ON EQUIVALENCY REASONING FOR CONFLICT DRIVEN CLAUSE LEARNING SATISFIABILITY SOLVERS

Azam Heydari

A thesis
in
The Department
of
Computer Science and Software Engineering

Presented in Partial Fulfillment of the Requirements
For the Degree of Doctor of Philosophy (Computer Science)
Concordia University
Montréal, Québec, Canada

June 2012

Concordia University

School of Graduate Studies

This is to certify that the thesis prepared

By: **Ms. Azam Heydari**

Entitled: **On Equivalency Reasoning for Conflict Driven Clause Learning Satisfiability Solvers**

and submitted in partial fulfillment of the requirements for the degree of

Doctor of Philosophy (Computer Science)

complies with the regulations of this University and meets the accepted standards with respect to originality and quality.

Signed by the final examining commitee:

_____ Chair

_____ External Examiner

_____ Examiner

_____ Examiner

_____ Examiner

_____ Supervisor

Approved _____

Chair of Department or Graduate Program Director

_____ 20 _____ _____

Dr. Robin A.L. Drew, Dean

Faculty of Engineering and Computer Science

Abstract

On Equivalency Reasoning For Conflict Driven
Clause Learning Satisfiability Solvers

Azam Heydari, Ph.D.

Concordia University, 2012

Satisfiability problem or SAT is the problem of deciding whether a Boolean function evaluates to true for at least one of the assignments in its domain. The satisfiability problem is the first problem to be proved NP-complete. Therefore, the problems in NP can be encoded into SAT instances. Many hard real world problems can be solved when encoded efficiently into SAT instances. These facts give SAT an important place in both theoretical and practical computer science.

In this thesis we address the problem of integrating a special class of equivalency reasoning techniques, the strongly connected components or SCC based reasoning, into the class of conflict driven clause learning or CDCL SAT solvers. Because of the complications that arise from integrating the equivalency reasoning in CDCL SAT solvers, to our knowledge, there has been no CDCL solver which has applied SCC based equivalency reasoning dynamically during the search. We propose a method to overcome these complications. The method is integrated into a prominent satisfiability solver: MiniSat. The equivalency enhanced MiniSat, Eq-MiniSat, is used to explore the advantages and disadvantages of the equivalency reasoning in conflict clause learning satisfiability solvers. Different implementation approaches for Eq-MiniSat are discussed. The experimental results on 16 families of instances shows that equivalency reasoning does not have noticeable effects for the instances in one family. The equivalency reasoning enables Eq-MiniSat to outperform MiniSat on eight classes of instances. For the remaining seven families, MiniSat outperforms Eq-MiniSat. The experimental results for random instances demonstrate that almost in all cases the number of branchings for Eq-Minisat is smaller than Minisat.

Acknowledgements

This thesis would not have been completed without the support and encouragement of many individuals, too numerous to mention here.

My greatest gratitude goes to my supervisor, Dr Clement Lam, whose encouragement, inspiration and support made this project possible. I am most deeply indebted for his invaluable guidance and patience throughout many hours of critical discussions, and for his considerate and understanding character. His devotion to the research, in-depth knowledge and generous attitude were most crucial in helping me navigate the difficult moments of graduate years.

I'm also grateful to Dr Vasek Chvatal, from whom I learnt a lot about my research topic and many aspects of comprehensible academic writing in the earlier years of my time at Concordia.

I must acknowledge the guidance and help from the other members of my defence committee: Dr Gene Cooperman, Dr Chris Cummins, Dr Thomas Fevens, and Dr Brigitte Jaumard.

Many friends have provided invaluable support and encouragement throughout this journey. I extend special gratitude to Hengameh Saberi, Neda Zare, Farzan Rohani, Bahareh Vazhbakht, Mohammad Ebn Alian, and Kevin Halifax.

And last but not least, I want to express my gratitude to my mom, Robabeh Saberi, whom her unconditional love and faith have always been the greatest motivation for me in life. I want to thank my dad, Hossein Heydari, for his love and support he has given me throughout my life.

Contents

List of Algorithms

List of Tables

Chapter 1

Introduction

Satisfiability problem, or SAT, is one of the most important problems in both practical and theoretical computer science. As a result, it is one the problems that have been extensively researched. In this thesis, we investigate the integration of new techniques into one of the most prominent SAT algorithms.

This chapter gives an overview of the structure of the thesis and its contributions. Section 1.1 gives a short introduction to the the thesis topic. Section 1.2 reviews the thesis contributions. Section 1.3 provides the thesis organization.

1.1 The Boolean Satisfiability Problem

Satisfiability problem is the problem of deciding whether a Boolean function evaluates to true for at least one of the assignments in its domain. The satisfiability problem is the first problem to be proved NP-complete [23, 71]. Moreover, the concept of NP-completeness was introduced by this proof. These facts give SAT an important place in theoretical computer science.

As a result of SAT being NP-complete, the problems in NP can be encoded into SAT instances. It has been shown that many hard real world problems can be solved when encoded efficiently into SAT instances. Some examples include problems like planning [62], software verification [54], hardware verification [15], bounded model checking [21], Quasigroup completion problem [44], automatic test pattern generation [70], and Van Der

Warden numbers [3].

Generally, the algorithms to solve SAT can be categorized in two groups: *complete* and *incomplete*. The complete algorithms, given sufficient time, guarantee to find a satisfiable assignment if there exists one or to prove unsatisfiability otherwise. The incomplete algorithms do not guarantee finding the solution, therefore they are not able to prove unsatisfiability. In the meanwhile, the incomplete algorithms, usually, perform better than complete algorithms on satisfiable instances.

Among incomplete SAT algorithms are local search [47, 89], tabu search [57, 78], and genetic algorithms [58].

Some of the complete satisfiability algorithms are Davis-Putnam-Logemann-Loveland or DPLL [28], Davis-Putnam or DP [29], Stalmarck's method [91], and binary decision diagrams (BDDs) [18].

DP algorithm, introduced about half a century ago, is believed to be one of the first SAT algorithms. Surprisingly, it is still the most widely used algorithm to solve SAT. In Chapter 2, we give a short overview of this algorithm. Many modern SAT solvers are based on this algorithm. Their differences are in the implementation details, and in the new techniques integrated into the original algorithm.

There are two main categories for the DPLL based SAT solvers: *look-ahead* and *look-back*. The former gathers information based on the current state of the SAT instance, the latter is based on learning from the assignment failures. The work in this thesis investigates the integration of a new technique in the class of look-back SAT solvers. The following section reviews the thesis contributions.

1.2 Thesis Contribution

DPLL is a branch and bound algorithm. One of the techniques to improve DPLL is to minimize the search space by maximizing the logical reasoning to reduce the size of the SAT instance. One of the reasoning methods is *equivalency reasoning* which finds equivalent variables in the SAT instance. The equivalency reasoning and its effects have been widely studied for different DPLL-based SAT solvers [8, 17, 39, 48, 72, 99].

Equivalency reasoning can be done statically only in the beginning of the search, or dynamically during the search. For look-ahead SAT solvers both dynamic and static integration have been studied [72, 99]. Integrating the dynamic equivalency reasoning into look-ahead SAT solvers resulted in solving a hard class of SAT instances that DPLL-based solvers are unable to solve in reasonable time [72].

In look-back SAT solvers, because of its complications, the equivalency reasoning has been only studied statically at the beginning of the search[8, 17]. Because of these complications, to our knowledge, there has been no look-back SAT solver with integrated dynamic equivalency reasoning during the search. The work in this thesis studies the complications that arise from the dynamic integration of the equivalency reasoning into look-back SAT solvers. It also presents a method to overcome these complications. The result is integrated into a minimal look-back SAT solver MiniSat [32].

The equivalency reasoning techniques in this thesis use strongly connected components of a directed graph called *SCC implication graph* to identify the equivalencies. In this thesis, we present a customized algorithm based on Tarjan's algorithm [96] to generate the strongly connected components of the SCC implication graph.

The SCC implication graph, not only helps in identifying the equivalent literals, but also has other valuable information that can be used to guide the search. This thesis describes some possible use of this information in subsumption and self-subsumption rules, and in identifying the conflict variables without BCP. The former simplifies the formula, the latter helps the solver by avoiding unnecessary propagations.

The experimental results are based on our equivalency reasoning enhanced SAT solver Eq-MiniSat. These results are used to investigate the benefits and disadvantages of dynamic equivalency reasoning in look-back SAT solvers. The results indicate that although the integration of equivalency reasoning is not always beneficial, but it helps Eq-MiniSat to consistently outperform MiniSat on some classes of instances. The experiments on random instances indicates that for a majority of instances Eq-Minisat has less number of branchings than Eq-MiniSat.

Another topic that we investigated in this thesis is the use of orthogonal lists or dancing links [52, 63] to represent a SAT instance. Our experiments, which are not reported in

the thesis, show that although the orthogonal lists data structure outperforms the counter-based data structures for SAT [59, 81, 92], it is slower than the watched literals scheme [81].

1.3 Thesis Organization

Chapter 2 gives an introduction to DPLL algorithm and its ancestor DP algorithm. The DPLL algorithm in its original recursive and its more recent iterative form is described.

A minimal solver based on DPLL, MiniSat [32], is the foundation for the work in this thesis. This solver belongs to a category of solvers that are called *Conflict Driven Clause Learning*, or *CDCL*. Therefore, in Chapter 2, we review the structure of CDCL solvers in general.

Chapter 3 provides an overview of the previous research on equivalency reasoning for look-back SAT solvers. In this chapter, we discuss the complications that arises in integrating the equivalency reasoning into CDCL SAT solvers. A method to overcome these complications is provided. We have implemented this method into the minimal CDCL SAT solver MiniSat [32]. The equivalency enabled MiniSat is called Eq-MiniSat.

Chapter 4 gives an overview of the implementation details of Eq-MiniSat. This chapter reviews the basics that are used in implementing Eq-MiniSat including the set union-deunion data structure [77] and Tarjan's strongly connected components algorithm [96] and the customized modifications that are applied on them. This chapter also provides an overview of the data structure used in the equivalency reasoning engine.

In Chapter 5, we present the experimental results for Eq-MiniSat. The results in this chapter are used to compare different methods to implement Eq-MiniSat. Also, the chapter provides a comparison of MiniSat and Eq-MiniSat on a selected set of instances.

Finally, in Chapter 6 the thesis conclusions and future possible work is provided.

Chapter 2

DPLL Algorithm: Past to Present

This chapter reviews the Davis Putnam Logemann Loveland (DPLL) algorithm [28, 29], which is the foundation for the research in this thesis. DPLL algorithm is the most widely used approach to solve SAT instances. Due to the extensivity of the research in the field of DPLL based SAT solvers, we only review the concepts that are directly related to our research. A comprehensive and relatively recent survey on the SAT solvers can be found in [16].

In Section 2.1, the basic definitions and terminologies for the SAT problem are presented. We review the DPLL algorithm in Section 2.2. Section 2.3 reviews the *preprocessing techniques*. Preprocessing techniques aim to simplify the formula at the beginning of the search. In Section 2.4, we discuss some of the most commonly used data structures for SAT. Section 2.5 provides a short review of one of the major improvements for DPLL based SAT solvers: *Conflict Driven Clause Learning*, or *CDCL* method. CDCL enables DPLL based solvers to solve problems that, previously, were widely considered out of the reach of DPLL algorithm. Section 2.6 briefly reviews some of the mostly used branching rules in DPLL solvers. Section 2.7 summarizes the material discussed in this chapter.

2.1 Basic Definitions and General Overview of Algorithms

A *variable* v is a symbol used to represent a Boolean statement in logic that can take the value either true or false (1 or 0, respectively). The *Negation* of v is denoted by $\bar{v}$ such that

$\bar{v} = 1 - v$. A *literal* is defined as a variable or its negation.

A *clause* is a set of zero or more literals. The *length* of a clause is the number of its literals. An *empty* clause has length zero.

A clause is *satisfied* if at least one of its literals is true. A *truth assignment* or *assignment* for a formula $\mathcal{F}$ is a set of assigned variables.

A *formula* or *SAT instance* is a set of clauses. A formula is *satisfiable* if there exists an assignment that satisfies all of its clauses[1]. A *complete assignment* for a formula $\mathcal{F}$, is an assignment in which all the variables in the formula $\mathcal{F}$ are assigned a value. Otherwise, it is a *partial* assignment. An assignment of length zero, the empty assignment is a partial assignment. An empty assignment for a SAT instance on n variables, can be extended to 2^n complete assignments.

A clause is *redundant* in a formula $\mathcal{F}$, if it can be inferred from other clauses in the formula.

Example 2.1. *In a formula $\mathcal{F}$ with clauses $C_1 = \{v_1\}$, $C_2 = \{v_1, v_2\}$, and $C_3 = \{v_1, \bar{v}_2\}$, the clauses C_2 and C_3 imply clause C_1. Therefore, the clause C_1 is a redundant clause.*

Let σ_1 and σ_2 to be two assignments on variables $v_1, \cdots, v_n$. The assignment σ_2 is called an *extension* of the assignment σ_1 if and only if for every v_i with $1 \leq i \leq n$ that have been assigned by σ_1, we have $\sigma_1(v_i) = \sigma_2(v_i)$.

If all the literals in a clause C in a formula $\mathcal{F}$ are assigned to false by an assignment σ, then the assignment is a *conflict* assignment for the formula $\mathcal{F}$. If a partial assignment is a conflict assignment, then all of its extended assignments are also conflict assignments.

If all the clauses in a formula are satisfied by an assignment σ, then σ is a *satisfying* assignment. A formula is *satisfiable* if it has at least one satisfying assignment.

If a literal w is assigned to true in a formula $\mathcal{F}$, then the *residual formula*, $\mathcal{F}|w$, is obtained from formula $\mathcal{F}$ as follows:

- For every clause C with literal $w \in C$, remove the clause C from the formula.
- For every clause C with literal $\bar{w} \in C$, remove the literal $\bar{w}$ from the clause C.

[1]This is equivalent to a formula in *Conjunctive Normal Form* (CNF).

A clause C is called a *unit* clause, if all of its literals except for one unassigned literal w are false. If a formula $\mathcal{F}$ has a unit clause C with unit literal w for a partial assignment σ, then σ can be extended to a satisfying assignment if and only if the literal w is assigned to true. Therefore, formula $\mathcal{F}$ is satisfiable if $\mathcal{F}|w$ is satisfiable. In this case, the clause C *implies* the literal w and literal w is an *implication* of clause C based on the partial assignment σ. The literal w is called an *implied* literal. The clause C is called the *reason* for the implied literal w.

Whenever a unit clause is found in a formula $\mathcal{F}$, then the formula $\mathcal{F}$ can be simplified by setting the unit literal w to true. The residual formula $\mathcal{F}|w$ might have new unit clauses because of the assignment of w to true. Therefore, assigning a unit literal might propagate into new unit assignments. This kind of reasoning to simplify a SAT formula is called *Unit Propagation (UP)* or *Boolean Constraint Propagation (BCP)*.

A variable v is called *monotone* in a formula $\mathcal{F}$ if there is no $\bar{v}$ in $\mathcal{F}$. If a formula $\mathcal{F}$ has a monotone literal w, then $\mathcal{F}$ is satisfiable if $\mathcal{F}|w$ is satisfiable. The *monotone rule* is the process of satisfying all the monotone literals in a formula.

The Satisfiability Problem (SAT) is the problem of finding a truth assignment for a formula or showing that no such assignment exists. SAT is one of the most important problems in computer science. It was the first problem shown to be NP-complete [23, 71]. Moreover, the concept of NP-completeness was originated by the proof of SAT being NP-complete.

Experimental results have established the fact that different SAT algorithms have different memory and run-time behavior on the same input instances. There has been extensive research to understand the complexity of SAT [24, 88] by investigating the instances that are hard for all the current SAT algorithms.

There are classes of SAT instances that are known to be solvable in polynomial time, for example 2-satisfiability [65] and horn-satisfiability [30]. Also, many practical problems that have been reduced to SAT can be solved in polynomial time. Some of the areas where SAT have been applied include planning [62], software verification [54], hardware verification [15], bounded model checking [21], Quasigroup completion problem [44], automatic test pattern generation [70], and Van Der Warden numbers [3]. Therefore, an efficient SAT solver can

be of practical use.

Because of its theoretical and practical importance, SAT has been widely studied in computer science. As a result, there have been extensive research to design efficient algorithms. The most widely used algorithm to solve SAT is the DPLL algorithm that was introduced in 1962. The DPLL algorithm [28] is based on Davis Putnam (DP) algorithm [29]. In Section 2.2, we review the DPLL algorithm and its ancestor DP algorithm.

2.2 DP and DPLL Algorithms: an Overview

Two formulae are called *equisatisfiable* if they both are satisfiable or both are unsatisfiable. The satisfying assignments for equisatisfiable formulae might be different.

Let C_1 and C_2 be clauses such that for exactly one literal w, we have $w \in C_1$ and $\bar{w} \in C_2$. As in [20], we call such clauses *clashing*. The *resolvent* clause $C = C_1 \triangledown C_2$ is defined as

$$C = (C_1 - w) \cup (C_2 - \bar{w}).$$

The operator $\triangledown$ is the *resolution operator*, and the operation is called the *resolution operation*.

For a formula $\mathcal{F}$ define the sets $\mathcal{S}_w$ and $\mathcal{S}_{\bar{w}}$ to be the set of clauses in $\mathcal{F}$ having literals w and $\bar{w}$ respectively. Their *resolvent set*, $\mathcal{S} = \mathcal{S}_w \triangledown \mathcal{S}_{\bar{w}}$, is defined as

$$\mathcal{S} = \{C | \exists C_1 \in \mathcal{S}_w, \exists C_2 \in \mathcal{S}_{\bar{w}} : C = C_1 \triangledown C_2\}.$$

The maximum size for the resolvent set $\mathcal{S}$ is $|\mathcal{S}_w| \times |\mathcal{S}_{\bar{w}}|$.

The resolution operation is the basis for Davis Putnam or DP algorithm [29] introduced in 1960. At each iteration, the DP algorithm generates a formula which is *equisatisfiable* to the first formula but with fewer variables. The algorithm terminates by either generating an empty clause or an empty formula. The former proves the unsatisfiability (because of that an empty clause is also called a *conflict* clause), the latter proves that the formula is satisfiable. A more detailed description follows.

At every iteration, the DP algorithm simplifies the formula by the monotone rule and

by unit propagation. When the simplification is done, the algorithm chooses a variable v in the formula $\mathcal{F}$. If no such variable exists, the formula is satisfiable. Otherwise, for the chosen variable v, the algorithm does the following:

- It adds the resolvent set $\mathcal{S}_v \triangledown \mathcal{S}_{\bar{v}}$ to $\mathcal{F}$.
- It removes the clauses in $\mathcal{S}_v$ and $\mathcal{S}_{\bar{v}}$ from $\mathcal{F}$.

The transformed formula is equisatisfiable to the original formula. The DP algorithm is shown in Algorithm 2.1.

Algorithm 2.1 DP Algorithm

Input: formula $\mathcal{F}$
Output: determines the formula $\mathcal{F}$ is satisfiable or not.

```
1: while (there exists a unit or monotone literal w in F) do
2:     F = F|w
3: end while
4: if (F has an empty clause) then
5:     return unsatisfiable
6: end if
7: if (F is empty) then
8:     return satisfiable
9: end if
10: choose a variable v in F
11: add S = S_v ▽ S_v̄ to F
12: remove clauses in S_v and S_v̄ from F
```

In the worst case, the resolution step in the DP algorithm results in an exponential growth in the number of clauses. Therefore, the memory requirement for DP algorithm is exponential in the worst case. To solve this problem, in 1962, Davis, Putnam, Loveland, and Logemann introduced a refinement of the DP algorithm resulting in the DPLL algorithm [28].

DPLL algorithm is a branch and bound algorithm. It is based on the fact that for every variable v, the formula $\mathcal{F}$ is satisfiable if and only if at least one of the formulae $\mathcal{F}|v$ or $\mathcal{F}|\bar{v}$ is satisfiable.

In order to find a satisfying assignment for a formula $\mathcal{F}$, the algorithm recursively finds the satisfying assignments for the formulae $\mathcal{F}|v$ and $\mathcal{F}|\bar{v}$. The algorithm is shown in Algorithm 2.2.

Algorithm 2.2 DPLL Algorithm

Input: formula $\mathcal{F}$

Output: determines the formula $\mathcal{F}$ is satisfiable or not. If the formula is satisfiable, it returns a satisfiable assignment.

```
1: while (there exists a unit or monotone literal w) do
2:     F = F|w
3: end while
4: if (F contains an empty clause) then
5:     return unsatisfiable
6: end if
7: if (all the literals are assigned) then
8:     return satisfiable
9: end if
10: choose an unassigned variable v
11: if (DPLL(F|v) = satisfiable) then
12:     return satisfiable
13: end if
14: if (DPLL(F|v̄) = satisfiable) then
15:     return satisfiable
16: end if
17: return unsatisfiable
```

The recursion tree for the DPLL algorithm is a binary tree. Every edge represents a choice of literal that is assigned to true. Every internal node is a residual formula for a partial assignment. The leaves are the complete assignments.

At every step, the algorithm divides the search space based on a chosen variable v. The variable v is called the *decision* or *branching* variable. The variables that are assigned because of the unit propagation are called *implied* variables.

A variable has level ℓ if it has been assigned at the level ℓ of the tree. We use the notation $v = 0@\ell$ if the variable v is assigned to false at level ℓ. Similarly, we use the notation $v = 1@\ell$ if the variable v is assigned to true at level ℓ.

There are two main categories for DPLL SAT solvers: *look-ahead* and *look-back*. The look-back techniques learn from the conflict assignments in the search space in order to avoid encountering them again. The look-ahead techniques spend time to choose the best variable to branch on. Empirically, it has been shown that while look-back SAT solvers perform better on large structured instances, the look-ahead solvers outperform on small hard random instances [5].

Among the SAT solvers that use look-back techniques are GRASP [92], Chaff [81], MiniSat [32] and PicoSat [14]. The solvers Satz [74] and March-dl [51] are among look-ahead solvers.

In this chapter, we discuss the main components of a modern look-back DPLL SAT solver. In Section 2.3 we review the preprocessing techniques for SAT solvers. Section 2.4 reviews some of the main data structures for DPLL SAT solvers.

The DPLL algorithm, in its original form, backtracks to the maximum level with a decision variable that has not been tested for both values. This is called *chronological backtracking*. Look-back SAT solvers discover the reason for the conflict, and then backtrack to the smallest level that is the reason for the conflict. Therefore, they backtrack *non-chronologically*. Non-chronological backtracking along with *clause learning* is one of the prominent improvements to the DPLL algorithms over the last decades. In Section 2.5 we review these methods.

The method to choose the decision variable is called the *branching rule*. It is well-known that for a formula $\mathcal{F}$, different branching rules might result in trees that differ exponentially in size for the same algorithm [53, 69, 82]. Therefore, there has been a lot of research on designing efficient branching rules. We review some of the branching rules in Section 2.6.

2.3 Preprocessing Methods

Preprocessing techniques are used to reduce the size of a formula, meaning the number of variables and clauses in the beginning of the search. The simplest preprocessing techniques are the unit literal and monotone literal rules. The preprocessing step is a place to apply the simplifications that are time-consuming if applied dynamically during the search. Different preprocessing techniques have been shown to have different performances on different classes of instances. Therefore, a key point in efficient SAT solving is to find a balance between the time that is spent in the preprocessing and the reduction in the size of the formula. In this section, we briefly review some of the most used techniques.

If for clauses C_1 and C_2, we have $C_1 \subset C_2$, then clause C_2 is *subsumed* by clause C_1. It is easy to see that a subsumed clause is redundant, therefore it can be removed from the

formula. The *Subsumption rule* is the process of removing subsumed clauses from a formula [75].

