Analysing Optimisation and Solving Techniques in QBF Solvers

A REPORT SUBMITTED TO THE UNIVERSITY OF MANCHESTER
FOR THE DEGREE OF BACHELOR OF SCIENCE
IN THE FACULTY OF SCIENCE AND ENGINEERING

2024

Tom Gill
Supervised by: Konstantin Korovin

Department of Computer Science

Contents

Word Count: 14712

Abstract

This project explores, implements, and evaluates an array of SAT solving algorithms and optimisation techniques for solving Quantified Boolean Formula (QBF) such that the relative strengths of different techniques can be quantified. Specifically, I provide an implementation and critical analysis of the Davis-Putnam-Logemann-Loveland (DPLL) and Conflict Driven Clause Learning (CDCL) back-tracking algorithms along with the optimisation techniques of pure literal deletion, universal reduction, and pre-processing. Further, this project presents pre-resolution, a novel optimisation technique based on applying Q-Resolution on the original clause set prior to solving. Additionally, I explore the algorithms used in current state-of-the-art QBF solvers.

During evaluation, I found that the core optimisations of pure literal deletion, universal reduction, and pre-processing provided the most benefit across all QBF structures, whereas pre-resolution provided benefit in only QBF-based encodings of the bomb-in-the-toilet planning problem. Additionally, I found CDCL outperformed DPLL in most instance suites. Finally, I found that performing Q-Resolution on both existential and universal literals is valid in the context of pre-resolution, however, it provides no benefit over performing Q-Resolution on only existential literals. Overall, all solvers produced correctly solve QBF and the tool produced allows the systematic evaluation of configured solvers.

Acknowledgements

I would like to thank my project supervisor, Konstantin Korovin, for his assistance throughout my project. In particular, I would like to thank him for proposing the idea of performing Q-Resolution prior to input to a solver, which resulted in my implementation of pre-resolution.

List of Tables

List of Figures

List of Algorithms

Copyright

List of Abbreviations

PCNF Prenex Conjunctive Normal Form

CNF Conjunctive Normal Form

SAT Satisfiability

UNSAT Unsatisfiable

QSAT Quantified Satisfiability

QBFLIB Quantified Boolean Formulas Satisfiability Library

QBFEVAL QBF Solver Evaluation

DPLL Davis-Putnam-Logemann-Loveland

QDPLL Quantified DPLL

CDCL Conflict Driven Clause Learning

QCDCL Quantified CDCL

UIP Unique Implication Point

QBF Quantified Boolean Formula

VSS Variable State Sum

LBD Literal Block Distance

CEGAR Counter Example Guided Abstraction Refinement

CAQE Clausal Abstraction for Quantifier Elimination

Chapter 1

Introduction

1.1 Project Proposal

The objectives of this project are to investigate, implement, and evaluate efficient algorithms used in quantified Boolean formula (QBF) solvers with a selection of optimisation techniques. Further, I will perform a critical analysis into the relative strengths of various algorithmic optimisations on different backtracking QBF solving techniques. Additionally, my objectives include an investigation into and implementation of a new optimisation not yet discussed in the field, which is rooted in the technique of Q-Resolution, and I will discuss reasons for and hyperparameters governing its success and limitations. QBF evaluators attempt to determine whether a given QBF problem is satisfiable or unsatisfiable and they are applicable to many real-world problems, ranging from the formal verification and synthesis of computing systems to planning and reasoning in artificial intelligence [37]. The academic field of QBF solvers is large, and each year the QBF Solver Evaluation (QBFEVAL) competition [31] takes place, where a wide range of efficient solvers compete. Therefore, my project will not aim to compete with well-established solvers but will provide guidance on which optimisations give the strongest increase in efficiency and where the applications of such techniques are recommended. As a result, the creation of this project will give additional resources to those creating QBF solvers in the following years. Therefore, in general, the focus of this project will be on the area of propositional logic and the utility of different QBF solving algorithms with the addition of various optimisation techniques.

1.2 Aims and Objectives

In this section, I will define the aims and objectives I outlined at the start of the project and any additional aims added as extensions throughout the project. Success criteria will be defined for each objective, which will be used to determine whether the objective was accomplished.

Aims and Objectives	**Success Criteria**
Explore the state of the top QBF-solving algorithms over the past years and understand their mechanisms.	This will be considered successful when there is a description of the algorithms and an explanation of the techniques used in the top QBF solvers submitted to the QBFEVAL competition [14] in the past and recent years. This will be explored in the technical background section.
Investigate efficient backtracking algorithms and optimisations to determine which features to develop and implement.	This will be considered successful when there is a description of the well-known DPLL and CDCL algorithms and the applicable optimisation techniques.
Implement various QBF-solving algorithms and optimisation techniques and evaluate the relative performance and efficiency differences using official data sets.	This will be considered successful given that there are evaluation metrics for each implemented algorithm and their applied optimisations, using the data sets provided by the Quantified Boolean Formulas Satisfiability Library (QBFLIB) [14].
Perform a critical analysis of which optimisations and solver techniques improved solving efficiency, where such techniques should be applied, and reasons for their success or failure.	This will be considered successful given that there is a critical analysis of the performance of implemented algorithms and optimisation features, providing insight into the relative strength of the techniques chosen and why some performed better than others for particular QBF instance structures.
Additional extension: Investigate, implement, and evaluate a novel optimisation technique for DPLL and CDCL based on Q-resolution.	This will be considered successful given that there is an explanation of the theory behind the optimisation, a description of the algorithm used, an evaluation of its effectiveness, and a discussion of the hyperparameters governing its success and limitations.

1.3 Report Design

This report will first provide the technical background knowledge needed to understand my achievements. This will include algorithm explanations and mathematical definitions surrounding propositional logic, satisfiability problems, backtracking, and optimisation techniques, as well as related work and a brief description of the top solvers used in QBFEVAL [31] competitions. Following this will be a description of the design, architecture, and implementation of my QBF solvers and optimisation techniques. Then there will be the testing, evaluation, and critical analysis of my solver implementations, with a discussion of the relative strengths of various techniques on a range of QBF instance suites. Finally, the summary and conclusion will discuss the achievements, provide a critical reflection, and propose future work.

1.4 Evaluation and Testing Strategy

In this section, I will briefly outline how I will test that my implementation works correctly and how I will test its performance. To test whether it works correctly, I will compare my solver outputs for a range of QBF instances with the outputs from a strong, well-established solver that has already been verified and has reached top rankings in the QBFEVAL competition. Further, I will provide a series of unit tests for individual functionality. To test the performance, I will compare different combinations of my implemented solver algorithms and optimisation techniques with each other to measure the relative strengths of different techniques and algorithms. I will measure data pertaining to the number of instances solved, time to solve instances, and statistical counts to gain insight into the performance of different solvers and reasons for performance differences. Note that I will not be comparing my work against state-of-the-art QBF solvers, as the QBF solvers presented in my project are not intended to compete with those in terms of performance. Instead, the focus of my project is to make contributions and provide insights to the field of QBF Satisfiability (SAT) solving with regards to the relative strengths of different solving techniques.

Chapter 2

Technical Background

This chapter will discuss the technical knowledge required to understand the achievements of the project. It will first give the definitions required to understand propositional logic, quantified Boolean formulae (QBF), and the required and expected format of the QBF. Following this will be an explanation of how unit propagation, the Davis-Putnam-Logemann-Loveland (DPLL), and the Conflict Driven Clause Learning (CDCL) algorithms work. Finally, it will discuss and explain the optimisation techniques used in various QBF solvers, and it will provide a case study into one of the top QBF solvers of the QBFEVAL competition in recent years.

2.1 Propositional Logic

Propositional logic, also known as statement logic, is an area of mathematical logic that specifies the rules of joining and modifying statements and propositions to form more complicated propositions and statements [23]. Truth-functional logic is a thoroughly researched area of propositional logic that studies logical operators and connectives that are used to produce more complex statements whose validity is determined by the simpler statements that produce them [23]. Firstly, I will define terms to introduce propositional logic. A proposition is a statement that is evaluated as either true or false.

Definition 2.1.1 *A **proposition** is a declarative sentence or statement that is either [22]:*

- true ($\top$ or 1) or
- false ($\bot$ or 0)

Definition 2.1.2 *A literal L is a propositional variable (atom) p or its negation $\neg p$. The complementary literal to L is defined as [25]:*

$$\overline{L} \stackrel{\text{def}}{=} \begin{cases} \neg p, & \text{if L is of the form p;} \\ p, & \text{if L has the form } \neg \text{p .} \end{cases} \tag{2.1}$$

Definition 2.1.3 *The set of **propositional formulas** is defined here [25] [18]:*

- The truth values $\top$ and $\bot$;
- Each propositional variable p_i;
- If $F_1, ..., F_n$ are formulas, then $(F_1 \vee ... \vee F_n)$ and $(F_1 \wedge ... \wedge F_n)$;
- If F is a formula, then $\neg F$;
- If F_1 and F_2 are formulas, then $(F_1 \rightarrow F_2)$ and $(F_1 \leftrightarrow F_2)$;
- If p is a propositional variable and F is a formula, then $\forall pF$ and $\exists pF$.

$\forall$ is the universal quantifier, and we say a variable p is universally quantified in the case it is bound, such that, $\forall p$. $\exists$ is the existential quantifier, and we say a variable p is existentially quantified in the case it is bound such that, $\exists p$.

Definition 2.1.4 *A **clause** is defined as a disjunction of literals [25]:*

$$p_1 \vee ... \vee p_n, \text{ where } n \geq 0.$$

Definition 2.1.5 *A **unit clause** is a clause containing a single literal such that n = 1.*

Definition 2.1.6 *An **empty clause** is a clause containing no literals such that n = 0, and is denoted by* $\square$.

The terminology that will be used frequently throughout this paper has now been defined. I will now describe the forms propositional formulae and quantified Boolean formulae will take, along with the form necessary for my solving algorithms.

2.2 Prenex Conjunctive Normal Form and QBF

The QBFEVAL competition has many tracks that are used to test different formula structures, and different QBF solving tools and techniques can be evaluated as being better or worse suited to certain tracks [34]. The most popular of these is the Prenex Conjunctive Normal

Form (PCNF) track, in which solvers compete in the evaluation of formulas in PCNF format. My solver implementations will solve QBF in PCNF, as this is the most popular track. This section will discuss different forms of propositional formulae and the propositional logic that forms a QBF in PCNF.

Definition 2.2.1 *A formula F is in **Conjunctive Normal Form (CNF)** if it is either $\top$, $\bot$, or a conjunction of disjunctions of literals. In other words, a conjunction of clauses [25]:*

$$F = \bigwedge_I \bigvee_J L_{ij}$$

CNF reduces the SAT problem for formulas to the SAT problem for sets of clauses, which is generally simpler to evaluate [25].

A prenex formula is a quantified Boolean formula that has a list of quantifiers and bound variables, followed by a quantifier-free part. This form is useful as it separates the quantifiers and variables, making the formula more readable. Combining this with CNF, we can define the normal form used in my solvers. Below, I define this explicitly.

Definition 2.2.2 *A formula is in **prenex normal form** when it has the form [25]:*

$$\exists\forall p_1 ... \exists\forall p_n \cdot C, \text{ where } C \text{ is a quantifier-free formula.}$$

***Note**: $\exists\forall p_i$ means that p_i can be either existentially or universally quantified.*

Definition 2.2.3 *Given a formula in **prenex normal form**, the **quantifier prefix** Q is a list of quantifier bound variables defined as [25]:*

$$\exists\forall p_1 ... \exists\forall p_n, \text{ where } p_i \text{ are propositional variables.}$$

Definition 2.2.4 *A quantified Boolean formula is **rectified** when [25]:*

- No variable appears both free and quantified in the formula;
- For every variable p_i, the formula contains at most one instance of quantifiers $\exists\forall p_i$.

Now that I have defined when a quantified Boolean formula is in prenex form, CNF, and rectified, I can combine these to give a definition of prenex conjunctive normal form.

Definition 2.2.5 *A quantified Boolean formula F is in PCNF if it is either $\top$, $\bot$, or rectified and prenex such that it has the form [25]:*

$\exists\forall p_1 ... \exists\forall p_n (C_1 \wedge ... \wedge C_m)$, where C_i is a propositional clause and p_i is a propositional variable.

Now that the formula format I will be solving has been defined, I provide some terminology for defining the satisfiability and unsatisfiability of a given QBF.

Definition 2.2.6 *For a formula F in PCNF [25]:*

- If F contains the empty clause, $\square$, it is unsatisfiable;
- If F contains the empty set of clauses, $\emptyset$, it is satisfiable.

This is the QBF form in which my implementation will take, and so the benchmark test data will be of this structure. There are other tracks in the QBFEVAL competition [14] such as Prenex Non-CNF, where the formula can contain a nesting of disjunctions, conjunctions, and negations that can be taken advantage of, and 2QBF, in which the data is in PCNF but contains a single rotation of $\exists\forall$ which allows several verification and practical synthesis problems to be encoded. As you can see, each track is diverse in the techniques required to solve their respective problems best [34] [32].

***Note**: This paper won't go through the techniques necessary to transform an equation into CNF or prenex form, as this isn't required to understand the achievements of this project.*

This concludes the core propositional knowledge background. In the following section, I will describe the main solving algorithms used for my QBF solvers.

2.3 SAT solvers

Quantified Satisfiability (QSAT) problems find whether a given quantified Boolean formula is satisfiable or unsatisfiable, in other words, they ask whether a sequence of quantifiers over a set of Boolean variables is true or false. Therefore, QSAT solvers are searching through a set of

logical statements to determine which statements are true or false, which are then combined to make the larger formula true or false.

QBF is the canonical complete problem for PSPACE, which in computational complexity theory is the class of problems that can be solved by a deterministic or nondeterministic Turing machine in polynomial space and unlimited time [12]. Given the formula in an abstract syntax tree form, which it is commonly portrayed as, it can be solved by a group of recursive procedures that evaluate the formula. Hence, an algorithm such as this uses space proportional to the height of the tree but executes in time exponential ($O(2^n)$) in the number of quantifiers [12] [7]. For this reason, it is important to find efficient algorithms to solve such problems. In this section, we will discuss two of these algorithms, DPLL and CDCL.

2.3.1 DPLL

A well-known and long-established SAT-solving algorithm is the David-Putnam-Logemann-Loveland (DPLL) algorithm, which is a backtracking-based search algorithm for solving CNF SAT problems developed by Martin Davis, George Logemann, Donald W. Loveland, and Hilary Putnam in the 1962 paper ”A Machine Program for Theorem-Proving” [8] [9]. In more recent years, this has been extended to Quantified DPLL (QDPLL) such that it can be used to solve QBF in a similar manner [5].

The QDPLL algorithm will take as input a quantified Boolean formula. In general terms, it works by recursively selecting a literal from the outermost quantifier and assigning it a truth value ($\top$ or $\bot$), then simplifying the formula via Boolean constraint propagation, commonly referred to as unit propagation, until we reach an evaluation of Satisfiable (SAT) or Unsatisfiable (UNSAT) . In the case of an existentially quantified variable p, we need only one assignment p or $\neg p$ to evaluate to true for that sub-section to be true. In the case of a universally quantified variable p, we need both assignments p and $\neg p$ to evaluate to true for that sub-section to be true. This is because $\exists$ is interpreted as ”there exists”, so we are only looking for one of our assignments to be evaluated as true, and $\forall$ is interpreted as ”for all”, so we want all assignments to be evaluated as true. Upon finding an evaluation for a given branch, we set that sub-section to evaluation and backtrack to the previous branch and evaluate it similarly. When evaluating a branch if:

- The branched variable p is universally quantified and the assignment results in a true evaluation, we must assign $\neg p$ and evaluate.

