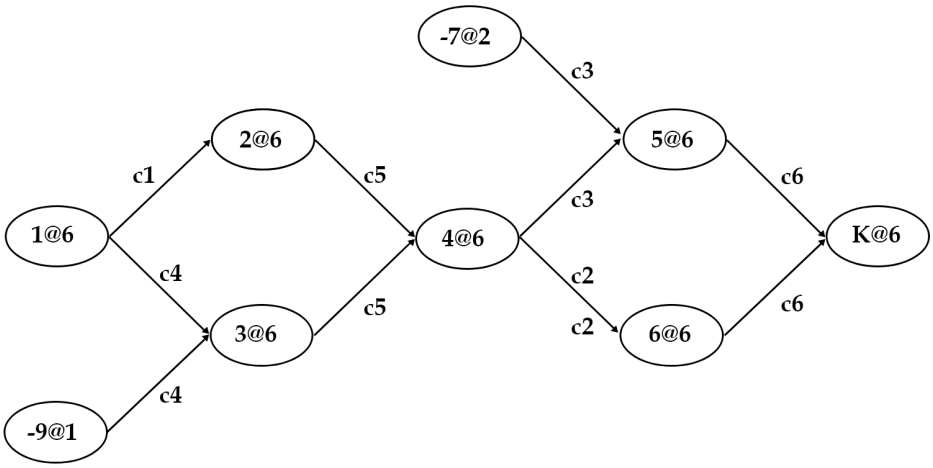


Figure 2.1: Example of an implication graph leading to a conflict

Every conflict must be analysed to undo the wrong decisions made by the solver. Conflict analysis involves two parts: learning a new clause and obtaining the appropriate decision level to backtrack to. We can learn useful clauses by resolving between two clauses which lead to the conflict. Two clauses are resolved by pivoting on a variable that is contained in both clauses. We can use the variables contained in the learned clause to imply a different and improved partial assignment.

A Unique Implication Point (UIP) is any node in the implication graph, other than the conflict node, that is on all paths from the first node at the conflict level to the conflict node. The first UIP is the UIP that is the closest to the conflict node. If we only resolve clauses up to the First UIP, the learned clause will be asserting; it will only contain one literal assigned at the conflict level [7][8][9]. The idea is to resolve clauses and pivot on the node at the first UIP, hence removing the literal that is on all paths from the first node at the conflict level to the conflict node. This node can, therefore, be considered to be the most responsible for the conflict. The first UIP strategy is widely used in CDCL SAT solvers to deduce a learned clause.

CDCL SAT solvers are not limited to backtracking to the previous decision level like DPLL, they are able to backtrack to any previous decision level. This is called non-chronological backtracking [1][7]. CDCL SAT solvers typically backtrack to the asserting level of the learned clause which is equal to the second highest decision level of any variable in that clause. However, if the clause is a unit clause its asserting level is equal to 0. Using asserting clauses and levels whilst backtracking have proven to be very useful for SAT solvers.

2.3 Branching Heuristics

SAT solvers rely on branching heuristics to choose the next variable. Branching heuristics have a significant effect on the performance as they shape the decision tree. Shai Haim and Marijn Heule propose a very interesting conjecture about heuristics [10]. They suggest that if a perfect branching heuristic exists which results in a solver that never needs to backtrack, and it is computable in polynomial time, then P is equal to NP. As a perfect heuristic does not exist, many techniques have been developed over the years in order to predict the ideal variable assignment.

The Dynamic Largest Individual Sum (DLIS) decision heuristic is perhaps the most intuitive heuristic. It suggests that we should simply choose the literal which satisfies the most clauses [11]. Although this heuristic is simple it is not very efficient because it requires us to iterate over all clauses for every variable in order to calculate the number of clauses they satisfy.

Perhaps the most popular heuristic to be used in modern SAT solvers is Variable State Independent Decaying Sum (VSIDS). VSIDS is a heuristic tailored for CDCL which assigns scores to each variable often referred to as their activity. The variable with the greatest score is chosen to be the next branching variable. The scores are based on the number of times the variable is in a learned clause. After every conflict, we increase the scores of every variable in the learned clause by some value, typically by 1. Periodically, the scores of all variables are multiplied by some constant between 0 and 1. This constant is called the decay factor [2]. This means variables that are in recently added learned clauses are prioritised over other variables.

Another efficient heuristic is the Jeroslow-Wang method. The Jeroslow-Wang method suggests that we calculate for every clause ω in φ and every literal l in ω:

$$J(l) = \sum_{l \in \omega, \omega \in \varphi} 2^{-|\omega|} \tag{2.1}$$

This gives an exponentially higher weight to literals in shorter clauses [3]. Shorter clauses are generally harder to satisfy as there are fewer literals in them. By satisfying shorter clauses first we are essentially reducing how frequently the SAT solver needs to backtrack; the number of possible variable assignments which satisfy smaller clauses is fewer than longer clauses.

Similarly to the Jeroslow-Wang method, the Maximum Occurence of clauses of Minimum size (MOM) heuristic also gives preference to literals in smaller clauses. MOM suggests that we

choose the literal l which maximises:

$$2^k(f(l) + f(!l)) + f(l)f(!l) \tag{2.2}$$

where $f(l)$ is the number of unsatisfied smallest clauses containing literal l.

Furthermore, this heuristic also prefers variables which are balanced over those which are not, making this heuristic very relevant to the project [11].

2.4 Restart Policies

Periodically restarting and backtracking to level 0, whilst retaining learned clauses, can improve the speed of SAT solvers greatly [12]. They help when a search is too focused on a small part of a search space. SAT solvers, initially, performed restarts after a fixed number of conflicts. Many efficient SAT solvers such as zChaff and BerkMin use a fixed sized restart strategies with sizes 700 and 550 respectively [10]. Executing a restart policy means that the entire search tree must be rebuilt. This can cause issues with completeness because it may limit the time provided to explore the search space properly [1]. MiniSAT 1.13 was the first to demonstrate a rather unique approach. It implements a restart policy which increases the number of conflicts required per restart geometrically. The geometric restart policy employed in MiniSAT has an initial restart interval of 100 conflicts. This interval is then increased by a factor of 1.5 after every restart [10] allowing the SAT solver to spend more time searching progressively. In modern days, restart policies have become essential because they counter heavy tailed behaviour, enhancing the performance of SAT solver, particulary in satisfiable instances [10].

2.5 Clause Deletion Policies

CDCL SAT solvers depend heavily on the clauses that they have access to. Having too many clauses at its disposal may result to a less efficient solver. On the other hand, removing too many learned clauses will limit the benefits of learned clauses [13]. It is important to find a balance between the two and only discard clauses which are not needed or result in a slower performance. It is important to note that clause deletion policies only discard learned clauses. Removing clauses contained within the original formula may lead to a completely different propositional formula and correctness issues.

Clauses deletion polices are used to identify which clauses to remove based on some properties. For example, one technique would be to discard clauses of a particular length. This strategy is called n-order learning [1]. Although this strategy is simple, it can be surprisingly effective for many instances.

More complex and advanced policies have also been realised allowing SAT solvers to recognise clauses which are not useful. A strategy used in some modern solvers discard clauses based on how active they are during clause learning and conflict analysis. Clauses with low activity are removed [1].

2.6 Existing SAT Solvers

Many effective SAT solvers have been developed in order to tackle the Boolean Satisfiability problem. SAT solvers such as MiniSAT and zChaff have been extremely influential in shaping the future of SAT solving. It is worth noting that the objective of this project is not to compete with these SAT solvers but instead to review their strategies. Some techniques will also be implemented in the accompanied SAT solver so they can be researched further and be tuned for imbalanced instances. MiniSAT is an extremely flexible and efficient SAT solver which had won every category in the 2005 SAT competition [14]. It performs very well on many different types of instances. The methods and heuristics employed in MiniSAT were often studied and evaluated on randomly generated imbalanced instances so that the accompanied SAT solver, PolarSAT, could be improved.

2.7 DIMACS Format

PolarSAT solves SAT instances in the DIMACS format. The DIMACS format is used to define a Boolean expression in conjunctive normal form. Each DIMACS CNF file includes a "problem" line which begins with "p cnf", followed by the number of variables and then the number of clauses. Each clause is listed one by one and on a different line. Each literal is represented by a single index: positive variables are indicated by using a positive index and negative variables are indicated using a negative index. Each variable is separated by a space. Each clause line ends with a "0". Hence, the index 0 is not allowed to be used to represent a literal. Additionally, comment lines begin with "c" [14].