Example 2.2. *Clause $C_1 = \{v_1, v_2, \bar{v}_3\}$ is subsumed by clause $C_2 = \{v_1, \bar{v}_3\}$. Therefore, if both of these clauses are in a formula $\mathcal{F}$, the clause C_1 is redundant in the formula $\mathcal{F}$.*

Let C_1 and C_2 be clashing clauses on variable v. If $C_1 - \{v\}$ is subsumed by $C_2 - \{\bar{v}\}$, then we have $(C_1 \triangledown C_2) \subset C_1$. Therefore, clause C_1 can be subsumed by $C_1 \triangledown C_2$. The clause C_1 is replaced by $C_1 \triangledown C_2$ in the formula which is equivalent to eliminating one literal from the original clause. The clause C_1 is *self-subsumed* using the clause C_2. The process of removing self-subsumed clauses is called the *self-subsumption rule* [31].

Example 2.3. *Let $C_1 = \{v_1, v_2, \bar{v}_3\}$ and $C_2 = \{\bar{v}_1, v_2\}$ be two clauses in a formula $\mathcal{F}$. The clause $C_1 - \{v_1\}$ is subsumed by clause $C_2 - \{\bar{v}_1\}$. Therefore, by self-subsumption rule, the clause C_1 can be replaced by the resolvent clause $\{v_2, \bar{v}_3\}$ in the formula $\mathcal{F}$.*

Hyper resolution [11] method generates new clauses based on resolving a set of input clauses so that the result is either a binary or unit clause.

Example 2.4. *The set of clauses $\{v_1, v_2, v_3, v_4\}$, $\{\bar{v}_2, v_5\}$, $\{\bar{v}_3, v_5\}$, and $\{\bar{v}_4, v_5\}$ results in the binary clause $\{v_1, v_5\}$.*

Hyper resolution method has been shown to be effective specially if combined with *equivalency reasoning* [11].

Equivalency reasoning discovers *equivalent literals* in a formula. Literals w_1 and w_2 are equivalent if one being true implies the other one being true, and one being false implies the other one being false.

Example 2.5. *The clauses $\{v_1, \bar{v}_2\}$ and $\{\bar{v}_1, v_2\}$ imply that literals v_1 and v_2 are equivalent.*

If two literals w_1 and w_2 are equivalent in a formula $\mathcal{F}$, then the literal w_1 can be replaced by w_2, and the literal $\bar{w}_1$ can be replaced by the literal $\bar{w}_2$ in the formula $\mathcal{F}$ resulting in a simplified formula with fewer number of variables, and fewer number of clauses and/or shorter clauses. The process of substituting the equivalent literals is called *equality*

reduction. In [17] experimental results suggest that equality reduction on the instances with lots of binary clauses is beneficial to the SAT solvers.

Failed literal probing [34] assigns a value to literal w. If the unit propagation results in a conflict, then the clause $\{\bar{w}\}$ can be added to the formula.

Variable elimination [29, 31] is a method based on resolution. A variable v is removed from the formula $\mathcal{F}$ by adding the resolvent of the clauses in S_v and $S_{\bar{v}}$ to $\mathcal{F}$ and removing clauses in S_v and $S_{\bar{v}}$ from $\mathcal{F}$. The choice of the variable to be eliminated is based on the number and the length of the clauses with that variable.

2.4 Data Structures from Counter Based to Watched Lists

Assigning a literal w to true in a formula $\mathcal{F}$ requires the DPLL SAT solver to create the residual formula $\mathcal{F}|w$. Therefore, the data structure for a SAT solver should be able to efficiently access the unsatisfied clauses with literals w and $\bar{w}$. The efficiency of this mechanism is important because SAT instances, in particular industrial instances, usually have lots of clauses.

After the residual formula is generated, the solver should be able to discover possible conflicts and unit clauses. The first indicates unsatisfiability, the second triggers unit propagation. Therefore, an efficient data structure should provide mechanisms to discover these kind of clauses.

In general, it is a known fact that Boolean Constraint Propagation, or BCP (Section 2.1), is an expensive operation in terms of runtime. Experimental results show that on average, up to 90 percent of the DPLL SAT solvers runtime is spent in BCP [81]. Therefore, there have been lots of efforts to design data structures to optimize the BCP. In this section, we provide a brief review of some of the main data structures for SAT solvers. A more thorough review can be found in [76, 104].

One of the traditional methods to represent a SAT formula is the *sparse matrix representation* [59, 81, 92]. Each row in the matrix represents a clause. Each column represents a variable. Every literal w has a list of clauses having literal w. In the sparse matrix representation data structure, one approach to keep track of clause lengths is associating counters

to every clause. A clause counter for a clause C indicates the number of unassigned literals in C. Using the clause counters, the identification of conflict or unit clauses takes $O(1)$. But when a clause is declared unit, finding the unit literal is $O(n)$ where n is the number of variables in the formula. Moreover, the counter values should be adjusted when a variable is being assigned a value going down the tree or is unassigned during backtracking.

One performance issue with the above method comes from the fact that whenever a literal w is assigned to true, all the clauses having $\bar{w}$, satisfied and unsatisfied, are being examined. Although only examining the unsatisfied clauses is necessary. Since the examining of the satisfied clauses is unnecessary, a refinement of this approach hides the satisfied clauses from the list of clauses of literal $\bar{w}$ [26]. This approach prevents the examination of the satisfied clauses, but it requires unhiding the clauses during backtracking. So far, these variations of the sparse matrix representation lacks the efficiency for handling large SAT instances.

A major improvement in the data structures is based on the following observation: Assigning a variable v implies unit or empty clause C only if the clause C is unsatisfied and has length one or two before assigning the variable v. In other words, the clause C has at most two unassigned literals, and all the other literals in C are assigned to false. As a result, in BCP, only clauses that meet this criteria need to be searched. In [76], the data structures that are designed based on this observation are called *lazy data structures*.

There are different methods to keep track of clauses that are of length one or two during the search. In the following we review some of these methods. An important aspect of these methods is their non-counter based approaches.

The first lazy data structure for SAT is *Head/Tail*(H/T) data structure, originally used in SATO [102]. This data structure associates two literals with every clause of length greater than one. The literals are called the *head* and the *tail*. Initially the head points to the first literal and the tail points to the last literal. Every literal w has the list of the clauses with w being the head or tail literal. Whenever a literal w is assigned to true, the list of clauses of $\bar{w}$ is traversed. For every clause C in this list, the literal w is either a head or a tail. The solver searches the clause to find a new unassigned literal other than original head and tail to be the new head or tail. If there is no such literal, and the clause is not satisfied, then

the clause is either unit or conflict based on the value of the head and tail. In this method the head literal should always be positioned before the tail literal. To ensure this property, the necessary changes might be done while backtracking or when assigning a new head or tail.

An improvement to this method is the use of *Watched Literals*, first introduced in Chaff [81]. Like the head/tail method, this method identifies two literals in each clause as watched. The difference with the head/tail method is that there is no ordering defined between the two literals. Because there is no order defined on the watched literals, there are no adjustments in the backtracking.

In the two watched literal scheme, for every clause with length greater than one, two non-zero elements are flagged as watched. Every literal has a list of clauses in which the literal is flagged as watched. When a literal w_1 is assigned to true, the list of watched clauses of $\bar{w}_1$ is traversed. For every clause C in the list, the other watched literal of C, w_2, is examined. There are two cases:

case 1: If w_2 is true, then the clause C is satisfied. Therefore, there is no need to process it.

case 2: Otherwise, the clause is processed to find a non-zero literal to make it watched instead of $\bar{w}_1$. If such a literal w' is found, then the clause C is removed from the list of watched clauses of $\bar{w}_1$ and is added to the list of watched clauses for w'. If it is not found, the clause is flagged as a conflict clause or unit clause based on the value of w_2.

Example 2.6. *Table 2.1 shows these cases. After assigning the variable v_2 to false, in the first clause the unassigned variable v_3 is chosen to be the new watched literal. In the second and fourth clauses, no other non-zero literal other than the other watched literal v_7 is found. In second clause, the fact that variable v_7 is unassigned implies that the clause is a unit clause. In the fourth clause, the variable v_7 having value false implies the current clause is a conflict clause. The third clause is satisfied because the other watched literal v_7 is already evaluated to true, therefore there is no need to update the watched literals.*

$\bar{v}_1$	$v_2{}^*$	v_3	v_6	$v_7{}^*$	v_8
0	0	U	U	U	U

The literal v_3 is the new watched literal.

$\bar{v}_1$	$v_2{}^*$	v_3	v_6	$v_7{}^*$	v_8
0	0	0	0	U	0

A unit clause is being identified.

$\bar{v}_1$	$v_2{}^*$	v_3	x_6	$v_7{}^*$	v_8
0	0	0	1	1	1

No change in the already satisfied clause.

$\bar{v}_1$	$v_2{}^*$	v_3	v_6	$v_7{}^*$	v_8
0	0	0	0	0	0

A conflict clause is found.

Table 2.1: The watched literal scheme after assigning the variable $v_2 = 0$. The second row shows the literal values. The character 'U' indicates that the literal is unassigned.

The watched literals scheme is the most widely used data structure in the current state-of-the-art SAT solvers. Some of the state-of-the-art SAT solvers using this scheme are [14, 32, 41, 81].

Dancing links [52, 63] is a data structure that have been successfully used in solving the exact cover problem [63]. Dancing links data structure also can be used to represent SAT instances. As in a sparse matrix representation, every row represents a clause, and every column represents a literal. All the clauses having the same literal are linked together. In the same way, all the literals in a clause are linked together. Whenever a literal w becomes true, the following operations are performed:

- The clauses having the literal w are delinked from the formula.
- For every clause C that is accessible in the column $\bar{w}$, the literal $\bar{w}$ is being delinked from the clause C.

In order to compare dancing links to watched literal scheme, we have implemented two DPLL SAT solvers that are identical unless for their data structures. Our experimental results suggest that watched literals scheme is more efficient than dancing links for DPLL SAT solving. The main reason is the amount of maintenance that the dancing links require in backtracking.

2.5 Conflict Driven Clause Learning SAT

Chronological backtracking is one of the main drawbacks of the DPLL algorithm because it usually results in unnecessary computations [42]. In order to overcome this weakness, non-chronological backtracking methods have been widely studied for DPLL algorithm. Non-chronological backtracking methods for DPLL method are closely related to the *clause learning* methods. Clause learning attempts to improve the DPLL search algorithm by adding new clauses, which are called *learnt clauses*. These learnt clauses are derived from the conflict assignments to prevent the solver to repeat the same conflict over and over again.

Non-chronological backtracking methods have been studied for SAT and other areas of combinatorial optimization research [34, 42, 79], but it was the work in GRASP [92] that made the non-chronological backtracking and clause learning a crucial part in modern SAT solvers. GRASP inspired a new class of solvers that are called *Conflict Driven Clause Learning*, or *CDCL*. In this section, we review the structure of CDCL SAT solvers.

Algorithm 2.3 represents a generalized algorithm for CDCL SAT solvers [81]. As it can be seen in Algorithm 2.3, there are some new additions to the original DPLL algorithm which includes the *restart* (line 9) and *analyze_conflict* (line 15) functions.

The original DPLL algorithm is a committed search: The choices of the variables make the search process committed to a part of search space. Therefore, a poor choice of variable to branch on in the early stages of the search affects the size of the tree dramatically. Modern algorithms use *restart strategies* [13, 45, 60] to overcome this weakness. For this reason, a *cutoff* value c is chosen that determines the restart point. Whenever the number of backtracks exceeds c (line 8), the current search backtracks to level zero (line 9), deleting the current assignment. The cutoff value is chosen by trial and error. Because the conflicts are added to the formula in terms of learnt clauses, the new search after the restart still has an implicit knowledge of the search history.

The decide_next_branch method (line 11) selects an unassigned variable to branch on based on the chosen branching rule. One of the main advances in CDCL SAT solvers was achieved by introducing *Variable State Independent Decaying Sum*, or *VSIDS*, branching

Algorithm 2.3 A generalized algorithm for CDCL SAT solvers

```
1: status = preprocess()
2: if (status != UNKNOWN) then
3:     return status
4:     counter = 0
5: end if
6: while (true) do
7:     counter = counter + 1
8:     if (counter > MAX) then
9:         restart()
10:    end if
11:    decide_next_branch()
12:    while (true) do
13:        status = deduce()
14:        if (status == CONFLICT) then
15:            blevel = analyze_conflict()
16:            if (blevel < 0) then
17:                return UNSATISFIABLE
18:            else
19:                backtrack(blevel)
20:            end if
21:        else if (status == SATISFIABLE) then
22:            return SATISFIABLE
23:        else
24:            break {continue the search}
25:        end if
26:    end while
27: end while
```

rule by SAT solver ZChaff [81]. We review VSIDS and some other branching rules in Section 2.6.

After the variable to branch on is selected, the formula is simplified as a result of this decision. The function deduce (line 13) performs some reasoning, including BCP, to determine variable assignments which are consequences of the current assignment.

During the simplification, if a conflict clause is discovered, then the current assignment can not lead to a satisfiable assignment. Therefore, the solver needs to backtrack. The method analyze_conflict (line 15) determines the decision level *blevel* to backtrack on. If the decision level *blevel* is less than zero (line 16), then the instance is unsatisfiable. Otherwise, the search backtracks to level *blevel* (line 19). The function $analyze\_conflict$ not only determines the backtrack level, but also adds a new learnt clause to the formula to prevent this conflict assignment in the future. As the search goes on, the number of learnt clauses increases exponentially which, as a result, slows down the search process. Therefore, periodically learnt clauses are removed from the formula. Usually the restart method is responsible to prune the learnt clauses. In Section 2.5.1, we briefly review the analyze conflict method.

2.5.1 Analyzing the Conflict

When a conflict is found, the solver needs to backtrack. In CDCL SAT solvers the backtracking level is determined by the conflict-driven learning methods. These methods usually use an *implication graph* to express the variable implications.

The vertices of the implication graph are the variable assignments. A vertex with label $v = 1@\ell$ shows a true assignment for the variable v at level ℓ. In the same way, a vertex with label $v = 0@\ell$ shows a false assignment for the variable v at level ℓ. Suppose a variable v is an implied variable because of a clause C with value $i \in \{0, 1\}$. This implication adds $|C| - 1$ edges from the variables of C other than v to the vertex labeled with $v = i@\ell$. Every edge is labeled by the clause C.

A *conflict clause* is a clause that all of its literals are assigned to false. An implication graph is in *conflict state* if it has both vertices $v = 0$ and $v = 1$ for a variable v. The variable v is called the *conflict variable*. The conflict variable is the last variable that is assigned to

zero in a conflict clause. Example 2.7 shows an implication graph in the conflict state.

The vertices for the conflict variables are called *conflict vertices*. The information in the connected component containing the conflict vertices is used to generate the learnt clause for the current component. The part of the implication graph that affects the learnt clause is the connected component containing the conflict vertices (Therefore, from this point by implication graph we refer to the part of graph that has the conflict vertices).

Example 2.7. *Figure 2.1 shows an implication graph in the conflict state. The clauses that correspond to this graph are:*

$$C_1 = \{v_2, \bar{v}_1, \bar{v}_4\}$$
$$C_2 = \{\bar{v}_{10}, v_4, v_8\}$$
$$C_3 = \{v_4, \bar{v}_{13}\}$$
$$C_4 = \{v_4, v_{13}, \bar{v}_8, v_6\}$$
$$C_5 = \{\bar{v}_6, \bar{v}_{11}, \bar{v}_3\}$$
$$C_6 = \{\bar{v}_6, v_{15}\}$$
$$C_7 = \{\bar{v}_{15}, v_3, v_5\}$$
$$C_8 = \{\bar{v}_6, \bar{v}_{15}, v_{17}\}$$
$$C_9 = \{\bar{v}_{17}, \bar{v}_7, \bar{v}_5\}$$

Assigning variable $v_1 = 1$ implies the variable $v_4 = 0$ because of the clause C_1. Therefore, two edges from vertices $v_1 = 1@6$ and $v_2 = 0@1$ to the vertex $v_4 = 0@6$ are added to the graph. Assigning this variable implies new variables, therefore, other edges are added to the graph. The graph is a conflict graph because it has both literals on variable v_5.

A learnt clause is generated by bipartitioning the implication graph in two sides: the *reason side* and the *conflict side*. The reason side has all the decision variables. The conflict side has the conflict vertices. Such a bipartition is called a *cut*. A *cut edge* is an edge with endpoints in both sides. The vertices in the reason side that are the endpoints of the cut edges are the *reasons* for the conflict.

Let the set $S = \{v_1, v_2, \cdots, v_i\}$ to be the set of reasons for the current conflict for a chosen cut. Based on the set S, a learnt clause C is generated as follows. For every variable $v \in S$, if v has value true , then $\bar{v}$ is added to the C. If v has value false, then v is added to the C. Different cuts define different set of reasons for a conflict. Therefore, the choice of

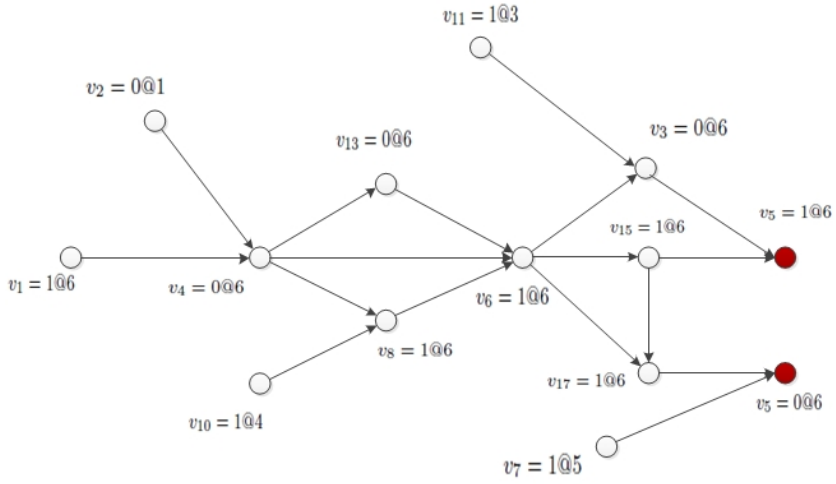


Figure 2.1: An implication graph in conflict state with conflict variable v_5.

the learnt clause to be added to the formula is dependent on the chosen cut. The following example demonstrates three different cuts for implication graph in Figure 2.1, and their corresponding learnt clauses.

Example 2.8. *Figure 2.2 shows three cuts for the implication graph in Figure 2.1. The learnt clause for cut 1 is* $\{v_3, \bar{v}_{15}, \bar{v}_{17}, \bar{v}_7\}$. *The learnt clause* $\{\bar{v}_{11}, v_{13}, v_4, \bar{v}_8, \bar{v}_7\}$ *corresponds to the cut 2. Finally, clause* $\{\bar{v}_{11}, v_2, \bar{v}_1, \bar{v}_{10}, \bar{v}_7\}$ *is the learnt clause identified by the cut 3.*

The effect of adding different learnt clauses have shown to result in different runtime for the same instance [105]. The most common approach to add learnt clauses is based on *Unit Implication Points* or UIPs. Let ℓ_v denotes the decision level of a variable v. Vertex v_1 dominates vertex v_2 in an implication graph if, and only if, any path from the decision variable at level ℓ_{v_2} to the vertex v_2 passes through v_1. A *Unit Implication Point* (UIP) [92] is a vertex at current level that dominates the conflicting vertices. Obviously, the decision variable at current level is always a UIP. In Figure 2.1, vertices v_4 and v_6 are also UIPs. Empirically, it has been shown that the UIP that is nearest to the conflict clause, the *first-UIP*, usually results in better performance [105].

Example 2.9. *In Figure 2.2, the vertex* v_6 *is the first UIP. Therefore, the learnt clause by first UIP approach is* $\{\bar{v}_{11}, v_{13}, v_4, \bar{v}_8, \bar{v}_7\}$.

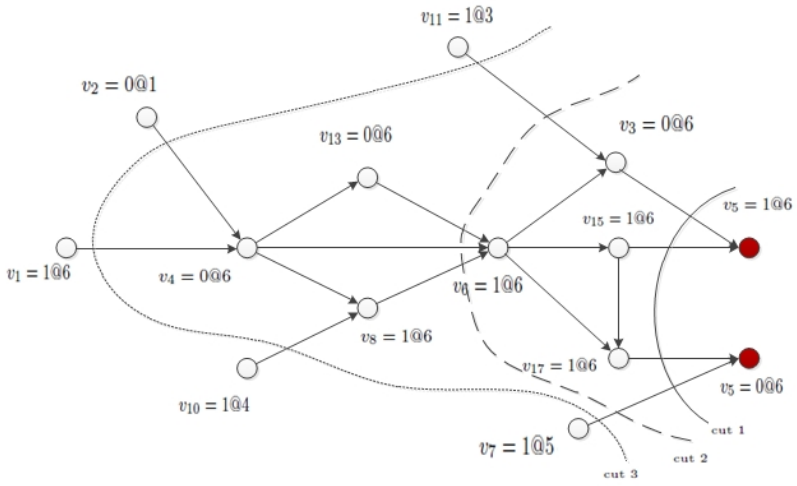


Figure 2.2: Different cuts for the implication graph in Figure 2.1

Learnt clauses are also used to determine the backtrack level. The backtracking level β is set to the second highest level of the variables in S (the first highest level is the current level). If $\beta = \ell - 1$ in which ℓ is the current level, then the method is equivalent to chronological backtracking. Otherwise, the search jumps over several levels in the tree and backtracks non-chronologically.

Example 2.10. *In Figure 2.2, for the first UIP learnt clause, we have $\beta = 5$. Therefore, the search backtracks chronologically. For the learnt clause defined by cut 3, $\{\bar{v}_{11}, v_2, \bar{v}_1, \bar{v}_{10}, \bar{v}_7\}$, we have $\beta = 4$. Therefore, if this learnt clause is added to the formula, the search backtracks non-chronologically from level 6 to level 4.*

2.6 Branching Rules

It is well-known that applying different branching rules may affect the size of a search space significantly [53, 69, 82]. Different branching rules have different performances on different benchmarks. Therefore, several different branching rules have been studied. A review of some of the branching rules and their experimental evaluations can be found in [53, 69, 73, 82, 92]. Generally, there is no known branching rule that outperforms the others on all the instances. In most cases, the choice of branching rules is application dependant.

In [53, 69] a class of branching rules is described which chooses the next variable based

on the number of literal occurrences in the formula. For example, the branching rule *MAXO* selects the branching variable based on the number of occurrences in the formula. It selects the variable with maximum number of occurrences in the formula. The branching rule *MOMS* counts the number of literal occurrences in the minimum size clauses, and chooses the variable with the maximum count. The *Jeroslaw-Wang rule* uses a weighting method to choose variables that are in many short clauses. These classes of branching rules are dependent on the current state of the formula.

The most widely used branching rule in modern CDCL SAT solvers is the *Variable State Independent Decaying Sum (VSIDS)* introduced in Chaff [81]. VSIDS selects literals to satisfy the most recently added clauses. In VSIDS every literal has a weight assigned to it. At the beginning, this weight is set to either zero for all the literals or to the number of literal occurrences in the formula for every literal. Whenever a learnt clause is added to the formula, the weight of the literals in the clause increases. In order to give more weight to recently added learnt clauses, periodically, the weights are divided by a constant. At every iteration, a literal with maximum weight is chosen to branch on. Ties are broken randomly. The method is called *state independent* because the selection of the literals is not related to the current state of the formula. Most of the modern CDCL SAT solvers use variations of VSIDS as their branching rule including [14, 32, 41, 81].