- The branched variable p is universally quantified and the assignment results in a false evaluation, we must set the current evaluation of the branch to false and backtrack.
- The branched variable p is existentially quantified and the assignment results in a true evaluation, we must set the current evaluation of the branch to true and backtrack.
- The branched variable p is existentially quantified and the assignment results in a false evaluation, we must assign $\neg p$ and evaluate.

Due to the fact we fork every time we select a literal, QDPLL has a worst-case run-time complexity of $O(2^n)$, and a worst-case space complexity of O(n) [25] [11]. Subsequently, QDPLL is inefficient for practical applications, and improvements are required to make proper use of QBF solving. These improvements take advantage of learning and non-chronological backtracking to prune large sections of the search space.

2.3.2 Conflict-Driven Clause Learning

Conflict-Driven Clause Learning (CDCL) was first proposed by Marques-Silva and Sakallah in their 1996 paper, "GRASP - A New Search Algorithm for Satisfiability" [28]. It discusses a conflict-driven technique that is built upon DPLL such that the algorithm can perform non-chronological backtracking (back-jumping) as opposed to DPLL, which only facilitates chronological backtracking. Figure 2.1 illustrates this idea. This allows the potential for large areas of the search space to be pruned, and this has been shown to be much more efficient than DPLL for most benchmark suites [28]. This can be extended to Quantified CDCL (QCDCL) such that it can be used to solve QBF [42] [26].

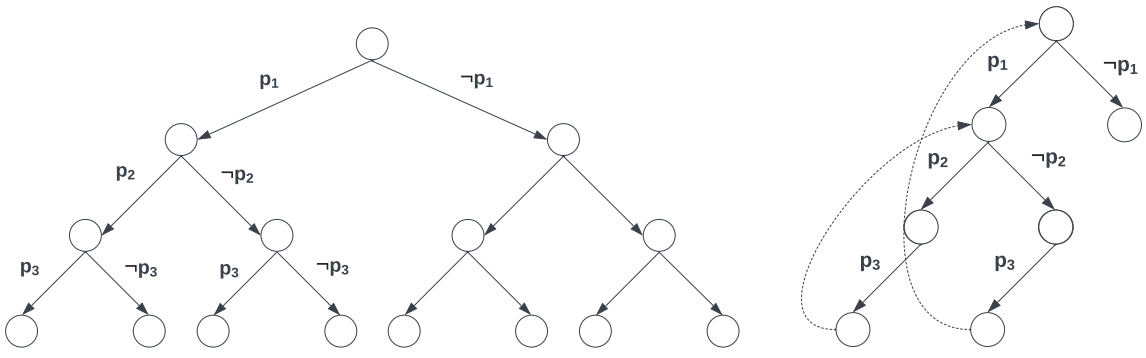

Figure 2.1: DPLL vs CDCL search space tree

The QCDCL algorithm will take as input a quantified Boolean formula. In general terms, it works by recursively selecting a literal from the outermost quantifier and assigning it a truth

value ($\top$ or $\bot$), called making a decision, simplifying the set of clauses via Boolean constraint propagation (unit propagation), and upon reaching an evaluation for a given decision branch, will backtrack appropriately depending on the evaluation and the quantification type of the most recent decision literal. When the algorithm reaches a branch that evaluates as unsatisfiable, it is said to have reached a conflict. In DPLL, after reaching a conflict, regardless of the assignments of the decision literals, it will always backtrack to the level above and continue the search down the other side of the branch. This is likely to result in another conflict, caused by the same assignments that caused the original conflict, which were made at earlier decision levels. QCDCL will analyse the cause of the conflict by checking which variable assignments are responsible and thus can derive a new clause that is the negation of these assignments, effectively removing the possibility of the same incorrect assignments being made in the future. This is called conflict analysis and is an essential part of any CDCL implementation. This new clause will be added to the set of original clauses, and the algorithm will backtrack to an earlier decision level in the tree, which could be across multiple decisions (back-jumping); this is called clause learning. Clause learning allows the algorithm to prune large areas of the search space, significantly increasing the efficiency of the search. The point at which the algorithm backtracks to depends on the individual back-jumping scheme being used. A technique with good performance is to stop clause learning at the first Unique Implication Point (UIP) and backtrack to the decision level which makes the learned clause a unit clause. Figure 2.2 illustrates a walk-through of applying CDCL to a QBF.

Unique Implication Point *- A Unique Implication point is any node at the current decision level such that any path from the decision node to the conflict node must pass through it [38].*

Implied variable *- A variable that has been assigned a truth value as a consequence of other assignments, such as a decision variable, usually during unit propagation.*

A technique for keeping track of assignments is to store a trail of assignments and implications as decisions are made and unit propagation is applied. A literals assignment is stored along with information on whether it was a decision or an implication. This list is in chronological order, so we can backtrack through it to reach the necessary point during conflict analysis for deriving a new clause for clause learning.

CDCL, and hence QCDCL, is a much more powerful approach to solving SAT problems and QBF respectively, than the simpler DPLL approach. In the following sections, I describe two components used in modern CDCL implementations that increase their effectiveness: **restarts**

and **clause database reduction**.

Example - CDCL [24]:

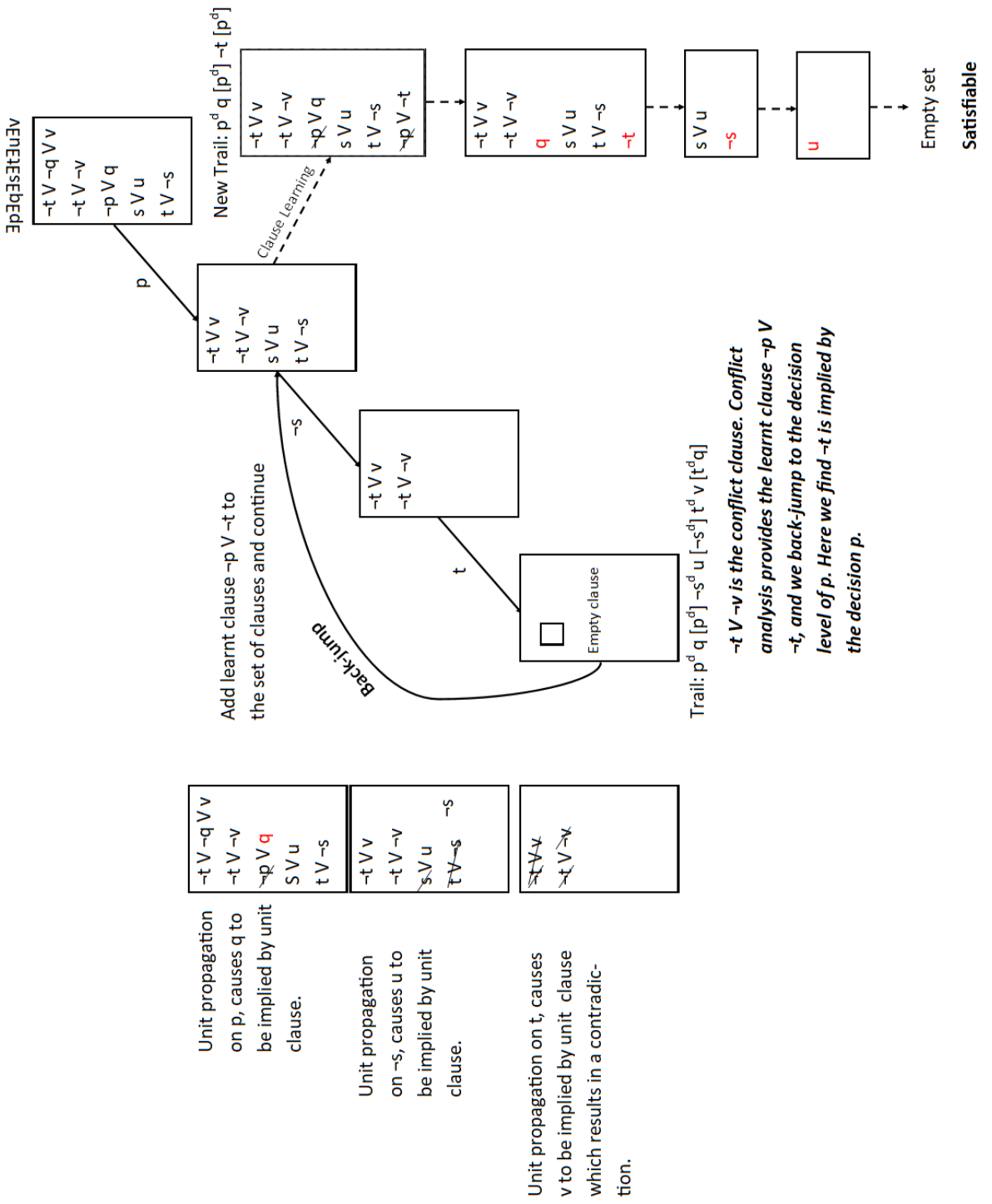

Figure 2.2: CDCL Example Walk-through

Restart Policy

CDCL algorithms, when searching, will progressively enter deeper areas of the search space, which can make the algorithm slower and can result in the algorithm entering branches that are dead ends or plateau regions, which are very unlikely to lead to a result. Restarts are a mechanism that allows the algorithm to reset the decision tree, which then allows the algorithm to explore new decision branches. CDCL performs a restart by reverting all current assignments and backtracking through the entire decision tree. They focus on entering a new state space of the decision tree by selectively keeping the most useful clauses learned prior to restarting and using literal selection schemes to prioritise searching through literals with the most potential. An additional idea behind restarts is that the solver can benefit from learned clauses that are repeatedly conflicting sooner than would be allowed otherwise when back-jumping using the regular back-jumping scheme [19]. This has been seen in practice, where frequent restarts have been found effective.

Determining an effective restart policy is an ongoing research area, and there are many available options to choose from. A restart policy consists of a scheme for scaling the restart intervals and a clause database reduction scheme, which in itself is its own area of research within SAT solvers. There are a range of available restart interval schemes consisting of static, uniform, non-uniform, and dynamic strategies [4]. In my implementation, I discuss a non-uniform method based on the Luby series that aims to utilise the effectiveness of frequent restarts whilst allowing the escape of dead-end branches.

Clause Database Reduction

When running CDCL on large instances, you are going to encounter situations where the learned clause database, the list of added clauses learned after conflicts, becomes very large. This can cause a slowdown in performance, especially when a lot of these clauses are low quality and aren't useful during the latter stages of execution. Therefore, it is important that we manage the clause database such that low-quality learned clauses are removed, which will reduce the effect of performance degradation. A strategy for clause database reduction is also necessary for use during a restart to reduce the set of learned clauses before beginning another search down the decision tree.

A complicated but effective deletion scheme proposed for MapleLCMDistChronoBT [30] uses the **Delete-Half** scheme, which sorts the learned clause store and deletes the lower-quality half. The quality heuristic for a newly learned clause, used to decide which store it is placed, is

largely based on literal block distance. The **Literal Block Distance (LBD)** of a clause is the number of decision levels in which literals within the clause have been assigned [39].

Age-Based Deletion is a simple, low-cost deletion scheme where each newly learned clause replaces the oldest clause in the list of learned clauses. This was shown to work surprisingly well, despite being trivial. It was shown to perform almost as well as MapleLCMDistChronoBT [20]. In my implementation, I propose a similar age-based deletion approach for my restart policy. In the following section, I describe the technique used in my conflict analysis and custom optimisation implementation.

2.4 Q-Resolution

Q-Resolution allows for the production of a new clause based on the implication of two other clauses containing complementary literals.

Definition 2.5.1 *The resolution rule for the SAT problem is defined as [27]:*

$$\frac{p_1 \vee ... \vee p_n \vee \mathbf{x}, \qquad q_1 \vee ... \vee q_n \vee \neg\mathbf{x}}{p_1 \vee ... \vee p_n \vee q_1 \vee ... \vee q_n}$$

- If for all literals l_i: $\{l, \neg l\} \nsubseteq p_1 \vee ... \vee p_n \vee q_1 \vee ... \vee q_n$.

Definition 2.5.2 *The resolution rule for the QSAT problem is defined as [27]:*

$$\frac{p_1 \vee ... \vee p_n \vee \mathbf{x}, \qquad q_1 \vee ... \vee q_n \vee \neg\mathbf{x}}{p_1 \vee ... \vee p_n \vee q_1 \vee ... \vee q_n}$$

- If for all literals l_i: $\{l, \neg l\} \nsubseteq p_1 \vee ... \vee p_n \vee q_1 \vee ... \vee q_n$;
- And x is existentially quantified.

Q-Resolution is a common technique used in CDCL conflict analysis procedures. When a conflict occurs, the clause from which the last literal assignment was implied can be resolved

with the conflict clause to produce a new clause. This process can be done iteratively with a trail of assignments all the way up until it contains only decision literals. Further, I use this technique in the **pre-resolution** optimisation, which will resolve clauses in the original clause list with each other to produce a new set of potentially useful clauses to use when executing the solving algorithm.

Example - Q-Resolution:

Given two clauses $C_1 \leftarrow (p_1 \vee p_2 \vee p_3)$ and $C_2 \leftarrow (\neg p_3 \vee p_4 \vee p_5 \vee p_6)$ with quantification $\exists p_1 p_2 p_3 p_6 \forall p_4 p_5$, Q-Resolution can be applied:

$$\frac{p_1 \vee p_2 \vee p_3 \qquad \neg p_3 \vee p_4 \vee p_5 \vee p_6}{p_1 \vee p_2 \vee p_4 \vee p_5 \vee p_6}$$

Hence, a new clause has been produced $C_3 \leftarrow (\mathrm{p}_1 \vee p_2 \vee p_4 \vee p_5 \vee p_6)$.

2.5 Unit Propagation Optimisations

Unit propagation, or Boolean constraint propagation, is a method of automated theorem proving [41] based on unit clauses that can simplify a set of clauses. It is an important part of both DPLL and CDCL implementations, and in practice, for most satisfiability problems, a major portion of the solver's execution time is spent in the unit propagation process [29]. Therefore, an efficient unit propagation function is desired to improve the overall efficiency of the given algorithm. Clauses that are composed of a single literal will always be satisfied in the case of $\exists$ and never satisfied in the case of $\forall$, this allows us to simplify the list of clauses. If the unit literal p is:

- quantified such that $\forall$p - replace the set of clauses with the set $\{\square\}$
- quantified such that $\exists$p or is not quantified:
  - remove all clauses containing the literal p from the set S of clauses.
  - replace all clauses of the form (C $\cup \neg$p) with the clause C in the set S of clauses. [25]

Example - Unit Propagation:

Before	During	After
$\exists p_1 p_2 \forall p_3 \exists p_4$	$\exists \sout{p_1} p_2 \forall p_3 \exists p_4$	$\exists p_2 \forall p_3 \exists p_4$
$\neg p_1$	$\sout{\neg p_1}$	$\neg p_2 \vee \ p_3$
$p_2 \vee \neg p_1$	$\sout{p_2 \vee \neg p_1}$	$\neg p_2 \vee \ p_3 \vee \ p_4$
$\neg p_2 \vee \ p_1 \vee \ p_3$	$\neg p_2 \vee \sout{p_1} \vee \ p_3$	
$\neg p_2 \vee \ p_3 \vee \ p_4$	$\neg p_2 \vee \ p_3 \vee \ p_4$	

Firstly, $\neg p_1$ is a unit clause and p_1 is existentially quantified, so we can remove the clauses $(\neg p_1)$ and $(p_2 \vee \neg p_1)$ completely. Further, we can remove p_1 from the clause $(\neg p_2 \vee p_1 \vee p_3)$. Finally, we can remove p_1 from the quantifier prefix, and we are left with the clauses $(\neg p_2 \vee p_3)$ and $(\neg p_2 \vee p_3 \vee p_4)$.