2.8 Imbalanced Instances

A balanced instance is an instance in which each variable occurs roughly the same number of times in positive and in negative polarity. An imbalanced instance is one that is not balanced. When dealing with imbalanced instances it is important to be able to quantify exactly how balanced an instance is so that it can be compared with others. Given two imbalanced instances, one instance could be more (or less) balanced than the other. A formula was constructed for this purpose:

$$b = \sum_{i=1}^{n} \frac{|(P_i - Q_i)|}{n(P_i + Q_i)} \tag{2.3}$$

where

P_i is the number of times literal i appears in positive polarity,

Q_i is the number of times literal i appears in negative polarity, and

n is the total number of distinct variables in the instance.

Instances which have a b coefficient of 0 are balanced. Instances which have a b coefficient that is greater than 0 are imbalanced. Also, note that b cannot be less than 0 or greater than 1.

2.8.1 Example of a balanced coefficient b

Consider the following instance:

p cnf 3 2

1 -2 3 0

-1 2 -3 0

$$\begin{aligned}
\text{b} &= \textstyle\sum_{i=1}^{n} \frac{|(P_i - Q_i)|}{n(P_i + Q_i)} \\
&= \textstyle\sum_{i=1}^{3} \frac{|(P_i - Q_i)|}{3(P_i + Q_i)} \\
&= \frac{|(P_1 - Q_1)|}{3(P_1 + Q_1)} + \frac{|(P_2 - Q_2)|}{3(P_2 + Q_2)} + \frac{|(P_3 - Q_3)|}{3(P_3 + Q_3)} \\
&= \frac{|(1-1)|}{3(1+1)} + \frac{|(1-1)|}{3(1+1)} + \frac{|(1-1)|}{3(1+1)} \\
&= 0 + 0 + 0 \\
&= 0
\end{aligned}$$

As the coefficient b is equal to 0, the instance is balanced.

2.8.2 Example of an imbalanced coefficient b

Consider the following instance:

p cnf 4 3

1 -2 4 0

-2 -3 -4 0

2 0

$$
\begin{aligned}
\text{b} &= \textstyle\sum_{i=1}^{n} \frac{|(P_i - Q_i)|}{n(P_i + Q_i)} \\
&= \textstyle\sum_{i=1}^{4} \frac{|(P_i - Q_i)|}{4(P_i + Q_i)} \\
&= \tfrac{|(P_1 - Q_1)|}{4(P_1 + Q_1)} + \tfrac{|(P_2 - Q_2)|}{4(P_2 + Q_2)} + \tfrac{|(P_3 - Q_3)|}{4(P_3 + Q_3)} + \tfrac{|(P_4 - Q_4)|}{4(P_4 + Q_4)} \\
&= \tfrac{|(1-0)|}{4(1+0)} + \tfrac{|(1-2)|}{4(1+2)} + \tfrac{|(0-1)|}{4(0+1)} + \tfrac{|(1-1)|}{4(1+1)} \\
&= 0.25 + 0.833 + 0.25 + 0 \\
&= 0.5833
\end{aligned}
$$

As the coefficient b is not equal to 0, the instance is imbalanced.

The balanced coefficient can be used to easily classify instances as balanced or imbalanced. A threshold can be set at any value between 0 and 1. If the balanced coefficient of a particular instance is below the threshold it should be considered to be balanced. The instance should be considered imbalanced if its balanced coefficient is above the threshold. However, the classification of instances depends greatly on how high the threshold is set. For example if the threshold is set at 1, it implies that every instance should be considered to be balanced. Of course, this is not an ideal proposition. Instead, the formula may be used by SAT solvers and the threshold should be tuned.

Chapter 3

Requirements and Specification

This chapter entails the requirements of PolarSAT and discusses how each of the requirements will be assessed.

3.1 Functional Requirements

The functional requirements of the program are:

- R1 - The program shall generate random SAT instances.
- R2 - The program shall be able to quantify how balanced the instances are and display this metric to the user.
- R3 - The program shall allow the user to influence how balanced the randomly generated files are.
- R4 - The SAT solver shall display whether or not the instance is satisfiable or unsatisfiable.
- R5 - The SAT solver must be complete and sound.
- R6 - The program shall output and display one satisfying assignment to the user if the instance is found to be satisfiable.
- R7 - The program shall output the time taken for the SAT solver to solve the instance before terminating.
- R8 - The program shall allow the user to input an existing problem file before attempting to solve it.

3.2 Non-Functional Requirements

The non-functional requirements of the program are:

- R9 - The SAT solver shall respond with a solution within 5 minutes for imbalanced instances containing no more than 500 variables and 1000 clauses.
- R10 - The Solution Verifier shall quickly and automatically verify the solutions found by the SAT solver for instances containing a maximum of 500 variables and 1000 clauses.
- R11 - The program shall provide useful help and documentation which conveys how to use the software.
- R12 - The randomly generated SAT instances shall be recoverable by the user.

3.3 Specification

- S1 - Use the number of variables and clauses, as well as the clause lengths, entered by the user to construct a .cnf problem file containing a SAT instance in the DIMACS format. Each file must be accessible from the root of the project directory.
- S2 - Use the balanced coefficient formula outlined in Chapter 2 to calculate how balanced instances are and display the value to the command line. The program shall also inform the user whether the instance is positively or negatively imbalanced by counting how frequently literals appear in positive and negative polarity.
- S3 - The user shall be able to input the probability that a particular variable appears in positive polarity as a parameter of the program. This probability should then be used to influence the program when randomly deciding the polarity of variables in the clauses.
- S4 - After finding a solution to an instance, the program shall immediately print SATISFIABLE/UNSATISFIABLE to the command line if the instance is found to be satisfiable/unsatisfiable.
- S5 - Every satisfiable instance should be verified by the program to ensure that all clauses in the propositional formula are satisfied by the assignment that was found. The program must also ensure that the assignment does not contain conflicting variables (the same variable in both positive and negative polarity). Solutions to unsatisfiable instances must be rigorously tested by cross-validating the solution with other SAT solvers.

- S6 - After the program has found a solution to a satisfiable instance, each literal in the assignment must be displayed to the user as an ordered list.
- S7 - The program shall record the start time before attempting to solve an instance, and also record the end time when a solution to the instance is found. Before terminating, the program shall calculate the time elapsed during its execution and display it in milliseconds.
- S8 - The program shall have a second mode which allows the user to run the Solver on an existing problem file that contains a SAT instance in the DIMACS format. The user shall be able to run the solver on a file located at the root of the project directory by entering a single parameter that specifies a valid file name.
- S9 - The SAT solver shall solve randomly generated imbalanced instances with a maximum of 500 variables, a maximum of 1000 clauses and a balanced coefficient of at least 0.2 within 5 minutes.

  The efficiency of the Solver will be assessed by using an Intel Core i5-8250U Processor at 1.6GHz with 8GB of RAM on the Windows 10 operating system.
- S10 - The program shall verify solutions for randomly generated problem files containing a maximum of 500 variables and 1000 clauses in under 5 seconds.
- S11 - The user shall be able to run the program in a third mode which displays information about how to run the program in the command line. The help and documentation shall explain how the parameters may be used to run the SAT solver on an existing or randomly generated problem file.
- S12 - Any SAT instance generated by the program shall be stored and organised in a folder at the root of the directory. Each file shall be identifiable for the user by including the number of variables and clauses contained in the file as the file name.

Chapter 4

Design

This chapter reviews the design of PolarSAT by discussing its structure and flow. Every design choice was motivated by at least one requirement or by the overall objective of the project.

4.1 Overview

Clause learning is one of the most greatest improvements made to SAT solvers in recent times; there is major empirical evidence to suggest that clause learning reduces time consumption when solving many instances [15][16]. PolarSAT employs the CDCL algorithm because the algorithm, and its many features, were also expected to perform well for imbalanced instances.

Gradle was utilised to automate some of the tasks involved in the project. Gradle is a platform independent build tool which is useful to organise the project, compile and run code, and test the program. When testing, it can be used to generate a test report in the form of a HTML document. The reports may help identify bugs in the code very quickly.

Additionally, Gradle allows the program to be run using simple and intuitive commands. It allowed the program to be run on multiple instances fairly quickly and, therefore, eases the user experience and makes it more efficient to run or test. The program can be run on a randomly generated file using the command:

```
gradle run --args="v4 c15 a2 b4"
```

The parameter including “v” refers to the number of variables, the parameter including “c” refers to the number of clauses, the parameters beginning with “a” and “b” specify the lower and upper bound clause lengths, respectively. When a random problem file is generated, the

file name includes the number of variables and the number of clauses that it contains. These design choices made the features of the program memorable and recognisable.

As the program is a backend engine, it is designed to be accessed via the command line. The command line behaves as an interface which allows the user to interact with the solver. Information about the SAT solver and its processes are displayed in the command line as text.