2.7 Chapter Summary

This chapter gives an overview of the DP and DPLL algorithms. It presents the basic terminology that is used throughout the thesis. It reviews the basics for a class of DPLL SAT solvers: Conflict-Driven Clause Learning (or CDCL) solvers.

This thesis investigates the dynamic integration of equivalency reasoning techniques in CDCL solvers. Chapter 3 discusses the complications of equivalency reasoning in a CDCL solver, and it presents a method to overcome these issues. The proposed solution is integrated into an existing CDCL SAT solver.

Chapter 3

Eq-MiniSat: Integrating Equivalency Reasoning in CDCL SAT solvers

One of the widely studied reasoning techniques in DPLL SAT solvers is *binary reasoning*. Binary reasoning, as its name suggests, uses binary clauses to simplify the formula by reducing the number of variables and clauses.

Example 3.1. *Binary clauses* $\{v_i, v_j\}$ *and* $\{\bar{v}_i, v_j\}$ *imply variable assignment* $v_j = 1$.

Example 3.2. *Binary clauses* $\{\bar{v}_i, v_j\}$ *and* $\{v_i, \bar{v}_j\}$ *imply that literals* v_i *and* v_j *are equivalent. Therefore, the formula might be simplified by replacing* v_i *with* v_j *(*$\bar{v}_i$ *with* $\bar{v}_j$*, respectively). The result of the simplification is a formula with fewer variables and possibly shorter clauses.*

So far, the binary reasoning has been mostly applied in the preprocessing steps of CDCL SAT solvers [11, 17, 31, 39, 50]. In Section 3.1, we briefly review the previous work for integrating equivalency reasoning into CDCL SAT solvers.

For look-back SAT solvers, the equivalency reasoning has been only applied in preprocessing steps [14, 17]. The equivalency reasoning in these solvers is based on binary clauses. Because of the complications that arise from integrating the equivalency reasoning in CDCL solvers, to our knowledge, there has been no CDCL solver which has implemented the binary clause based equivalency reasoning during the search.

In this chapter, we discuss the complications that arise from incorporating an equivalency reasoning engine into the CDCL based DPLL procedure. These complications include:

Identifying binary clauses during the search The solver needs to be able to identify these clauses efficiently.

Equivalence literal assignment In a CDCL solver, every implied literal has a reason clause. Therefore, for every literal that is assigned based on literal equivalency a reason clause needs to be identified.

Literal assignment order The learnt clause mechanism in CDCL solvers is dependent on the order of literal assignments. Therefore, to comply with CDCL solver requirements, the literals that are assigned due to equivalency reasoning are required to have an ordering.

This chapter is organized as follows. Section 3.1 reviews the prior work on binary reasoning in general, and equivalency reasoning in particular.

In Section 3.2, we present some definitions which are needed throughout the chapter.

Section 3.3 provides a data structure to identify and use the equivalent literals efficiently without explicitly replacing the equivalent literals. Replacing the equivalent literals results in fewer variables, and shorter clauses but its runtime overhead overweighs these benefits. Every time a conflict is found, the search needs to backtrack. In backtracking, the procedure needs to undo the replacements. Practically, the runtime overhead caused by the replacements outweighs the beneficial effects gained by having fewer variables. Therefore, we decided to avoid substituting the equivalent literals.

One of the important features of CDCL SAT solvers is their conflict engine that discovers the reason for every conflict and adds it to the formula as a learnt clause. As was discussed in Section 2.5, in order to construct the learnt clauses, the solver maintains a graph called the implication graph. In order to successfully integrate equivalency reasoning into a CDCL solver, the implication graph should be maintained correctly. We discuss the complications and solutions to this problem in Section 3.4.

3.1 Related Work

Logical reasoning is used to simplify SAT instances and minimize the search space. The logical rules include, but are not limited to, unit propagation [29], the subsumption rule [75], the unit literal rule [29], blocked clause elimination [66], and binary reasoning [39, 72]. Many current DPLL SAT solvers apply at least one kind of logical reasoning: the unit propagation. The effects of the other rules are application dependant. For example, Ouyang [83] showed that unit literal rule and the subsumption rule are not effective for random formulae. In [56], Biere et al. showed that blocked clause elimination is effective on CNFs resulting from a standard CNF encoding for circuits.

More reasoning usually means fewer number of branchings. Nonetheless, fewer number of branchings does not necessarily imply better performance. The negative runtime performance impact of logical reasonings makes the solvers more cautious in using them. To our knowledge, there have not been any theoretical results on the computational trade-off between reasoning and searching. Empirical methods have been used to determine the effectiveness of reasoning techniques [9, 83].

It is a well-known fact that the problem of evaluating a set of binary clauses, *2-satisfiability*, is in P [7, 33, 65]. Using this fact, many algorithms first solve the 2-SAT clauses in a formula with a polynomial algorithm. The solution to the 2-SAT problem is used to simplify the formula. Then, a general algorithm is called to solve the simplified problem [19, 37, 70].

Let $C_1 = \{v, v_1\}$ and $C_2 = \{\bar{v}, v_2\}$ to be clashing binary clauses. Then, their resolvent C (Section 2.2) is either a unit or a binary clause based on the variables v_1 and v_2. There are two cases:

case 1: If $v_1 = v_2$, then the clause C is a unit clause. Therefore, the formula can be simplified through BCP.

case 2: Otherwise, the clause C is a binary clause which can be added to the formula.

The DPLL solver in [8] applies the resolution operation extensively at every node of the search tree on all the unsatisfied binary clauses. The result has been shown to be effective

for some classes of instances, while ineffective for others.

The *Krom subsumption resolution* [40] uses binary clauses to reduce the length of longer clauses. If clauses $\{\bar{v}, v_1\}$ and $\{v, v_1, \cdots, v_k\}$ are in the formula, then by using resolution and subsumption rules (Section 2.3), the second clause is subsumed to $\{v_1, \cdots, v_k\}$.

Hyper binary resolution [9] applies the resolution rule on a set of clauses $\{v_1, \bar{v}_2\}$, $\{v_1, \bar{v}_3\}, \cdots, \{v_1, \bar{v}_k\}$ and $\{v_2, \cdots, v_k, v_{k+1}\}$, adding the binary clause $\{v_1, v_{k+1}\}$ to the formula [9, 41].

So far, all the mentioned binary reasonings have been based on resolution. Another kind of inference that can be derived from binary clauses is detecting *equivalent literals* using *implication graphs* introduced by Apsvall, Plass, and Tarjan [7]. The implication graph represents the binary clauses in a CNF formula. The implication graph has the property that all the literals in a strongly connected component, or *SCC*, are equivalent.

Many SAT solvers use the implication graph introduced by Tarjan et al. to identify the equivalent literals in the preprocessing steps. After identifying the equivalent literals, the formula is simplified by replacing the equivalent literals. Due to run-time overhead in backtracking, in most SAT solvers, this kind of equivalency reasoning is done only during the preprocessing steps [11, 17, 31, 39, 50].

In this chapter, we examine a heuristic to identify the SCCs of the implication graph during the search for look-back SAT solvers. We proceed to give some basic definitions in the next section.

3.2 Definitions

Identifying the equivalent literals in the formula is one of the main tasks of equivalency reasoning in SAT solvers. One of the main approaches to identify equivalent literals is *Tarjan's strongly connected components* method [96]. We review this method in Section 3.2.1. Section 3.2.2 presents definitions that are needed throughout the chapter. Section 3.2.3 gives the definition of reason sets for strongly connected compnents.

3.2.1 SCC Implication Graphs

Apsvall, Plass, and Tarjan [7] represent binary clauses in a SAT instance by a graph called the *implication graph*. The vertices of the graph are the literals of the formula. For every binary clause $\{v_i, v_j\}$, two edges $\{\bar{v}_i, v_j\}$ and $\{\bar{v}_j, v_i\}$ are added to the graph. The strongly connected components of the implication graph partition the literals into equivalency classes. To avoid confusion with conflict implication graph, we call this graph the *SCC implication graph*. Example 3.3 presents a SAT instance with its SCC implication graph. For the examples in this chapter, we use the formula presented in this example.

Example 3.3. *Figure 3.1 shows a SAT instance on 12 variables and 17 clauses. This SAT instance has 5 binary clauses C_2, C_3, C_4, C_5, and C_{12}.*

$$
\begin{array}{ll}
C_1 = \{\bar{v}_1, v_2, \bar{v}_5\} & C_2 = \{v_1, v_3\} \\
C_3 = \{\bar{v}_1, \bar{v}_2\} & C_4 = \{\bar{v}_6, v_8\} \\
C_5 = \{v_7, \bar{v}_8\} & C_6 = \{v_1, v_2, v_9\} \\
C_7 = \{v_6, \bar{v}_7, v_9\} & C_8 = \{v_2, \bar{v}_3, v_{10}\} \\
C_9 = \{\bar{v}_1, v_2, \bar{v}_{10}\} & C_{10} = \{\bar{v}_2, v_6, v_{10}\} \\
C_{11} = \{\bar{v}_5, v_3, v_6\} & C_{12} = \{\bar{v}_3, \bar{v}_{13}\} \\
C_{13} = \{\bar{v}_1, \bar{v}_2, \bar{v}_9\} & C_{14} = \{v_1, \bar{v}_2, \bar{v}_{10}\} \\
C_{15} = \{v_{10}, \bar{v}_1, v_2, v_6\} & C_{16} = \{\bar{v}_1, v_2, v_{11}\} \\
C_{17} = \{v_3, \bar{v}_7, v_{10}\} &
\end{array}
$$

Figure 3.1: A SAT instance

The SCC implication graph for this instance is shown in Figure 3.2. The binary clauses in this instance add ten edges to the SCC implication graph, two for every binary clause. For example the binary clause C_2=$\{v_1, v_3\}$ adds edges $\{\bar{v}_1, v_3\}$ and $\{\bar{v}_3, v_1\}$ to the SCC implication graph.

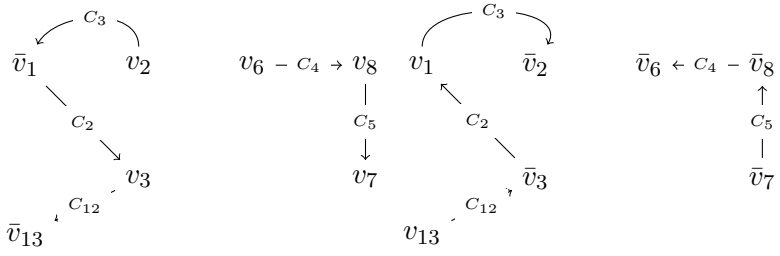


Figure 3.2: SCC implication graph for SAT instance in Figure 3.1

The *transpose graph* of a directed graph is formed by reversing all the edges in the graph.

A graph is *skew-symmetric* if it is isomorphic to its transpose graph. An SCC graph G is a skew-symmetric graph where an edge $\{v_i, v_j\} \in G$ if and only if $\{\bar{v}_j, \bar{v}_i\} \in G$.

Example 3.4. *The SCC implication graph for the formula in Example 3.3, as can be seen in Figure 3.2, is a skew-symmetric graph with the following pairs of edges:*

- $\{\bar{v}_1,\ v_3\}$, *and* $\{\bar{v}_3,\ v_1\}$
- $\{v_2,\ \bar{v}_1\}$, *and* $\{v_1,\ \bar{v}_2\}$
- $\{v_6,\ v_8\}$, *and* $\{\bar{v}_8,\ \bar{v}_6\}$
- $\{v_8,\ v_7\}$, *and* $\{\bar{v}_7,\ \bar{v}_8\}$
- $\{v_3,\ \bar{v}_{13}\}$, *and* $\{v_{13},\ \bar{v}_3\}$

We use the skew-symmetric property of the SCC implication graphs to simplify the SCC graph drawings. If the literals of a variable are in two different connected components of the SCC implication graph, then we draw only one of the components. The other component, can be easily derived from this component by replacing every literal with its negation and reversing all the edges.

Example 3.5. *The simplified graph for the SCC implication graph in Figure 3.2 is shown in Figure 3.3.*

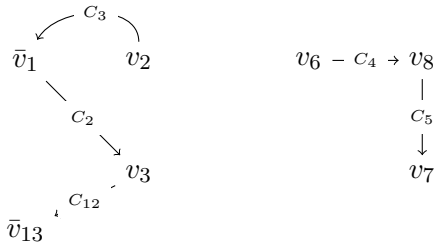


Figure 3.3: Simplified SCC graph for Example 3.3 at level zero

Example 3.6 depicts the evolution of the SCC implication graph is Figure 3.3 as some of the variables are assigned values.

Example 3.6. *This example represents the changes in the SCC implication graph for the instance in Example 3.3 as some of the variables are assigned values.*

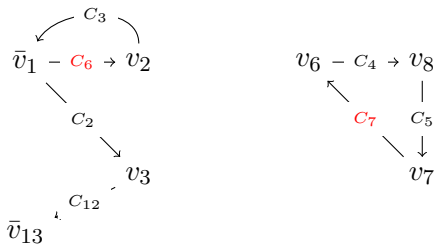


Figure 3.4: SCC graph for the Example 3.3 with $v_9 = 0@1$

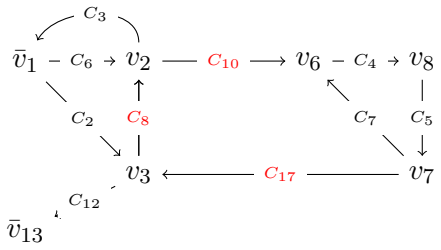


Figure 3.5: SCC graph for the Example 3.3 with $v_9 = 0@1$, $v_{10} = 0@2$

level zero: *The Figure 3.3 represents the SCC implication graph at the end of level zero with all the variables being unknown. As can be seen in the Figure 3.3, all the strongly connected components are of length one. Therefore, all the equivalency classes have length one.*

level one ($v_9 = 0@1$): *Suppose the decision variable at level one is $v_9 = 0$. As a result of assigning $v_9 = 0$, ternary clauses C_6 and C_7 become binary clauses. Therefore, they can be added to the SCC implication graph. The result graph is shown in Figure 3.4. At the end of level one, the non-trivial equivalency classes are $\{v_1, \bar{v}_2\}$, $\{\bar{v}_1, v_2\}$, $\{v_6, v_7, v_8\}$, $\{\bar{v}_6, \bar{v}_7, \bar{v}_8\}$.*

level two ($v_{10} = 0@2$): *Suppose $v_{10} = 0$ is the decision variable at level two. Assigning $v_{10} = 0$ turns ternary clauses C_8, C_{10}, and C_{17} into binary clauses. The updated graph is shown in Figure 3.5. As can be seen in Figure 3.5, The non-trivial connected components at level two are $\{\bar{v}_1, v_2, v_3, v_6, v_7, v_8\}$ and $\{v_1, \bar{v}_2, \bar{v}_3, \bar{v}_6, \bar{v}_7, \bar{v}_8\}$.*

A *conflict assignment* is an assignment for which at least one of the clauses in the formula evaluates to false. If a partial assignment is a conflict assignment, then no satisfying assignment can be found by extending it. If both literals of a variable appear in the same strongly connected component, then the strongly connected component is a *conflict compo-*

nent. An SCC implication graph with a conflict component is in a *conflict state.* A graph in a conflict state might have more than one conflict component. If an SCC implication graph is in a conflict state, then the partial assignment associated with the SCC implication graph is a conflict assignment.

In Example 3.6, we depicted the SCC implication graphs for the instance in Example 3.3 with values $v_9 = 0@1$ and $v_{10} = 0@2$. In the following example, we choose different values for the variables v_9 and v_{10}.

Example 3.7. *For the formula in Figure 3.1, Let* $v_9 = 1@1$ *and* $v_{10} = 1@2$*. The strongly connected component having the literals on* v_1 *and* v_2 *at the end of level two is shown in Figure 3.6. As it can be seen in Figure 3.6, both literals of variables* v_1 *and* v_2 *are in the same strongly connected component. Therefore, the partial assignment* $v_9 = 1$ *and* $v_{10} = 1$ *is a conflict assignment.*

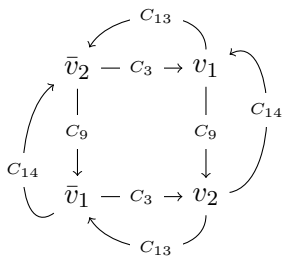


Figure 3.6: SCC implication graph in conflict state

3.2.2 Eq-Unit and Eq-Binary Clauses

Let C be an unsatisfied clause such that some of its unassigned literals are equivalent. Substituting the equivalent literals results in a shorter clause. We define a clause to be *eq-unit* if substituting the equivalent literals results in a unit clause. A clause is called *eq-binary* if substituting the equivalent literals results in a binary clause. In the following, the formal definitions for eq-unit and eq-binary clauses are given.

Let E be an equivalency class of literals. Let C be an unsatisfied clause with all literals evaluated to false except for a subset $S \subseteq E$ for which the values for its literals are unknown. The equivalency of literals in S implies every literal in S has the value true. The clause C is an eq-unit clause. Every unit clause is an eq-unit clause with $|S| = 1$.

In DPLL, every unit clause only results in one implied literal. In order to be consistent with the terminology of DPLL, the literals in S are divided in two categories: *implied* and *eq-implied*. One of the literals in S is chosen to be the implied literal. The other elements of S are called *eq-implied* literals which are implied by *extended implication*.

Example 3.8. *The SCC implication graph of the instance in Figure 3.1 shown in Figure 3.4, shows that the literals $\bar{v}_1$ and v_2 are equivalent. If we assign value true to v_{10}, then the clause $C_9 = \{\bar{v}_1, v_2, \bar{v}_{10}\}$ becomes an eq-unit clause implying $v_1 = 0$ and $v_2 = 1$. The solver chooses one of them to be the implied variable. The other variable is assigned by extended implication.*

For an equivalency class of literals E, the *ground set* is defined to be the set of variables v such that one of the literals on v is in E. Two disjoint equivalency classes are called *complementary* if their ground sets are equal.

Example 3.9. *In Figure 3.5, the ground set for equivalency class $E_1 = \{\bar{v}_1, v_2\}$ is the set of variables $\{v_1, v_2\}$. The ground set for equivalency class $E_2 = \{v_1, \bar{v}_2\}$ is also the set $\{v_1, v_2\}$. Therefore, the sets E_1 and E_2 are complementary.*

Let E_1 and E_2 be two disjoint equivalency classes that are not complementary. Let unsatisfied clause C be a clause with all literals evaluated to false except for the literals in sets $S_1 \subseteq E_1$ and $S_2 \subseteq E_2$. The clause C is called an *eq-binary clause* with respect to E_1 and E_2. Every binary clause is an eq-binary clause with $|S_1| = 1$ and $|S_2| = 1$.

Example 3.10. *In Figure 3.4, the two equivalency clauses are $\{v_1, \bar{v}_2\}$ and $\{v_6, v_7, v_8\}$ for $v_9 = 0@1$. The clause $C = \{v_1, v_6, v_7, v_9\}$ is an eq-binary clause for this assignment.*

Whenever an eq-binary clause C with respect to sets E_1 and E_2 is discovered, the two edges $\{\bar{v}_i, v_j\}$ and $\{\bar{v}_j, v_i\}$ are added to the SCC implication graph in which v_i and v_j are randomly chosen from E_1 and E_2, respectively. The edges $\{\bar{v}_i, v_j\}$ and $\{\bar{v}_j, v_i\}$ are labeled by C. Different choices of literals v_i and v_j result in different SCC implication graphs. From the fact that the vertices in E_1 and E_2 are subsets of strongly connected components, all these graphs have the same strongly connected components.

Example 3.11. *For the clause* $C = \{v_1, v_6, v_7, v_9\}$ *from example 3.10, we add edges* $\{\bar{v}_1, v_6\}$ *and* $\{v_6, \bar{v}_1\}$ *to the SCC implication graph.*

Let G be the SCC implication graph for a formula $\mathcal{F}$. The label for every edge $e = \{v_i, v_j\} \in G$, $label(e)$, has the following properties:

- $label(e) = C$ for a $C \in \mathcal{F}$
- Literals $\bar{v}_i, v_j$ are in $label(e)$.
- For every $w \in label(e)$, either the value of w is false, or the literal w is equivalent to one of the two literals $\{\bar{v}_i,\ v_j\}$.

The edge labels are used to determine the reason for the equivalency of the literals in a strongly connected component. Section 3.2.3 provides the formal definition for the component reasons.

3.2.3 Reason Clauses

Let clause C be an eq-binary clause with respect to sets E_1 and E_2. We define the *reason set* of C, $\mathcal{R}(C)$, to be the set of literals whose assignment to true makes the clause eq-binary. Therefore, we have

$$\mathcal{R}(C) = \{\bar{w} | w \in C - (E_1 \cup E_2)\}.$$

Example 3.12. *For the clause* $C = \{v_1, v_6, v_7, v_9\}$ *from example 3.10, the reason clause is* $\mathcal{R}(C) = \{\bar{v}_9\}$.

Let H be a strongly connected component of an SCC implication graph G. Define $c(H)$ be the set of labels of graph H. Therefore, we have

$$c(H) = \{C | \exists \text{ an edge } e \in H \text{ such that } C = label(e)\}.$$

The *reason* for graph H, $\mathcal{R}(H)$ is defined as

$$\mathcal{R}(H) = \bigcup_{C \in c(H)} \mathcal{R}(C).$$

The reason for a strongly connected component represents the set of literals such that their assignment to true generates the edges of the component.

Example 3.13. *In Figure 3.4, let H_1 and H_2 to be the strongly connected components representing equivalency classes $\{\bar{v}_1, v_2\}$ and $\{v_6, v_7, v_8\}$, respectively. Then we have $\mathcal{R}(H_1) = \{\bar{v}_9\}$, and $\mathcal{R}(H_2) = \{\bar{v}_9\}$.*

If for a strongly connected component H of an SCC implication graph, we have $|\mathcal{R}(H)| = 0$, then the literals in the component are equivalent regardless of the assignment. This information can be used to simplify the formula at preprocessing or restarts.

The following example shows a simple SCC implication graph with a reason set of length zero.

Example 3.14. *Let $C_1 = \{v_1, \bar{v}_2\}$ be a clause in a formula $\mathcal{F}$ with SCC implication graph G. Suppose H is the connected component having literal v_1. Suppose at level $\ell - 1$, for $\ell > 0$, the connected component H has only one edge $\{v_2,\ v_1\}$. If a learnt clause $C_2 = \{\bar{v}_1, v_2\}$ is added to the formula at level ℓ, then the connected component H will become a strongly connected component. The strongly connected component H is shown in Figure 3.7. The strongly connected component H states that literals v_1, v_2 are equivalent with $|\mathcal{R}(H)| = 0$. Therefore, these two literals are equivalent regardless of the chosen assignment.*

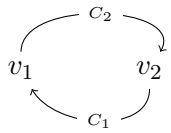


Figure 3.7: SCC component H with $|\mathcal{R}(H)| = 0$

Suppose SCC implication graph G is in a conflict state. Let H to be a conflict component with reason $\mathcal{R}(H)$. This means that the assignment of literals in $\mathcal{R}(H)$ leads to a conflict. Therefore, in order to avoid the conflict, at least one of the variables in $\mathcal{R}(H)$ should be flipped. In other words, the clause

$$C_H = \{w | \bar{w} \in \mathcal{R}(H)\},$$

should be satisfied. The clause C_H is a conflict clause under the current assignment and is called the *SCC conflict clause*. Whenever an SCC conflict clause is found, it is added to the formula. The analyze conflict method is called to determine the backtrack level and the learnt clause for this conflict.