Now that you know how unit propagation is performed, the following sections will outline techniques to further simplify QBF and reduce the time spent in unit propagation.

2.5.1 Pure Literal Elimination

The identification and removal of pure literals play an important role in QBF, as the removal of such literals may cause the detection of unit literals, which can reduce the number of decisions required and hence decrease the search space. However, the identification of pure literals is difficult as we usually need to check every literal in a clause when the clause is removed. In this section, I will define pure literals, and in the implementation section, I will discuss my pure literal detection technique [13].

Definition 2.6.1 *A literal p is **pure** in a set of clauses S if its negation $\neg p$ does not occur in S [25].*

Definition 2.6.2 *If a literal p is **pure** in a set of clauses S with quantifier prefix Q, we can assign p such that [25]:*

- If p is existentially quantified in Q ($\exists p$), we can remove all clauses containing p.
- If p is universally quantified in Q ($\forall p$), we can remove p from all clauses containing p.

This is called **pure literal elimination**.

Example - Pure Literal Elimination:

Before	During	After
$\exists p_1 p_2 \forall p_3 \exists p_4$	$\exists p_1 p_2 \forall p_3 \exists p_4$	$\exists p_1 \forall p_3 \exists p_4$
$\neg p_1 \wedge \ p_2$	~~$\neg p_1 \wedge \ p_2$~~	$\neg p_1 \wedge \neg p_3 \wedge \ p_4$
$\ p_1 \wedge \ p_2 \wedge \ p_3$	~~$\ p_1 \wedge \ p_2 \wedge \ p_3$~~	
$\neg p_1 \wedge \neg p_3 \wedge \ p_4$	$\neg p_1 \wedge \neg p_3 \wedge \ p_4$	
$\ p_2 \wedge \ p_3 \wedge \neg p_4$	~~$\ p_2 \wedge \ p_3 \wedge \neg p_4$~~	

Firstly, p_2 is pure and it is existentially quantified, so we can remove the clauses $(\neg p_1 \vee p_2)$, $(p_1 \vee p_2 \vee p_3)$, and $(p_2 \vee p_3 \vee \neg p_4)$. The clause database is now simplified and consists of a single clause.

2.5.2 Universal Reduction

Given a QBF, the quantification level $\mathcal{L}$ of a variable is defined by the number of quantifier alternations from left (outer) to right (inner) for that variable. For example, the formula $\forall p_1 p_2 \exists p_3 \forall p_4 \cdot C$ where C is a set of clauses, has $\mathcal{L}(p_1) = 1, \mathcal{L}(p_2) = 1, \mathcal{L}(p_3) = 2$ and $\mathcal{L}(p_4) = 3$ [3].

Theorem 2.6.3 *Let Q be a quantifier prefix, S a set of clauses, and a clause $C \in S$. Given that:*

- *C* is a non-tautological clause in *S*;
- A literal *p* in *C* is universally quantified in *Q*;
- The literal *p* has the highest quantification level in *C*.

*Then the removal of the literal p from C does not change the truth value of the QBF [25]. This is called **Universal Reduction** or **Universal Literal Deletion.***

The application of Universal reduction simplifies clauses and may cause the detection of unit literals.

Example - Universal Reduction:

$\exists p_1 \forall p_2 \exists p_3 p_4$	$\exists p_1 \forall p_2 \exists p_3 p_4$	$\exists p_1 \forall p_2 \exists p_3 p_4$
$p_1 \wedge \neg p_2$	$p_1 \wedge \neg p_2$	p_1
$\neg p_3 \wedge \ p_2$	$\neg p_3 \wedge \ p_2$	$\neg p_3 \wedge \ p_2$
$\neg p_1 \wedge \ p_2 \wedge \ p_4$	$\neg p_1 \wedge \ p_2 \wedge \ p_4$	$\neg p_1 \wedge \ p_2 \wedge \ p_4$
$\neg p_1 \wedge \ p_2 \wedge \ p_3 \wedge \neg p_4$	$\neg p_1 \wedge \ p_2 \wedge \ p_3 \wedge \neg p_4$	$\neg p_1 \wedge \ p_2 \wedge \ p_3 \wedge \neg p_4$
Before	**During**	**After** [25]

In the first clause, $(p_1 \vee \neg p_2)$, p_2 is a universally quantified literal with a quantification level higher than p_1, therefore we can remove $\neg p_2$ from the clause. We now have a unit clause (p_1) to which we can apply unit propagation. In the following section, I will discuss another powerful SAT-solving approach, Counter Example Guided Abstraction Refinement (CEGAR).

2.6 Related Work

This section will discuss algorithms which won't be explored any further in this paper. I will describe their importance, and I hope to give you a well-rounded understanding of a variety of algorithms used in state-of-the-art QBF solvers [14].

2.6.1 CEGAR

Counterexample-Guided Abstraction Refinement or CEGAR-based algorithms for solving QBF with an arbitrary number of quantifiers were first introduced in the 2012 paper, "Solving QBF with Counterexample Guided Refinement" [21]. The authors provide a CEGAR implementation that progressively expands a given QBF formula into a propositional formula that can then be solved by a traditional SAT solver. In this section, I will briefly describe CEGAR and discuss a CEGAR-based implementation that has won the QBFEVAL competition in recent years.

CEGAR-based algorithms function by iteratively constructing and refining abstractions up until we receive a candidate solution that satisfies the original formula or we find no solution [17]. Firstly, they build an abstraction of the given formula, which we can then use to derive a candidate solution for the winning move. The initial abstraction is progressively refined as the algorithm advances using a counterexample of the candidate solution. This means the abstraction is strengthened by reducing the set of winning moves in the candidate solution, which

is achieved by applying conjunction and disjunction to simplify the set of combined formulae [21]. Note that a solution that satisfies the abstraction does not necessarily satisfy the original formula from which the abstraction is derived. This is why you need to verify that it also satisfies the original formula, which is achieved by checking for counterexamples; if there are no counterexamples, we can return the candidate solution as it has been verified.

Essentially CEGAR-based algorithm will follow the following steps:

1. Build an abstraction of the given formula;
2. Find a candidate solution using the abstraction;
3. If there is no candidate solution that provides a winning move, return false;
4. Find a counter example of the candidate solution for the winning move;
5. If there is no counter example, return the candidate solution;
6. If there is a counter example, refine and strengthen the current abstraction by adding the counterexample to the abstraction.
7. Repeat from step 2.

Note: *Depending on the implementation, this can be implemented recursively or iteratively such that quantifiers are gradually expanded and eliminated via abstraction creation.*

2.6.2 Case Study - CAQE

Traditional CEGAR-based approaches currently present two shortcomings: poor performance for QBF instances with many quantifier alternations and no ability to certify their results. Clausal Abstraction for Quantifier Elimination (CAQE) is a CEGAR-based algorithm presented in the paper ”CAQE: A Certifying QBF Solver” [35] which improves on these shortcomings. CAQE solvers have placed highly in the QBFEVAL competition [31] in recent years.

CAQE uses clausal abstractions to provide a new idea of refinement, which leads to a separate variant of CEGAR-based algorithms. Clausal abstractions are the decomposition of QBF formulae into sequences of propositional formulae. It works on the idea that each clause with one quantifier alternation can be separated into two parts with additional variables that detail their relationship. This provides a new technique for refinement, which leads to the algorithm performing efficiently for QBFs with numerous quantifier alternations. CAQE also incorporates well-established preprocessing techniques, which include: pure literal deletion, universal

reduction, unit clauses, tautology clauses, and miniscoping.

CAQE allows the certification of the result provided by the solver. The solver outputs a ”clausal abstraction proof”, which is a sequence of various variable assignments and navigation symbols for determining quantification levels. From the clausal abstraction proof, they can effectively extract Skolem and Herbrand functions [40] which can be used for certification. The authors provide a tool chain and proof format for the certification process, which has been shown to outperform prior certifying QBF solvers [35].

The authors' experimental evaluation shows that CAQE performed competitively against the top-performing solvers in the 2014 QBF Gallery, particularly for hard instances where there were many quantifier alternations. Further, the results of the QBF Evaluation competition undoubtedly show that variations of CAQE are the current top QBF solvers.

This is the end of the technical background section. This section will have given you an understanding of the algorithms and optimisations that will be implemented and discussed in the following sections. Further, it will have helped you understand my contribution to the field and the achievements outlined earlier. In the following section, I will describe the design of my QBF solvers.

Chapter 3

Implementation and Design

In this section, I will describe the formulation of the QBF problems, how I modelled the problem in data structures, and finally the implementation of optimisations and algorithms. In particular, I will describe the implementation of **pre-resolution**.

I have chosen to write the QSAT solver in Rust for the following reasons:

- Rust is memory-safe and efficient, leading to speed increases when handling large amounts of data.
- It forces the programmer to be memory conscious and keep track of memory in use.
- Rust is a language seen in recent solvers in the QBFEVAL competitions, showing strong solving capabilities.

3.1 Problem Definition

The first step of implementation is to parse a given problem into data structures that are usable and readable by my solver. The format used in the QBFEVAL competition for the PCNF track is QDIMACS [15], hence, I will be assuming this file format for given QBF problems.

3.2 Architecture

Figure 3.1 illustrates the path a QBF instance can take through the solver, along with the data structures created through parsing the QBF instance. The two steps prior to the solver are optional and can be omitted. Figure 3.2 illustrates each technique that can be applied within a given solver.

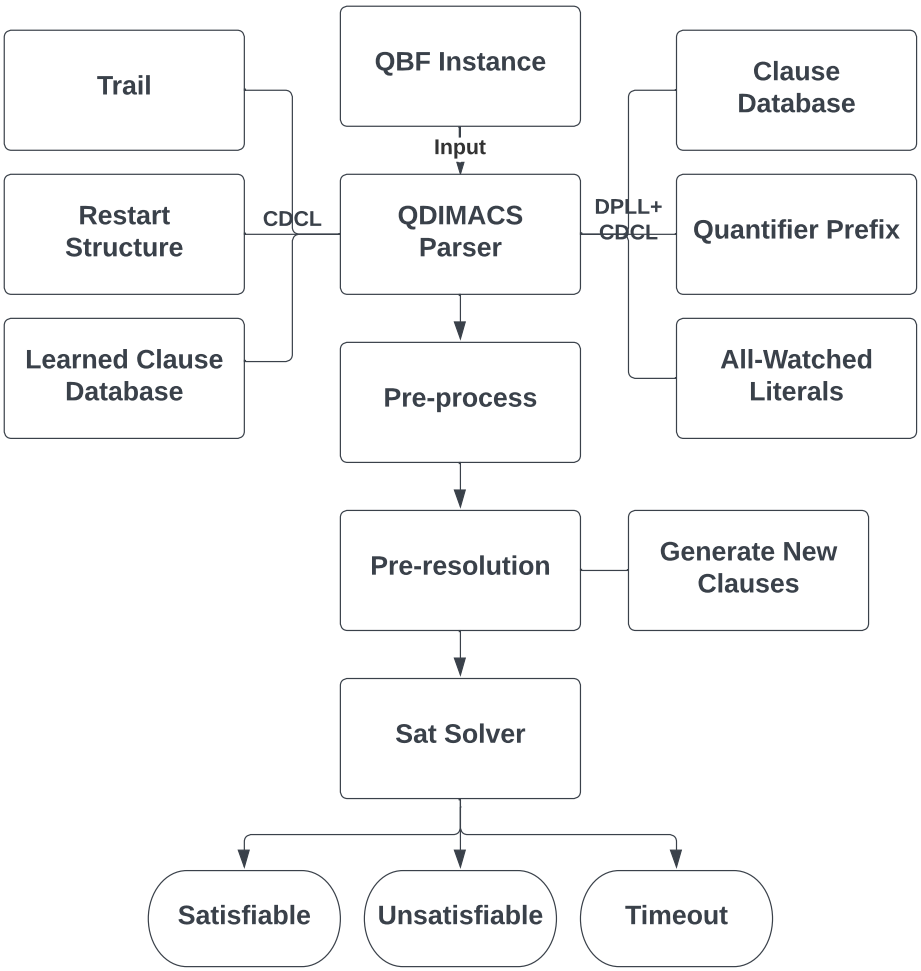

Figure 3.1: The path a QBF instance takes prior to SAT solving

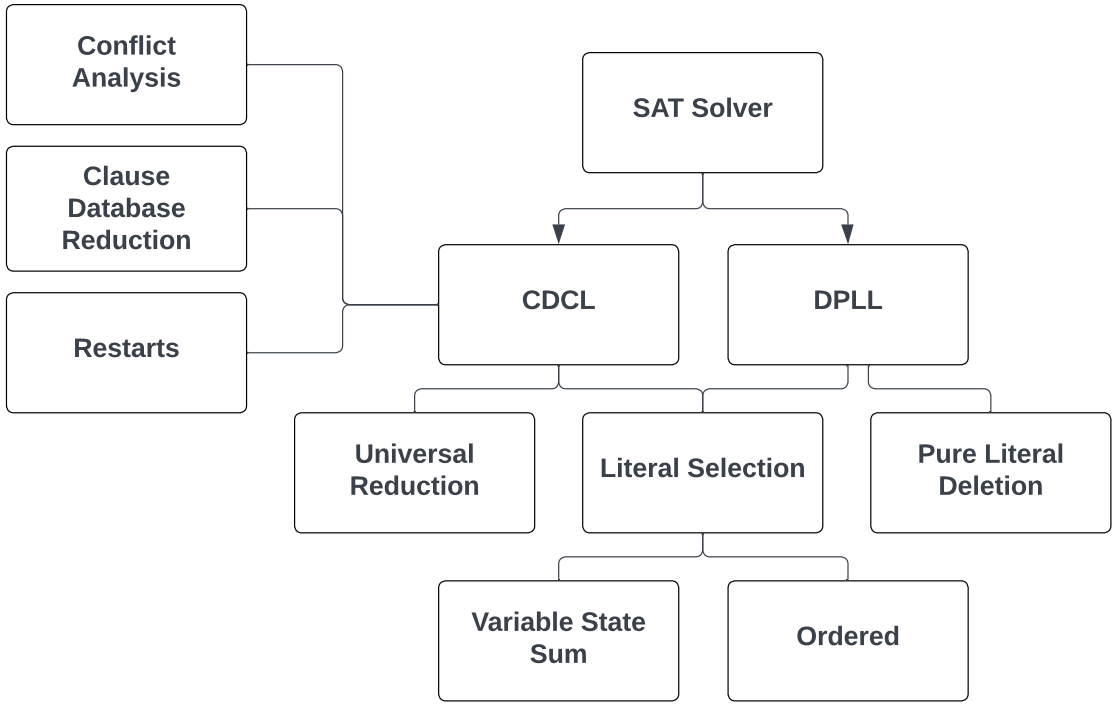

Figure 3.2: The techniques applied within a SAT Solver

3.3 Design Methodologies and Practices

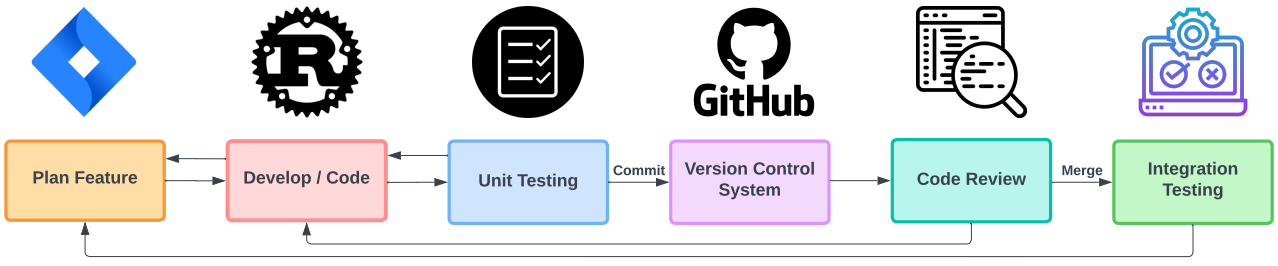

Figure 3.3: Continuous Development Pipeline

Figure 3.3 illustrates the development pipeline used throughout the development of my QBF solvers. I adopted an agile approach to my development, with an iterative process of introducing new features. I used Jira for creating a task board that has the features I planned to implement split up into small components that can be implemented separately. It was also used for planning

the timeline of implementation alongside my Gantt chart, which has an abstract overview of the implementation timeline. Unit testing was applied all throughout implementation using Rust's built-in testing functionality. GitHub was used for version control management and storing the project safely, as well as facilitating code reviews, which I performed myself after the implementation of a feature. If code review passed, the new feature code was merged with the rest of the code base, where I then ran integration tests to ensure it all worked together correctly.