The program is responsible for three key processes:

- It generates random problem files.
- It solves the entered problem file.
- It verifies any satisfying assignments found by the solver.

These processes were separated structurally using different classes, making the program loosely coupled.

4.2 Program Structure

The program consists of seven classes:

- Variable: contains data about a single variable such as its polarity, value, antecedent clause and the level at which it was assigned.
- Clause: represents clauses in the formula. Each clause consists of one or more literals. Each clause object holds information specifying whether it is learned and its clause ID.
- Pair: a very simple data structure which allows two objects to be returned simultaneously.
- Problem Generator: responsible for generating and storing problem files with respect to the constraints specified by the user.
- Solver: uses the CDCL algorithm and attempts to solve the SAT instance. It outputs a satisfying assignment if the instance is satisfiable.
- Solution Verifier: verifies the satisfying assignment returned by the SAT solver by checking if all the clauses in the formula are satisfied by the literals in the assignment.
- Master: contains the main method and is responsible for instantiating the other class objects.

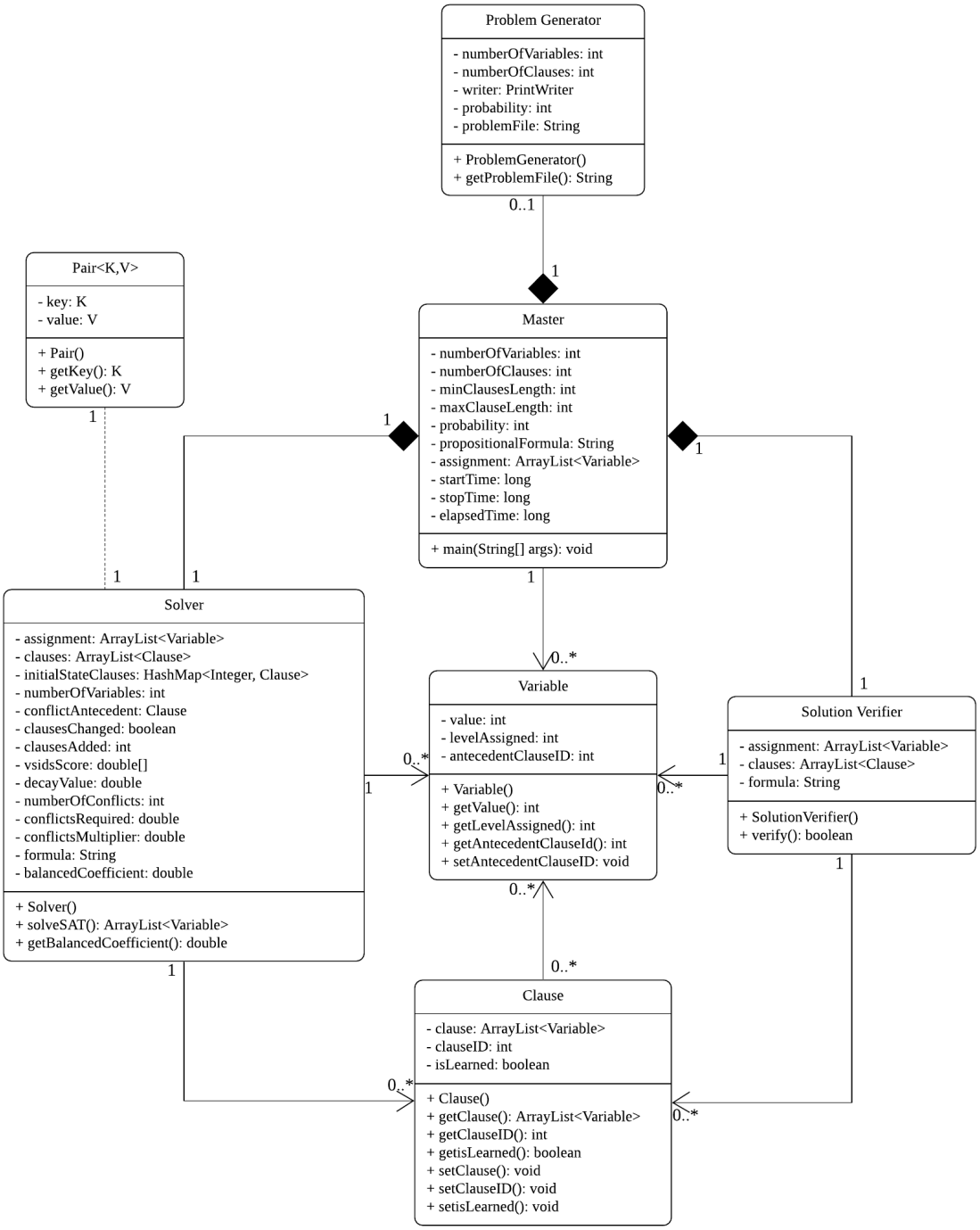


Figure 4.1: Class Diagram

For presentational purposes, the private methods in the class diagram have been omitted. However, it still conveys how the classes interact with each other and the overall structure of the program.

4.3 Flow of the Program

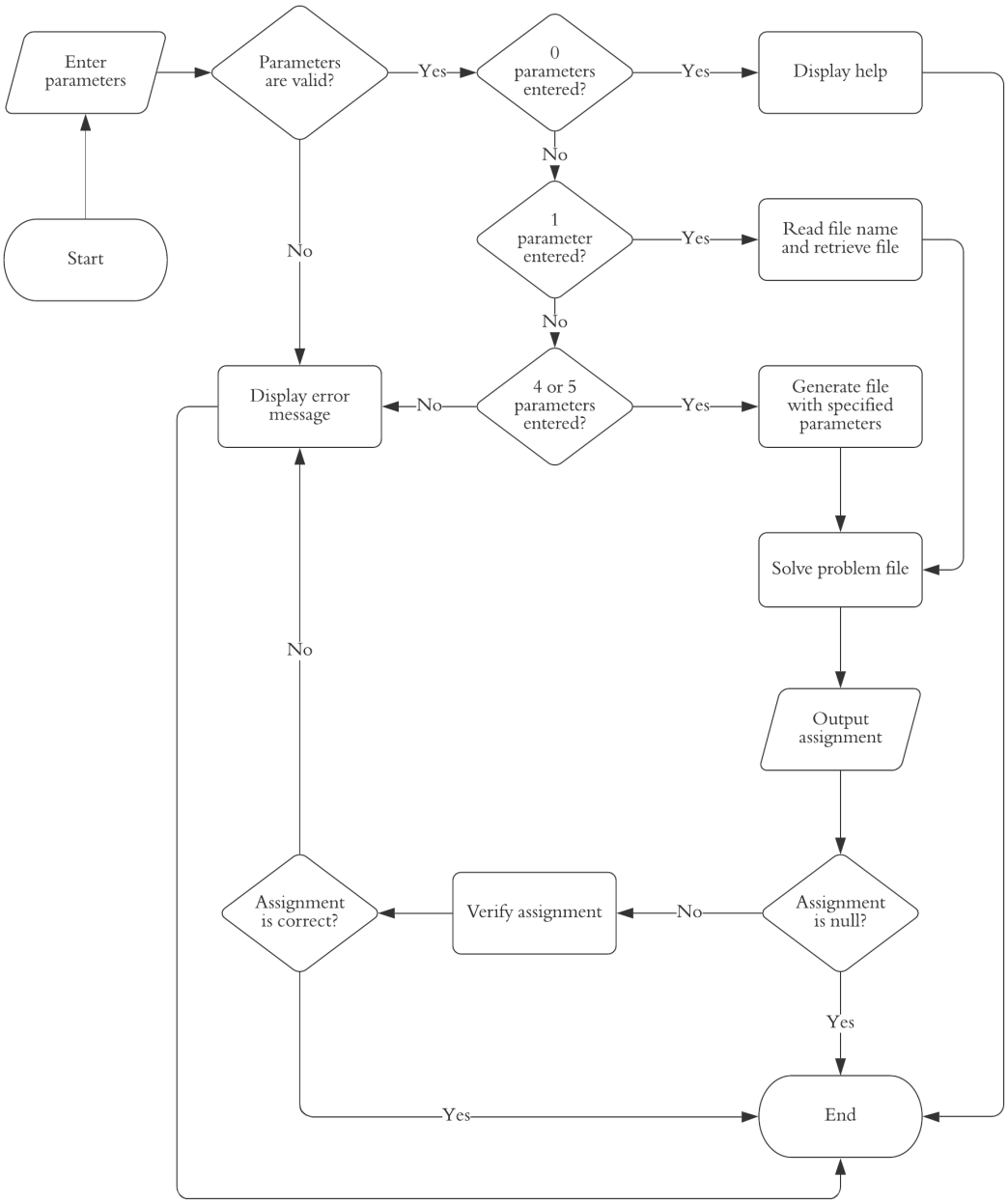


Figure 4.2: Flowchart

To run the program, the user must input some parameters. As the flowchart conveys, the program allows the user to input different parameters to carry out different tasks. If one parameter is entered, it suggests that the user wishes to run the SAT solver on an existing file, where the parameter is equal to the file name. If 4 parameters are entered, the program uses the parameters as constraints to construct a random file. The four parameters correspond to:

- The number of variables.
- The number of clauses.
- The minimum clause length.
- The maximum clause length.