The SCC conflict clauses act in the same way as conflict clauses in the CDCL solvers. The difference is, unlike conflict clauses that are either original clauses or learnt clauses, these clauses are added to the formula because of the SCC implication graph.

If for the SCC implication H in a conflict state we have $|\mathcal{R}(H)| = 0$, then the conflict component implies that the instance is unsatisfiable, therefore the search terminates.

Example 3.15. *For the conflict component H in Figure 3.6, we have*

$$c(H) = \{C_3,\ C_9,\ C_{13}, C_{14}\}.$$

For the clauses in $c(H)$ we have: $\mathcal{R}(C_3)=\emptyset$, $\mathcal{R}(C_9)=\{v_{10}\}$, $\mathcal{R}(C_{13})= \{v_{10}\}$, $\mathcal{R}(C_{14})=\{v_9\}$. Therefore, we have $\mathcal{R}(H)= \{v_9,\ v_{10}\}$. As a result the conflict clause C_H is $C_H = \{\bar{v}_9, \bar{v}_{10}\}$.

Generally, if a clause C has literals that have been assigned at level zero, it can be simplified by removing those literals. We use this fact to simplify the SCC conflict clause C before adding it to the formula. If a literal in C_H has level zero, then we do not add it to the formula. Therefore, we define

$$C_H = \mathcal{R}(H) - \{\bar{w} | w \in \mathcal{R}(H) \text{ and } level(w) \neq 0\}.$$

In the following sections, we discuss the complications that occur after integrating equivalency reasoning into a CDCL solver. In Section 3.3, a slight change in the two watched literal scheme [81] is proposed that enables the solver to identify the eq-unit and eq-binary clauses during BCP. The BCP with eq-unit and eq-binary clauses is called *equivalency propagation*. Section 3.4 discusses the complications that arise in identifying the reasons for the assigned variables.

3.3 Equivalency Propagation

In a CDCL SAT solver, the Boolean unit propagation is implemented by keeping a queue of literals, the *implication queue*. When a unit clause is discovered, the unit literal is added to the queue to process its list of clauses. Discovering the unit clauses is one of the most expensive operations in a SAT solver. There are several methods to identify unit clauses including counter based methods [27], head/tail method [101] and two-watched literal scheme [81]. In most modern CDCL solvers, the list of clauses for each literal is maintained by using the two watched literal scheme. An explanation of this method was given in Section 2.4

In an equivalency reasoning enhanced conflict clause DPLL SAT solver, we want to add the ability to identify eq-unit and eq-binary clauses. In order to achieve this goal, we add a constraint to the watched literals. The watched literals in a clause should belong to different equivalency classes. We call this scheme the *eq-watched literal scheme*.

As in CDCL SAT solvers, every literal w has a list of watched clauses. When a literal w is assigned to true, the solver searches every clause C in the watched list of $\bar{w}$ to find a new literal w' such that

1. the literal w' is either true or unknown,
2. it is not equivalent to either of the current watched literals in the clause C.

The identification of the new watched literal determines whether the clause is a conflict, eq-unit or eq-binary clause. For a clause C, there are four possible cases:

case 1: If such literal w' exists, the literal w' is marked as the new watched literal instead of w. The clause C is removed from the list of watched clauses of $\bar{w}$, and is added to the list of watched clauses of w'.

case 2: Otherwise, if the value of the other watched literal is true, then the clause is satisfied. Therefore there is nothing to do.

case 3: Otherwise, if the value of the other watched literal for clause C is false, then the clause is a conflicting clause.

$\bar{v}_1$	$v_2{}^*$	v_3	v_6	$v_7{}^*$	v_8	
0	0	U	U	U	U	The literal v_3 is the new watched literal.
0	0	0	U	U	U	An eq-unit clause is being identified.
0	0	0	1	1	1	No change in a satisfied clause.
0	0	0	0	0	0	A conflict clause is found.

Table 3.1: The eq-watched literal scheme after assigning $v_2 = 0$. The equivalency classes are $\{\bar{v}_1, v_2\}$ and $\{v_6, v_7, v_8\}$. The second row shows the literal values. The character 'U' indicates that the literal is unassigned.

case 4: Otherwise, the clause is an eq-unit clause.

Example 3.16. *Table 3.1 shows an example for these cases for clause* $C = \{\bar{v}_1, v_2, v_3, v_6, v_7, v_8\}$ *with watched literals* v_2 *and* v_7 *after assigning* v_2 *to false in different scenarios. The equivalency classes are* $\{\bar{v}_1, v_2\}$ *and* $\{v_6, v_7, v_8\}$*. For example, in the first case in Table 3.1, the literal* v_3 *is the new watched literal. In the second case, all the unassigned literals are equivalent to the other watched literal* v_7*. The variable* v_7 *is unassigned, therefore the clause is identified as eq-unit clause.*

These four cases identify conflict and eq-unit clauses. For eq-binary clauses the solver only needs to search among the clauses in the first case. After discovering the new watched literal, the search continues to find a literal w_i such that it is not equivalent to either of the current watched literals. If no such literal exists, the clause is being flagged as an eq-binary clause.

Example 3.17. *For example, in the first case in Table 3.1, the literal* v_3 *is the new watched literal. The search to find another literal that is not equivalent to* v_3 *and* v_7 *is unsuccessful. Therefore, the clause is identified as an eq-binary clause.*

It can be noted that not all the eq-binary clauses can be discovered with this method. For example, suppose we have a clause $C = \{v_1, v_2, v_3\}$ with v_1 and v_2 as watched literals. If the literal v_3 is set to false, then the clause C becomes an eq-binary clause. But because

v_3 is not watched in this clause, the clause is not being processed by the solver during unit propagation. Therefore, this type of eq-binary clauses are not identified during BCP. Due to efficiency concerns, we have limited the search for eq-binary clauses to those that are found during BCP.

3.4 Identifying the Reasons

The unit propagation in a DPLL algorithm divides the variable assignments into two categories: decision and implied. In CDCL SAT solvers, each implied variable has a corresponding *antecedent* or *reason* clause. The antecedent of a variable v is a unit clause that causes v to have its implied value.

Example 3.18. *Let $v_1 = 0$, $v_3 = 1$ and $v_5 = 0$. Then, clause $C = \{v_1, \bar{v}_3, v_4, v_5\}$ is a reason for variable v_4 have value true. The variable v_4 is an implied variable with antecedent clause C.*

The variables are ordered based on the time they have been added to the implication queue. Let $pos(v_i)$ to be the time stamp of v_i in the implication queue. If the value for variable v_i is unknown, i.e. it is not in the implication queue, then $pos(v_i)$ is set to infinity. If the assignment of the variable v_i implies the assignment of the variable v_j, then we have $pos(v_i) < pos(v_j)$.

In a CDCL SAT solver, the reason clause for a variable v_i, $r(v_i)$, has the following properties:

1. The value for all the literals in $r(v_i)$, except for the literal on v_i, is false.
2. For all $v_j \in C$ such that $v_j \neq v_i$, we have $pos(v_j) < pos(v_i)$.

We call these properties the *reason clause properties*. The first condition in reason clause properties implies the variable v_i to have its value. The second condition makes the implication graph acyclic. The correctness of the clause learning in CDCL solvers is dependant on the implication graph being acyclic.

The following example shows that in an equivalency reasoning enhanced SAT solver, the eq-unit clauses might not satisfy the reason clause properties.

Example 3.19. *For the SAT instance shown in Figure 3.1, let* $v_9 = 0@1$ *and* $v_{10} = 0@2$. *The corresponding SCC implication graphs are shown in Figure 3.4 and 3.5. At level two, the clause* $C_{16} = \{\bar{v}_1, v_2, v_{10}\}$ *implies* $v_1 = 0$ *because* $v_{10} = 0$ *and* $\bar{v}_1$ *is equivalent to* v_2. *The clause* C_{16} *does not satisfy the conditions for a reason clause for variable* v_1 *because:*

- *variable* v_2 *is unknown,*
- *and for variable* v_2 *we have:* $pos(v_2) > pos(v_1)$.

In order to comply with reason clause properties, our goal is to replace the eq-unit clause C with a clause C' such that the clause C' implies $\bar{v}_1$ and satisfies the reason clause conditions.

In an equivalency enhanced CDCL SAT solver, we have two kinds of implied literals. The ones that are implied in BCP by eq-unit clauses and the eq-implied literals that are implied by equivalency reasoning. In Section 3.4.1, we propose a method to identify valid reason clauses for implied literals by eq-unit clauses. Section 3.4.2 discusses the complications in identifying the reason clauses for eq-implied literals.

3.4.1 Reason Clause Adjustments for Implied Literals

For an implied variable v_i, let clause C be its reason, i.e. $C = r(v_i)$. Let w_i to be the literal on v_i in the clause C. We partition the literals in the reason clause C in two sets. The set E_{C,w_i} has the literals equivalent to the literal w_i in C. The set N_{C,w_i} has the negation of literals that are not equivalent to w_i in the clause C. The assignment of the literals in N_{C,w_i} to true makes the clause C an eq-unit clause. Therefore,

$$E_{C,w_i} = \{w_j | w_j \in C : w_j \text{ is equivalent to } w_i \text{ or } w_j = w_i\},$$
$$N_{C,w_i} = \{\bar{w}_j | w_j \in (C - E_{C,w_i})\}.$$

If the equivalent literals w_i and w_j are in a clause C, then for this clause C we have

$$E_{C,w_i} = E_{C,w_j} \text{ and } N_{C,w_i} = N_{C,w_j}.$$

Example 3.20. *In Example 3.6, at level two, instead of assigning* $v_{10} = 0$*, we set* $v_{10} = 1$*. This assignment makes the clause* C_9=$\{\bar{v}_1, v_2, \bar{v}_{10}\}$ *an eq-unit clause. We choose* v_1 *to be the implied variable, making* v_2 *the eq-implied variable. Therefore, we have* $E_{C_9,\bar{v}_1} = \{\bar{v}_1, v_2\}$*, and* $N_{C_9,\bar{v}_1} = \{v_{10}\}$*.*

If $|E_{C,w_i}| = 1$ for a clause C, then the clause C satisfies both properties for a reason clause for the literal w_i. Otherwise, as it was shown in Example 3.19 none of the conditions are satisfied.

Suppose for a clause C we have $|E_{C,w_i}| \neq 1$ for a variable v_i. Suppose the literal w_i is the literal associated with variable v_i in clause C, and H is the strongly connected component containing the literal w_i in the SCC implication graph. The set $\mathcal{R}(H)$, as defined in Section 3.2, is the reason for the literals in H to be equivalent. As a result, it is the reason for the literals in E_{C,w_i} to be equivalent. Therefore,

$$\mathcal{R}(H) \Rightarrow \text{ the literals in } E_{C,w_i} \text{ are equivalent.}$$

We can rewrite the above as

$$(\mathcal{R}(H) \text{ AND } N_{C,w_i}) \Rightarrow (\text{ the literals in } E_{C,w_i} \text{ are equivalent AND } N_{C,w_i}).$$

From the fact that the clause C is an eq-unit clause implying the literal w_i we have

$$(\text{ the literals in } E_{C,w_i} \text{ are equivalent AND } N_{C,w_i}) \Rightarrow w_i.$$

As a result

$$(\mathcal{R}(H) \text{ AND } N_{C,w_i}) \Rightarrow w_i,$$

which is equivalent to a disjunctive clause

$$C' = w_i \cup \{\bar{w_j} | w_j \in (\mathcal{R}(H) \text{ OR } N_{C,v_i})\}.$$

Lemma 3.1. *The clause C' satisfies the reason clause properties for an unassigned literal w_i.*

Proof. From the definition of $\mathcal{R}(H)$ and N_{C,w_i}, the value for all the literals in the clause C', except for v_i, is equal to false. The literal w_i is unassigned therefore the second condition is true. As a result this clause satisfies the reason clause properties. □

The clause C' satisfies the conditions for a reason clause. Therefore, for every eq-unit clause C which is a reason for literal w_i with $|E_{C,w_i}| > 1$, we set the reason clause for literal w_i to the clause constructed as discussed. The clause C' is called *adjusted reason clause*. The process of generating an adjusted reason clause for a variable v_i is shown in Algorithm 3.1. The input to the algorithm is a clause C and a literal $w \in C$ such that C is an eq-unit clause for the literal w.

Algorithm 3.1 Adjusting Reason Clauses for Implied Literals

Input: clause C and literal $w \in C$ such that C is an eq-unit clause for w
Output: clause C' which complies with reason clause conditions for literal w

```
1: clause C' = ∅
2: add w to C'
3: for (literals w_i in C) do
4:     if (w_i is not equivalent to w) then
5:         add w_i to C'
6:     end if
7: end for
8: H = strongly connected component containing w
9: for (all the edge labels C_i in H) do
10:    for (all the literals w_j in C_i) do
11:        if (w_j is not equivalent to w) then
12:            add w_j to C'
13:        end if
14:    end for
15: end for
16: v = variable(w)
17: r(v) = C'
```

Example 3.21. *In Example 3.20, let H be the SCC containing $\bar{v}_1$. We have $\mathcal{R}(H)$=$\{\bar{v}_9\}$, and $N_{C_9,\bar{v}_1}$=$\{v_{10}\}$. Therefore, adjusted reason clause for v_1 is C'=$\{v_9$,$\bar{v}_{10}$, $\bar{v}_1\}$.*

3.4.2 Reason Clause Adjustments for Eq-Implied Literals

As was discussed in Section 3.2, other than decision and implied variables, we have another category of variable assignments: eq-implied. Another complication arises in identifying reason clauses for eq-implied literals. The eq-implied literals are implied because of the equivalency reasoning. The eq-implied variables are assigned through extended implication scheme. Whenever a literal w_i is assigned a value, all the literals w_j that are equivalent to w_i are being assigned as well. For these assignments the solver needs to identify reason clauses. The reason clauses for the eq-implied assignments are implicitly stated in the edge labels of the strongly connected components. The solver needs to extract this information from the SCC implication graph. A depth first search on the SCC implication graph can be used to identify reason clauses for the eq-implied variables.

In a directed graph G, a vertex v_j is called a *direct successor* of a vertex v_i if there is an edge $\{v_i, v_j\} \in G$.

Let H be a strongly connected component in an SCC implication graph. Suppose there exists a literal w in H with value true. Define

$$S_w = \{w_1, \cdots, w_t\},$$

to be the set of direct successors of w in H. After assigning the literal w to true, we want to identify reason clauses for literals in S_w.

Let w_i be a literal in S_w. For the edge $e_i = \{w, w_i\}$, let clause $C_i = label(e_i)$. As in section 3.4.1, we partition the clause C_i into two sets E_{C_i,w_i} and N_{C_i,w_i} based on the successor vertex w_i. From the definition of the edge labels in an SCC implication graph, all the literals in $C_i - E_{C_i,w_i}$ are false. Therefore, the clause C_i is an eq-unit clause for literal w_i. There are two cases:

case 1: If $|E_{C_i,w_i}| = 1$, then the clause C_i is an antecedent clause for the literal w_i.

case 2: Otherwise, we use the method discussed in Section 3.4.1 to generate an adjusted reason clause for the literal w_i.

The same method can be used on the direct successors of w_i to identify their reason clauses.

Therefore, a DFS on the component H starting from vertex w generates antecedents for all the literals in H except for the literal w.

Let w be a literal with value true. To find reason clauses for the literals that are equivalent to w, we do a DFS on the strongly connected component containing literal w. The extended implication procedure is shown in Algorithm 3.2. The input is an SCC implication graph G on $2n$ vertices, and a literal w with value true.

Algorithm 3.2 Identifying Reason Clauses for Eq-Implied Literals

Input: SCC implication graph G on $2n$ vertices
Input: literal w with value true
Output: all the literals equivalent to w are set to true

```
1: H = strongly connected component containing w
2: stack S
3: array seen[2n]
4: initialize array seen to false
5: if (SCC implication graph G in conflict state) then
6:     return
7: end if
8: S.push(w)
9: seen[w] = true
10: while (S is not empty) do
11:     current = S.top
12:     for (all direct successors w_i of current in H) do
13:         if (seen[w_i]) then
14:             continue
15:         end if
16:         seen[w_i] = true
17:         S.push(w_i)
18:         set the value of literal w_i to true
19:         v = variable(w_i)
20:         e = label({current,w_i})
21:         r(v) = adjust(e)
22:     end for
23: end while
```

Example 3.22. *In Figure 3.5, the SCC implication graph is shown for formula 3.1 with* $v_9 = 0@1$ *and* $v_{10} = 0@2$. *Suppose literal* v_1 *is set to true. A DFS on the strongly connected component results in the following literals assignments:* $v_2 = 1$, $v_6 = 1$, $v_8 = 1$, $v_7 = 1$ *and* $v_3 = 1$. *The reason for these clauses are* C_6, C_{10}, C_4, C_5, *and* C_{17}, *respectively. All the reason clauses comply with the reason clause rules, therefore no further adjustments are required.*

3.5 SCC Implication Graph Reasoning

Other than eq-implied literals, there is other information that can be retrieved from the SCC implication graph. In this section, we review two reasoning methods based on the SCC implication graph.

3.5.1 Extended Implication Procedure

A vertex v_j is reachable from a strongly connected component H, if there exists a path from v_i to v_j for a vertex $v_i \in H$.

Let w be a literal that is set to true. Suppose H is the strongly connected component having w. Therefore, the extended implication procedure assigns values to all the literals in H. Using the same argument as for equivalent literals, it is easy to see that all the reachable literals v_j from H are implied too. Therefore, we can change the extended implication Algorithm 3.2 to include these implications as well. The procedure is shown in Algorithm 3.3.

In the general extended implication procedure, instead of only exploring the neighbors in the strongly connected component, all the reachable neighbors of a literal are explored. In terms of Algorithm 3.2 and Algorithm 3.3 the difference is in the while loop that explores the direct successors. The first algorithm only looks at the direct successors in H, while the second algorithm explores all the direct successors including those in G.

Example 3.23. *In example 3.22, using the general extended implication procedure also sets the literal $\bar{v}_{13}$ to true with reason clause C_{12}.*

3.5.2 Discovering the Conflicts

The SCC implication graph can be used to discover that a partial assignment is a conflict assignment without propagation. A strongly connected component H is called *conflict-aware* if both literals of a variable are reachable from H. For example, a conflict component is a conflict-aware SCC.

A slight change in Algorithm 3.3 enables the solver to determine whether an SCC is conflict-aware. We add the following change to the while loop in Algorithm 3.3:

Algorithm 3.3 General Extended Implication Procedure

Input: SCC implication graph G on $2n$ vertices
Input: literal w with value true
Output: all the literals equivalent to w are set to true

```
1: stack S
2: array seen[2n]
3: initialize array seen to false
4: if (SCC implication graph G in conflict state) then
5:     return
6: end if
7: S.push(w)
8: seen[w] = true
9: while (S is not empty) do
10:    current = S.top
11:    for (all direct successors w_i of current in G) do
12:        if (seen[w_i]) then
13:            continue
14:        end if
15:        seen[w_i] = true
16:        S.push(w_i)
17:        set the value of literal w_i to true
18:        v = variable(w_i)
19:        e = label({current,w_i})
20:        r(v) = adjust(e)
21:    end for
22: end while
```

if ($seen[w_i]$ AND $seen[\bar{w}_i]$) **then**

let G_{w_i} be the connected component having w_i

add $R(G_{w_i})$ as the conflict clause

mark the current assignment as a conflict assignment

return

end if

After the conflict clause is added to the formula, the analyze conflict method is called to determine the backtrack level and generate the learnt clause.

In the next section, we summarize the chapter by presenting the equivalency enhanced CDCL algorithm.

3.6 Equivalency Enhanced CDCL Procedure

Our equivalency enhanced CDCL algorithm is shown in Algorithm 3.4. The differences with Algorithm 2.3 are explained in this section.

preprocess (line 1) The equivalent literals are identified and replaced. The preprocess method also sets the restart_flag to zero.

restart (line 7) The equivalent literals are identified and replaced, and sets the restart_flag to zero.

deduce (line 11) The function deduce is responsible to update the SCC implication graph. Whenever an eq-binary clause is discovered during BCP, the corresponding edges are added to the SCC implication graph. Also, whenever a literal w is set to true, the extended implication Algorithm 3.2 is called to set all the literals that are equivalent to w to true.

update_scc (line 13) The function update_scc is shown in Algorithm 3.5. The function update_scc in Algorithm 3.5 updates the strongly connected components of the SCC implication graph. If the level returned by the update_scc is equal to zero, then there are literals that are equivalent regardless of the assignment. Therefore, the restart flag is set to one so

Algorithm 3.4 Equivalency Enhanced DPLL CDCL Algorithm

```
1: status = preprocess()
2: if (status ! = UNKNOWN) then
3:     return status
4: end if
5: while (true) do
6:     if (fulfilling the criteria for the restart strategy or restart_flag is 1) then
7:         restart()
8:     end if
9:     decide_next_branch()
10:    while (true) do
11:        status = deduce()
12:        current_level = blevel
13:        blevel = update_SCC()
14:        if (blevel is 0) then
15:            restart_flag = 1
16:            break
17:        else if (blevel is equal to the current_level) then
18:            if (status is CONFLICT) then
19:                blevel = analyze_conflict()
20:            end if
21:        end if
22:        if (blevel < 0) then
23:            return UNSATISFIABLE
24:        else
25:            backtrack(blevel)
26:        else if  (status is SATISFIABLE) then
27:            return SATISFIABLE
28:        else
29:            break {continue the search}
30:        end if
31:    end while
32: end while
```

that the solver substitutes these equivalent literals. If the level is equal to the current level, and the status is set to 'conflict', it means the SCC implication graph is in a conflict state.

If the SCC implication graph is found to be in a conflict state, the reason for the conflict is added to the formula (as described in Section 3.2). Then, the analyze_conflict function is called to find the backtrack level.

Otherwise, if for a strongly connected component H with $|\mathcal{R}(H)| = 0$, then the literals in H are equivalent regardless of the assignment. Therefore, to take advantage of this information, the search backtracks to level zero. It calls the preprocess function to replace the equivalent literals.

Algorithm 3.5 Updating Strongly Connected Components

```
1: run Tarjan's algorithm to update SCCs
2: if SCC implication graph is in conflict state then
3:     Define H to be the conflict strongly connected component
4:     conflict clause = R(H)
5:     blevel = analyze_conflict
6:     return blevel
7: else if (there exists an SCC H with |R(H)| = 0) then
8:     backtrack(0)
9:     preprocess()
10:    return 0
11: else
12:    return current_level
13: end if
```

backtrack (line 25) In backtracking to level ℓ, the SCC implication graph is updated by removing all the edges that have level greater than ℓ. The set union-deunion structure is used to update the strongly connected components so that they reflect the edge removals.

Figure 3.8 shows a flowchart of the algorithm. The gray parts are added due to equivalency reasoning.

3.7 Chapter Summary

In this chapter, we discussed a method to integrate equivalency reasoning into CDCL SAT solvers. The complications that arise from the integration were discussed. A method to overcome these complications was provided.