3.4 QDPLL

Algorithm 1 depicts the abstracted functionality of the QDPLL algorithm used in my implementation. Unit propagate will perform unit propagation as defined in Section 2.5 as well as pure literal deletion and universal reduction iteratively as the conditions to allow the application of these techniques are detected. The methods of detecting when certain techniques can be applied are a core part of my implementation, which I describe in the next section.

Algorithm 1: QDPLL(θ, Q)

```
// If there has been a selected literal to propagate
if θ contains a unit clause then
    unit_propagate(θ);
end
if θ contains the empty set of clauses then
    return SAT;
end
if θ contains the empty clause then
    return UNSAT;
end
l ← select_literal(Q);
result ← QDPLL(θ ∧ l, Q) ;
switch (result, Q(l)) do
    case SAT ∧ ∃ do
        return SAT;
    end
    case SAT ∧ ∀ OR UNSAT ∧ ∃ do
        return QDPLL(θ ∧ ¬ l, Q);
    end
    case UNSAT ∧ ∀ do
        return UNSAT;
    end
end
```

3.4.1 Data Structures & Requirements

The data structures into which the problem is modelled during parsing are an important aspect of the solver, as they determine how the solver detects conditions that allow for simplification and the extent to which the data needs to be manipulated as it goes through the solving process. Techniques to reduce data structure manipulation can significantly speed up the solving process as less time is required in frequently used parts of the solver, such as unit propagation.

Firstly, I outline the main conditions my data structures aim to uphold [13]:

- Allows the detection of unit literals, pure literals, and universals literals for reduction - ***For unit propagation, pure literal removal, or universal reduction respectively***;
- Allows the detection of the empty clause and the empty set of clauses - ***For returning UNSAT or SAT respectively***;
- Allows the tracking of the clause database and quantifier prefix state - ***For accurate traversal down the decision tree***;
- Allows the detection of void quantifiers in the prefix - ***For the removal of unnecessary quantifiers***.

Now I outline how each of these conditions are met.

Unit Literals: Detected during unit propagation and pure literal deletion when the negation of a literal is removed from a clause. Also, during universal reduction, when universally quantified literals are removed from a clause. This is achieved by checking whether the clause contains a single literal, as we are removing literals from a clause as we traverse the tree. *Sorted clauses* improve the efficiency of literal removal.

Pure Literals: Since I am using *all-watched literals*, pure literals are detected when a given variable has either only positive occurrences in the clause database or only negative occurrences in the clause database. This is checked initially and at the end of each unit propagation cycle.

Universal Literals: Detected during unit propagation and pure literal deletion when the negation of a given literal is removed from a clause as universal literals in the clause may now have the highest quantification level. Quantification levels are stored for literals and checked within clauses when necessary.

Empty Set of Clauses: I store a clause counter showing the remaining number of clauses in the clause database. This is decreased whenever a clause is removed during unit propagation or pure literal deletion. When this reaches zero, I have the empty set of clauses. It is checked after unit propagating a decision variable.

Empty Clause: The clause counter is set to -1 to signal the empty clause, and the solver starts backtracking whenever:

1. A universal literal is propagated that isn't a decision variable - it will always result in a contradiction;
2. During unit propagation or pure literal deletion when the negation of a given literal is removed from a clause containing only that literal;
3. During universal reduction, if there is no remaining existentially quantified literal in the clause.

Void Quantifiers [13]: A void quantifier is a literal in the quantifier prefix that does not appear in any clause, either negatively or positively. Detected during literal selection by checking whether the literal in the quantifier prefix has no clause references within the *all-watched literals* data structure.

Clause database and quantifier prefix state: The clause database is stored as a list of clauses, where each clause holds its literals in their current state in the tree traversal. Similarly, the quantifier prefix is initialised and then updated as literals are selected and propagated.

These data structures are updated as the decision tree deepens, and upon backtracking, the changes must be undone. Below, I outline two techniques for storing the data within the structures to reduce traversal of the clause database.

All Watched Literals [13]: For each variable, store the index of the clauses it appears in positively and negatively. This is an efficient method for updating clauses, as you don't need to loop through every clause and their corresponding literals, and it allows for pure literal and void quantifier detection.

Sorted Clauses: For each clause, separate its literals into universal and existential, and sort into quantifier prefix order. This improves the speed of literal removal during unit propagation, pure literal deletion, and universal reduction as you don't need to search through the whole set of literals when removing a variable.

Below I list the core data structures I parse the problem into, which I will be referring to within this paper:

- Q **stores the quantifier prefix** - A list of literals with their quantification appearing in the order they appear in the quantifier prefix.
- θ **stores the list of clauses** - A list of clauses and their literals, separated into existential and universal, and ordered corresponding to their order in the quantifier prefix.
- φ **stores the clause references for each literal** - Storing a literals positive and negative clause appearances separately.

You should now have an understanding of how the QSAT problem is modelled after parsing and how various conditions are detected to instigate unit propagation, pure literal deletion, and universal reduction.

3.4.2 Pre-processing

Pre-processing is a method of reducing the QBF problem prior to it being passed to the solver to make decisions. The methods I apply during pre-processing are pure literal deletion, universal reduction, and unit propagation, where each technique is applied iteratively until it can no longer be applied.

3.5 QCDCL

Now that the QDPLL algorithm has been structured, I have the core foundation to extend the functionality to the implementation of QCDCL. **Algorithm 2** depicts the abstracted functionality for my QCDCL algorithm. The key differences compared to QDPLL are conflict analysis following an unsatisfiable assignment during unit propagation, a restart policy dictating when and how to perform a restart and how to reduce the learned clause database, and finally, the back-jump handling of how far to backtrack up the decision tree before adding the newly learned clause. To perform these functions, I require additional data structures on top of those used in QDPLL. Further, pure literal deletion is not implemented for CDCL due to it being too expensive to keep track of the implications for conflict analysis. However, universal reduction was implemented and showed a strong performance increase in many instances.

An interesting component of my CDCL procedure is how it handles learned clauses which are unit clauses. If conflict analysis produces a learned clause that is unit, it means I set the

backtrack level to 0 and propagate this as its own unit clause prior to starting the solver again. This is because no matter what path the solver decides to take from now on, this unit clause must be true.

3.5.1 Data Structures

QCDCL uses the same core data structures as QDPLL but also requires additional structures essential for algorithmic components unique to QCDCL. These include:

- The **trail of assignments** - A chronologically ordered list of variable assignments. For each assignment, it stores the literal assigned and whether it was assigned by decision or implication. It will store the clause from which it was implied, if applicable.
- The **restart structure** - A custom data structure containing the conflict number for the current restart, the conflicts allowed until the next restart, and the restart counter. It is also responsible for updating the restart interval preceding each restart using a geometric equation.
- The **learned clause database** - A list storing the clauses learned during conflict analysis.

3.5.2 Pre-processing

Pre-processing for CDCL is identical to DPLL, as we can apply all the same techniques. Pure literal deletion can be applied here as we don't need to identify the clauses that were responsible for assignments that are made during pre-processing.

3.5.3 Conflict Analysis

Conflict analysis is the critical component of any CDCL implementation. It controls how a newly learned clause will be constructed, which can vary depending on the form of the implementation. As a result of this, it also governs how far the algorithm will back-jump after analysis, therefore determining how much the decision tree can be effectively pruned.

Algorithm 3 depicts a simplified version of my conflict analysis procedure. Its foundation is built upon iteratively performing Q-Resolution on the conflict clause and the clauses responsible for the implied literals within the conflict clause and the subsequent resolved clauses. This procedure is based upon the criteria defined in the paper "Conflict Driven Learning in a Quantified Boolean Satisfiability Solver" [42], however, it is extended using some of the ideas proposed

Algorithm 2: QCDCL(θ, Q)

```
// If there has been a selected literal to propagate
if θ contains a unit clause then
    unit_propagate(θ);
end
if θ contains the empty set of clauses then
    return satisfiable(); // Invariant indicating satisfiability
end
if θ contains the empty clause then
    if restart should be performed then
        return restart();
    end
    (learned_clause, backtrack_level) ← analyse_conflict();
    increment_conflict_count();
    return (learned_clause, backtrack_level, UNSAT);
end
increment_decision_level();
l ← select_literal(Q);
result ← QCDCL(θ ∧ l, Q) ;
switch (result, Q(l)) do
    case SAT ∧ ∃ do
        return result;
    end
    case SAT ∧ ∀ do
        return QCDCL(θ ∧ ¬ l, Q);
    end
    case UNSAT ∧ ∃ OR UNSAT ∧ ∀ do
        // Depending on quantification, handling is slightly different
           but ommitted for simplicity
        if backtrack_level == decision_level then
            add_learned_clause();
            // If learned clause is unit, backtrack level is set to 0
               and it's propagated as its own fact
            if learned clause is a unit clause then
                unit_propagate(θ);
            end
            repeat_current_propagation();
        else
            return result; // Continue backjump
        end
    end
    case RESTART do
        reduce_learned_clauses();
        restart();
    end
end
```

Algorithm 3: analyse_conflict(θ, trail, conflict)

```
learned_clause ← conflict;
// Iterate through the trail in reverse
for assignment in trail do
    if assignment is a decision variable then
        clause_responsible ← get_clause_responsible(assignment);
        learned_clause ← resolve(learned_clause, clause_responsible, assignment);
    end
    if stopping constraints met then
        backtrack_level ← calculate_backtrack_level();
        break;
    end
end
if unsatisfiability criteria met then
    return unsatisfiable();
else
    return (learned_clause, backtrack_level);
end
```

in the MiniSAT solver [10]. Q-resolution is halted upon fulfilling the following stopping constraints [42]:

1. Amongst the existential literals within the clause, only one of them has the highest decision level (not necessarily the current decision level).
2. This particular literal is at a decision level with an existential literal as the decision variable.
3. All the universal literals within the clause with a quantification level less than this literal are falsely assigned prior.

Constraints one and three ensure that this clause will become unit after back-jumping. Constraint two ensures that the solver reverts at least one existential decision; reverting only universal decisions doesn't resolve a conflict. The constraints governing whether a resolved clause results in unsatisfiability are [42]:

1. All existential literals within the clause are at decision level 0, or;
2. The clause consists entirely of universal literals (can be reduced to the empty clause via universal reduction).

Finally, the backtrack level calculation determines how far the solver needs to revert decisions made so that the learned clause can be utilised effectively. Once the stopping constraints are

met, we want to **backtrack such that the learned clause becomes unit**, as this allows the solver to perform unit propagation on the highest decision level literal, bringing the solver to a new state space and avoiding the decision branch that led to the previous conflict. Therefore, the backtrack level is equal to the second highest decision level of a literal in the learned clause. My scheme for conflict analysis results in backtracking to the **first UIP**, which has been shown to produce good results in experiments, as opposed to backtracking in which the learned clause consists entirely of decision variables.

3.5.4 Restart Policy

I implement a non-uniform restart interval policy based on the Luby series [4] such that the number of conflicts until a restart should be performed increases linearly as the number of restarts increases. As the optimal restart interval size is usually unknown, non-uniform restarts are desired. Further, a given problem will benefit from frequent restarts due to clause learning; however, to be able to solve the problem, it may require a deeper tree search with more conflicts. A non-uniform policy that allows occasional exponential increases accounts for both of these. The Luby series is a geometric equation that satisfies both of these desired conditions. It is defined as follows [19]:

$$L(i) \stackrel{\text{def}}{=} \begin{cases} 2^{k-1}, & \text{if } i = 2^{k-1} \\ L_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1 \end{cases}, \text{ where } i,k \in \mathbb{N} \tag{3.1}$$

Sequence Extract:	1,	1,	2,	1,	1,	2,	4,	1,	1,	2,	4,	8...

The sequence extract illustrates the nature of the Luby sequence, where i equates to the number of restarts. It is scaling exponentially non-exponentially. In other words, it allows frequent restarts while also occasionally allowing restart runs with larger amounts of conflicts to explore deeper into the decision tree. Further, it does this without scaling up the number of conflicts too quickly, as some geometric equations are known to do. The output of the sequence is multiplied by a starting constant. In my implementation I use 100, which defines the number of initial restarts allowed.

3.5.5 Clause Database Reduction Strategy

For reducing the list of learned clauses during a restart, I implemented an age-based deletion strategy. During a restart, I delete the oldest half of the learned clauses. This is a simple

approach to clause database reduction that combines the ideas proposed in Maple [30] and the age-based deletion approach [20]. Despite its simplicity, it serves its purpose well within my implementation and demonstrates the important ideas behind clause database reduction in CDCL.

3.6 Literal Selection Scheme

During DPLL and CDCL, the solver needs to select which literal to expand next by assigning it a 'guessed' value, true or false, and apply unit propagation to the given literal. The order with which literals are selected can significantly affect the speed at which a solution can be found, as a satisfiable or unsatisfiable assignment can be found sooner through different paths. However, there are restrictions on which literals must be selected first.

I define a QBF $Q_1.Q_2...Q_n \cdot \theta$ where Q_i are quantifier blocks with alternating quantification types and sets of variables and θ is a set of clauses. During literal selection for this QBF, all variables in the outermost quantifier block must be selected prior to the variables in subsequent quantifier blocks. However, within this block, they can be selected in any order. This allows me to implement two different methods of variable selection: **Ordered** and **Variable State Sum (VSS)**.

3.6.1 Ordered

Ordered selection is a simple scheme where, during literal selection, I select the first literal in the outermost quantifier block of the quantifier prefix.

3.6.2 Variable State Sum

Variable State Sum (VSS) is a selection scheme where, from the set of literals in the outermost quantifier block, I select the literal with the most occurrences within the set of clauses. This is determined by using the All-Watched literals data structure. The motivation is that by selecting the literals with the most occurrences, most clause elimination can occur when removing the clauses where the literal occurs, and unit propagation and other optimisations can occur in clauses where the negation of the literal occurs. In addition, I take this idea a step further, and for the variable p with the most clause occurrences:

- If p appears more often negatively than positively in the clause database, I select the literal as negative;

- Otherwise, I select the literal as positive.