The user may also enter an optional fifth parameter which is used to influence how balanced the random instance is. This fifth parameter is used to decide how probable a variable is to be in positive polarity. If it is set to 100 the polarity of every variable will be positive. Whereas, if it is set to 0 the polarity of every variable will be negative. Indeed, this value can be set to any value in between 0 and 100 to generate a problem file containing both positive and negative variables. For instance, to generate a reasonably balanced instance the parameter can be set to 50. If the user does not enter any parameters, they are presented with help and documentation that explains how to run the program.

The Solver then attempts to solve either an existing or a randomly generated instance depending on how many parameters were entered. After the SAT solver has found the solution to the problem, it outputs the satisfying assignment (if applicable) and the Master is responsible for displaying this data to the user as text.

The assignment is then verified by the Solution Verifier. If the assignment returned by the solver is equal to null it implies that the instance was found to be unsatisfiable. In this case, the Master informs the user that the instance is unsatisfiable and the program terminates without verifying the solution.

4.4 Usability

Whilst developing the program, each of the 10 Usability Heuristics for User Interface Design developed by Jakob Nielsen were carefully considered. These usability heuristics outline key design practices to follow when developing any sort of software to be used by a person [17]:

- Visibility of system status.
- Match between system and the real world.
- User control and freedom.
- Consistency and standards.

- Error prevention.
- Recognition rather than recall.
- Flexibility and efficiency of use.
- Aesthetic and minimalist design.
- Help users recognize, diagnose, and recover from errors.
- Help and documentation.

It was important for the program to, even without a graphical user interface, be transparent about its processes and provide useful feedback to the user. The program conveys the system status by printing relevant information, such as the satisfying assignments and any error messages, as text in an understandable and consistent manner.

The program also includes a help and documentation for the SAT solver which is simple and clear. The user may easily learn how to run the solver, specifically how to input any parameters. The error messages are also fairly transparent, allowing the user to diagnose any exceptions thrown by the program.

The SAT solver provides freedom and flexibility to the user by allowing them to run the solver on any SAT instance which is in the DIMACS format. This feature is especially useful to test the solver on particular benchmarks.

Chapter 5

Implementation

This chapter outlines and explains the types of methods that were incorporated into PolarSAT and how heuristics were adjusted to make the solver run faster.

5.1 Programming Language

PolarSAT was programmed using Java because its object-orientated nature allowed each element in the program to be modelled as a distinct component. It was important for each method and class to be organised. Every method in the program does exactly one task. This modular design helps to remove or change heuristics very easily. Moreover, the Java library contains many data structures which can be utilised very easily with the help of the useful documentation that is provided. Other object-orientated languages such as C++ could also have been used to implement the program, however, they do not provide any additional advantages over Java in the context of the project.

5.2 The Solver

As stated in the Chapter 4, PolarSAT implements the CDCL algorithm. The SAT solver utilises many techniques such as clause learning and non-chronological backtracking as well as many heuristics. A discussion about the implementation of each of these aspects follows.

5.2.1 Data Structures

Many aspects of the Solver are implemented using array lists. For instance, the Solver keeps a record of the assignment as an array list of Variables and the formula is represented as an array list of clauses. This data structure keeps the program simple; each time a variable is assigned, it is simply added to the list. Similarly, every time a learned clause needs to be added to the formula it is simply added to the clause list. Whenever a clause is satisfied it is removed from the list.

The assignment list could have been implemented using a set data structure to make sure that there are no duplicate or conflicting variables, however, the order of the elements in a set do not always correlate to the order in which they were entered. Using an array list was helpful for backtracking and removing variables after a certain level because it meant the the Solver only needed to consider elements at one end of the list.

The Pair class is a data structure which is only used during conflict analysis. It is used to return the backtrack level and the learned clause simultaneously. The learned clause is then added to the clause array list and the current decision level is set to the backtrack level.

5.2.2 Unit Propagation

```
private boolean propagate(int level){
        clausesChanged = false;
        unitPropagation(level);
        simplifyClauses(assignment.size(), level);
        if(checkForConflicts()){
                return true;
        }
        if(clausesChanged){
                return propagate(level);
        }
        return false;
}
```

Propagation is implemented as a recursive method which simplifies the clauses in the formula repeatedly whilst assigning literals in unit clauses until it can no longer do so. The clausesChanged variable is a boolean value accessible from any method in the class. When a clause

is simplified or a new variable is assigned, the variable is assigned to true and as a result the propagate method is called in order to attempt to simplify the clauses further.

5.2.3 Non-Chronological Backtracking

Each clause is characterised by:

- the variables contained in it, as an array list.
- whether or not it is a learned clause, as a boolean attribute.
- its clause ID, as an integer.

The Solver keeps a record of every clause that had been created in a separate hash map called initialStateClauses. The hash map stores all clauses, including learned clauses, in their initial state and is used to access particular clauses using the clause ID as the key.

```
private void backtrack(int backtrackLevel){
        ArrayList<Variable> varsRemove = new ArrayList<Variable>();
        for(Variable v: assignment){
                if(v.getLevelAssigned() >= backtrackLevel){
                        varsRemove.add(v);
                }
        }
        for(Variable v : varsRemove){
                assignment.remove(v);
        }
        clauses = new ArrayList<Clause>();
        for(Clause c : initialStateClauses.values()){
                clauses.add(new Clause(c));
        }
}
```

To backtrack the state of the clauses, every clause in the initialStateClauses hash map are added to the clause array list and then simplified based on the partial assignment. Using a stack to backtrack was also considered during the implementation phase. The idea was to push clause states onto the stack before the decision level is incremented and pop states off the stack to backtrack. Although, this may have been more efficient it meant that multiple states, which

includes the clauses and the partial assignment, needed to be stored for every decision level. This ultimately would have led to more memory consumption than necessary.

5.2.4 Clause Learning

Just like any CDCL SAT solver, the program learns clauses by resolving upon the antecedent clauses of variables assigned at the conflict level. As discussed in Chapter 2, it is beneficial to backtrack up to the First UIP in the implication graph. The implication graph is modelled through variable assignments and their antecedent clauses. The program uses the method soleLitAtLevel() to resolve clauses until it encounters a conflict clause containing exactly one variable assigned at the conflict level. The learned clause is then added to the clause array list and the initialStateClauses hashmap. Simultaneously, the VSIDS activity scores of the variables contained in the learned clause are updated.

5.3 The Problem Generator

Random CNF problems are generated using the parameters entered into the Problem Generator class. To construct a number of clauses, the Problem Generator begins by randomly deciding on the length of the clause.

```
while(clause.size() != clauseLength){
        double polarity = Math.random();
        Random r = new Random();
        int nextVar = r.nextInt((numberOfVariables - 1) + 1) + 1;
        if(polarity < (double)probability/100) {
                clause.add(nextVar);
        }
        else{
                clause.add(-(nextVar));
        }
}
```

Of course, the length of the clause respects the constraints entered by the user. For every clause, the Problem Generator writes random variables to a file until the number of literals in the clause is equal to the length decided. As the variables in the clauses are chosen randomly, it is not guaranteed that the problem file will contain the exact number of variables specified by

the user. When the number of clauses constructed is equal to the number of clauses required by the user, the Problem Generator terminates its construction and closes the writer. The problem file is then saved in a folder and the path of the file is printed to the console. It is worth noting that the problem file is created and is returned to the Solver as a string. It is the responsibility of the Solver to understand and parse the string by storing it in useful data structures such as an array list.

5.4 The Solution Verifier

The Solution Verifier is a simple assignment checker. After the Solver has found a solution to a particular problem, the Master inputs the problem formula and the assignment into the Solution Verifier. It then reads the input and organises the clauses into lists similarly to the Solver.

```
public boolean verify(){
        if(formula.trim().isEmpty()){
                return true;
        }
        simplifyClauses();
        return clauses.size() == 0 && checkAssignment();
}
```

The Solution Verifier contains a method called verify(). The method begins by checking if the file contains any clauses. If it does not, the instance is satisfiable regardless of the assignment. It then simplifies the clauses based on the assignment by discarding clauses if they include an assigned variable. If the list of clauses is empty it means that all the clauses were satisfied and the assignment is correct. If the list of clauses is not empty, it means that the assignment is incorrect. Finally, the Verifier checks if the assignment does not contain any conflicting variables. An assignment is incorrect if it contains a particular variable in both positive and negative polarity. The verify() method returns true if the the clauses are all satisfied and the assignment is correct, and false otherwise.