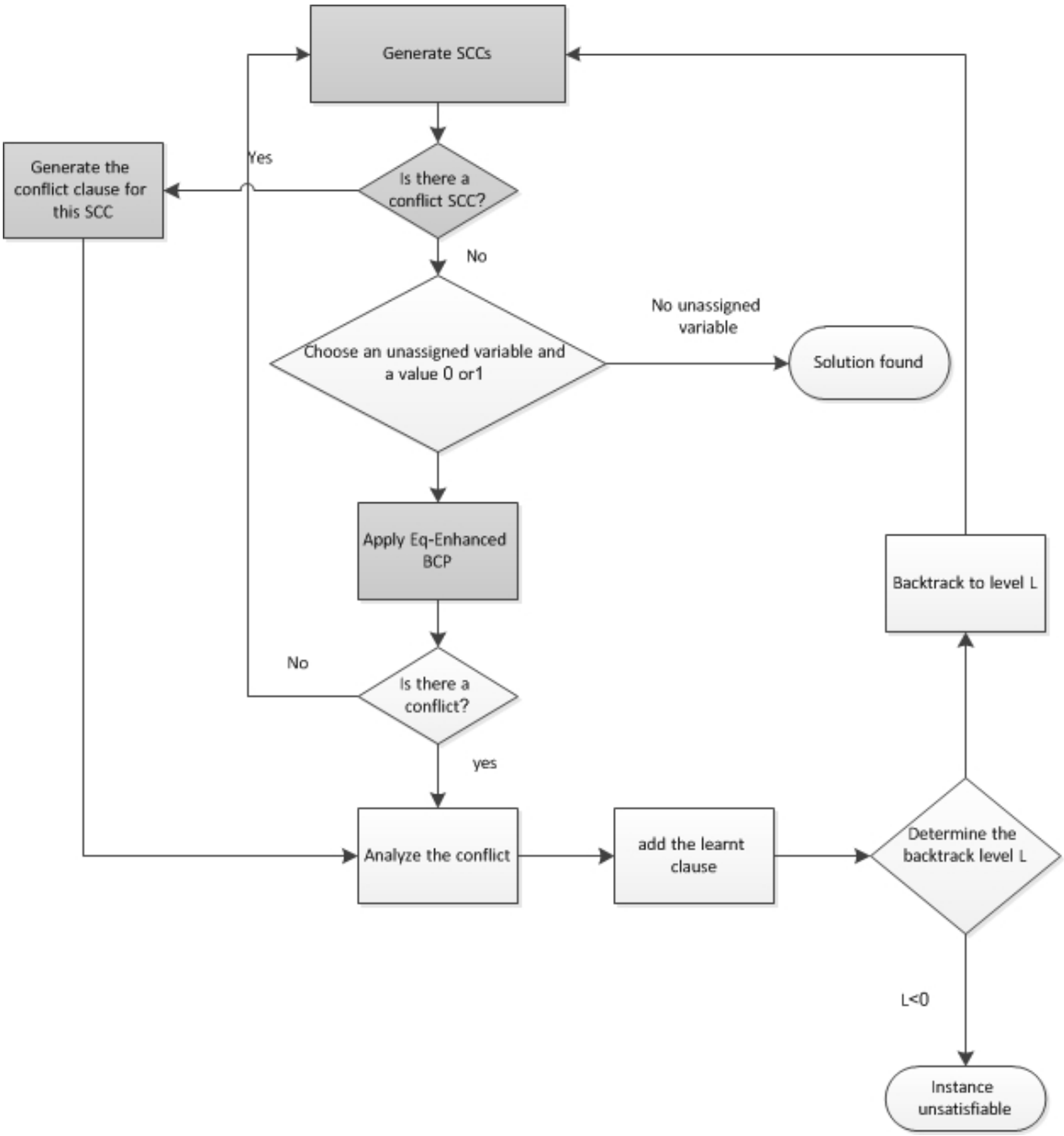


Figure 3.8: A flowchart for an equivalency enhanced CDCL SAT solver. The gray parts are related to the equivalency reasoning logic.

For experimental results, we have incorporated our equivalency reasoning engine into an existing state-of-the-art CDCL based SAT solver MiniSat [32]. Because of its easily modifiable code, the SAT solver MiniSat is traditionally used to analyze innovative ideas which are to be integrated into CDCL based SAT solvers (for example in SAT competitions 2009 and 2011, there was a special track called *MiniSAT* hack track).

The equivalency enhanced MiniSat is called Eq-MiniSat. Chapter 4 gives an overview of implementation details of Eq-MiniSat. Chapter 5 provides performance statistics to demonstrate the possible effectiveness of the equivalency reasoning for CDCL solvers.

Chapter 4

Eq-MiniSat: Implementation Details

In order to evaluate the effectiveness of the proposed method in Chapter 3 to integrate equivalency reasoning into CDCL SAT solvers, we have incorporated this method into an existing state-of-the-art SAT solver: MiniSat [32]. Because of its easily modifiable code, the SAT solver MiniSat is traditionally used to analyze innovative ideas which are to be integrated into CDCL based SAT solvers[1]. Our equivalency enhanced CDCL solver is called Eq-MiniSat.

A key point in equivalency reasoning integration in a SAT solver is implementation efficiency. In this Chapter, we present some implementation details of the solver Eq-MiniSat.

One of the major operations in Eq-MiniSat is calculating strongly connected components, or SCCs, at every node of the search tree. In Section 4.1, we provide the details of the method used in Eq-Minsat to calculate SCCs. The well-known Tarjan's algorithm [96] was our first choice to calculate the SCCs. Section 4.1.1 presents this algorithm. Eq-MiniSat uses the data structure union-deunion [77] to store the SCCs. We review this data structure at Section 4.1.2. The SCC implication graphs in the search have special characteristics. In Section 4.1.3, we discuss these characteristics. In Section 4.1.4, we propose a variation of Tarjan's algorithm, the *LIFO Tarjan algorithm*, that generates the strongly connected components using the special structure of SCC implication graphs.

Section 4.2 discusses the strongly connected components maintenance in backtracking.

In Section 4.3, we review some of the implementation details of the LIFO Tarjan algo-

[1]For example in SAT competitions 2009 and 2011, there was a special track called *MiniSAT* hack track.

rithm. Section 4.3.1 reviews the idea of timestamps in the initialization step of Tarjan's algorithm. Sections 4.3.2 and 4.3.3 discuss two methods to reduce the number of visited vertices in every call of the LIFO Tarjan algorithm.

Another runtime impact of the SCC based equivalency reasoning is due to SCC implication graph maintenance. In Section 4.4, we present a simple data structure that is efficient in maintaining the SCC implication graph in terms of runtime.

Section 4.5 presents a small modification to the VSIDS branching rule based on the SCC implication graph information.

In the following section, we introduce a variation of Tarjan's algorithm to calculate strongly connected components during the search.

4.1 LIFO Tarjan Algorithm

In this section, we present the LIFO Tarjan algorithm. Section 4.1.1 gives a short overview of the original Tarjan's algorithm. Section 4.1.2 presents the data structure that is used to store the strongly connected components in the LIFO Tarjan algorithm.

The dynamic of the changes in SCC implication graphs during the DPLL search has the Last-In-First-Out characteristic in edge addition and removal operations. Section 4.1.3 reviews this property of the SCC implication graphs. This characteristic is used in designing the LIFO Tarjan algorithm specially in the deunion of the strongly connected components.

Section 4.1.4 reviews the LIFO Tarjan algorithm to generate the SCCs during the search.

4.1.1 Original Tarjan's Strongly Connected Component Algorithm

Tarjan's algorithm [96] is one of the well-known algorithms used to find strongly connected components in a digraph. The algorithm is based on a depth first search on the graph. The vertices are positioned in a stack in the order that they have been visited. The strongly connected components are the subtrees of the search tree. The algorithm determines a special vertex *root* for every subtree. The root of a subtree is the first vertex of the subtree encountered in the depth first search. The algorithm finds the strongly connected components by determining the roots of the strongly connected components. When a vertex v is

identified as a root, all the vertices in the stack that are on the top of v form a strongly connected component.

The root is identified by using a special counter for each vertex: *index*. The index is incremented for vertices in the order that they are discovered. Also, every vertex has a value *lowlink* which is smaller than or equal to its index. Let $l(v)$ denote the minimum index of the nodes reachable from v. The lowlink for a vertex v is equal to the minimum of the index of v and $l(v)$. A vertex v is identified as the root of a strongly connected component if $v.lowlink$ is equal to $v.index$. The original Tarjan's algorithm is shown in Algorithm 4.1 and Procedure 1.

Algorithm 4.1 Original Tarjan's Strongly Connected Components Algorithm

Input: graph $G = (V, E)$
Output: set of strongly connected components

```
1: index = 0
2: stack S = empty
3: for all v ∈ V do
4:     if (v.index is undefined) then
5:         strongconnect(v)
6:     end if
7: end for
```

Procedure 1 Procedure *strongconnect*(v)

```
1: v.index = index
2: v.lowlink = index
3: index = index + 1
4: S.push(v)
5: if (v.index is undefined) then
6:     for all ((v, w) ∈ E) do
7:         strongconnect(w)
8:         v.lowlink = min(v.lowlink, w.lowlink)
9:     end for
10: else if (w ∈ S) then
11:     v.lowlink = min(v.lowlink, w.index)
12: end if
13: if (v.lowlink == v.index) then
14:     start a new strongly component
15:     repeat
16:         w = S.pop()
17:         add w to the current strongly connected component
18:     until (v == w)
19: end if
```

For a graph $G(V, E)$ the Tarjan's algorithm runtime is $O(|V| + |E|)$.

4.1.2 Set Union-Deunion Data Structure

The classical set union data structure [25] maintains a collection of disjoint sets. The set union data structure is usually used to represent the strongly connected components of a digraph. In this data structure, every set has a special element which is called the *representative*. The union data structure supports the following operations:

MakeSet(x): Creates a singleton set $\{x\}$ with representative x. The element x should not be in any other set.

FindSet(x): Returns the set containing element x.

UnionSet(x,y): Replaces the sets containing x and y with the union of the two sets. It updates the set of representatives to reflect this change.

The classical set union problem does not support the deunion of the sets. In [77], Mannila and Ukkonen proposed a variant of the set union problem that supports backtracking over the union operations. They added the following operation to the problem:

Deunion(): Cancels the last union operation that already has not been cancelled. This function partitions the result of the UnionSet(x,y) to the original sets containing x and y before the union. It updates the representatives to reflect this change.

In Eq-MiniSat, we use this data structure to represent the strongly connected components of the SCC implication graph. This data structure is particularly useful in updating the SCCs in backtracking which is discussed in Section 4.2.

The set union-deunion data structure in Eq-MiniSat uses adjacency lists to represent sets and an additional stack to keep track of the unions.

For every element x, a variable rep is used to keep the representative for this variable. If for an element x we have $rep = x$, then the element x is a representative.

For every element x, an array A_x lists the elements that are in the set with the representative x. We call a set A_x an *alive* set if the variable x is a representative. The disjoint sets are formed by the union of the alive sets.

For an adjacency list A_x, we set the element $A_x[0]$ to be the number of elements that are in the set of A. For a variable y, we have $y \in A_x$ if for some $1 \leq i \leq A_x[0]$, we have $A_x[i] = y$.

The function $FindSet(x)$ returns the variable rep for the variable x. Therefore, the runtime complexity for $FindSet(x)$ is $O(1)$.

The function $MakeSet(x)$ sets the values of $A_x[0]$ and $A_x[1]$ to one and x, respectively. It also sets the variable rep for the variable x equal to x. Therefore, the runtime complexity for $MakeSet(x)$ is $O(1)$.

For the $UnionSet(x, y)$, let $x.rep$ and $y.rep$ to be the representatives for x and y, respectively. We assume $A_{x.rep}[0] > A_{y.rep}[0]$. The union function applies the following changes:

1. For every element in $z \in A_{y.rep}$ set $z.rep = x$.
2. Set $A_{x.rep}[0] = A_{x.rep}[0] + A_{y.rep}[0]$.
3. Add the elements in $A_{y.rep}$ to the adjacency list of the element x.

The runtime complexity for $UnionSet(x, y)$ is $O(n)$ in the worst case in which n is the total number of elements in the set union-deunion data structure.

The $Deunion(x, y)$, reverses all the above operations:

1. For every element in $z \in A_{y.rep}$ set $z.rep = y$.
2. Set $A_{x.rep}[0] = A_{x.rep}[0] - A_{y.rep}[0]$.

The runtime complexity for $UnionSet(x, y)$ is $O(n)$ in the worst case.

The runtime complexity for m find operations, and k deunions on a set union-deunion with n elements in the above data structure is $O(m + kn)$. In [77], using a different data structure, the complexity is $O((m + k) \log \log n)$. In experimental results we have observed that the set union-deunion operations take less than .5 percent of the total runtime. Due to its relatively small runtime impact, we have not changed the set union-deunion data structure from adjacency lists to the ones with better theoretical runtime complexity. In the future work, the set union-deunion data structure can be replaced by using other data structure such as *path compressions* [36] instead of adjacency lists.

4.1.3 LIFO-Dynamic Graphs

In an SCC implication graph, the edges are added to the graph going down the search tree, and are removed from the graph in backtracking. The addition and removal of the edges is always in Last-In-First-Out order. We call such graphs *LIFO-dynamic* graphs. In this Section, we review the properties of these graphs.

Let e be an edge in a LIFO-dynamic digraph G. We define the *level* of the edge e, $\ell(e)$, to be the level when the edge e added to G. At every level, more than one edge can be added to G.

Let μ be the maximum level for the edges in a graph G. If an edge e is added to G, then its level is at least μ. In other words, the edges are ordered based on the time they have been added to the graph.

The edges of a LIFO-dynamic graph can be partitioned based on their levels. Let G_i to be the spanning subgraph of G with edges at level i. Then, we have

$$G = G_0 \cup G_1 \cup G_2 \cup \cdots \cup G_\mu.$$

We say G is a LIFO-dynamic graph at level μ.

The remove function for a LIFO-dynamic graph has the property that it removes the edges in the reverse order of the edge additions. Moreover, the remove function deletes all the edges in G_μ, in which μ is the maximum level.

Example 4.1. *An SCC implication graph is a LIFO-dynamic graph. The edge levels are the DPLL tree levels. For example in Figure 3.5, we have* $\ell(\{v_2, \bar{v}_1\}) = 0$, $\ell(\{v_7, v_6\}) = 1$, $\ell(\{v_7, v_3\}) = 2$. *The remove function deletes edges* $\{v_2, v_6\}$, $\{v_3, v_2\}$, *and* $\{v_7, v_3\}$ *in Figure 3.5.*

4.1.4 LIFO Tarjan Algorithm

The strongly connected components of an SCC implication graph represent the equivalency classes of the literals in the formula. Therefore, to discover the equivalent literals of a formula using SCC implication graphs, we need an efficient algorithm to compute the strongly

connected components for a given graph.

The algorithms can be divided into two categories based on how they process the input: *offline* and *online*. An offline algorithm requires the complete input to be given to the algorithm in advance. An online algorithm accepts dynamic inputs in a sequential manner; at each moment, an online algorithm make decisions based on the current state of the input.

There are several efficient offline algorithms to compute strongly connected components of a graph: Tarjan's algorithm [96], Kosaraju's algorithm [4] and Cheriyan-Mehlhorn-Gabow's algorithm [35].

There are different categories of online algorithms for the SCC problem:

Fully dynamic algorithms They support both adding and deleting edges to the graph.

Incremental algorithms They support edge additions, but not edge removals.

The algorithms by Roditty and Zwick [87], and Pearce and Kelly [85] are examples of fully dynamic algorithms. Haeupler, Sen and Tarjan [49] presented an incremental algorithm for the SCC problem.

The LIFO-dynamic graphs, including SCC implication graphs, lie in between these two categories. The incremental algorithms do not suffice to solve the problem of finding strongly connected components in LIFO-dynamic graphs because they do not support edge removal. In the meanwhile, because of the LIFO ordering on the edge removals, these graphs are more restricted than the input graphs in the fully dynamic algorithms. Although we are able to use the algorithms in the first category, we might get a better performance by designing an algorithm based on this restriction.

For every directed graph G, let the set union-deunion S_G represent the strongly connected components of G.

Suppose $G = (V, E_G)$ and $H = (V, E_H)$ are directed graphs on the same vertex set $\{v_1, v_2, \cdots, v_n\}$. The union of the graphs G and H, $U = G \cup H$, is the graph $U = (V, E_G \cup E_H)$.

Suppose $H = (V, E_H)$ is a directed graph. Let E be a set of directed edges defined on the vertex set V. We define graph $G = H + E$ to be $G = (V, E_H \cup E)$.

Suppose H is a directed graph. Let G to be a directed graph such that $G = H + E$ in which E is a set of directed edges not in H. Suppose the strongly connected components of H, S_H, is known. If we run Tarjan's algorithm on the graph G, it reconstructs the strongly connected components of H again. We want to modify Tarjan's algorithm to find the strongly connected components of G without reconstructing the strongly connected components of H.

In the graph $G = H + E$, an edge $e \in G$ is called an *inter-SCC* edge with respect to the union-deunion set S_H, if its endpoints belong to different sets in S_H, i.e. different strongly connected components of H.

Example 4.2. *The inter-SCC edges of SCC implication graph for graphs in 3.3, 3.4 and 3.5 are shown in Figure 4.1 by solid lines. The inter-SCC edges of SCC implication graph in Figure 3.3 with respect to empty set of SCCs are shown in Figure 4.1a. Figure 4.1b and Figure 4.1c show the inter-SCC edges of the graphs in 3.4 and 3.5 with respect to strongly connected components* $\{\{\bar{v}_1, v_2\}, \{v_3\}, \{v_6, v_7, v_8\}, \{\bar{v}_{13}\}\}$ *and* $\{\{\bar{v}_1, v_2, v_3, v_6, v_7, v_8\}, \{\bar{v}_{13}\}\}$ *respectively.*

Let G be a LIFO-dynamic graph at level μ. Define graph H to be

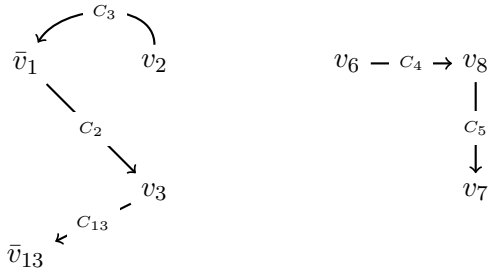
$$H = G_0 \cup G_1 \cup \cdots G_{\mu-1}.$$

If $\mu = 0$, then the graph H is empty. Let E be the set of edges in G_μ. Therefore, we have $G = H + E$. In order to avoid rediscovering the already known strongly connected components of G in the LIFO Tarjan algorithm, we modify Tarjan's original algorithm such that it only explores the inter-SCC edges in G with respect to S_H.

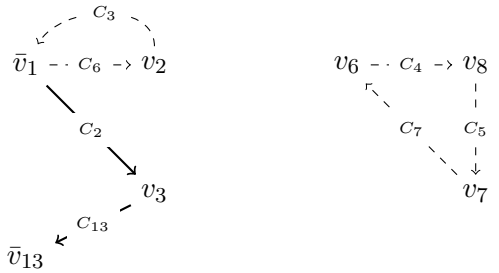
The input to the LIFO Tarjan algorithm is a graph G and a union-deunion set S_H which represents the strongly connected components for a subgraph H of G. The algorithm returns the set union-deunion S_G representing the strongly connected components for graph G.

For a vertex w in a LIFO-dynamic graph $G = H \cup G_\mu$, let $eq(w)$ represent all the vertices in the strongly connected component in H containing the literal w. Therefore, the literals in $eq(w)$ are equivalent to w.

(a) Inter-scc edges at level zero with respect to S = {{$\bar{v}_1$} ,{v_2}, {v_3}, {v_6}, {v_7}, {v_8}, {$\bar{v}_{13}$}}



(b) Inter-SCC edges at level one with $v_9 = 0$ with respect to $S = \{\{\bar{v}_1, v_2\}, \{v_3\}, \{v_6, v_7, v_8\}, \{\bar{v}_{13}\}\}$



(c) Inter-SCC edges at level two $S = \{\{\bar{v}_1, v_2, v_3, v_6, v_7, v_8\}, \{\bar{v}_{13}\}\}$

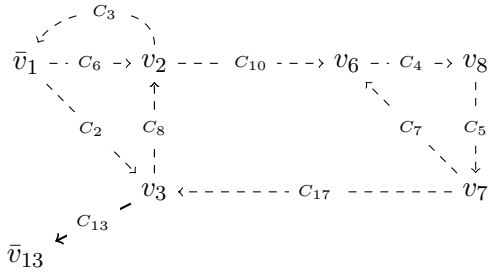


Figure 4.1: Inter-SCC edges for different levels are shown by solid lines.

The *extended neighbors* of w, $en(w)$, is the set of vertices w' that are not in $eq(w)$ but for which there exist at least one edge between one of the vertices of $eq(w)$ and w'. Therefore, for every literal $w' \in en(w)$ we have

- the literal w' is not equivalent to the literal w,
- the literal w' is a neighbor to a vertex in $eq(w)$.

Therefore,

$$en(w) = \{w_j \mid \exists w_i \in eq(w), \{w_i, w_j\} \in G \text{ AND } w_j \notin eq(w)\}.$$

It can be noted that all the edges in the graph G that are from a literal in $eq(w)$ to a literal in $en(w)$ are inter-SCC edges.

Example 4.3. *The extended neighbors of the vertices for Figure 3.4 are:* $en(\bar{v}_1) = \{v_3\}$, $en(v_2) = \{v_3\}$, $en(v_3) = \{\bar{v}_{13}\}$, $en(v_6) = \emptyset$, $en(v_7) = \emptyset$, $en(v_8) = \emptyset$.

Whenever Tarjan's algorithm explores a vertex w, then DFS explores all of its neighbors. The Tarjan's algorithm starts from a union set having singleton elements. Therefore, it always rediscovers the previous information about the strongly connected components.

LIFO Tarjan algorithm starts generating the strongly connected components on the top of the already known ones. To do this instead of exploring the neighbors of the vertex w, its extended neighbors, $en(w)$ are explored. Also, whenever a literal w is explored, all the vertices in $eq(w)$ are marked as explored without DFS exploring them. Therefore, the vertices that are know to be equivalent from earlier levels, are treated as one super vertex[2]. This prevents the Tarjan's algorithm from discovering the same strongly connected component over and over again. A general overview of the algorithm follows.

The LIFO Tarjan algorithm chooses an unexplored vertex w and marks w and all the vertices in $eq(w)$ as explored. Then, the algorithm does a DFS on the set $en(w)$. As in Tarjan's algorithm, the vertices are placed on a stack. A vertex is identified as being the root of a strongly connected component exactly in the same way as in Tarjan's algorithm.

[2]This is equivalent to the definition of a vertex in a condensation graph.

The difference is in identifying the strongly connected component elements. In the Tarjan's algorithm, if a vertex w is identified as a root of the strongly connected component, then the vertex w and all the vertices that are on the top of the w on the stack are in the same connected component. In the LIFO Tarjan algorithm, the strongly connected component consists of the union of the literals in $eq(w)$ and $eq(w')$ for all the vertices w' that are on the top of w on the stack. The LIFO Tarjan algorithm is shown in Algorithm 4.2. The 'strongconnect' method in the line 7 is defined in Algorithm 4.3.

Algorithm 4.2 LIFO Tarjan's Strongly Connected Components Algorithm

Input: graph $G = (V, E)$
Input: union-deunion set S_H representing the strongly connected components for a subgraph H of G
Output: union-deunion set S_G representing the strongly connected components set of G

```
1: index = 0
2: stack S = empty
3: for all v ∈ V do
4:     v.index = undefined
5: end for
6: for all v ∈ V do
7:     if (v.index is undefined) then
8:         strongconnect(v)
9:     end if
10: end for
11: S_G = S_H
```

Observation 4.1. *Let G be a directed graph, and H a subgraph of G with strongly connected components S_H. The LIFO Tarjan's algorithm computes S_G based on the input (G, S_H).*

The LIFO Tarjan algorithm runtime in the worst case is the same as the original Tarjan's algorithm. Theoretically the LIFO Tarjan algorithm complexity in the worst case is not better than Tarjan's algorithm. But our experiments show, in practice, it outperforms the Tarjan's algorithm specially when the strongly connected components are discovered during early levels of the search.