This maximises clause elimination.

3.6.3 Void Quantifier Removal

In both literal selection schemes, if the selected literal does not appear in any clause, either negatively or positively, I remove the literal from the quantifier prefix and select the next literal, performing the same check. This is an important optimisation, as selecting these literals as decision variables results in searching branches and hence performing unit propagation, which doesn't reduce the clause database. This causes a significant increase in run-time, so in all solvers I use void quantifier removal.

3.7 Pre-resolution

This section will describe pre-resolution, a SAT solver optimisation based on Q-Resolution that aims to identify new and useful clauses prior to running the solver in the hopes of reducing the search space and therefore the solver's runtime. Q-Resolution is used within QCDCL for conflict analysis, but it has not been investigated whether doing Q-Resolution on the original set of clauses prior to solving will provide benefit.

The base algorithm is shown in **Algorithm 4**, where the configuration contains tuned hyperparameters found experimentally. The algorithm will loop through the original clause database, resolving all clause pairs where valid resolution can occur according to the rules of Q-Resolution defined in *definition 2.4.2*.

In addition, I have extended pre-resolution to apply resolution iteratively with the newly resolved clauses. The idea is that small, and theoretically better, clauses may not initially be found in the first resolution iteration. Therefore, by taking the longer initial resolved clauses and iteratively performing resolution, we can obtain more useful smaller clauses. I can then filter out the longer clauses, only adding the smaller clauses to the original clause database for use in the solver.

3.7.1 Hyperparameters

In this section, I outline the hyperparameters that govern the type and amount of resolved clauses that pre-resolution will add to the original clause database. These hyperparameters are:

Algorithm 4: PRE-RESOLUTION(θ, φ, Q, config)

```
resolutions_per_literal ← config.get_min_resolutions() / len(Q);
for (quantifier, literal) in Q do
    if quantifier is ∃ AND literal ∈ φ AND ¬literal ∈ φ then
        for positive_reference in φ(literal) do
            clause_one ← θ(positive_reference);
            for negative_reference in φ(¬literal) do
                clause_two ← θ(negative_reference);
                resolved_clause ← resolve(clause_one, clause_two, literal);
                // resolved_clause is not a duplicate
                if resolved_clause ∉ θ then
                    θ ← add_clause(resolved_clause);
                    if len(resolved_clause) > config.get_repeat_above() then
                        continue;
                    end
                else
                    continue;
                end
                if number of resolutions for literal > resolutions_per_literal then
                    break;
                end
            end
            if number of resolutions for literal > resolutions_per_literal then
                break;
            end
        end
    end
    if number of new resolved clauses > config.get_max_resolutions() then
        break;
    end
end
```

- **max_resolutions** - Limits the number of resolved clauses that can be added to the clause database. This is based on a percentage of the original clause database size.
- **min_resolutions** - Controls how many resolved clauses to add to the clause database at a minimum. This is based on a percentage of the original clause database size.
- **repeat_above** - When the length of a resolved clause is above this limit, perform another resolution for the current literal regardless of the resolutions_per_literal parameter.

The relationship between **max_resolutions** and **min_resolutions** is to enforce a lower and upper bound on the amount of newly resolved clauses to add. **min_resolutions** is a heuristic indicating the minimum number of newly resolved clauses I want to add to the clause database, where an approximately equal amount of resolved clauses is added for each literal in the given instance. Moreover, **max_resolutions** indicates the number of resolved clauses to stop pre-resolution at, regardless of whether there is an equal number of resolutions per literal. This is a requirement as the **repeat_above** heuristic will allow literals to have many resolutions when the length of a resolved clause is above a certain length, and in some cases the amount of added clauses can exceed the **min_resolutions** heuristic value significantly, and the large addition to the clause database will outweigh the benefits gained from the additional clauses. There is a margin between the **max_resolutions** and **min_resolutions** values to allow for beneficial clauses to be found before the final cutoff. Below, I provide some example heuristic values and explain their usage.

Example Heuristic Values

max_resolutions = 0.5: The algorithm will stop adding resolved clauses if the count is $> 50\%$ of original clause database size, regardless of whether there are an equal number of resolutions per literal.

min_resolutions = 0.25: The algorithm will,at minimum, add resolved clauses such that the count is 25% of the original clause database size. It does this by calculating the number of resolutions allowed per literal ($min\_resolutions/len(Q)$) and adding approximately an equal amount of new resolved clauses per literal in the QBF instance.

repeat_above = 3: When the resolved clause contains more than 3 literals, perform another resolution for the current literal regardless of whether the number of resolutions allowed per literal has already been exceeded.

Note that it may not be clear as to why resolutions allowed per literal is included as a heuristic rather than a minimum amount of clauses to add. However, during experimentation, if I were just to cap the number of resolutions allowed at a certain cutoff point without resolving equally on each literal, there was no benefit to applying pre-resolution. Further, in many cases, including pre-resolution in this format actually gave a negative performance benefit. Therefore, when applying pre-resolution, it is important to resolve on each literal in the quantifier prefix by approximately the same amount.

Note that during testing, I noticed that when iteratively applying pre-resolution, a large amount of duplicate clauses were being created, which enlarged the clause database exponentially throughout the iterations, slowing down the solving time drastically. To prevent this, I implemented a hash table to keep track of all clauses that exist and only add resolved clauses if they are completely new; this significantly sped up the pre-resolution run-time.

Now that the design has been outlined, the solvers and techniques used should be replicable. In the following section, I test and critically analyse the implemented solvers.

Chapter 4

Testing, Results, and Evaluation

This section will include how I tested the correctness and performance of my solvers, a comparison of different algorithms and optimisations, and a critical analysis of performance differences. Firstly, I will describe how to customise a solver and run it on instances or benchmarks. Then I will describe how I tested the correctness of my solvers. Following this will be the testing and critical analysis of DPLL, CDCL, and pre-resolution. In each of these sections, I will discuss how different optimisation methods affected the results, along with the impact that literal selection methods had on different solvers.

4.1 State of the Art

This chapter does not compare my work against state-of-the-art SAT solvers, as the SAT solvers presented in my project are not intended to compete with the state-of-the-art in terms of performance. The aims of my project are to explore the relative strengths of different SAT solving techniques and optimisations, rather than to achieve the highest possible level of performance. Therefore, the testing and results of this chapter consist of comparisons between my own implementations of different SAT solving methods and optimisation techniques, and the focus should be on the contributions and insights that this brings to the field of SAT solving.

4.2 Usability

Figure 4.1 shows the format of the JSON file, which stores the configuration of the solver. The **RunBenchmark** marks whether the solver should be run on a directory of QBF instances or a singular QBF instance. The **SolverType** key is either CDCL or DPLL, the **LiteralSelection** key is either VSS or Ordered, and the **PreResolutionConfig** keys contain their respective number

types or ”infinity”, excluding the **iterations** key. The remaining **SolverOptions** keys are flags on whether a particular feature should be used in the given solver. Finally, once you have Cargo setup in Rust, use the command ***”cargo run --release”*** to run the solver.

```
{
    "RunBenchmark": Bool,
    "BenchMarkPath": String,
    "InstancePath": String,
    "OutputFileName": String,
    "SolverOptions": {
        "SolverType": <SolverType>,
        "LiteralSelection": <LiteralSelectionType>,
        "Preprocess": Bool,
        "UniversalReduction": Bool,
        "PureLiteralDeletion": Bool,
        "Restarts": Bool,
        "PreResolution": Bool,
        "PreResolutionConfig": {
            "min_ratio": Float,
            "max_ratio": Float,
            "max_clause_length": Integer,
            "repeat_above": Integer,
            "iterations": Integer
        }
    }
}
```

Figure 4.1: JSON Configuration file determining solver composition

4.3 Testing Correctness

The best way to test the correctness of my solvers is to compare the solution output for a wide range of problem sets against strong, well-established solvers that have already been verified. CAQE is a certifying QBF solver [35] with very strong QBF solving capabilities. I am comparing the outputs from my solvers to the results outputted from CAQE to verify their correctness. Further, I test a wide range of different instance types, downloaded from the QBFLIB [14],

such that edge cases can be caught as best as possible to ensure the solver works correctly for instances with more complex structures, large sizes, and varying variable size, clause size, and quantifier alternations. All throughout implementation and evaluation, the correctness of the results is tested.

4.4 Data Sets

Deciding which instances to run the solvers on is a difficult task, as the hardness of an instance is not wholly dependent on the size of the instance but rather the complexity of the structure it has. Therefore, I use a mixture of different instance families from QBFLIB [14] for testing. The data sets I will be using are:

- **Castellini Suite** - Consists of various QBF-based encodings of the bomb-in-the-toilet planning problem [6].
- **Pan Suite** - Consists of QBF encodings of modal K formulas satisfiability [33].
- **Ayari Suite** - A family of 5 different QBF problem sets related to the formal equivalence checking of partial implementations of circuits [2].
- **Tacchella Suite** - A family of randomly generated QBF instances [16]

4.5 Testing DPLL

Throughout the following sections, I will refer to different solver configurations by various abbreviations, which are defined as follows:

- ***DPLL*** or ***CDCL***: refers to the QBF solving algorithm used.
- ***LR***: pure literal removal enabled.
- ***UR***: universal reduction enabled.
- ***PP***: preprocessing enabled.
- ***VSS***: variable state sum selection used.
- ***O***: ordered selection used.
- ***PR***: pre-resolution enabled.
- ***R***: restarts enabled.

For evaluating my DPLL implementation, I will start by demonstrating the strengths of the core optimisations. As a result, you will understand the relative strengths of different optimisation methods as well as the overall strength of the DPLL algorithm at solving different types of QBF instances.

Note that due to limited computing resources and the large number of instances being used, I've decided to time-out instances that have been running longer than 30 seconds.

4.5.1 Overall Performance

I evaluate overall performance by first finding the optimal DPLL solver configuration and applying it across a range of data sets. Table 4.1 shows results for DPLL using ordered selection on the Castellini suite [14], where bare DPLL is a configuration of DPLL that does not use any of the core optimisations.

Table 4.1: Solver results on the Castellini suite showing instances solved and CPU time for DPLL using a combination of universal reduction, pure literal deletion, and pre-processing with ordered selection

Type	DPLL · Ordered · Castellini						
Optimisation	Bare	LR	UR	LR+UR	PP	PP+LR	PP+LR+UR
Total solved	36	44	36	44	39	46	60
Satisfiable	20	20	20	20	20	20	20
Unsatisfiable	16	24	16	24	19	26	40
Timeout	40	32	40	32	37	30	16
Total Time (s)	1299	1079	1298	1081	1213	1034	591
Time for solved (s)	56.8	84.7	55.7	87.6	64.7	103	74.8
Avg Time (s)	1.58	1.93	1.55	1.99	1.66	2.24	1.27

The results show that the best performance solver used a combination of pre-processing, universal reduction, and pure literal removal techniques, where adding universal reduction to the pre-processing solver resulted in a drastic increase in the number of instances solved. I expected to see a strong performance increase from using each of the techniques as they are well-established optimisations; however, from these results, I see a particularly large increase in the number of unsatisfiable instances being solved when adding universal reduction. This is due to instances with clauses consisting entirely of universally quantified literals are able to be reduced to the empty clause, which makes the QBF unsatisfiable. This can be done prior to being passed to the DPLL procedure, which can reduce the run-time significantly as no

branching needs to occur. Further, universal reduction can result in the detection of more pure literals or unit clauses, which can then be removed. I will now consolidate these results by applying the optimal DPLL configuration to a range of QBF instance structures.

Table 4.2: Solver results for 4 suites showing instances solved and CPU time for DPLL with no optimisations and DPLL with all applicable optimisations, both using ordered selection

Suite	Castellini		Ayari		Pan		Tacchella	
Solver Type	Bare	Optimal	Bare	Optimal	Bare	Optimal	Bare	Optimal
Total	76	76	18	18	57	57	200	200
Solved	36	60	6	18	24	24	166	166
Timeout	40	16	12	0	33	33	34	34
Time (s)	1299	591	401	56.8	1098	1073	1161	1160

Table 4.2 shows results comparing a bare DPLL solver with a fully optimised DPLL solver for 4 different data suites whilst using ordered literal selection. The results show a huge performance increase when using the fully optimised DPLL, with a 66% increase in instances solved for the Castellini suite and a 200% increase in instances solved for the Ayari suite. Despite there being no increase in instances solved for the Pan and Tacchella suites, there is no performance decrease as the total time to solve the instances did not increase, and in the Pan suite, it improved. In the following section, I will do a similar evaluation but for the VSS literal selection method to establish whether my optimal DPLL configuration is the same for both selection methods.

4.5.2 Literal Selection Method

Table 4.3 shows results comparing a bare DPLL solver with a fully optimised DPLL solver for 4 different data suites whilst using VSS literal selection. The results show that for VSS literal selection, when applying all optimisations to DPLL, we see a significant increase in the instances solved for the Castellini, Ayari, and Pan suites, with a 150%, 183%, and 19% increase in instances solved respectively. Despite the Tacchella suite showing no improvement when applying the optimisations, there is no performance decrease, which means the optimisations can be applied without causing any harm to the solving ability for all data suites shown here.

Table 4.3: Solver results for 4 suites showing instances solved and CPU time for DPLL with no optimisations and DPLL with all applicable optimisations, both using VSS selection

Suite	Castellini		Ayari		Pan		Tacchella	
Solver Type	Bare	Optimal	Bare	Optimal	Bare	Optimal	Bare	Optimal
Total	76	76	18	18	57	57	200	200
Solved	22	55	6	17	27	32	174	174
Timeout	54	21	12	1	30	25	26	26
Time (s)	1750	750	405	84.2	1026	915	890	889

Comparing the solved instances for different data sets for ordered selection and VSS selection, there are obvious strengths to using a mixture. Ordered selection solves more instances on the Castellini and Ayari suites, while VSS solves more instances on the Pan and Tacchella suites. The results show that no one literal selection method is optimal for all QBF instance structures.

Since the combination of pure literal deletion, universal reduction, and pre-processing show a strong performance increase across most data sets with no performance decrease in any data set, I will by default use them in each DPLL solver when evaluating my CDCL and pre-resolution implementations. Now that the DPLL implementation has been thoroughly tested by analysing the relative strengths of different optimisations, I can conclude that DPLL has strong solving capabilities across a wide range of instance structures.

4.6 Testing CDCL

In this section, I evaluate and critically analyse my CDCL solver implementation by comparing it against the optimal DPLL solver presented in Section 4.5. I will provide insight into reasons why particular solver configurations may have performed differently than expected. In addition, I will discuss the effectiveness of my restart policy's implementation.

4.6.1 Overall performance

I will evaluate the overall performance of my CDCL algorithm by making comparisons to the performance of DPLL. This way I can evaluate whether CDCL provides the significant performance increase I expect.