5.5 Tuning the Solver

Many heuristics are included in the SAT solver. These heuristics will certainly need to be tuned so that they are efficient for tackling imbalanced instances. The program automatically outputs the balanced coefficient of the instance and the time taken to solve the instance before terminating. These two data values are helpful for examining the efficiency of the SAT solver and to compare its performance on instances which are imbalanced to varying extents.

Many experiments were conducted to tune the SAT solver and its heuristics for imbalanced instances. All experiments were carried out using the same 12 randomly generated benchmarks.

Benchmark No.	No. of Variables	No. of Clauses	Satisfiability	Balanced Coefficient
1	700	6000	SAT	0.13 (+)
2	500	3000	SAT	0.26 (−)
3	200	3300	UNSAT	0.51 (−)
4	200	1000	SAT	0.35 (−)
5	450	5000	SAT	0.11 (−)
6	320	2400	SAT	0.15 (+)
7	200	2400	UNSAT	0.12 (+)
8	600	3000	SAT	0.26 (+)
9	600	4250	SAT	0.31 (+)
10	250	2000	UNSAT	0.15 (−)
11	300	4550	SAT	0.69 (−)
12	400	8550	SAT	0.74 (+)

Table 5.1: Benchmark features

Each benchmark is relatively imbalanced but are different in size. Also, some instances are satisfiable and some are not. The referenced benchmarks can be found in the cnf folder at the root of the project directory. The SAT solver had a maximum of 5 minutes to solve each instance. If the time limit was exceeded the upper bound time of 5 minutes was recorded for that particular instance. All tests were carried out using an Intel Core i5-8250U Processor at 1.6GHz with 8 GB of RAM on the Windows 10 operating system.

5.5.1 Branching Heuristic

The branching heuristic is a significant aspect of the Solver. An experiment was conducted to learn which heuristic reduces the time consumption of the Solver the most. Four heuristics were tested on the 12 benchmarks: DLIS, VSIDS, MOM and the Jeroslow-Wang Method. The time required by the Solver when employing each of these heuristics was recorded for each benchmarks and the total time was used to compare the performance. For this experiment, VSIDS employs a decay factor of 0.95 as this is the value that is typically utilised by other SAT solvers such as MiniSAT.

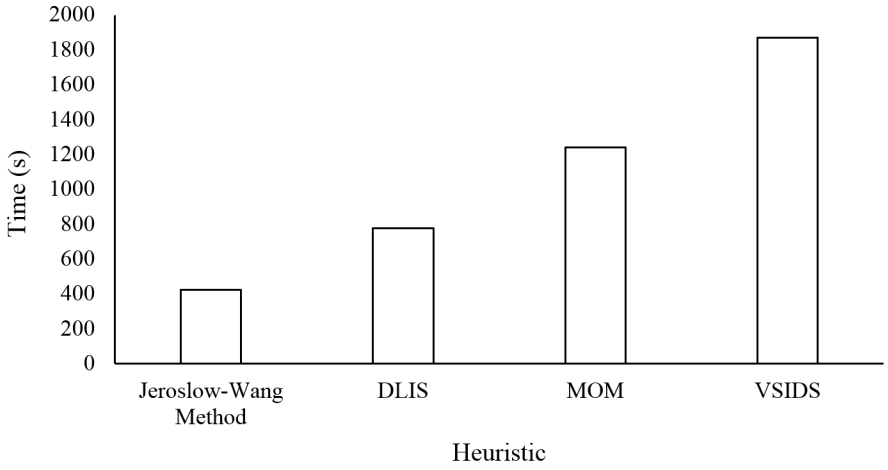


Figure 5.1: Comparison of heuristics

Surprisingly, VSIDS performed the least efficient. The Jeroslow-Wang method performed over 78% quicker than VSIDS making it the most efficient heuristic. The Solver did not perform well for balanced instances, particularly instances with a balanced coefficient of less than 0.2. In fact, the Solver was unable to find a solution to benchmark 6 within 5 minutes using any of the heuristics.

Evidently, DLIS and MOM are not the ideal heuristics to employ for imbalanced instances. The implementation of MOM in particular was found to be quite complex as it required looping over many of the clauses in order to find the next branching assignment. As these heuristics are used very regularly during the execution of the program, the inefficiency of any structurally complex heuristics are amplified and ultimately led to a higher time to solve all 12 benchmarks.

5.5.2 Combining Branching Heuristics

Although the Jeroslow-Wang method performed significantly better than VSIDS, another experiment was conducted to learn if combining heuristics with VSIDS would improve its speed. VSIDS is a great heuristic which is tailored for CDCL, however, it is unable to choose variables sensibly before a conflict has occurred. Therefore, it seemed reasonable to use another branching heuristic before a conflict has occurred and then use VSIDS only when the SAT solver has access to scores which contain useful information about the variables. The Jeroslow-Wang method, DLIS and MOM were all combined with VSIDS during the following experiment to find the best combination and to see if it performed better than using the Jeroslow-Wang method individually. Similarly to the previous experiment, the time taken to solve all 12 benchmarks was recorded for each combination.

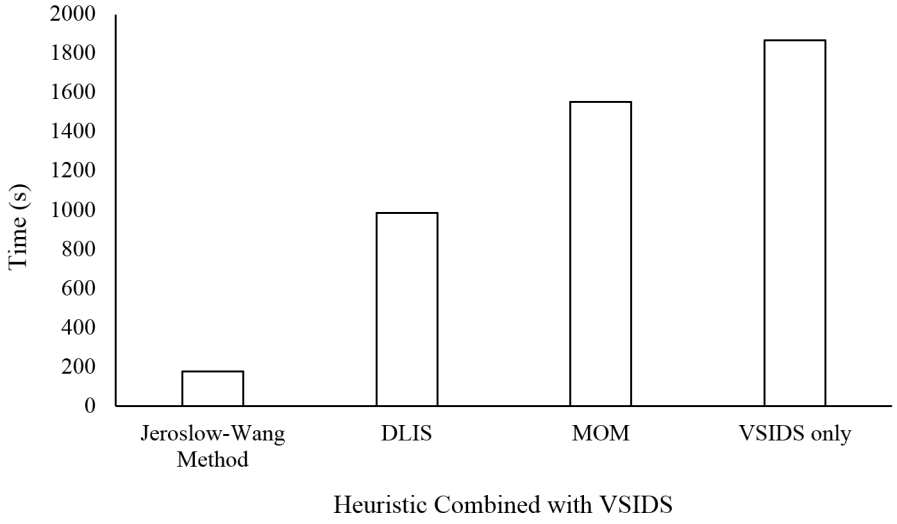


Figure 5.2: Comparison of heuristics when combined with VSIDS

As the Jeroslow-Wang method, DLIS and MOM all individually performed more efficiently than VSIDS, it was expected that the combination of VSIDS with any other heuristic would enhance the performance of VSIDS. It is clear from the findings that the best heuristic to combine with VSIDS is the Jeroslow-Wang method. Interestingly, combining the Jeroslow-Wang method with VSIDS made the SAT solver perform significantly faster compared to using the Jeroslow-Wang method individually. Due to the findings of this experiment, the VSIDS heuristic combined with the Jeroslow-Wang method was chosen and implemented as the branching heuristic for PolarSAT.

5.5.3 VSIDS Decay Factor

The previous experiments employ a VSIDS decay factor of 0.95, however, this value was not calculated or tuned for imbalanced instance but instead just extracted from MiniSAT. It was important to verify if 0.95 is in fact the best decay factor because it has a great effect on the overall performance of the heuristic, and therefore was of great interest when tuning. The time taken for the SAT solver to solve all 12 benchmarks was measured and the decay factor was changed to monitor its effect on the performance of the solver.