The LIFO Tarjan algorithm presents an online maintenance method to generate the strongly connected components while edges are added to the graph. The following Section presents a method to maintain the strongly connected components while edges are removed from the graph.

Algorithm 4.3 Procedure $strongconnect(v)$

```
1: v.index = index
2: v.lowlink = index
3: index = index + 1
4: S.push(v)
5: for all (element s ∈ S_H.find(v)) do
6:       s.index = index
7:       for all (w ∈ en(s)) do
8:             if (w.index is undefined) then
9:                   strongconnect(w)
10:                  v.lowlink = min(v.lowlink, w.lowlink)
11:            else if (w ∈ S) then
12:                  v.lowlink = min(v.lowlink, w.index)
13:            end if
14:      end for
15: end for
16: if (v.lowlink == v.index) then
17:      start a new strongly component
18:      repeat
19:            w = S.pop()
20:            S_H.union(v, w)
21:      until (v == w)
22: end if
```

4.2 SCC Maintenance in Backtracking

If a conflict is found, the search needs to backtrack. Therefore, the SCC implication graph G and its strongly connected components, S_G, should be updated.

If the search backtracks to level ℓ, all the edges with level greater than ℓ should be removed from G. In order to keep track of the edge levels, every level ℓ keeps a list of edges e with $level(e) = \ell$. In backtracking to level ℓ, the edges that are in the lists with levels greater than ℓ are removed from the graph. A more detailed explanation of the graph maintenance operations is given in Section 4.4.

In order to maintain the strongly connected components, in backtracking to the level ℓ, all the unions that have been done after level ℓ should be deunioned. A slight change in the set union-deunion data structure (Section 4.1.2) provides a mechanism to determine the level for every union.

In the set union-deunion method, for every union, we keep track of the level of the union. Let $z = UnionSet(x, y)$ to be a union done at level t. In the set union-deunion data

structure, this union adds the 3-tuple (z, x, y) to an stack which can be used in retrieving x and y in deunion. Because we need to have the union level information, we add the 4-tuple (z, x, y, t) to also keep track of the levels. The deunion function is called on all the unions with level greater than ℓ. Algorithm 4.4 shows the procedure. The algorithm runtime for the deunion operation is constant.

Algorithm 4.4 Deunion function to level ℓ

Input: a union-deunion data structure with stack S containing the union information
Input: backtrack level ℓ
1: $current = S.top$
2: **while** $(current.level > \ell)$ **do**
3: $z = current.z$
4: $x = current.x$
5: $y = current.y$
6: replace the set z with sets x and y
7: update the representatives
8: $S.pop$
9: $current = S.top$
10: **end while**

4.3 LIFO Tarjan Algorithm Implementation

This Section briefly reviews some of the implementation details of the LIFO Tarjan algorithm. Section 4.3.1 reviews the known idea of timestamps to avoid initializing all the vertices whenever the LIFO Tarjan algorithm is called. Sections 4.3.3 and 4.3.2 introduce methods to limit the number of visited vertices during every call of the LIFO Tarjan algorithm.

4.3.1 Initialization with Timestamps

Tarjan's algorithm, and in the same way LIFO Tarjan algorithm, are DFS-based. They use indices to keep track of the visited vertices during the DFS. Therefore, in Tarjan's algorithm, whenever the method is called, all the vertices should be marked as unseen. In the LIFO Tarjan algorithm, the vertices that are representatives in the set union-deunion data structure should be marked as unseen. In the traditional version of Tarjan's algorithm, this is achieved by initializing the indices to zero every time the method is called. According

to our experiments, this initialization might have an impact up to ten percent of the runtime. In order to reduce this impact, we use the well-known idea of timestamps.

In the timestamp method, there is a variable the value of which states the current timestamp. Any variable with a timestamp less than this value is unseen. Any variable with a timestamp greater than this value has been already visited.

Let variable t to be the current timestamp. A vertex is unexplored if its index is smaller than t. The timestamp t has the following properties:

1. At the beginning of the Tarjan's algorithm its value is greater than all the vertex indices in the graph.

2. During one call of the LIFO Tarjan algorithm its value is constant.

At the beginning all the vertex indices are initialized to zero. The timestamp t is initialized to one. The timestamp t and variable indices might be needed to reinitialize if the timestamp t becomes greater than a predefined threshold value.

The LIFO Tarjan algorithm using this method is shown in Algorithm 4.5. The 'strong-connect' method in Algorithm 4.5 is shown in Algorithm 4.6.

4.3.2 Touched Vertices

Let G be a digraph. Let E be the set of edges added to G, and $V(E)$ denote the end vertices of the edges in E. The SCCs that might be affected by adding the edges in E are those that are reachable (defined in Section 3.5.1) by vertices in $V(E)$.

Lemma 4.1. *If a vertex v is not reachable by $V(E)$, then adding E to G does not change its strongly connected components. Therefore, the LIFO Tarjan Algorithm does not need to explore that vertex.*

Proof. Let $C(v)$ be the SCC containing v in G. Let $C'(v)$ be the SCC containing v in $G \cup E$. We know $C(v) \subset C'(v)$. if $C'(v) \neq C(v)$, then there exists an edge $e = \{v_1, v_2\} \in E$ such that $e \in C'(v)$ and $e \notin C(v)$. If $e \notin C(v)$, then at least one of its end points is not in $C(v)$. Suppose $v_1 \notin C(v)$. Because $C'(v)$ is a strongly connected component, it has a Hamiltonian

Algorithm 4.5 LIFO Tarjan's Strongly Connected Components Algorithm with Timestamps

Input: graph $G = (V, E)$
Input: union-deunion set S_H representing the strongly connected components for a subgraph H of G
Output: union-deunion set S_G representing the strongly connected components set of G

```
1: stack S = empty
2: if (t > MAX − |V| or t is not initialized) then
3:     t = 1
4:     for all v ∈ V do
5:         v.index = 0
6:     end for
7: end if
8: maxIndex = t
9: for all v ∈ V do
10:    if (v.index < t) then
11:        strongconnect(v)
12:    end if
13: end for
14: t = maxIndex + 1
15: S_G = S_H
```

Algorithm 4.6 Procedure *strongconnect*(v) with Timestamps

```
1: v.index = maxIndex
2: v.lowlink = maxIndex
3: maxIndex = maxIndex + 1
4: S.push(v)
5: for all (element s ∈ S_H.find(v)) do
6:     s.index = index
7:     for all ((s, w) ∈ en(s)) do
8:         if (w.index < t ) then
9:             strongconnect(w)
10:            v.lowlink = min(v.lowlink, w.lowlink)
11:        else if (w ∈ S) then
12:            v.lowlink = min(v.lowlink, w.index)
13:        end if
14:    end for
15: end for
16: if (v.lowlink == v.index) then
17:    start a new strongly component
18:    repeat
19:        w = S.pop()
20:        S_H.union(v, w)
21:    until (v == w)
22: end if
```

cycle T. Therefore, there is a path from vertex v to v_1, i.e. vertex v is reachable by $V(E)$ which is a contradiction. □

As a result, in the depth-first search of LIFO Tarjan algorithm, it is enough to explore the vertices that are in $V(E)$ or are reachable by $V(E)$ for the set of added edges E. We call the vertices in $V(E)$, the *touched* vertices. Whenever an edge is added to the graph, its endpoints are added to the touched list. In the DFS, we only explore the vertices in the touched list. After the strongly connected components are calculated the touched list is cleared.

The LIFO Tarjan algorithm using touched lists is shown in Algorithm 4.7. The difference with the Algorithm 4.5 is in line 7. Instead of all the vertices, the algorithm only explores the touched vertices. The 'strongconnect' is the same as in Algorithm 4.6.

Algorithm 4.7 LIFO Tarjan's Strongly Connected Components Algorithm with Touched Vertices

Input: graph $G = (V, E)$
Input: union-deunion set S_H representing the strongly connected components for a subgraph H of G, a set S_V of the touched vertices
Output: union-deunion set S_G representing the strongly connected components set of G

```
 1: stack S = empty
 2: if (t > MAX − |V| or t is not initialized) then
 3:     t = 1
 4:     for all v ∈ V do
 5:         v.index = 0
 6:     end for
 7: end if
 8: maxIndex = t
 9: for all v ∈ S_V do
10:     if (v.index < t) then
11:         strongconnect(v)
12:     end if
13: end for
14: t = maxIndex + 1
15: S_G = S_H
```

4.3.3 Alive Vertices

The set of vertices in a strongly connected component $\mathcal{C}$ and the vertices that are reachable from $\mathcal{C}$ are called *reach(C)*.

If a literal w in a strongly connected component $\mathcal{C}$ is assigned a value, then by unit propagation all the literals in $reach(\mathcal{C})$ are assigned values. A strongly connected component is *assigned*, if the value of its literals are assigned.

An edge $e = \{v_1, v_2\}$ is added to SCC implication graph, if both v_1 and v_2 are unassigned. Therefore, if a strongly connected component $\mathcal{C}$ is assigned, then no new edges are added to or from the vertices in $reach(\mathcal{C})$. As a result, the LIFO Tarjan algorithm does not need to process the vertices in $reach(\mathcal{C})$. We call these vertices *dead*. Therefore, a vertex is dead if it is assigned.

At the beginning of the search, all the vertices are flagged *alive*. Whenever a literal w is assigned a value, the vertices in $reach(\mathcal{C})$, in which $\mathcal{C}$ is the strongly connected component containing the literal w, are flagged *dead*. In backtracking, when the value of a variable is unset, the variable flag is set to alive. The LIFO Tarjan algorithm explores only the alive vertices in the graph.

The LIFO Tarjan algorithm using alive vertices is shown in Algorithm 4.8. The 'strongconnect' is shown in Algorithm 4.9.

Algorithm 4.8 Tarjan's Strongly Connected Components Algorithm with Alive Vertices

Input: graph $G = (V, E)$
Input: union-deunion set S_H representing the strongly connected components for a subgraph H of G, a set S_V of the touched vertices
Output: union-deunion set S_G representing the strongly connected components set of G

```
1: stack S = empty
2: if (t > MAX − |V| or t is not initialized) then
3:     t = 1
4:     for all v ∈ V do
5:         v.index = 0
6:     end for
7: end if
8: maxIndex = t
9: for all v ∈ S_V do
10:    if (v.index < t AND v is alive) then
11:        strongconnect(v)
12:    end if
13: end for
14: t = maxIndex + 1
15: S_G = S_H
```

Algorithm 4.9 Procedure *strongconnect*(v) with Alive Vertices

```
1: v.index = maxIndex
2: v.lowlink = maxIndex
3: maxIndex = maxIndex + 1
4: S.push(v)
5: for all (element s ∈ S_H.find(v)) do
6:     s.index = index
7:     for all ((s, w) ∈ en(s)) do
8:         if (w.index < t AND (w is alive) then
9:             strongconnect(w)
10:            v.lowlink = min(v.lowlink, w.lowlink)
11:        else if (w ∈ S) then
12:            v.lowlink = min(v.lowlink, w.index)
13:        end if
14:    end for
15: end for
16: if (v.lowlink == v.index) then
17:    start a new strongly component
18:    repeat
19:        w = S.pop()
20:        S_H.union(v, w)
21:    until (v == w)
22: end if
```

4.4 SCC Implication Graph Data Structure

In this section, we present a simple data structure for SCC implication graphs. This data structure is efficient in maintaining the SCC implication graph during the search. The downside to this approach is its memory requirement. Due to memory requirements, we have limited the maximum number of variable instances in our experiments to 10,000. For the purpose of the experiments in this thesis, this data structure suffices. Because our goal is to provide an insight into the possible effectiveness of the equivalency reasoning in CDCL solvers. The wide range of available SAT benchmarks provides lots of relatively small instances to achieve this goal. A further possible research project is to use the sparse matrix representation to manage the large instances.

The efficiency of the data structure representing an SCC implication graph in our solver is dependent on the following operations:

add(e): Whenever an eq-binary clause C on equivalency sets E_i and E_j is found, for a literal $w_i \in E_i$ and a literal $w_j \in E_j$ new edges $\{\bar{w}_i, w_j\}$ and $\{\bar{w}_j, w_i\}$ are added to

the graph. These edges are labeled by the pair $\{C, \ell\}$ in which ℓ is the level that eq-binary clause C is discovered. It is possible to discover several eq-binary clauses on equivalency sets E_i and E_j during the search. If for the chosen literals $w_i \in E_i$ and $w_j \in E_j$ an edge already exists in the graph, then it is not overridden. The edge label always represents the eq-binary clause having smallest level.

isEdge(w_i, w_j): It returns true if there is an edge between w_i and w_j, Otherwise, it returns false. This function is always called before adding an edge $\{w_i, w_j\}$ to the graph.

remove(e): In backtracking to level ℓ, all the edges with labels greater than ℓ are removed from the graph.

neighbors(w): Lists all the neighbors for the vertex w. This function is used in LIFO Tarjan algorithm to discover the strongly connected components.

An efficient data structure for adding and removing edges, and neighbor testing for two vertices is the adjacency matrix representation. All these operations for adjacency matrix is $O(1)$. Though, listing the neighbors of a vertex is $O(n)$ in which n is the number of vertices. Let δ represent the degree of a vertex v. The adjacency list data structure has a $O(\delta)$ runtime for listing the neighbors of a vertex v. Edge removal and addition can also be implemented in $O(1)$. Though, the neighbor testing for two vertices is $O(n)$ in the worst case.

In order to benefit from both data structures, we have used a combination of the two data structures. Every SCC implication graph on n vertices is represented by an $n \times n$ adjacency matrix M and an adjacency list L of size n simultaneously. In the beginning all the elements of the adjacency matrix are 0. For every adjacency list i, the first element $L[i][0]$ represents the degree of vertex i. The value of $M[i][j]$ represents the position of literal j in $L[i]$. The value stored in $M[i][j]$ is useful while removing the edge $\{i, j\}$. Algorithms 4.10, 4.11, 4.12 and 4.13 represents the main operations for this data structures. The runtime for all these operations is constant.

The main issue with this approach is memory constraint, specially for the instances with

Algorithm 4.10 Adding Edge

Input: Edge $\{i, j\}$

1: $L[i][0] = L[i][0] + 1$
2: $sz = L[i][0]$
3: $L[i][sz] = j$
4: $M[i][j] = sz$

Algorithm 4.11 Removing Edge

Input: Edge $\{i, j\}$

1: $sz = L[i][0]$
2: $cr = M[i][j]$
3: Swap $L[i][sz]$ and $L[i][cr]$
4: $M[i][j] = 0$
5: $L[i][0] = L[i][0] - 1$

Algorithm 4.12 Neighbor Test

Input: Vertices i and j

1: **if** $(M[i][j] == 0)$ **then**
2: return false
3: **else**
4: return true
5: **end if**

Algorithm 4.13 Listing Neighbors

Input: Vertex i

1: $sz = L[i][0]$
2: **for** $i = 1 \rightarrow sz$ **do**
3: print neighbor $L[i][j]$
4: **end for**

lots of variables. Therefore, in our experiments, we have limited the maximum number of variables in our test cases to be less than $10,000$. The maintenance of the SCC implication graph for large instances eliminates the possible benefits of the equivalency reasoning. A possible future research project is to use a sparse matrix representation for the SCC implication graphs to manage large instances efficiently.

4.5 Decision Heuristic

In the VSIDS branching rule, whenever a variable is selected to branch on, its value is decided randomly. Instead, we use the information in the SCC implication graph to choose the value.

If a literal w is assigned to true, the number of unit implications implied by this assignment is at least equal to the degree of the vertex for $\bar{w}$. We use this fact to determine the value for a decision variable. Whenever a variable v is chosen by VSIDS, its value is set to 0, if $degree(v) > degree(\bar{v})$. If $degree(v) < degree(\bar{v})$, then the value of v is set to 1. In this way, we try to maximize the number of implications for the selected branching variable.

4.6 Chapter Summary

In this Chapter, we provided some of the implementation details of the equivalency based CDCL SAT solver Eq-MiniSat. The next Chapter presents some experimental results for this solver.

Chapter 5

Comparisons and Measurements

This chapter presents the experimental results for our equivalency reasoning enabled CDCL solver: Eq-MiniSat. The first part of the chapter discusses the effects of different implementations for Eq-MiniSat. The second part compares the performances for the solvers MiniSat [32] and Eq-MiniSat.

Section 5.1 gives a short overview of the benchmarks for which Eq-MiniSat outperforms MiniSat. Section 5.2 discusses the kind of statistical data that are presented in this chapter. Section 5.3 provides some comparisons of different approaches implementing Eq-MiniSat. Finally, Section 5.4 compares the results from the standard CDCL SAT solver MiniSat [32] and our equivalency reasoning enabled MiniSat based solver Eq-MiniSat.

5.1 Benchmark Description

The SAT problem is NP-complete [23, 71]. Therefore, unless $P = NP$, there is no SAT algorithm that solves all the instances of SAT in polynomial time. In practice, the hardness of a particular instance is determined by solving it with the state-of-the-art SAT solvers. Different SAT solvers have different performances on different SAT benchmarks.

For experimental results in this thesis, we have chosen benchmarks from different areas of research. Because of the memory constraints of Eq-MiniSat (Section 4.4), the selected benchmarks have at most 10,000 variables. This section provides a description of the benchmarks for which Eq-MiniSat outperforms MiniSat. This set of instances is used in the first

part of this chapter to provide comparisons of different implementation approaches for Eq-MiniSat. A complete list of benchmarks can be found in Section 5.4.

SGI The Subgraph Isomorphism problem or SGI is the problem of deciding whether a graph G is isomorphic to a subgraph of graph H. The practical importance of the SGI problem comes from the fact that it has lots of applications in different areas of research including data mining [68], bioinformatics [86] and social network analysis [95]. Random Subgraph Isomorphism instances that are converted to SAT are shown to be hard for the current state-of-the-art SAT solvers [6]. Therefore, they are often used to compare SAT solver performances.

QG SAT encoded Quasigroup or Latin square instances that satisfy some constraints [103]. This set of instances has both satisfiable and unsatisfiable instances.

FRB A set of forced satisfiable instances encoded from constraint satisfaction problems proposed by Xu and Li [100]. The instances in the benchmark are relatively small, with variable range between 450 and 1534. Nonetheless, many of these instances are hard for most of the current prominent CDCL SAT solvers.

QWH The Quasigroup Completion Problem or QCP [22] is the problem of determining whether it is possible to complete a partially filled Latin square to obtain a complete Latin Square. The Quasigroup With Holes (QWH) [2] is a variation of the Quasigroup Completion Problem. Every instance in QWH is generated starting from a complete Latin square and removing some of its entries. Therefore, for every instance in QWH there exists at least one solution. As a result, after encoding into SAT, all the QWH SAT instances are satisfiable.

EZFACT Dan Pehoushek submitted the SAT encoding of factorization circuits in SAT competition 2002 [93].

RBSAT Random CSP problems encoded to SAT by Mohamedou in SAT 2009 [67].

Table 5.1: Number of instances, average number of variables, and average percentage of binary clauses in every benchmark

Family Name	#instances	Avr. #Vars	Bin. Cl. Prc.
SGI	28	1779	99%
FRB	20	687	99%
QG	20	1825	4%
QWH	5	2329	99%
EZFACT	39	1441	0%
RBSAT	46	1636	99%
GT	10	1668	77%
VMPC	6	1027	39%

GT A set of instances based on partial order and counting problems submitted by Sabharwal to SAT 2005 [10].

VMPC SAT encoded instances based on Variably Modified Permutation Composition on open cryptographic problem by Grieu in SAT 2005 [10].

In this thesis we investigate the integration of equivalency reasoning in CDCL SAT solvers. Intuitively, it seems that equivalency reasoning should perform better on SAT instances with lots of binary clauses. In order to examine the effect of the number of the binary clauses we have chosen benchmarks with a variety of number of binary clauses. Table 5.1 shows the number of instances and the average percentage of binary clauses in every benchmark.

In order to discuss the experimental results, we have chosen some instances from every benchmark. Table 5.2 shows the number of variables, clauses and binary clauses for the chosen instances. The name part that is shown in bold in every instance name is the alias name for that instance from now on in this chapter. In the result column, the values 'S' and 'U' stand for 'Satisfiable' and 'Unsatisfiable', respectively.

In the next section, we discuss the type of statistical data that is presented in this chapter.

Table 5.2: The information for a set of selected Instances. In the 'RSL' column, the values 'S' and 'U' stand for 'Satisfiable' and 'Unsatisfiable', respectively.

Instance Name	Instance Family	Vars	Cls	Bin Cls	RSL
srhd-sgi-m27-q225-n25-p15-s58217873	SGI	550	35586	35561	S
QG7-dead-dnd001.sat05-3419.reshuffled-07	QG	1040	13020	1086	U
QG7a-gensys-icl004.sat05-3825.reshuffled-07	QG	2401	15960	427	U
frb40-19-4	FRB	760	43780	43740	S
frb45-21-4	FRB	945	61855	61810	S
qwh.40.528.shuffled-as.sat03-1652	QWH	2511	18906	17509	S
qwh.40.560.shuffled-as.sat03-1654	QWH	3100	26345	24756	S
ezfact64_1.shuffled	EZF.	3073	19785	0	S
ezfact64_5.shuffled	EZF.	3073	19785	0	S
rbsat-v760c43649gyes6	RBS.	760	43649	43609	S
rbsat-v760c43649g9	RBS.	760	43649	43609	S
gt-ordering-unsat-gt-030.sat05-1307.reshuffled-07	GT	900	24825	435	U
gt-ordering-unsat-gt-035.sat05-1308.reshuffled-07	GT	1225	39900	595	U
vmpc_35.renamed-as.sat05-1921	VMPC	1225	211785	83405	S

The bold part in every instance name is the alias name used for that instance throughout the chapter.

5.2 Statistic for Comparison

Comparing DPLL based SAT solvers involves many considerations including memory requirements, the search tree size, and the CPU time for every solver. Each of these measurements provides different insights into the SAT solver performance and they are not necessarily co-related. For example, a SAT solver might spend a lot of time in every node of the search tree to prune the formula using logical inference rules. This approach usually results in smaller tree sizes, but it does not necessarily imply less CPU time to solve the instance. Or as another example, an equivalency based DPLL SAT solver usually generates smaller search trees, but it requires more memory.

In this section, we review the measurements that are used in this chapter to compare the experimental results.

5.2.1 Memory

Generally, the hardness of a SAT instance is not related to its size. There are SAT instances with less than a hundred variables that the current state-of-the-art SAT solvers can not solve

in a reasonable amount of time. While SAT instances with thousands of variables are solved in fractions of a second.

Large instances require data structures that provide means to access the variables and clauses efficiently. A key point in the success of state-of-the-art SAT solvers in solving large instances is their innovative data structures [76]. These data structures enable SAT solvers to efficiently access and maintain the variables and clauses.

A known SCC-based equivalency reasoning weakness is its memory requirements for representing the SCC graph. This weakness restricts the input for the SCC-based equivalency enhanced SAT solvers to instances with a relatively small number of variables [40]. Eq-MiniSat suffers from the same problem. The memory requirement for Eq-MiniSat for an instance on n variables and m clauses is $O(n^2 + mn)$. Also, it can be noted that in CDCL solver the memory requirement increases during the search due to the addition of learnt clauses. Therefore, we have restricted the number of variables to be less than 10,000. In Section 5.3.6 we provide the statistics for Eq-MiniSat memory requirements for instances in Table 5.2.

5.2.2 Search Tree Size

The Search tree size, or number of branchings, is dependant on the chosen branching rule and the inference rules used in every node to simplify the formula. Most of the DPLL SAT solvers have at least one inference rule: unit propagation. Many other inference rules have been also studied including pure literal rule [28], equivalency reasoning [72], resolution [28], and subsumption rule [28]. The effects of these logical reasonings in SAT solving have been shown to be application dependant. While one kind of reasoning might decrease the tree size for an instance by orders of magnitude, the same kind of reasoning might increase the tree size for another instance by orders of magnitude.