Table 4.4: Solver results for 4 suites showing instances solved and CPU time for optimised DPLL and CDCL using VSS selection

Solver	DPLL · VSS · LR · UR · PP —**OR**— CDCL · VSS · UR · PP · R							
Suite	Castellini		Ayari		Pan		Tacchella	
Solver Type	DPLL	CDCL	DPLL	CDCL	DPLL	CDCL	DPLL	CDCL
Total	76	76	18	18	57	57	200	200
Solved	55	55	17	14	32	53	174	179
Timeout	21	21	1	4	25	4	26	21
Time (s)	750	732	84.2	219	915	425	889	885

Table 4.4 shows results comparing a fully optimised DPLL solver with the CDCL solver for 4 different data suites whilst using VSS literal selection. Taking the overall results across all solvers, we see CDCL solve 25 more instances, which is a 9% increase compared to DPLL. *Note that this increase would be larger if the Pan Suite had an equal share of the number of instances*. However, this performance increase is not equal across all suites, as CDCL gives a huge performance increase in the Pan suite, a minimal performance increase in the Castellini and Tacchella suites, and a small decrease in performance for the Ayari suite. I will analyse these results separately in more depth to give reasons for these differences.

Pan Suite

On the Pan suite, the CDCL solver solved 66% more instances than the optimised DPLL solver and had half the CPU run-time. Therefore, it is clear that CDCL performs significantly better than DPLL on this suite. I will now analyse the reasons for its success so we understand what other QBF instance structures it will be strong at solving.

Firstly, by analysing the conflict count for each instance, I can see that 81% of the instances encountered conflicts that led to a learned clause being added to the clause database. These conflict counts ranged from 1 conflict to 403 conflicts, which resulted in a learned clause, with the instances with higher conflict counts being solved much faster by CDCL than DPLL. This is because when CDCL encounters a conflict and analyses the reason that the conflict occurred during conflict analysis, it can potentially learn a new and useful clause. These newly learned clauses can then prune large sections of the search space as we can back-jump up multiple decision levels at a time, as opposed to DPLL, which backtracks one level at a time. This saves huge amounts of time by avoiding branches that will repeatedly result in conflicting assignments produced by the same decisions that were made at earlier decision levels. We can see data supporting this conclusion in figure 4.3.

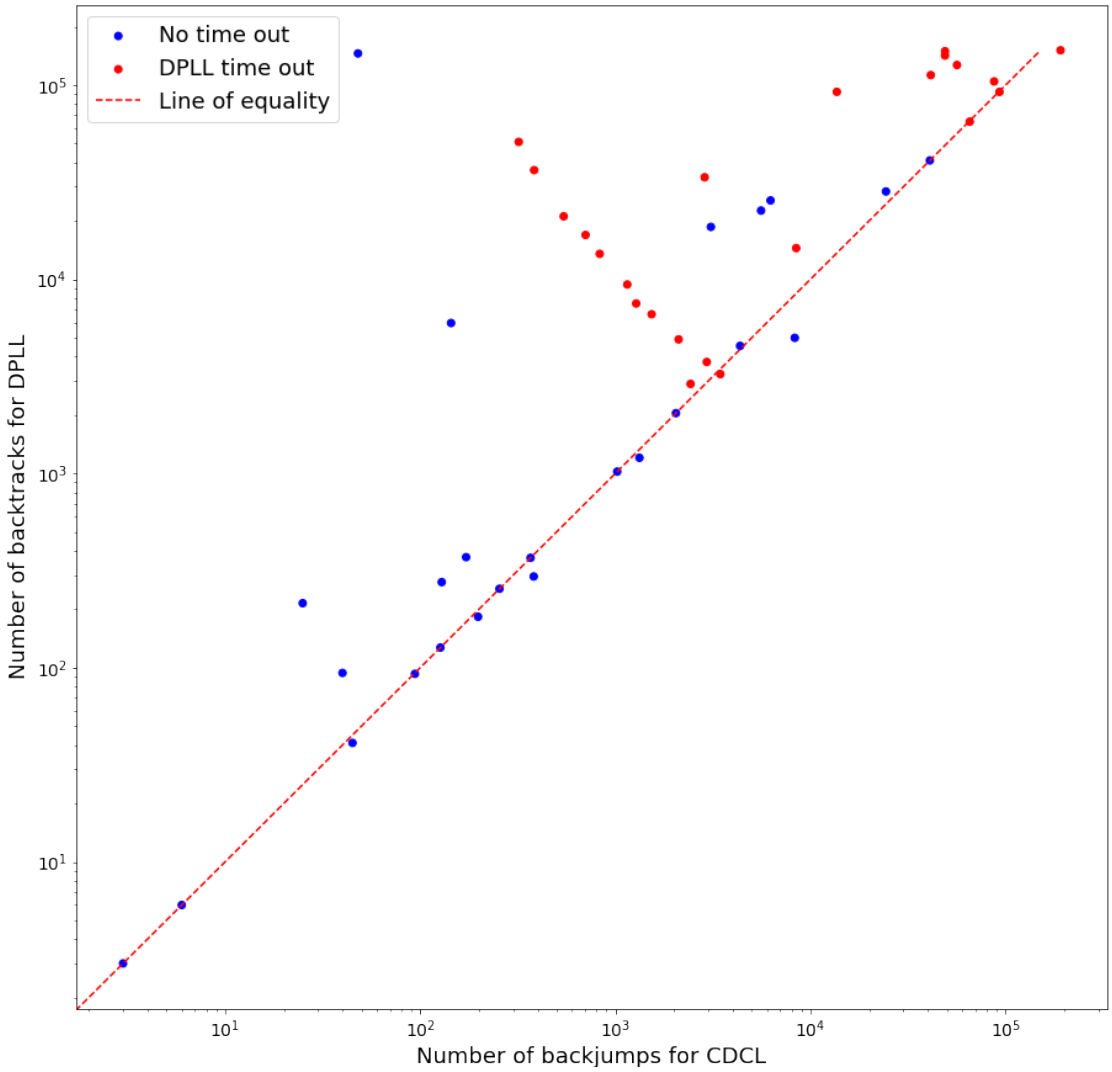

Figure 4.2: Solver comparison on backtrack/back-jump count for the Pan Suite

Figure 4.2 shows a comparison of the number of backtracks or back-jumps performed by DPLL and CDCL on the Pan suite [14]. The graph illustrates that for most instances, CDCL required significantly fewer backtracks to solve a given instance. This supports my analysis of why the CDCL performance was stronger, as we can see through this data that CDCL back-jumped across multiple decision levels frequently, which resulted in a significant time reduction. In addition, many of the instances were unsolved by DPLL and timed out after 30 seconds. If these instances were left to run to completion, we'd see the graph much more skewed above the line of equality.

Castellini

On the Castellini suite, CDCL didn't solve any additional instances and improved the speed by 2.4% when compared with DPLL. Therefore, for the Castellini suite, CDCL doesn't provide much of an improvement over DPLL; however, CDCL should still be preferred due to the small speed improvement. The reasons for the minimal improvement can be seen through an analysis of the conflict count. Overall, CDCL didn't have many instances where it produced newly learned clauses by performing conflict analysis. Since CDCL provides significant improvement when learning new clauses, in the cases where it doesn't learn it provides minimal improvement.

Tacchella

On the Tacchella suite, CDCL solved 2.9% more instances. Despite this only being a small improvement, it further consolidates the idea that CDCL should be preferred over DPLL for most data set suites. The reasons for the small improvements are similar to those for the Castellini suite. Take into consideration that the Tacchella suite consists of randomly generated instances that aren't representative of real-world problems, so these results are only used for comparing the relative strengths of different algorithms.

Ayari

On the Ayari suite, CDCL solved 18% fewer instances. This is an interesting result, as you would expect CDCL to perform better or at least as well as DPLL in all instances since CDCL is built upon DPLL. The CDCL solver doesn't use the pure literal removal optimisation while DPLL does, and as a result, instances that produce many pure literals during unit propagation are likely to be better solved by DPLL. From my DPLL analysis, we see the strength of pure literal removal, and so we can see why not using this optimisation can lead to a performance decrease in some QBF structures.

4.6.2 Restarts

Table 4.5: Solver results on the Pan suite showing instances solved and CPU time for CDCL with and without restarts enabled

Solver	CDCL · VSS · UR · PP	
Suite	Castellini	
Restart Type	No Restarts	Restarts
Total	57	57
Solved	48	53
Timeout	9	4
Time (s)	556	408

Table 4.5 shows a comparison of a CDCL solver performing restarts against a CDCL solver performing no restarts on the Pan suite, using VSS selection. We see that it is beneficial to include a restart policy, even if it uses a simple clause database reduction policy like mine, which uses age-based deletion. Performing restarts increased the number of instances solved by 10% which shows the success of performing restarts on instances that produce many learned clauses, such as those in the Pan suite do.

4.6.3 Literal Selection Method

In this section, I will show the performance difference when using a CDCL solver that uses ordered literal selection and suggest reasons for the observed results.

Table 4.6: Solver results for 4 suites showing instances solved and CPU time for optimised DPLL and CDCL using ordered selection

Solver	DPLL · O · LR · UR · PP **—OR—** CDCL · O · UR · PP · R							
Suite	Castellini		Ayari		Pan		Tacchella	
Solver Type	DPLL	CDCL	DPLL	CDCL	DPLL	CDCL	DPLL	CDCL
Total	76	76	18	18	57	57	200	200
Solved	60	61	18	12	24	8	166	164
Timeout	16	15	0	6	33	49	34	36
Time (s)	591	570	56.8	222	1073	1526	1160	1321

Table 4.6 shows a comparison between the optimal DPLL solver and the CDCL solver for 4 different data suites while using ordered literal selection. For the Castellini, Ayari, and Tacchella suites, the results show similar performance changes as VSS selection, but there is a huge performance drop in the Pan suite. CDCL solves 85% fewer instances when it uses ordered selection rather than VSS selection. This is a consequence of having no literal selection scheme which dynamically changes as the clause database changes. For example, with VSS

selection, when conflicts are learned and additionally when restarts occur, the scores for the literals appearing in the learned clauses are updated according to VSS, which in turn increases the likelihood of these literals being selected, guiding the solver towards making use of the learned clauses. With ordered selection, none of this is present, and the literals are selected in the same order regardless of the learned clauses; therefore, it does not guide the search whatsoever, resulting in a significant decrease in performance. Since the other instance suites don't benefit from learned clauses to the same extent as the Pan suite does, they don't suffer from a similar performance decrease.

This concludes the testing and critical analysis of my CDCL implementation. Overall, I've shown that CDCL provides a huge performance increase over DPLL, especially when used on instances that encounter conflicts that can be learned from. Overall, CDCL is preferred over DPLL for most instance suites.

4.7 Testing Pre-Resolution

In this section, I will evaluate the efficiency and success of my custom implementation pre-resolution. I will first discuss the hyperparameters that provide the strongest solving capabilities and the reasons for their success. Following this, I will discuss the general performance of the optimisation on various data sets.

4.7.1 Hyperparameters

During testing, I tuned the pre-resolution hyperparameters to establish an optimal balance between them such that the best-performing values could be used for the evaluation.

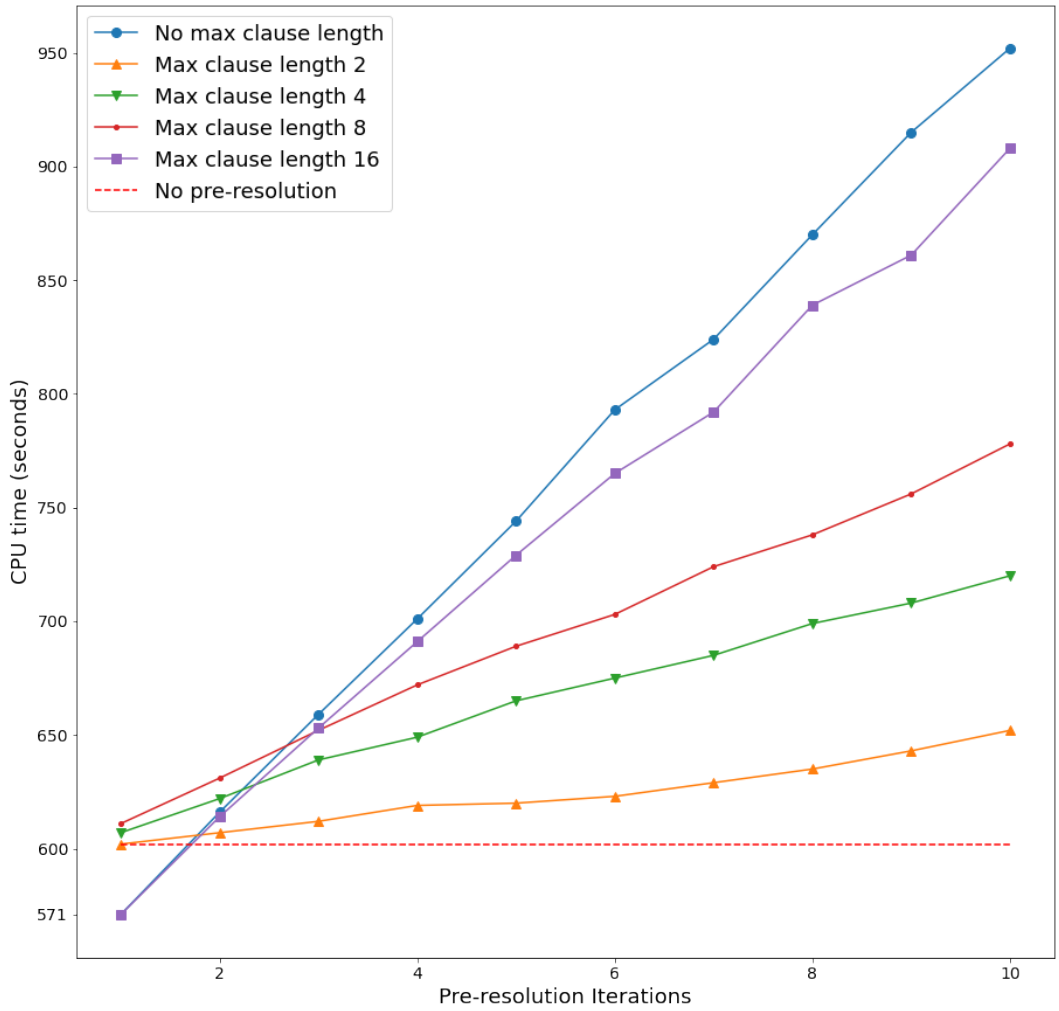

Figure 4.3: A graph showing CPU time to solve the Castellini suite instances at different pre-resolution hyperparameter configurations for DPLL

Figure 4.3 shows a graph depicting the CPU run-time to solve the Castellini suite at different combinations of hyperparameters. I vary the number of pre-resolution iterations to perform and the maximum clause length I allow to be added to the clause database after running pre-resolution.

My initial hypothesis was that performing more iterations would result in finding higher-quality clauses that the solver could use to find a solution to an instance faster. The results shown contradict this, showing that as the number of pre-resolution iterations performed increases, the run-time increases linearly, whereas the best performing configurations only performed one pre-resolution iteration.