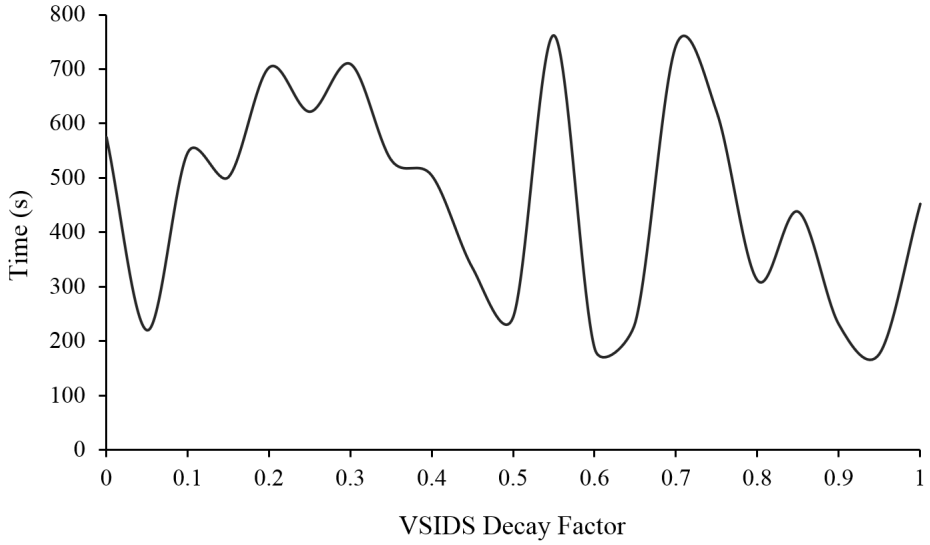


Figure 5.3: The effect of different VSIDS decay factors

The graph illustrates that the best decay value for the VSIDS heuristic is 0.95; it required the least time to solve all 12 benchmarks. As MiniSAT employs the same VSIDS decay factor, it adds credibility to the findings.

It was surprising to find that a decay factor of 0.05 and 0.60 performed so well on the benchmarks. This really emphasises that there is not a very clear correlation between the decay factor and the time taken to solve instances. Increasing the sample size of benchmarks perhaps would have led to a more clearer trend between the two variables, however, this was not within the scope of the project due to time constraints.

5.5.4 Polarity of Variable Assignments in VSIDS

When using VSIDS it is common to always assign variables in negative polarity, this is the approach that MiniSAT takes. However, this could be quite inefficient for imbalanced instances

because they contain variables which appear more often in one polarity. We can use this information to make better decisions about the polarity of the variable assignments.

A new technique called polarity tuning is implemented in PolarSAT, hence the name of the program. This technique assigns variables in a particular polarity based on how balanced the instance is:

- If the instance is imbalanced, variables are assigned in the polarity that appears the most frequently in the instance.
- If the instance is balanced, variables are assigned in the polarity that appears the least frequently.

This method should increase the chances of satisfying a clause with a particular variable assignment in imbalanced instances.

```
int sign = 1;
if(balancedCoefficient < 0){sign = -1;}
if(Math.abs(balancedCoefficient) >= threshold){
        nextVarVal = nextVarVal*sign;
}
else{
        nextVarVal = nextVarVal*sign*-1;
}
```

Depending on what is considered to be a balanced instance, polarity tuning can be implemented in different ways. We can use the balanced coefficient to set a threshold which specifies what a balanced instance is. For example if the threshold is set at 0.5, it means that instances with a balanced coefficient of 0.5 or greater shall be considered to be imbalanced, and instances with a balanced coefficient of less than 0.5 shall be considered to be balanced.

The following experiment was conducted to learn if the time consumption of the SAT solver decreases after implementing polarity tuning and to find the best threshold for polarity tuning.

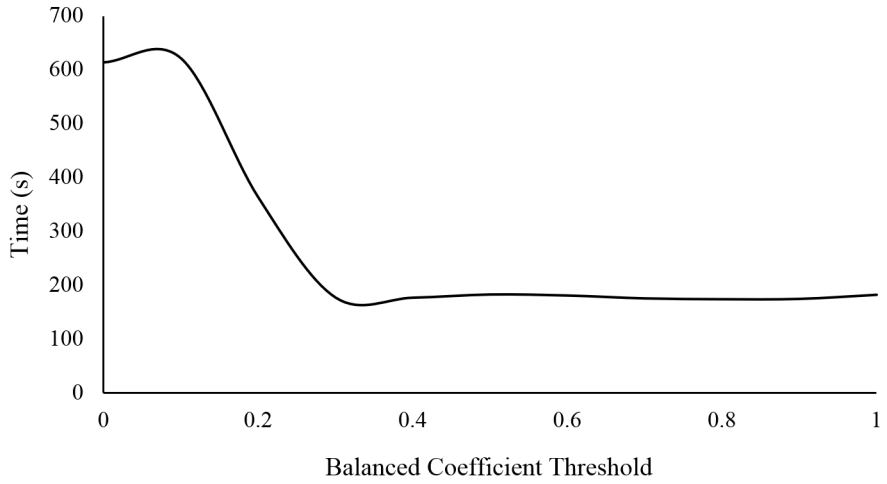


Figure 5.4: The effects of polarity tuning when combined with VSIDS

According to the results, polarity tuning is most effective when instances with a balanced coefficient of 0.3 or greater are considered to be imbalanced. As expected, polarity tuning is not efficient when instances with a balanced coefficient of less than 0.3 are considered to be imbalanced.

To demonstrate the efficiency of polarity tuning, two implementations of the SAT solver were compared: one implemented polarity tuning with a threshold of 0.3, and the other did not implement polarity tuning at all. All other aspects of the SAT solver were the same.

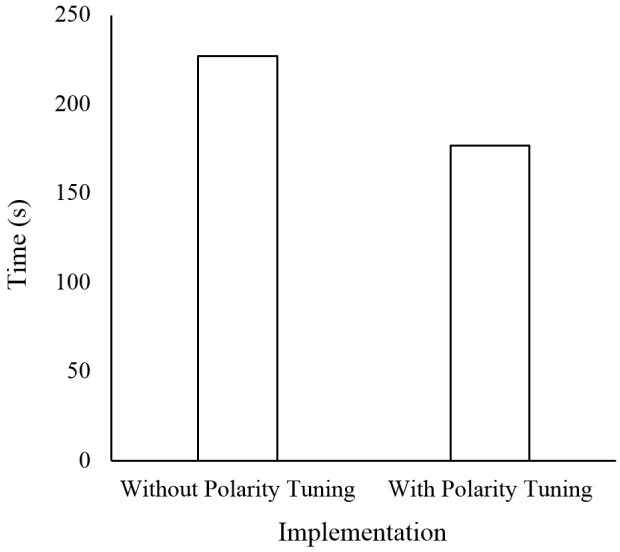


Figure 5.5: The effectiveness of polarity tuning

Polarity tuning does in fact reduce the overall time consumption of the SAT solver. The SAT solver that implemented polarity tuning performed significantly better for imbalanced instances than the SAT solver which did not. In fact, the SAT solver which implemented polarity tuning dominated in performance in every benchmark except benchmark 5. This is not surprising because benchmark 5 has a balanced coefficient of 0.11, meaning that it is relatively balanced.

5.5.5 Restart Policy

The restart policy that is implemented is extracted from MiniSAT. It is a geometric restart policy where the SAT solver restarts whenever the number of conflicts that have occured is higher than a certain threshold. Initially the threshold is set at 100 conflicts but it is increased by 1.5 after every restart.

5.5.6 Clause Deletion Policy

Finally, the deletion policy was implemented and tuned. The SAT solver employs a n-order deletion policy, which discards clauses of length greater than n. In the accompanied SAT solver, the deletion policy is only executed after every restart. To tune the policy different values of n, ranging from 0 to 12, were tested on each benchmark.

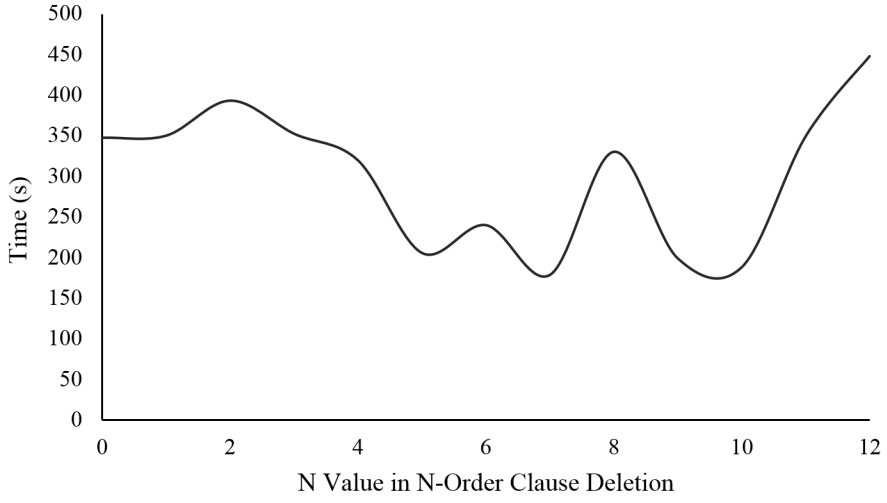


Figure 5.6: Tuning the n-order learning deletion policy to find the most efficient value of n.