The logical reasoning usually decreases the tree size. But, from the fact that it increases the amount of time spend in every node, smaller tree size does not necessarily mean a better run-time. Therefore, the number of branchings is not an accurate measure to compare different solvers. Though, theoretically, if all the solvers spend the same time at every node, a smaller search tree results in better performance.

There is another measurement that can be used in order to predict the run-time performance: number of propagations. The number of propagations is the number of variable assignments both for decision and implied variables at every node. Division of this number by the number of branchings is an indication of the average effort at every node. In this chapter we provide statistics for both number of branchings and number of propagations.

5.2.3 Runtime

The practical aspects of SAT solving demands SAT solvers be able to solve real-world industrial SAT instances efficiently. The SAT solvers are expected to solve instances in a reasonable time. Therefore, one measurement in comparing SAT solvers' efficiency is comparing their CPU runtime on the same computing environment for the same input.

So far, there has been no solver that outperforms all the other solvers on all the instances. Specially because there are application-based SAT solvers that are configured to solve a special class of instances efficiently. These solvers usually outperform other solvers in their special set of instances, but have poor performance on other instances.

We use the CPU runtime statistics to compare MiniSat and Eq-MiniSat. Also, the CPU runtime is used to compare different approaches that we have used to implement Eq-MiniSat.

The CPU runtime in this chapter are presented in seconds. All the experiments have been run on an AMD Opteron (tm) Processor 875, 2200 GHZ, 64GB RAM machine. The cut-off time is set to 7200 seconds.

The following section discusses the effects of different implementations of Eq-MiniSat.

5.3 Effects of Different Implementations

This section gives statistics on some of the different implementations for Eq-MiniSat. It also provides statistics about strongly connected components in the SCC implication graph. The instances in Table 5.2 are used throughout this section.

Tarjan's algorithm is a linear algorithm, but when it is called at every node, its negative performance impact is noticeable. The LIFO Tarjan's method, introduced in Chapter 4.1, is

one attempt to decrease this negative impact. Section 5.3.1 provides statistics on comparing LIFO Tarjan algorithm versus the original Tarjan's algorithm.

The branching rule VSIDS is the incumbent branching rule in CDCL SAT solvers. In Eq-MiniSat, as in MiniSat, the decision variable v is chosen by VSIDS branching rule. But instead of randomly choosing a value for the variable v, we use the information from the SCC graph to choose the value. Section 5.3.2 presents a detailed explanations of the changes.

Section 5.3.3 provides the statistics for different preprocessing methods.

Section 5.3.4 gives some statistics about the size and level of the strongly connected components.

Section 5.3.5 gives some statistics on the time that is spent in equivalency reasoning related methods. The statistics of this kind can be used to investigate the runtime bottlenecks. This information identifies the possible scopes for future optimization.

Section 5.3.6 presents the memory requirements for Eq-MiniSat.

5.3.1 LIFO Tarjan Algorithm vs. Original Tarjan's Algorithm

The strongly connected components of the SCC graph are updated in the beginning of every level and in backtracking. Tarjan's algorithm is a memoryless algorithm. Therefore, the algorithm evaluates all the vertices of the graph every time it is called. The LIFO Tarjan algorithm, uses the existing strongly connected components to generate new ones going down the search tree. Our experiments show that the LIFO Tarjan algorithm, on average, improves the runtime performance of Eq-MiniSat from 20 up to 50 percent.

Table 5.3 presents the runtime percentage for Eq-MiniSat using original and LIFO Tarjan algorithm on an instance from FRB family. The information is obtained using the profiler *gprof* [46].

5.3.2 A Variation of VSIDS

The SCC graph provides information that might be useful in the variable selection process. For example, the degree of the vertex for a literal w shows the minimum number of variable implications if the literal w is assigned to true. The exact number of variable implications

Table 5.3: Comparing Original Tarjan's Algorithm with LIFO Tarjan Algorithm for FRB30-15-01. The runtime for original and LIFO algorithms are 29.17, 15.97 seconds, respectively.

	Original Tarjan		LIFO Tarjan	
Procedure	Percentage	Seconds	Percentage	Seconds
	SCC related			
Extended Propagation	10.10	2.94	14.4	2.29
Maintaining Edges	1.20	.35	1.00	.15
Tarjan's Alg.	13.60	3.96	2.70	.43
	Non-SCC related operations			
Propagate	73.10	21.32	85.60	13.67
Get a Literal Value	24.40	7.11	8.40	1.34
Conflict Analyze	1.60	.26	8.00	1.27
Backtrack	2.00	.58	2.70	.43.00

is the number of vertices that are reached when running a DFS algorithm on the vertex for the literal w. The latter approach have a runtime impact due to the fact that at every node, a DFS on all unassigned vertices is performed to find the best possible choice. This runtime impact usually overweights its benefits. The former approach requires the solver to find the unassigned vertex with maximum degree. From the fact that the degree of every vertex is presented by the first element in its adjacency list, finding the vertex with maximum degree does not impose a significant runtime impact.

One possible approach is to choose the literal with maximum degree. Our experiments show that the results of this approach is usually less favorable comparing to VSIDS.

Another approach is to use this information with the branching rule VSIDS. In VSIDS, when a variable v is chosen to be the decision variable, its value is determined randomly. Instead, we choose the value for variable v to be the value that makes the literal on v with the maximum degree true.

Table 5.4 shows the results for VSIDS with random value selection versus VSIDS with SCC based value selection. As it can be seen in this table, the SCC based value selection results in better runtime performance for this selection of instances.

5.3.3 Preprocessing Effects

In this section, we present statistical results for different preprocessing methods: literal substitution, failed literal detection, and clause strengthening.

Table 5.4: Experiments on branching rule VSIDS. Comparing SCC Value Selection (A) vs. Random Value Selection (B).

Instance	A	B
srhd	.04	207.1
QG7	3367.8	I
QG7a	1061.9	5428.4
frb40-19-4	93.3	1646.4
frb45-21-4	569.1	5744.7
qwh.40.528	472.9	3486.1
qwh.40.560	637.7	5610.8
ezfact64_1	1133.4	I
ezfact64_5	2022.9	I
rbsat-v760c43649gyes6	I	I
rbsat-v760c43649g9	I	I
gt-ordering-unsat-gt-030	1399.2	I
gt-ordering-unsat-gt-035	6659.5	I
vmpc_35	I	I

The sign (I) means solver was unable to solve the instance in two hours.

Literal Substitution The strongly connected components of an SCC implication graph denote the literal equivalencies in a SAT instance. Substituting equivalent literals leads to fewer number of variables and shorter clauses. Though, because of expensive maintenance that is required during the backtracking, applying literal substitution during the search is not efficient. In the meanwhile, if the equivalent literals are discovered in level zero, there is no runtime overhead in backtracking if the literal substitution is performed. This approach have been used in preprocessing steps of different SAT solvers [14, 17].

Failed Literals If the failed literal rule (Section 2.3) implies a literal, the implication might generate new binary clauses. Therefore, the SCC implication graph might find more equivalencies in preprocessing.

Clause Strengthening The information in the SCC implication graph can be used to facilitate subsumption [75] and self-subsumption [31] rules. Self-subsuming a clause is the process of shortening the clause by logical inferences using other clauses in the formula. This process is also called *clause strengthening*. In order to strengthen a clause C, we use the edge labels in SCC implication graph. The edges that are particularly useful are the ones with labels having clauses of length two. Let literal w be one of the literals in the clause C. For every

other literal $w_i \in C$, we verify the following edges:

Case 1: If there is an edge $\{w, w_i\}$ with the label $C_i = \{\bar{w}, w_i\}$, then the clause C is self-subsumed to $C - \{w\}$.

Case 2: If there is an edge $\{\bar{w}, \bar{w}_i\}$ with label $C_i = \{w, \bar{w}_i\}$, then the clause C is self-subsumed to $C - \{w_i\}$.

Case 3: If there is an edge $\{\bar{w}, w_i\}$ with clause $C_i = \{w, w_i\}$, then by subsumption rule, the clause C can be removed from the formula.

Table 5.5, presents a combination of the discussed methods: equivalent literal substitution, failed literals, and clause strengthening. As it can be seen in Table 5.5 different preprocessing methods have different runtime impacts in Eq-MiniSat.

In Table 5.5 Column 'E' shows the runtime without any preprocessing. Columns 'L', 'F', and 'C', respectively, represents the runtime when only equivalent literal substitution, failed literals, or clause strengthening is applied in preprocessing. Column 'LF' shows the runtime for the combination of literal substitution and failed literals in preprocessing. In column 'LC', the combination of literal substitution and clause strengthening is presented. Column 'FC' provides the runtime for the combination of failed literals and clause strengthening in preprocessing. Finally, column 'LFC' presents the result when all the three methods are used in preprocessing.

As it can be seen in Table 5.5, the benefits of different combinations is application dependant. For example, the combination of literal substitution and clause strengthening enables the solver to solve 'rbsat-v760c43649gyes6', while it is harmful for 'frb45-21-4'.

5.3.4 Strongly Connected Components Information

The strongly connected components in an SCC implication graph reveal the equivalent literals. The larger the size of a strongly connected component is, the more equivalency information it provides. Another important factor is the level the strongly connected component is found. The lower the level is, the possibility of more impact on the search increases. Table 5.6 provides this information for the instances in Table 5.2. The information in this table is generated by running Eq-MiniSat without any preprocessing.

Table 5.5: Experimental results for Eq-MiniSat with different preprocessing method combinations. In this Table, 'E', 'L', 'F', and 'C' stands for no preprocessing, literal substitution, failed literal detection, and clause strengthening respectively.

Instance	E	L	F	C	LF	LC	FC	LFC
srhd	.4	.4	.5	.07	.08	.05	.03	.06
QG7	3367.8	I	6152.4	6000.6	I	6970.1	I	I
QG7a	1061.9	I	2335.4	1861.8	2923.7	1314.1	1995.1	3371.6
frb40-19-4	93.3	688	1779.8	30.4	562.6	705.3	2108.9	399
frb45-21-4	569.1	I	361.5	561.8	7070.2	I	422.1	5028.7
qwh.40.528	472.9	233.6	1464.2	418.6	366.7	240.4	1546.18	265.6
qwh.40.560	637.7	68.7	62.8	668.9	172.6	69.3	56.2	115.5
ezfact64_1	1133.4	2227	3795.1	I	921.7	311.7	760.4	2148.7
ezfact64_5	2022.9	3937.2	I	I	I	773.3	I	6608.4
v760c43649gyes6	I	I	I	I	I	3266.4	I	I
v760c43649g9	I	608.8	I	I	I	616.1	I	I
gt-030	1399.2	247.4	I	2828	5326.7	270.5	I	3694.4
gt-035	6659.5	5915.9	I	I	I	6116.5	I	I
vmpc_35	I	I	I	I	I	I	I	I

The sign (I) means solver was unable to solve the instance in two hours.

In this section, by strongly connected component we mean a non-trivial strongly connected component, i.e. a strongly connected component with size greater than one. Table 5.6 presents the following information:

First SCC: The level and size of the first strongly connected component that has been discovered. It can be noted that this value is not necessarily the minimum level with a strongly connected component, because after the search restarts we might find a strongly connected component in a lower level.

Maximum SCC: The level and size of the strongly connected component with maximum size.

Average: The average size of the strongly connected components throughout the search.

The information in Table 5.6 might be used to avoid unnecessary calculations in the solver. For example for the instances in the GT benchmark, the first levels of search (up to level 50 before first restart) do not have any strongly connected components. In future work, dynamic methods will be designed to enable the solver to postpone or disable the

Table 5.6: Strongly Connected Component Information

Instance	First		Max.		Avr.
Name	level	size	level	size	size
srhd	6	2	6	2	2
QG7	0	2	7	106	3
QG7a	0	2	12	84	2
frb40-19-4	1	2	6	66	2
frb45-21-4	1	2	6	76	2
qwh.40.528	0	5	31	286	2
qwh.40.560	0	3	45	106	2
ezfact64_1	0	2	10	124	4
ezfact64_5	0	2	12	146	4
rbsat-v760c43649gyes6	5	2	8	70	1
rbsat-v760c43649g9	6	2	7	82	2
gt-ordering-unsat-gt-030	50	7	39	226	6
gt-ordering-unsat-gt-035	90	8	50	126	6
vmpc_35	1	3	5	110	2

Table 5.7: First SCC Level Information for Every Benchmark

Family	Avr. Level	Avr. Size
SGI	8.5	3.3
FRB	4.3	2.0
QG	0.0	2.0
QWH	0.0	7.0
EZFACT	0.0	2.0
RBSAT	8.2	2.0
GT	29.9	7.9
VMPC	1.0	3.1

SCC calculations based on the history of the discovered strongly connected components throughout the search.

Table 5.7, for the benchmarks in Table 5.1, shows the average value of the levels that for the first time a non-trivial SCC is discovered throughout the search.

5.3.5 Runtime Overhead

Our experiments show that the equivalency reasoning has a runtime effect of 20 to 40 percent. This runtime impact mostly depends on the number of variables and number of binary clauses in the beginning and during the search. The larger these numbers are, the more expensive equivalency reasoning is.

Table 5.8 shows the runtime statistics for equivalency reasoning for an instance from

FRB family. The information is obtained using the profiler *gprof* [46]. Because running the profiler takes a considerable amount of time, we have chosen a relatively small instance as a sample. The information in the table shows that for this instance the equivalency reasoning takes about about 34% of the runtime when the preprocessing in enabled, and 31% of the runtime without preprocessing.

Table 5.8: Runtime Percentage Statistics for running Eq-MiniSat on frb35-17-1 with 595 variables and 29,707 clauses. The runtime with and without processing are 22.41 and 60.81 seconds, respectively.

	Preprocessing Runtime		No Preprocessing Runtime	
Procedure	Percentage	Seconds	Percentage	Seconds
	SCC related			
Extended Propagation	14.9	3.33	14.2	8.63
Maintaining Edges	11.6	2.59	12.6	7.66
Tarjan's Alg.	2.6	.58	2.7	1.62
Literal Substitution	2.2	.49	-	-
Failed Literal	1.7	.38	-	-
Clause Strengthening	0.1	0.02	-	-
	Non-SCC Related Operations			
Propagate	83.7	18.57	85.3	51.87
Get a Literal Value	8.5	1.90	10.1	6.11
Conflict Analyze	9.1	2.03	8.2	4.98
Backtrack	2.4	.53	2.9	1.76

5.3.6 Memory Requirement

One of the main weaknesses of the Eq-MiniSat is its memory requirements. The SCC implication graph and the set union-deunion require lots of memory. Table 5.9 presents the memory requirements for Eq-MiniSat for the instances in Table 5.2. As it can be seen the memory requirement increases considerably when the number of variables increases. For example an instance with 550 variables requires 39.8 MB, while an instance with 760 variables requires 81.44 MB which is more than twice the first instance.

5.4 MiniSat and Eq-MiniSat: A Comparison

In order to evaluate the effects of equivalency reasoning in CDCL solvers, in this section we overview the results of running MiniSat and Eq-MiniSat on a selected set of benchmarks.

Table 5.9: Memory Information in MB

Instance	Vars	Cls	Memory(MB)
srhd	550	35586	39.80
QG7	1040	13020	113.18
QG7a	2401	15960	469.04
frb40-19-4	760	43780	81.44
frb45-21-4	945	61855	148.09
qwh.40.528	2511	18906	518.10
qwh.40.560	3100	26345	785.63
ezfact64_1	3073	19785	759.39
ezfact64_5	3073	19785	762.89
rbsat-v760c43649gyes6	760	43649	144.04
rbsat-v760c43649g9	760	43649	146.13
gt-ordering-unsat-gt-030	900	24825	167.85
gt-ordering-unsat-gt-035	1225	39900	384.91
vmpc_35	1225	211785	341.62

The complete set of results for the selected benchmarks is provided in Appendix A.

The computing environment is an AMD Opteron (tm) Processor 875, 2200 GHZ, 64GB RAM machine. The cut-off central processing unit (CPU) time for every run is set to 7200 seconds. The MiniSat clause minimization mode is disabled.

For the experiments, different crafted and industrial benchmarks are used. All the SAT benchmarks used in this thesis are from the SAT competitions and are accessible through the SAT competition web page [1]. Other than the benchmarks described in Section 5.1, the other benchmarks are 2 dimensional strip packing or 2SPP, advanced encryption standard or AES, equivalence checking multiplier designs, MOD circuits, graph pebbling, automata synchronization, battleship, and Van Der Waerden or VDW numbers. A description of these benchmarks follows.

2SPP The 2-dimensional strip packing or 2SPP [12] is a special case of the bin packing problem [97]. The SAT encodings of the 2SPP was submitted to SAT competition 2011 by Daniel Le Berre.

AES A SAT encoding of advanced encryption systems or AES [84] was submitted by Oliver Kullmann to the SAT competition 2011.

Equivalency Checking Multiplier Designs The SAT encodings of equivalency checking hardware designs for integer multiplication was submitted by Matti Jarvisalo to the SAT competition 2007 [55].

MOD Circuits The encodings of circuits for MOD-functions was submitted by Yaroslavtsev to the SAT competition 2009 [64].

Graph Pebbling The encoding of the graph pebbling game [80] was submitted by Ashish Sabharwal to the SAT competition 2005.

Automata Synchronization The encoding of the synchronization of random automata [94] into SAT was submitted to the SAT competition 2011 by Evgeny Skvortsov.

Battleship The encoding of the battleship puzzle (a description can be found here [90]) into SAT was submitted to the SAT competition 2011 by Evgeny Skvortsov.

VDW The encoding of Van Der Warden [98] problem into SAT was submitted to the SAT competition 2011 by Oliver Kullmann.

The total number of families is sixteen. Based on the performance comparisons of MiniSat and Eq-MiniSat on these benchmarks, we have observed three different outcomes

Category 1: For one class out of the sixteen chosen families, the equivalency reasoning does not have a noticeable effect on improving or worsening the performance. Examples include MOD circuits family (Table A.24 in Appendix A).

Category 2: In eight classes, the Eq-MiniSat outperforms the original MiniSat. The families SGI (Table A.8), FRB (Table A.10), QG (Table A.12), QWH (Table A.14), EZFACT (Table A.16), GT (Table A.18), RBSAT (Table A.20) and VMPC (Table A.22) have this property. The result tables are in Appendix A.

Category 3: Finally, there are six families of instances that equivalency reasoning slows down the SAT solver MiniSat. Examples of these families include: 2SPP (Table A.2), AES

Table 5.10: Summary of the Results for MiniSat Versus Eq-MiniSat

Family	Total	Solved by Both			Only	Only
Name	#Ins.	#Ins	M. Runtime	Eq-M. Runtime	MiniSat	Eq-M.
MOD	7	3	2555.1	1505.0	0	0
SGI	28	1	601.7	.1	0	24
FRB	20	14	6339.9	5904.5	0	4
QG	20	11	9365.7	6194.7	0	3
QWH	5	4	3487.4	1788.6	0	1
EZFACT	39	31	525.6	1846.3	0	4
GT	10	7	2590.9	1088.5	0	3
RBSAT	46	1	2620.9	67.0	1	5
VMPC	6	2	5691.3	801.6	0	1
2SPP	11	6	474.5	461.8	4	0
AES	5	2	3.2	14.3	1	0
MULT.	7	3	341.9	753.5	1	0
AUT.	10	7	505.3	5109.8	1	0
BATT.	24	2	336.7	192.8	17	0
PEBB.	9	7	26.7	758.9	3	0
VDW	12	9	14,921.0	16,289.0	3	0

(Table A.4), equivalency checking multiplier designs (Table A.6), automata synchronization (Table A.26), battleship (Table A.28), graph pebbling (Table A.30), and VDW (Table A.32). The result tables are in Appendix A.

Table 5.10 summarizes the results for these benchmarks. The information in this table includes the total number of instances in every family, the number of instances solved by both solvers and the runtime every solver spent to solve these instances, and finally the number of instances solved by only one of the solvers.

In order to have an insight into the SCCs generated during the search, Table 5.11 presents the average first level a non-trivial SCC is found for every family, alongside the average length of strongly connected components for every family at the first level. The information in this table suggests that the lower the first level, and the higher the average length of SCCs, the probability of Eq-MiniSat outperforming MiniSat is higher.

Table 5.12 shows a random selection of instances from these families. Figure 5.1 shows the number of branchings for Minisat and Eq-Minisat for these instances. In this Figure, it can be seen that the number of branchings for most of the instances is less for Eq-Minisat. Figure 5.2 shows the CPU runtime in seconds for Minisat and Eq-Minisat for these instances. As can be seen in these figures, fewer number of branchings for Eq-Minisat

Table 5.11: Number of instances, average percentage of binary clauses, and average level for the first nontrivial strongly connected component in the benchmarks that MiniSat outperforms Eq-MiniSat. The 'Avr. Size' column shows the average size for the SCCs found the first time a non-trivial SCC is found.

Family Name	#instances	Avr. #Vars	Bin. Cl. Prc.	Avr. Frst. lvl	Avr. Size
MOD	7	602	.01%	25.8	3.8
SGI	28	1779	99%	8.5	3.3
FRB	20	675	99%	4.3	2.0
QG	20	1825	4%	0.0	2.0
QWH	5	2329	99%	0.0	7.0
EZFACT	39	1441	0%	0.0	2.0
GT	10	1668	77%	29.9	8.0
RBSAT	46	1636	99%	8.2	2.0
VMPC	6	1027	39%	1.0	3.1
2SPP	11	6488.9	14%	0	2.0
AES	5	708	4%	0	2.0
MULT.	7	1185	43%	1	2.0
AUT.	10	4097	13%	41.75	20.6
BATT.	24	266	92%	11.5	22.0
PEBB.	9	1772	.1%	1220	5.5
VDW	12	201	.4%	10.8	2.4

Table 5.12: A random selection of instances from every family. The column 'label' presents the label used for every instance in Figure 5.1 and Figure 5.2

Family	Instance	Graph Label
MOD	mod3_4vars_6gates	mod3_4
SGI	srhd-sgi-m27-q225-n25-p15-s58217873	s58217873
FRB	frb40-19-1	frb40-19-1
QG	QG-gensys-brn008.sat05-2685.reshuffled-07	QG-brn008
QWH	qwh.40.560.shuffled-as.sat03-1654	qwh.40.560
EZFACT	ezfact16_1.shuffled	ezfact16_1
GT	counting-easier-fphp-012-010.sat05-1214.reshuffled-07	sat05-1214
RBSAT	rbsat-v760c43649g7	v760c43649g7
VMPC	vmpc_29.renamed-as.sat05-1916	vmpc_29
2SPP	E03N17	E03N17
AES	aes_32_2_keyfind_1	aes_32_2
MULT.	eq.atree.braun.8.unsat	braun.8
AUT.	rnd_100_28_s	rnd_100_28_s
BATT.	battleship-10-10-unsat	10-10
PEBB.	sat-pbl-00400.sat05-1322.reshuffled-07	sat-pbl-00400
VDW	VanDerWaerden_2-3-12_135	2-3-12_135

does not necessarily imply less runtime.

In the following section, we summarize the results provided in this section.

5.4.1 Conclusion and Future Work

The experimental results demonstrate that the beneficial aspects of equivalency reasoning in CDCL solvers is application dependant. The equivalency reasoning enabled CDCL solver Eq-MiniSat is able to solve families of instances that are generally considered hard for CDCL solvers. At the same time, for some instances Eq-MiniSat has considerably slower performance than MiniSat.