In addition, the results show that taking only the clauses with fewer literals doesn't provide

the best performance. After analysing the instances in more detail, I found that there were some instances that produced useful clauses containing over 10 literals, which goes against the assumption that the shorter the length of the clause, the better. Finally, I can conclude that **the best pre-resolution configuration was having no limit on the length of clauses to add to the database and only performing one pre-resolution iteration.**

4.7.2 Overall Performance

Table 4.7: Solver results for 4 suites showing instances solved and CPU time for optimised DPLL with and without pre-resolution applied, using ordered selection

Solver	DPLL · O · LR · UR · PP							
Suite	Castellini		Ayari		Pan		Tacchella	
Pre-resolution	No PR	PR	No PR	PR	No PR	PR	No PR	PR
Total	76	76	18	18	57	57	200	200
Solved	60	60	18	14	24	24	166	165
Timeout	16	16	0	4	33	33	34	35
Time (s)	591	565	56.8	153	1073	1096	1160	1222

Now that I have the optimal configuration for pre-resolution, it is important to know its relative strength across different types of solvers and different types of QBF instances, so that it is known where it's best to apply the optimisation. Table 4.7 shows a comparison between DPLL using pre-resolution and DPLL not using pre-resolution for 4 different data suites while using ordered literal selection. The results show that resolution is not beneficial for the Ayari, Pan, and Tacchella suites, as applying pre-resolution increased the CPU time to run the suites and solved fewer instances. However, the results show that for the Castellini suite, pre-resolution provides a speed improvement as it takes less CPU time to solve the same number of instances, which suggests pre-resolution is beneficial when applied to QBF structures that consist of QBF-based encodings of the bomb-in-the-toilet planning problem, as is the case with Castellini.

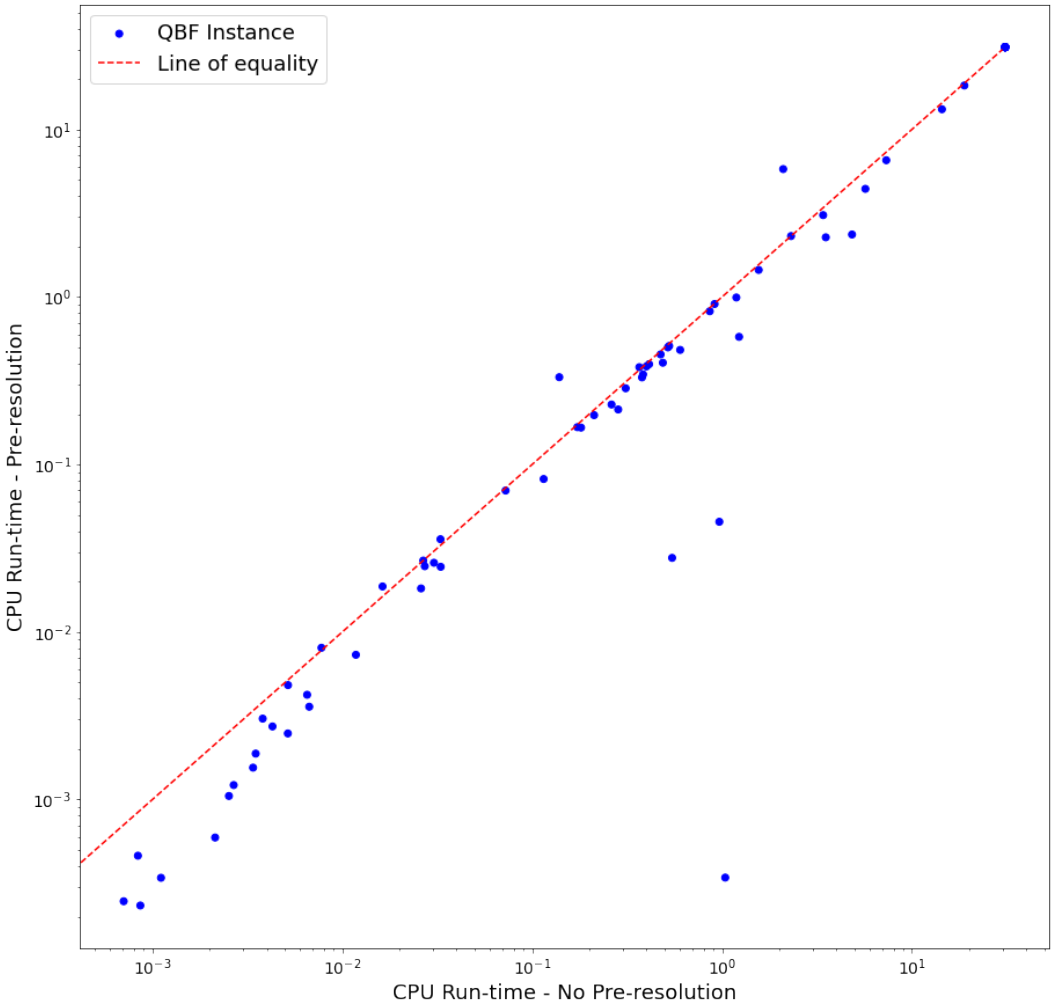

Figure 4.4: A graph comparing CPU run-time using DPLL with and without pre-resolution on instances in the Castellini suite

Figure 4.4 shows a comparison of CPU time to solve an instance with DPLL when using pre-resolution and no pre-resolution. The results further consolidate the fact that applying pre-resolution gave a speed increase across a majority of the instances, removing the possibility that there was a 'lucky' instance that was solved much faster, which was skewing the data.

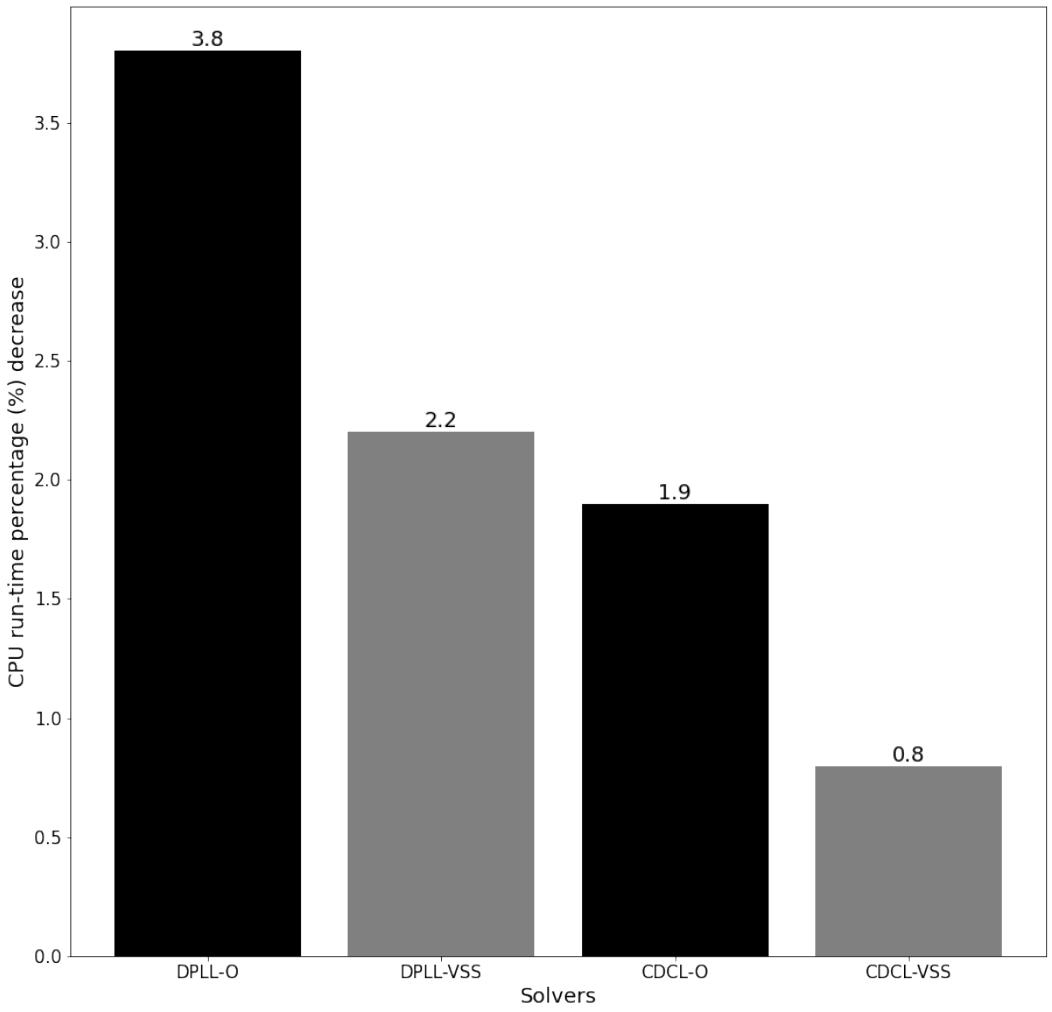

Figure 4.5: A graph showing the CPU run-time percentage decrease for various solvers on the Castellini suite

Now that pre-resolution has been shown to be useful, it is important to investigate whether it is beneficial across different types of solvers and to investigate its strength relative to other optimisations. Figure 4.5 displays the CPU run-time improvement for CDCL and DPLL solvers on the Castellini suite when using both literal selection methods. The results show that pre-resolution provides improvement across each solver.

To conclude, relative to the core optimisations of pure literal deletion, universal reduction, and pre-processing, pre-resolution provides significantly less performance increase. However, despite this and the fact that pre-resolution is non-beneficial in several instance suites, the evaluation has still been beneficial in guiding which instances it should be applied to for improving performance.

4.7.3 Investigating Resolution Constraints

Throughout my time researching the background for my pre-resolution implementation, I found that the definitions for Q-Resolution explicitly mention that it should only be performed on existentially quantified literals. In this section, I will be discussing an investigation into whether resolving on both universally and existentially quantified literals is valid and whether it is beneficial in the context of pre-resolution.

Table 4.8: Solver results for 4 suites showing instances solved and CPU time for DPLL when pre-resolution is applied to existential literals or both existential and universal literals

Solver	DPLL · O · LR · UR · PP · PR							
Suite	Castellini		Ayari		Pan		Tacchella	
Resolved On	$\exists$	$\exists\wedge\forall$	$\exists$	$\exists\wedge\forall$	$\exists$	$\exists\wedge\forall$	$\exists$	$\exists\wedge\forall$
Total	76	76	18	18	57	57	200	200
Solved	60	60	14	14	24	24	165	165
Timeout	16	16	4	4	33	33	35	35
Time (s)	565	575	153	152	1096	1099	1222	1223

Table 4.8 shows a comparison between pre-resolution resolving on only existential literals and resolving on both existential and universal literals for 4 different data suites while using DPLL and ordered selection. The results show that there is no difference in the number of instances solved and minimal differences in the time taken to solve the instances. Resolving on both universal and existential literals produces valid clauses, as no instance was solved incorrectly.

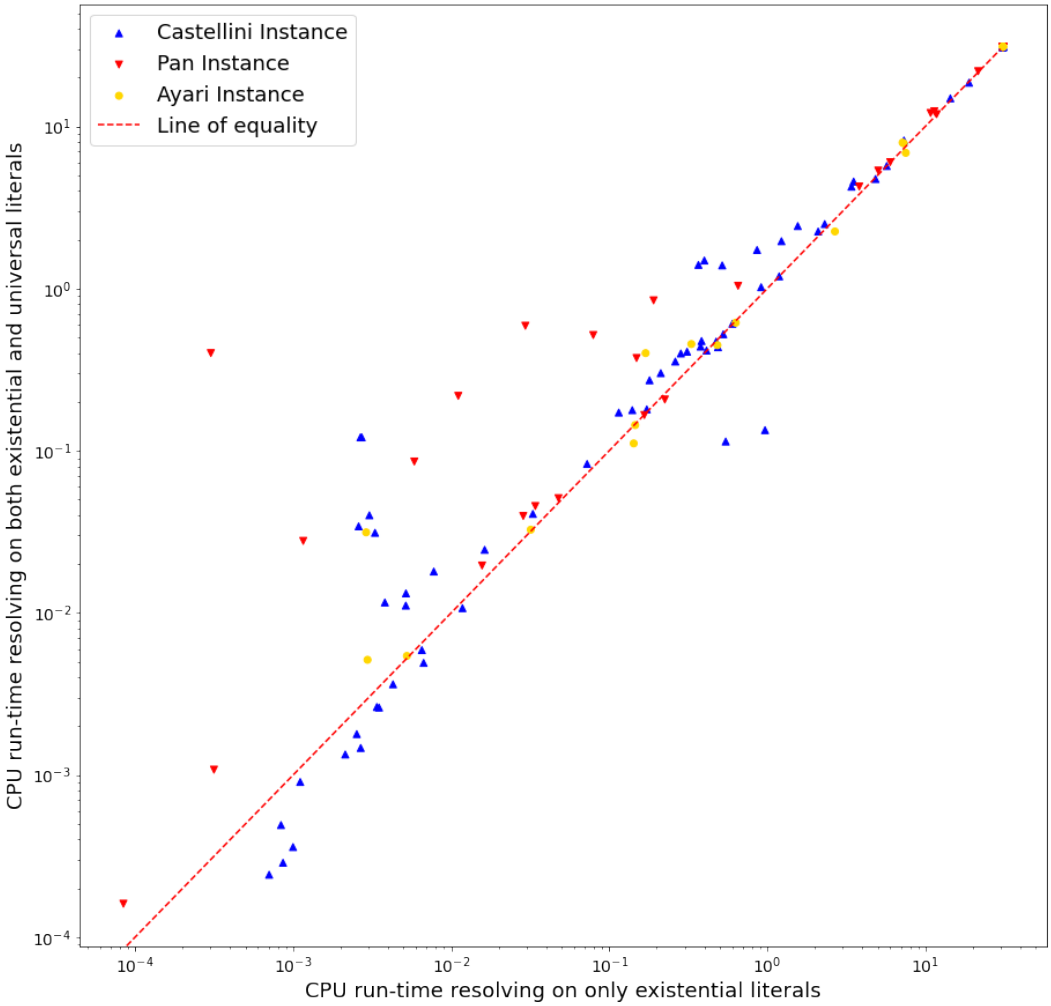

Figure 4.6: A graph showing the CPU time comparison of resolving on existential literals only and both existential and universal literals for different instance suites

Figure 4.6 is an illustration of the data shown in Table 4.8. The graph shows that the CPU time spent solving a given instance in the Pan Suite skews higher when resolving on both existential and universal literals. Further, the Castellini instances skew both above and below the line of equality, which also aligns with the fact that overall time is minimally impacted.

To conclude, in this section I've shown that performing Q-Resolution on both existential and universal literals is valid in the context of pre-resolution and very minimally impacts the solving time. However, it is still overall faster when resolving only existential literals.

4.8 Overall Remarks

In summary, you should now have an understanding of the strengths of different solvers, optimisations, and literal selection methods and why they perform better or worse for different data sets and QBF instance structures. All in all, I have produced an efficient tool that accurately solves QBF and allows for the evaluation of a wide range of QBF solving techniques and optimisation methods. Further, my results have provided insight into the effectiveness of pre-resolution in solving QBF. Some of my findings include:

- DPLL benefits significantly from pure literal removal, universal reduction, and pre-processing, and they should be used when solving every QBF instance structure.
- CDCL outperforms DPLL in every QBF instance apart from the Ayari suite, which benefits from pure removal. In particular, it outperforms DPLL the most in the Pan suite.
- CDCL benefits significantly from universal reduction and should be used for each QBF instance.
- CDCL benefits from a restart scheme, even if the restart scheme is relatively simple.
- Determining which literal selection method to use on a particular instance is important, and variable state sum is preferred for CDCL.
- Incorporating pre-processing into DPLL and CDCL has the largest improvement, allowing them to solve many more instances and reducing CPU time to run benchmarks.
- The pre-resolution optimisation does not provide as much benefit as the core optimisations.