Based on the findings, the best value of n is 10. This means that clauses which contain more than 10 literals are to be removed during the execution of the policy. This value of n is, therefore, used in PolarSAT.

Chapter 6

Legal, Social, Ethical and Professional Issues

SAT solvers in general are used in many practical applications, therefore, returning an incorrect solution can have negative repercussions in real world situations. The software was tested sufficiently and the solutions of many instances were verified rigorously over the course of the project to establish credibility.

The software was developed to research SAT solvers and the effects on imbalanced instances, however, the SAT solver can always try to solve an existing problem file that relates to a real problem. As the problem files are encoded using the DIMACS format it is difficult to understand exactly what the variables refer to. This means that it is not possible for any confidential information that may be entered in the form of a problem file to be disclosed to any other party.

As the software is a back end engine designed to be used by very a specific group of people, many of the social and ethical principles that would normally be associated with a standard software do not apply to this project.

Chapter 7

Evaluation

This chapter reviews the requirements of the software to evaluate if they were achieved successfully and discusses the limitations of the SAT solver.

7.1 Evaluation of Requirements

1. **The program shall generate random SAT instances.**

   As stated previously, the user may enter the number of variables, number of clauses and clause lengths by using parameters. Each of the parameters must include a prefix denoting the meaning of the parameters. Each of the prefixes were chosen in a way that made running the program memorable for the user. The program generates files by randomly choosing valid variables to construct clauses. The Problem Generator does this until the problem contains the exact number of clauses specified by the user.

2. **The program shall be able to quantify how balanced the instances are and display this metric to the user.**

   The SAT solver implements the balanced coefficient formula in the calculateBalancedCoefficient() method in order to quantify how balanced instances are. This method is called by the Master and it returns a double.

   ```
   Solving...
   Balanced coefficient: 0.1 (-)
   ```

   The program also informs the user if the file is positively imbalanced or negatively imbalanced using the plus or minus symbol. This was not specifically a requirement of the Solver, however, it is additional information which may be useful to the user.

3. **The program shall allow the user to influence how balanced the randomly generated files are.**

```
C:\SAT-Solver>gradle run --args="v2 c5 a1 b3 p70"
```

The user may use the optional fifth parameter with the prefix "p" to specify the probability that a variable will be in the positive polarity. Adjusting this probability will change the balanced coefficient of the randomly generated instance. Typically, setting the probability to a value that is greater than 50 will generate a file which is positively imbalanced, and setting the probability to a value which is less than 50 will generate a file which is negatively imbalanced.

4. **The SAT solver shall display whether or not the instance is satisfiable or unsatisfiable.**

```
SATSIFIABLE.
```

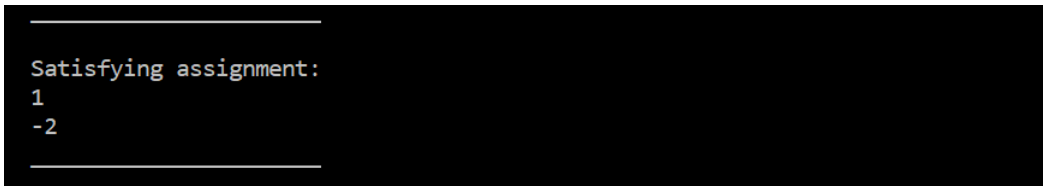
When a satisfying assignment is found for a particular instance, the program informs the user that the instance is satisfiable. Similarly if the instance is unsatisfiable, the program prints unsatisfiable to the command line.

5. **The SAT solver must be complete and sound.**

Every time the Solver finds a satisfying assignment for a particular instance, the Solution Verifier is instantiated by the Master and the verify() method is called. If the solution is found to be correct, the program does not output anything to the command line. However, when the solution is found to be incorrect an error message is displayed, indicating to the user that the assignment is incorrect. It is implemented in the program to act as a safety net, and to identify any bugs in the code.

Although it is impossible to ensure that the SAT solver behaves appropriately in all cases, the thorough experimentation and testing of the program helped minimise the number of errors in the code. Many black box tests were written to evaluate the effectiveness of the SAT solver and unit tests were used to assess the correctness of key methods used during the execution of the Solver. Black box testing was carried out by running the SAT solver on many satisfiable and unsatisfiable instances. As the solution to these instances were known, it was straightforward to test the output of the SAT solver to verify if it is correct.

6. **The program shall output and display one satisfying assignment to the user if the instance is found to be satisfiable.**



The program displays the first satisfying assignment found by printing each variable in an ordered list. If the instance is unsatisfiable the program does not output a satisfying assignment, as expected.

7. **The program shall output the time taken for the SAT solver to solve the instance before terminating**

Before solving an instance, the program begins a timer. The timer is stopped before the termination of the program. The time taken for the program to solve the instance is then displayed to the user in milliseconds.

8. **The program shall allow the user to input an existing problem file before attempting to solve it.**

The user is able to input an existing file, that contains a valid SAT instance in the DIMACS format, by entering the file name as a single parameter. If an existing problem file is entered, the Problem Generator is not instantiated by the Master. Instead, the text from the file is entered into the SAT solver as a string.

7.2 Evaluation of Non-Functional Requirements

1. **The SAT solver shall respond with a solution within 5 minutes for imbalanced instances containing no more than 500 variables and 1000 clauses.**

Although the SAT solver performs very well on randomly generated imbalanced instances, this requirement was not completely fulfilled. Section 7.2 discusses the limitations of the program and discusses a brief experiment that was conducted to test the effectiveness

of the Solver. The findings of that experiment conveyed that the SAT solver did not always find a solution to instances with the properties outlined by the requirement within 5 minutes. Failure to fulfill a requirement, especially a non-functional requirement, does not signify that the entire project was unsuccessful. However there is definitely room for improvement, particularly in the implementation.

2. **The Solution Verifier shall quickly and automatically verify the solutions found by the SAT solver for instances containing a maximum of 500 variables and 1000 clauses**

   The Solution Verifier has proved to be very efficient during the testing and experimentation phase of the project. Instances with 500 variables or less are verified very quickly, typically within 5 seconds. Instances with more variables, however, do tend to take more time to be verified but this is expected due to the way the verifier is implemented.

3. **The program shall provide useful help and documentation which conveys how to use the software.**

   The help and documentation of the program may be used by a user to understand the different tasks available. It explains, with examples, how to enter parameters into the program.

4. **The randomly generated SAT instances shall be recoverable by the user.**

   The program displays the path of the generated file, allowing the user to open, view or change it however they wish. The file name of any generated instance specifies the number of variables and clauses in the instance, making each generated file identifiable.

7.3 Limitations

Although the software fulfilled many of the requirements outlined in Chapter 3, it is not perfect. There is always room for improvement. The implementation of the SAT solver could have been improved in many areas. It is important for methods in the SAT solver to be as efficient as possible because SAT solvers in general regularly call the same methods over the course of their execution. This means that any inefficient behaviour in the SAT solver is typically amplified over time as the algorithm loops. After some analysis of PolarSAT, it was transparent that unit propagation consumes the most time.

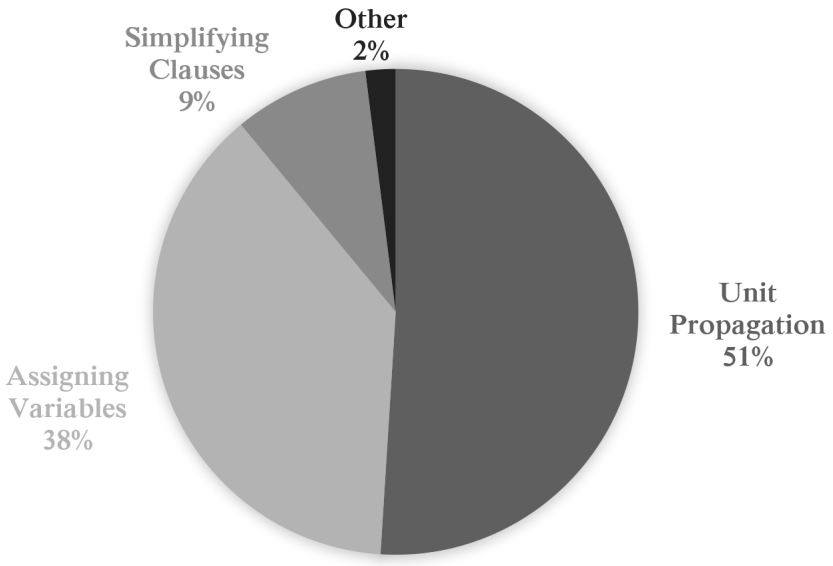


Figure 7.1: Average time consumption of key methods in the solver

In fact, unit propagation consumes up to 51% of the time on average to solve instances. This is a major portion of the programs execution time. The implementation of unit propagation in the accompanied SAT solver is admittedly quite naive. It recursively simplifies the formula and assigns literals contained in unit clauses until it can no longer do so. By implementing the two watched literal data structure the performance of the SAT solver could be significantly improved, making it even more effective and efficient for larger and complex instances.

Furthermore, the implementation of many aspects of the SAT solver depended heavily on the experiments conducted whilst tuning. Although the benchmarks that were used to tune the SAT solver were carefully chosen and were diverse, it is difficult to say whether the findings are generalisable to all types of imbalanced instances. Perhaps the tuning of the SAT solver had overfit the benchmarks. Additionally, due to the time constraints of the project the number of benchmarks used were limited to 12. The relatively small sample size reduced the chances of being able to visualise a pattern in some cases.

To assess the effectiveness of the Solver, it was imperative to analyse the time taken by the Solver to solve instances which differed in the ratio of clauses per variable. The SAT solver was tested on 200 randomly generated imbalanced instances. Every instance was 3SAT to ensure that there were no extraneous variables which affected the difficulty of the instances. Furthermore, the tested instances contained a different number of clauses, ranging from 100 to 1000 with an interval of 100. Each clause range category was tested with clauses to variable ratios, ranging from 1.0 to 10.0 with an interval of 0.5.

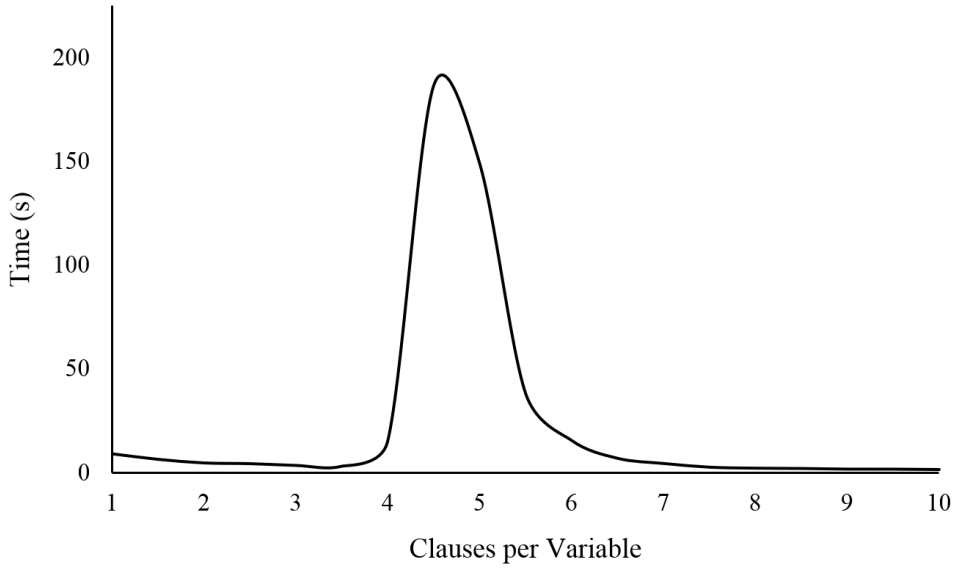


Figure 7.2: Average time consumption of the solver on varying clauses per variable ratios

By conducting these tests, it was evident that the SAT solver struggles with randomly generated instances where the variables to clauses ratio is between 4.5 and 5.5. The Solver does, however, perform very well on instances where the clauses per variable ratio is greater than 5.5 or less than 4.5. Although the program is effective for most randomly generated imbalanced instances, it does not always output a solution for instances within 5 minutes. In fact, the Solver failed to output a solution for two large instances, containing 900 and 1000 clauses respectively, that had a clauses per variable ratios of 5.0.

Chapter 8

Conclusion and Future Work

8.1 Conclusion

PolarSAT employs a successful CDCL algorithm which makes use of many effective heuristics such as VSIDS, clause deletion policies and restart policies. All of these techniques were combined in the SAT solver and tuned to enhance the performance of the program for imbalanced instances.

By quantifying how balanced instances are, the VSIDS branching heuristic was improved by constructing a technique that aimed to assign variables to the ideal polarity. This technique proved to be fairly efficient as it improved the response time of the program significantly during many experiments.

Although the report focuses heavily on the Solver itself, the accompanied program includes a very productive Problem Generator which allowed for effective testing and tuning through various experiments.

The difficulty of SAT problems are affected by many factors such as the number of variables and clauses, especially the ratio of variables to clauses. Instances that contain more than 5.5 clauses per variable are typically found to be unsatisfiable very quickly by the program. Similarly, instances which contain less than 4.5 clauses per variable are generally easy to solve because the search space consists of many satisfiable assignments. Additionally instances which are imbalanced with a balanced coefficient that is greater than or equal to 0.4, are typically extremely easy to solve than relatively balanced instances, especially when employing the polarity tuning technique discussed in Chapter 5.

8.2 Future Work

There are many ways to take this project further and improve the SAT solver. As discussed in Chapter 7, the two watched literals data structure is an implementation strategy that is most likely to reduce the time consumption of the program. It is a data structure that keeps track of two literals in each clause. When the negation of a watched literal is assigned, the Solver shall keep track of another literal in the clause. When the Solver is unable to watch another literal, this implies that the clause only contains one literal. We can, therefore, use this technique to identify unit clauses more efficiently.

As assigning variables and the execution of decision heuristics are the second most time consuming aspect of the Solver, it would also be strategic to research and implement variations of VSIDS. Currently, PolarSAT employs a variation of VSIDS where the activities of only variables in the newest learned clause are bumped up by 1 after each conflict, known as cVSIDS. Alternatively, mVSIDS bumps the activities of all variables resolved during conflict analysis that led to the learned clause. The variation of this heuristic shall be implemented and experimented with in order to understand its effectiveness on imbalanced instances [2].

Other usability issues could also be addressed. Currently, the SAT solver is only able to solve one instance at a time. The user must manually run the Solver on each instance. It would be convenient if the user is able to input a collection of instances at once. The SAT solver could then solve each instance in the collection sequentially. This also suggests that a time limit feature should be implemented which terminates the program on an instance automatically when it exceeds the limit.

References

[1] Joao Marques-Silva, Ines Lynceand Sharad Malik. Conflict Driven Clause Learning SAT Solvers. 2009.

[2] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, Krzysztof Czarnecki. Understanding VSIDS Branching Heuristics in Conflict-Driven Clause-Learning SAT Solvers, 2015.

[3] Jingchao Chen. Phase Selection Heuristics for Satisfiability Solvers, 2011.

[4] Ian P. Gent, Chris Jefferson, Ian Miguel. Watched Literals for Constraint Propagation in Minion, 2006.

[5] Michael Alekhnovich, Edward A. Hirsch, Dmitry Itsykson. Exponential Lower Bounds for the Running Time of DPLL Algorithms on Satisfiable Formulas, 2004.

[6] Munmun Dey and Amitava Bagchi. Satisfiability Methods for Colouring Graphs, 2013.

[7] Louis Abraham. SAT solving techniques: a bibliography, 2018.

[8] Nachum Dershowitz, Ziyad Hanna, Alexander Nadel. A Clause-Based Heuristic for SAT Solvers. 2005.

[9] Niklas S¨orensson and Armin Biere. Minimizing Learned Clauses, 2009.

[10] Shai Haim and Marijn Heule. Towards Ultra Rapid Restarts, 2014.

[11] Fadi A. Aloul. Search techniques for SAT-based Boolean optimization, 2006.

[12] Jinbo Huang. The Effect of Restarts on the Efficiency of Clause Learning, 2007.

[13] Gilles Audemard, Laurent Simon. Predicting Learnt Clauses Quality in Modern SAT Solvers, 2009.

[14] Marijn Heule, Matti Järvisalo, Martin Suda, Markus Iser, Tomáš Balyo. SAT competition. www.satcompetition.org, 2020. [Online, accessed 20 April 2020]

[15] Paul Beame, Henry Kautz, Ashish Sabharwal. Understanding the Power of Clause Learning, 2003.

[16] Artur Niewiadomski, Wojciech Penczek, Teofil Sidoruk. Comparing Efficiency of Modern SAT-solvers for Selected Problems in P, NP, PSPACE, and EXPTIME, 2017.

[17] Jakob Nielsen. 10 Heuristics For User Interface Design, Nielsen Norman Group. www.nngroup.com/articles/ten-usability-heuristics, 1994. [Online, accessed 20 April 2020]