Chapter 5

Summary and Conclusions

5.1 Summary

In this project, I have investigated and implemented a tool for solving QBF instances using a customisable solver configuration that additionally allows for performance evaluation of the solvers using QBF instance suites. With this, I explored and developed the core backtracking algorithms DPLL and CDCL, with applicable optimisations of pure literal deletion, universal reduction, and pre-processing, along with the literal selection methods Ordered and Variable State Sum literal selection. Further, I investigated, implemented, and evaluated a new optimisation pre-resolution based on the technique of Q-Resolution. To conclude the implementation, I then tested and produced a critical analysis and comparison of all the implemented components on a variety of QBF instance suites, such that the relative performance difference of the QBF solver algorithms and optimisation techniques can be quantified for further application in the field of QBF solvers. Finally, I have also explored related work, in particular, the knowledge and techniques behind the top QBF evaluators of recent years, and produced a brief case study on CAQE, a certifying QBF solver, so that the reader understands the scope of the academic field of QBF solvers.

5.2 Achievements

The initial aims and objectives have been achieved and exceeded, with additional objectives being added and accomplished throughout. I have gained an in-depth understanding of the techniques and concepts used in the field of QBF and SAT solvers above and beyond what was required for my implementation, and as a result, I've further sparked my interest in the field. In addition, I have created a tool that allows the comparison of combinations of different QBF

solving algorithms and optimisation techniques by evaluating them on QBF benchmark suites such that performance differences can be quantified. Using this tool, I've been able to provide insights into the relative performance differences of a variety of QBF solving algorithms and optimisations, which gives future academics quantifiable knowledge when they're planning their implementation of a QBF solver. With this, I've produced a deep critical analysis of the performance of each component of my implementations so that the reasons for the performance gains and losses can be understood in the context of propositional logic. Finally, the achievement of which I am most proud is the investigation, development, and evaluation of the novel optimisation pre-resolution, which can provide foundational knowledge for future endeavours and extensions to the concept of pre-resolution. Despite pre-resolution being an extension of my original aims for the project, I am proud of the insights and conclusions it provides.

5.3 Critical Reflection and Limitations

Despite the project's achievements, there are several limitations and areas for improvement. Firstly, although my QBF solvers are able to solve a large variety of instances, their speed and overall solving ability are not comparable to or competitive with the current top QBF solvers submitted to the QBFEVAL competition. This was not in the scope of the aims of my project, but it has reduced the variety of QBF instance suites I've been able to utilise in my evaluation. Secondly, I didn't have the time to implement additional restart policies and clause data base reduction policies such that different methods could be evaluated against each other in the context of CDCL. Again, this was not in the scope of the aims, but I understand a comparison of different restart policies would have been beneficial. Finally, my implementation of pre-resolution isn't a strong optimisation when applied to CDCL and DPLL, and it was only beneficial in one of the tested instance suites, Castellini, whereas in the remaining suites it decreased the performance. Despite this limitation of pre-resolution, I've still provided valuable knowledge about the utility of the optimisation. Looking back to the project aims and objectives (Section 1.2) you can see all the described goals and success criteria have been met and exceeded. All in all, I am proud of the range and depth of the knowledge I have gained and my achievements.

5.4 Future Work

In this section, I will propose avenues for future work. Firstly, I would investigate and implement the two watched literals technique discussed in the paper 'Watched Data Structures for QBF' [13]. This optimisation would reduce the number of data manipulations required on the

clause database throughout unit propagation, as well as the size of the data structures that need to be restored upon backtracking. Therefore, this addition would provide a significant speed increase due to most time being spent within unit propagation currently, and as a result, more instances would be solved. Secondly, I would implement a restart scheme based on Literal Block Distance (LBD) discussed in the paper 'Refining Restarts Strategies for SAT and UNSAT' [1]. I could then evaluate the performance differences between a complex and simple restart scheme within the context of CDCL. Finally, the main area for future development would be to implement a CEGAR-based algorithm for solving QBF, which is discussed in the paper 'Solving QBF with Counterexample Guided Refinement' [21]. CEGAR-based solvers have very strong solving capabilities. In doing this, I would also be gaining a wider range of knowledge surrounding SAT solvers. Following this, I would incorporate pre-resolution into CEGAR so that I can investigate whether pre-resolution is more beneficial in a non-backtracking-based SAT solver. On the whole, the primary proposed future developments will be the implementation of a CEGAR-based QBF solver and additional optimisations for DPLL and CDCL.

Bibliography

[1] Gilles Audemard and Laurent Simon. "Refining Restarts Strategies for SAT and UNSAT". In Michela Milano, editor, *Principles and Practice of Constraint Programming*, pages 118–126, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-33558-7_11.

[2] Abdelwaheb Ayari and David Basin. "Bounded Model Construction for Monadic Second-Order Logics". In E. Allen Emerson and Aravinda Prasad Sistla, editors, *Computer Aided Verification*, pages 99–112, Berlin, Heidelberg, 2000. Springer Berlin Heidelberg. https://doi.org/10.1007/10722167_11.

[3] Valeriy Balabanov and Jie-Hong Jiang. "Unified QBF certification and its applications". *Formal Methods in System Design*, 2012. https://doi.org/10.1007/s10703-012-0152-6.

[4] Armin Biere and Andreas Fröhlich. "Evaluating CDCL Restart Schemes". In *Proceedings of Pragmatics of SAT 2015 and 2018*, pages 1–17. https://doi.org/10.29007/89dw.

[5] Marco Cadoli, Andrea Giovanardi, and Marco Schaerf. "An Algorithm to Evaluate Quantified Boolean Formulae". In *AAAI/IAAI*, 1998. https://dl.acm.org/doi/10.5555/295240.295609.

[6] Claudio Castellini, Enrico Giunchiglia, and Armando Tacchella. "SAT-based planning in complex domains: Concurrency, constraints and nondeterminism". *Artificial Intelligence*, pages 85–117, 2003. https://doi.org/10.1016/S0004-3702(02)00375-2.

[7] Ashok K. Chandra, Dexter C. Kozen, and Larry J. Stockmeyer. "Alternation". *J. ACM*, page 114–133, 1981. https://doi.org/10.1145/322234.322243.

[8] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-Proving". *Journal of Symbolic Logic*, 1967. https://doi.org/10.2307/2271269.

[9] Martin Davis and Hilary Putnam. "A Computing Procedure for Quantification Theory". *J. ACM*, 1960. https://doi.org/10.1145/321033.321034.

[10] Niklas Eén and Niklas Sörensson. "An Extensible SAT-solver". In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-24605-3_37.

[11] Matt Fredrikson. "Lecture Notes on Solving SAT with DPLL", 2021. https://www.cs.cmu.edu/~15414/lectures/14-sat-dpll.pdf.

[12] Michael R. Garey and David S. Johnson. *"Computers and Intractability; A Guide to the Theory of NP-Completeness"*. W. H. Freeman Co., USA, 1990.

[13] Ian Gent, Enrico Giunchiglia, Massimo Narizzano, Andrew Rowley, and Armando Tacchella. "Watched Data Structures for QBF Solvers". In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 25–36, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-24605-3_3.

[14] Enrico Giunchiglia, Massimo Narizzano, Luca Pulina, and Armando Tacchella. "Quantified Boolean Formulas satisfiability library (QBFLIB)", 2005. www.qbflib.org.

[15] Enrico Giunchiglia, Massimo Narizzano, Luca Pulina, and Armando Tacchella. "Quantified Boolean Formulas satisfiability library (QBFLIB)", 2005. www.qbflib.org/qdimacs.html.

[16] Enrico Giunchiglia, Massimo Narizzano, Armando Tacchella, and Moshe Y. Vardi. "Towards an Efficient Library for SAT: a Manifesto". *Electronic Notes in Discrete Mathematics*, pages 290–310, 2001. https://doi.org/10.1016/S1571-0653(04)00329-4.

[17] Ákos Hajdu and Zoltan Micskei. "Efficient Strategies for CEGAR-Based Model Checking". *Journal of Automated Reasoning*, 2020. https://doi.org/10.1007/s10817-019-09535-x.

[18] Holger H. Hoos and Thomas Stützle. "1 - INTRODUCTION". In Holger Hoos and Thomas Stützle, editors, *Stochastic Local Search*, The Morgan Kaufmann Series in Artificial Intelligence, pages 13–59. Morgan Kaufmann, San Francisco, 2005. https://doi.org/10.1016/B978-155860872-6/50018-4.

[19] Jinbo Huang. "The Effect of Restarts on the Efficiency of Clause Learning.". pages 2318–2323, 2007. https://dl.acm.org/doi/10.5555/1625275.1625649.

[20] Sima Jamali and David Mitchell. *"Simplifying CDCL Clause Database Reduction"*, pages 183–192. 2019. https://doi.org/10.1007/978-3-030-24258-9_12.

[21] Mikoláš Janota, William Klieber, Joao Marques-Silva, and Edmund Clarke. "Solving QBF with Counterexample Guided Refinement". In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 114–128, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-642-31612-8_10.

[22] Penelope Kirby. "Logic Chapter 2", 2011. https://www.math.fsu.edu/~pkirby/mad2104/SlideShow/s2_1.pdf.

[23] Kevin C. Klement. "Propositional Logic". In *Internet Encyclopedia of Philosophy*. 2004. https://iep.utm.edu/propositional-logic-sentential-logic/.

[24] Konstantin Korovin. "DPLL / CDCL, Lecture Notes". 2020.

[25] Konstantin Korovin. "Propositional Logic. Lecture Notes". 2020.

[26] Florian Lonsing. "An Introduction to QBF Solving", 2017. http://www.florianlonsing.com/talks/Lonsing-SAT-SMT-school-india-2017.pdf.

[27] Florian Lonsing, Uwe Egly, and Martina Seidl. "Q-Resolution with Generalized Axioms". pages 435–452, 2016. https://doi.org/10.1007/978-3-319-40970-2_27.

[28] J.P. Marques Silva and K.A. Sakallah. "GRASP-A New Search Algorithm for Satisfiability". In *Proceedings of International Conference on Computer Aided Design*, pages 220–227, 1996. https://doi.org/10.1109/ICCAD.1996.569607.

[29] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. "Chaff: engineering an efficient SAT solver". In *Proceedings of the 38th Design Automation Conference (IEEE Cat. No.01CH37232)*, pages 530–535, 2001. https://doi.org/10.1145/378239.379017.

[30] Alexander Nadel and Vadim Ryvchin. *"Chronological Backtracking"*, pages 111–121. 2018. https://doi.org/10.1007/978-3-319-94144-8_7.

[31] Massimo Narizzano, Luca Pulina, and Armando Tacchella. "The QBFEVAL web portal". In *Logics in Artificial Intelligence*, pages 494–497. Springer Berlin Heidelberg, 2006. https://doi.org/10.1007/11853886_45.

[32] Andreas Nonnengart and Christoph Weidenbach. "Chapter 6 - Computing Small Clause Normal Forms". In Alan Robinson and Andrei Voronkov, editors, *Handbook of Automated Reasoning*, Handbook of Automated Reasoning, pages 335–367. North-Holland, Amsterdam, 2001. https://doi.org/10.1016/B978-044450813-3/50008-4.

[33] Guoqiang Pan, Uli Sattler, and Moshe Y. Vardi. "BDD-Based Decision Procedures for K". 2002. https://doi.org/10.1007/3-540-45620-1_2.

[34] Luca Pulina and Martina Seidl. "The 2016 and 2017 QBF solvers evaluations (QBFEVAL'16 and QBFEVAL'17)". *Artificial Intelligence*, pages 224–248, 2019. https://doi.org/10.1016/j.artint.2019.04.002.

[35] Markus Rabe and Leander Tentrup. "CAQE: A Certifying QBF Solver". pages 136–143, 2015. https://doi.org/10.1109/FMCAD.2015.7542263.

[36] Andrew G. D. Rowley. "Watching Clauses in Quantified Boolean Formulae". In Francesca Rossi, editor, *Principles and Practice of Constraint Programming – CP 2003*, pages 994–994, Berlin, Heidelberg, 2003. Springer Berlin Heidelberg. https://doi.org/10.1007/978-3-540-45193-8_118.

[37] Ankit Shukla, Armin Biere, Luca Pulina, and Martina Seidl. "A Survey on Applications of Quantified Boolean Formulas". In *2019 IEEE 31st International Conference on Tools with Artificial Intelligence (ICTAI)*, pages 78–84, 2019. https://doi.org/10.1109/ICTAI.2019.00020.

[38] Richard Tichy and Thomas Glase. "Clause Learning in SAT", 2006. https://www.cs.princeton.edu/courses/archive/fall13/cos402/readings/SAT_learning_clauses.pdf.

[39] Vincent Vallade, Ludovic Le Frioux, Souheib Baarir, Julien Sopena, Vijay Ganesh, and Fabrice Kordon. "Community and LBD-Based Clause Sharing Policy for Parallel SAT Solving". In Luca Pulina and Martina Seidl, editors, *Theory and Applications of Satisfiability Testing – SAT 2020*, pages 11–27, Cham, 2020. Springer International Publishing. https://doi.org/10.1007/978-3-030-51825-7_2.

[40] Eric W. Weisstein. "Tree. From MathWorld—A Wolfram Web Resource". https://mathworld.wolfram.com/SkolemFunction.html.

[41] Hantao Zhang and Mark Stickel. "An Efficient Algorithm for Unit Propagation". 1996. https://www.researchgate.net/publication/2508830_An_Efficient_Algorithm_for_Unit_Propagation.

[42] Lintao Zhang and Sharad Malik. "Conflict Driven Learning in a Quantified Boolean Satisfiability Solver". page 442–449, New York, NY, USA, 2002. Association for Computing Machinery. https://doi.org/10.1145/774572.774637.

Appendix A

Additional Watched Data Structures

A.1 Two Watched Literals

Two Watched literals is a unit propagation optimisation first proposed in the 2001 paper ”Chaff: Engineering an Efficient SAT Solver” [29], which showed an efficient way to quickly visit all clauses that become a new unit clause by a single addition to the set of assignments. It is used for the removal of clauses and the removal of literals from clauses [13]. The simplest way to implement this is to look at every clause in the set of clauses that contain a literal that the current assignment sets to 0, and keep a count of how many 0 assignments there are. We would then modify the counter every time a literal in the clause is set to 0. However, we would be visiting the clause every time there was an assignment. This would waste time, as we only need to know when the clause becomes a unit clause, and hence we only need to visit the clause when the counter indicates there is only one literal not set to 0 [29].

The technique proposed in Chaff says we can pick any two literals not assigned to 0 in each clause and observe them. We then know that until one of these literals is assigned to 0, the clause is not unit. This removes the unnecessary checking of each clause every time an assignment is made, and it is shown to give a performance improvement on difficult SAT benchmarks by one or two orders of magnitude [29].

This technique was extended to QBF in the 2003 paper ”Watched Data Structures for QBF Solvers” [13] which discusses some of the issues with implementing watched literals in QBF.

A.2 Watched Clauses

The detection of pure literals is important in QBF problems, the existence of these allows us to perform pure literal deletion, reducing the problem and potentially leading to further unit propagation [36]. Clause-watching is a method to efficiently detect pure literals. For this, each literal has two associated values: one representing a positive occurrence of the literal in a clause, and the other representing a negative occurrence of the literal in a clause. For each variable, the invariants are:

- The variable is pure in one of the signs - one of the values is empty;
- The variable is removed - both values are empty;
- The variable is not removed and not pure - both values aren't empty.

This literal invariant is defined in the paper "Watched Data Structures for QBF Solvers" [13]

The literals' respective values update when the clause that the literal occurs in is removed. First, search for the existence of the literal in the same sign within another clause; if one is found, this is the new value. After this update, we will know whether the variable is non-pure (another value is found for that sign), pure (one of the values is now empty), or removed (no value exists for either sign).