In order to fully take advantage of the benefits of the equivalency reasoning in CDCL SAT solvers, it is desirable to make the equivalency reasoning configurable during the search. For example, a solver capable of enabling or disabling the equivalency reasoning during the search acquires the means to prevent equivalency reasoning whenever it is not beneficial for the SAT solving. In that case the solver requires to have some measurements to decide based on those measurements to enable or disable the equivalency reasoning.

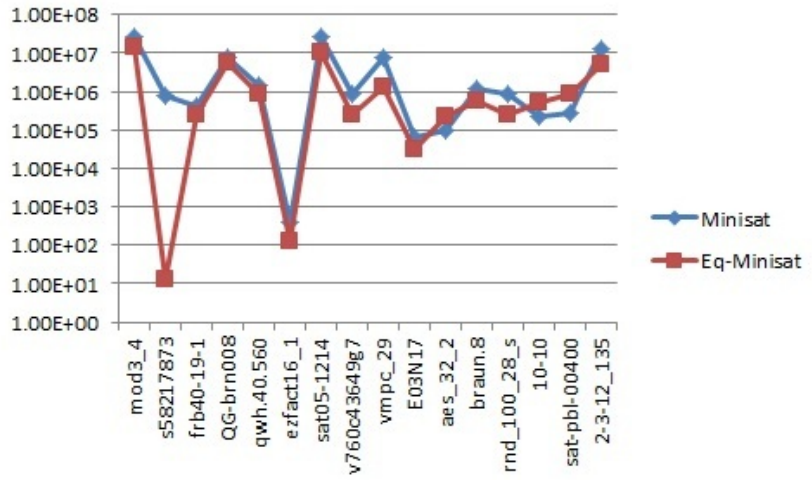


Figure 5.1: Number of branchings for instances in Table 5.12 in logarithmic scale.

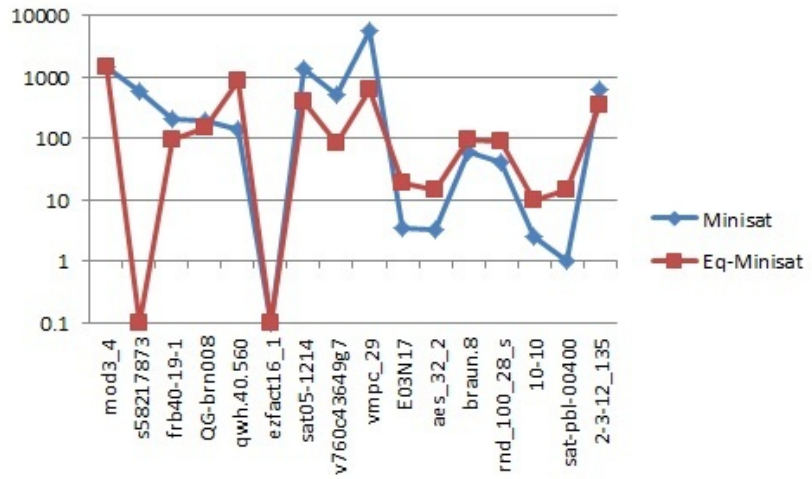


Figure 5.2: CPU runtime in seconds for instances in Table 5.12 in logarithmic scale.

The first intuitive measurement is the number of binary clauses. But as the experimental results shows, the number of binary clauses does not have a direct relation with the runtime performance of Eq-MiniSat. For example, Eq-MiniSat outperforms MiniSat for the instances in the EZFACT family though the number of binary clauses at the beginning of the search is zero. For the automata synchronization family, the average percentage of the binary clauses is 42 percent, but MiniSat performs better than Eq-MiniSat.

The information in Table 5.11 demonstrates that for the instances for which the first SCC level is low, and the average SCC length in this level is high, the probability of equivalency reasoning being beneficial to the solver is higher. This observation provides a possible measurement to decide whether equivalency reasoning is beneficial to the solver or not. For example, one possible approach is to provide different checkpoints for the solver. At every checkpoint, if the equivalency reasoning is not already on, the solver calculates the SCC components. If there is any non-trivial SCCs, the solver enables the equivalency reasoning. Otherwise, the solver continues the search without equivalency reasoning.

A future possible work is to investigate the effects of a configurable equivalency reasoning for CDCL solvers.

5.5 Chapter Summary

In this chapter, some experimental results for Eq-MiniSat was presented. Statistical data was provided to demonstrate the fact that equivalency reasoning can be useful in solving some classes of SAT instances for the CDCL SAT solvers. The following chapter presents the conclusions and possible future research on this topic.

Chapter 6

Conclusions and Future Work

This chapter summarizes the research in this thesis by providing conclusions and possible future work. In Section 6.1, we provide the conclusions from the research in this thesis. Section 6.2 reviews some possible future work.

6.1 Conclusions

In look-back SAT solvers, because of its complications, the equivalency reasoning has been only studied statically at the beginning of the search[8, 17]. In this thesis, we proposed a method to integrate an SCC based equivalency reasoning engine in CDCL SAT solvers. This engine enables the CDCL solvers to simplify the formula based on the identified equivalent literals during the search. As a result, it helps the solver to have a smaller search space.

The equivalency reasoning engine has been integrated into an state-of-the-art CDCL SAT solver MiniSat [32]. The equivalency reasoning enabled solver, Eq-MiniSat, has been used to examine the effects of literal equivalency reasoning in CDCL SAT solvers. The results of the experiments show that equivalency reasoning is beneficial in solving some classes of SAT instances. For example, Eq-MiniSat is able to solve a class of instances [38] that are considered hard for DPLL-based solvers. The experimental results on random instances demonstrates that equivalency reasoning in the majority of the instances generates a smaller search tree.

One of the main features of the SAT solvers is their implementation efficiency. There-

fore, in order to efficiently calculate the strongly connected components in Eq-MiniSat, a customized Tarjan's algorithm has been introduced.

The SCC implication graph, not only helps in identifying the equivalent literals, but it also has other valuable information that can be used to guide the search. This thesis describes some possible use of this information in subsumption and self-subsumption rules, and identifying the conflict variables without BCP.

Another topic that was investigated was the representation of the SAT instances by orthogonal lists or dancing links [52, 63]. Our experiments, which are not reported in this thesis, show that although the orthogonal lists data structure outperforms the counter-based data structure for SAT [59, 81, 92], it is slower than the watched literals scheme [81].

6.2 Future Work

The work in this thesis mainly discusses the integration process of the equivalency reasoning in CDCL SAT solvers. We have done some experiments on a selected set of benchmarks, but more experiments are required to have a better understanding of the equivalency reasoning enabled CDCL solver behaviours.

One of the main questions is to find out when equivalency reasoning is beneficial for an instance. So far, there is no understanding whether the equivalent reasoning is beneficial for an instance without solving it. Such information can be used dynamically during the search to decide whether to apply equivalency reasoning or not. From the fact that calculating SCCs and maintaining the SCC implication graph is an expensive task, this might help to improve the runtime performance.

The other area is the decision making process considering the equivalent literals. The solver Eq-MiniSat uses a variation of VSIDS based on the SCC implication graph. But even this variation does not take into account the equivalency of the literals and the effect of these equivalencies on the formula.

Other improvements include designing data structures for SCC implication graph to minimize the restrictions on the number of variables for the instances that can be solved by the SCC-based equivalency reasoning enabled CDCL solvers.

Another future work includes using *path compressions* [36] to represent the set union-deunion data structure.

Different CDCL SAT solvers have different strengths and weaknesses. In this thesis, we have integrated the equivalency reasoning engine into MiniSat [32]. One possible future work is to integrate the equivalency reasoning engine into other prominent CDCL SAT solvers. The comparison of the results might give a better understanding of the equivalency reasoning effects on CDCL SAT solvers.

Bibliography

[1] *SAT Competition*, `http://www.satcompetition.org/`. 85, 106

[2] Dimitris Achlioptas, Carla P. Gomes, Henry A. Kautz, and Bart Selman, *Generating Satisfiable Problem Instances*, in Kautz and Porter [61], pp. 256–261. 73

[3] Tanbir Ahmed, Oliver Kullmann, and Hunter S. Snevily, *On the van der waerden numbers w(2;3,t)*, CoRR **abs/1102.5433** (2011). 2, 7

[4] Alfred V. Aho, John E. Hopcroft, and Jeffrey D. Ullman, *Data Structures and Algorithms*, Addison-Wesley, 1983. 57

[5] Anbulagan and John K. Slaney, *Lookahead Saturation with Restriction for SAT*, CP (Peter van Beek, ed.), Lecture Notes in Computer Science, vol. 3709, Springer, 2005, pp. 727–731. 10

[6] Calin Anton and Lane Olson, *Generating Satisfiable SAT Instances Using Random Subgraph Isomorphism*, in Gao and Japkowicz [38], pp. 16–26. 73

[7] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan, *A Linear-Time Algorithm for Testing the Truth of Certain Quantified Boolean Formulas*, Inf. Process. Lett. **8** (1979), no. 3, 121–123. 26, 27, 28

[8] Fahiem Bacchus, *Enhancing Davis Putnam with Extended Binary Clause Reasoning*, AAAI/IAAI, 2002, pp. 613–619. 2, 3, 26, 92

[9] ———, *Exploring the Computational Tradeoff of More Reasoning and Less Searching*, Proceedings of Fifth International Symposium on Theory and Applications of Satisfiability Testing, 2002, pp. 7–16. 26, 27

[10] Fahiem Bacchus and Toby Walsh (eds.), *Theory and Applications of Satisfiability Testing, 8th International Conference, SAT 2005, St. Andrews, UK, June 19-23, 2005, Proceedings*, Lecture Notes in Computer Science, vol. 3569, Springer, 2005. 74, 98

[11] Fahiem Bacchus and Jonathan Winter, *Effective Preprocessing with Hyper-Resolution and Equality Reduction*, in Giunchiglia and Tacchella [43], pp. 341–355. 12, 24, 27

[12] Brenda S. Baker, Edward G. Coffman Jr., and Ronald L. Rivest, *Orthogonal Packings in Two Dimensions*, SIAM J. Comput. **9** (1980), no. 4, 846–855. 85

[13] Armin Biere, *Adaptive Restart Strategies for Conflict Driven SAT Solvers*, SAT (Hans Kleine Büning and Xishun Zhao, eds.), Lecture Notes in Computer Science, vol. 4996, Springer, 2008, pp. 28–33. 17

[14] ———, *PicoSAT Essentials*, JSAT **4** (2008), no. 2-4, 75–97. 11, 16, 23, 24, 80

[15] Armin Biere, Edmund M. Clarke, Richard Raimi, and Yunshan Zhu, *Verifiying Safety Properties of a Power PC Microprocessor Using Symbolic Model Checking without BDDs*, CAV (Nicolas Halbwachs and Doron Peled, eds.), Lecture Notes in Computer Science, vol. 1633, Springer, 1999, pp. 60–71. 1, 7

[16] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh (eds.), *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, vol. 185, IOS Press, 2009. 5

[17] Ronen I. Brafman, *A Simplifier for Propositional Formulas with Many Binary Clauses*, IEEE Transactions on Systems, Man, and Cybernetics, Part B **34** (2004), no. 1, 52–59. 2, 3, 13, 24, 27, 80, 92

[18] Randal E. Bryant, *Graph-Based Algorithms for Boolean Function Manipulation*, IEEE Trans. Computers **35** (1986), no. 8, 677–691. 2

[19] Michael Buro and Hans Kleine Büning, *Report on a SAT competition*, Tech. Report Reihe Informatik 110, Universität-Gesamthochschule Paderborn, 1992. 26

[20] Vasek Chvátal, *Resolution Search*, Discrete Applied Mathematics **73** (1997), no. 1, 81–99. 8

[21] Edmund M. Clarke, Armin Biere, Richard Raimi, and Yunshan Zhu, *Bounded Model Checking Using Satisfiability Solving*, Formal Methods in System Design **19** (2001), no. 1, 7–34. 1, 7

[22] Charles J. Colbourn, *The Complexity of Completing Partial Latin Squares*, Discrete Applied Mathematics **8** (1984), no. 1, 25–30. 73

[23] Stephen A. Cook, *The Complexity of Theorem-Proving Procedures*, STOC, ACM, 1971, pp. 151–158. 1, 7, 72

[24] Stephen A. Cook and David G Mitchell, *Finding Hard Instances of the Satisfiability Problem: A Survey*, Satisfiability Problem: Theory and Applications (Du, Gu, and Pardalos, eds.), Dimacs Series in Discrete Mathematics and Theoretical Computer Science, vol. 35, American Mathematical Society, 1997, pp. 1–17. 7

[25] Thomas H. Cormen, Charles Eric Leiserson, Ronald Linn Rivest, and Clifford Stein, second ed., ch. Data Structures for Disjoint Sets, pp. 498–524, MIT Press and McGraw-Hill, 2001. 54

[26] Olivier Coudert, *On Solving Covering Problems*, DAC, 1996, pp. 197–202. 14

[27] James M. Crawford and Larry D. Auton, *Experimental Results on the Crossover Point in Satisfiability Problems*, AAAI, 1993, pp. 21–27. 36

[28] Martin Davis, George Logemann, and Donald W. Loveland, *A Machine Program for Theorem-Proving*, Commun. ACM **5** (1962), no. 7, 394–397. 2, 5, 8, 9, 76

[29] Martin Davis and Hilary Putnam, *A Computing Procedure for Quantification Theory*, J. ACM **7** (1960), no. 3, 201–215. 2, 5, 8, 13, 26

[30] William F. Dowling and Jean H. Gallier, *Linear-Time Algorithms for Testing the Satisfiability of Propositional Horn Formulae*, J. Log. Program. **1** (1984), no. 3, 267–284. 7

[31] Niklas Eén and Armin Biere, *Effective Preprocessing in SAT Through Variable and Clause Elimination*, in Bacchus and Walsh [10], pp. 61–75. 12, 13, 24, 27, 80

[32] Niklas Eén and Niklas Sörensson, *An Extensible SAT-solver*, in Giunchiglia and Tacchella [43], pp. 502–518. 3, 4, 11, 16, 23, 50, 51, 72, 92, 94

[33] Shimon Even, Alon Itai, and Adi Shamir, *On the Complexity of Timetable and Multicommodity Flow Problems*, SIAM J. Comput. **5** (1976), no. 4, 691–703. 26

[34] J. W. Freeman, *Improvements to Propositional Satisfiability Search Algorithms*, Ph.D. thesis, Department of Computer and Information Science, University of Pennsylvania, 1995. 13, 17

[35] Harold N. Gabow, *Searching*, Discrete Math. and its Applications: Handbook of Graph Theory (Jonathan L. Gross and Jay Yellen, eds.), CRC Press, 25 ed., 2003, pp. 953–984. 57

[36] Bernard A. Galler and Michael J. Fischer, *An Improved Equivalence Algorithm*, Commun. ACM **7** (1964), no. 5, 301–303. 55, 94

[37] Giorgio Gallo and Daniele Pretolani, *A New Algorithm for the Propositional Satisfiability Problem*, Discrete Applied Mathematics **60** (1995), no. 1-3, 159–179. 26

[38] Yong Gao and Nathalie Japkowicz (eds.), *Advances in Artificial Intelligence, 22nd Canadian Conference on Artificial Intelligence, Canadian AI 2009, Kelowna, Canada, May 25-27, 2009, Proceedings*, Lecture Notes in Computer Science, vol. 5549, Springer, 2009. 92, 95

[39] Allen Van Gelder, *Toward Leaner Binary-Clause Reasoning in a Satisfiability Solver*, Ann. Math. Artif. Intell. **43** (2005), no. 1, 239–253. 2, 24, 26, 27

[40] Allen Van Gelder and Y. K. Tsuji, *Satisfiability Testing with More Reasoning and Less Guessing*, Second DIMACS Implementation Challenge, D.S. Johnson and M.A. Trick (1993). 27, 76

[41] Roman Gershman and Ofer Strichman, *HaifaSAT: A New Robust SAT Solver*, Haifa Verification Conference (Shmuel Ur, Eyal Bin, and Yaron Wolfsthal, eds.), Lecture Notes in Computer Science, vol. 3875, Springer, 2005, pp. 76–89. 16, 23, 27

[42] Matthew L. Ginsberg, *Dynamic Backtracking*, CoRR **cs.AI/9308101** (1993). 17

[43] Enrico Giunchiglia and Armando Tacchella (eds.), *Theory and Applications of Satisfiability Testing, 6th International Conference, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*, Lecture Notes in Computer Science, vol. 2919, Springer, 2004. 96, 98

[44] C. Gomes and D. Shmoys, *Completing Quasigroups or Latin Squares: A Structured Graph Coloring Problem*, Proceedings of the Computational Symposium on Graph Coloring and its Generalizations (Ithaca, New York, USA) (D. S. Johnson, A. Mehrotra, and M. Trick, eds.), 2002, pp. 22–39. 1, 7

[45] Carla P. Gomes, Bart Selman, and Henry A. Kautz, *Boosting Combinatorial Search Through Randomization*, AAAI/IAAI, 1998, pp. 431–437. 17

[46] Susan L. Graham, Peter B. Kessler, and Marshall K. McKusick, *gprof: a Call Graph Execution Profiler*, SIGPLAN Symposium on Compiler Construction, 1982, pp. 120–126. 78, 84

[47] Jun Gu, *Local Search for Satisfiability (SAT) Problem*, IEEE Transactions on systems, man, and cybernetics **23** (1993), 1108,1129. 2

[48] Harri Haanpää, Matti Järvisalo, Petteri Kaski, and Ilkka Niemelä, *Hard Satisfiable Clause Sets for Benchmarking Equivalence Reasoning Techniques*, JSAT **2** (2006), no. 1-4, 27–46. 2

[49] Bernhard Haeupler, Siddhartha Sen, and Robert Endre Tarjan, *Incremental Topological Ordering and Strong Component Maintenance*, CoRR **abs/0803.0792** (2008). 57

[50] Marijn Heule, Mark Dufour, Joris van Zwieten, and Hans van Maaren, *March_eq: Implementing Additional Reasoning into an Efficient Look-Ahead SAT Solver*, SAT

(Selected Papers (Holger H. Hoos and David G. Mitchell, eds.), Lecture Notes in Computer Science, vol. 3542, Springer, 2004, pp. 345–359. 24, 27

[51] Marijn Heule and Hans van Maaren, *March dl: Adding Adaptive Heuristics and a New Branching Strategy*, JSAT **2** (2006), no. 1-4, 47–59. 11

[52] Hirosi Hitotumatu and Kohei Noshita, *A Technique for Implementing Backtrack Algorithms and its Application*, Inf. Process. Lett. **8** (1979), no. 4, 174–175. 3, 16, 93

[53] John N. Hooker and V. Vinay, *Branching Rules for Satisfiability*, J. Autom. Reasoning **15** (1995), no. 3, 359–383. 11, 22

[54] Daniel Jackson and Mandana Vaziri, *Finding Bugs with a Constraint Solver*, ISSTA, 2000, pp. 14–25. 1, 7

[55] Matti Järvisalo, *Equivalence Checking Hardware Multiplier Designs*, 2007, SAT Competition 2007 benchmark description. Available at `http://www.satcompetition.org/2007/contestants.html`. 86

[56] Matti Järvisalo, Armin Biere, and Marijn Heule, *Blocked Clause Elimination*, TACAS (Javier Esparza and Rupak Majumdar, eds.), Lecture Notes in Computer Science, vol. 6015, Springer, 2010, pp. 129–144. 26

[57] Brigitte Jaumard, Mihnea Stan, and Jacques Desrosiers, *Tabu search and a quadratic relaxation for the satisfiability problem*, Cliques, Coloring and Satisfiability. American Mathematical Society, 1996. Proceedings of the second DIMACS Implementation Challenge, American Mathematical Society, 1996, pp. 457–477. 2

[58] Kenneth A. De Jong and William M. Spears, *Using Genetic Algorithms to Solve NP-Complete Problems*, ICGA (J. David Schaffer, ed.), Morgan Kaufmann, 1989, pp. 124–132. 2

[59] Roberto J. Bayardo Jr. and Robert Schrag, *Using CSP Look-Back Techniques to Solve Real-World SAT Instances*, AAAI/IAAI, 1997, pp. 203–208. 4, 13, 93

[60] Henry A. Kautz, Eric Horvitz, Yongshao Ruan, Carla P. Gomes, and Bart Selman, *Dynamic Restart Policies*, AAAI/IAAI, 2002, pp. 674–681. 17

[61] Henry A. Kautz and Bruce W. Porter (eds.), *Proceedings of the Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on on Innovative Applications of Artificial Intelligence, July 30 - August 3, 2000, Austin, Texas, USA*, AAAI Press / The MIT Press, 2000. 95, 102

[62] Henry A. Kautz and Bart Selman, *Planning as Satisfiability*, ECAI, 1992, pp. 359–363. 1, 7

[63] Donald Ervin Knuth, *Dancing links*, Millenial Perspectives in Computer Science **18** (2009), no. arXiv:cs/0011047. KNUTH MIGRATION 11-2004, 4, Comments: Abstract added by Greg Kuperberg. 3, 16, 93

[64] Arist Kojevnikov, Alexander S. Kulikov, and Grigory Yaroslavtsev, *Finding Efficient Circuits Using SAT-Solvers*, in Kullmann [67], pp. 32–44. 86

[65] Melven R. Krom, *The Decision Problem for a Class of First-Order Formulas in Which all Disjunctions are Binary*, Mathematical Logic Quarterly **13** (1967), no. 1-2, 15–20. 7, 26

[66] Oliver Kullmann, *On a Generalization of Extended Resolution*, Discrete Applied Mathematics **96-97** (1999), 149–176. 26

[67] Oliver Kullmann (ed.), *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, Lecture Notes in Computer Science, vol. 5584, Springer, 2009. 73, 101

[68] Michihiro Kuramochi and George Karypis, *Frequent Subgraph Discovery*, ICDM (Nick Cercone, Tsau Young Lin, and Xindong Wu, eds.), IEEE Computer Society, 2001, pp. 313–320. 73

[69] Michail G. Lagoudakis and Michael L. Littman, *Learning to Select Branching Rules in the DPLL Procedure for Satisfiability*, Electronic Notes in Discrete Mathematics **9** (2001), 344–359. 11, 22

[70] Tracy Larrabee, *Test Pattern Generation Using Boolean Satisfiability*, IEEE Trans. on CAD of Integrated Circuits and Systems **11** (1992), no. 1, 4–15. 1, 7, 26

[71] Leonid Anatolievich Levin, *Universal Sequential Search Problems*, Probl. Peredachi Inf. **9** (1973), 115–116. 1, 7, 72

[72] Chu Min Li, *Integrating Equivalency Reasoning into Davis-Putnam Procedure*, in Kautz and Porter [61], pp. 291–296. 2, 3, 26, 76

[73] Chu Min Li and Anbulagan, *Heuristics Based on Unit Propagation for Satisfiability Problems*, IJCAI (1), 1997, pp. 366–371. 22

[74] ———, *Look-Ahead Versus Look-Back for Satisfiability Problems*, CP (Gert Smolka, ed.), Lecture Notes in Computer Science, vol. 1330, Springer, 1997, pp. 341–355. 11

[75] Donald W Loveland, *Automated Theorem Proving: A Logical Basis (Fundamental Studies in Computer Science)*, sole distributor for the U.S.A. and Canada, Elsevier North-Holland, 1978. 12, 26, 80

[76] Inês Lynce and João P. Marques Silva, *Efficient Data Structures for Backtrack Search SAT Solvers*, Ann. Math. Artif. Intell. **43** (2005), no. 1, 137–152. 13, 14, 76

[77] Heikki Mannila and Esko Ukkonen, *The Set Union Problem with Backtracking*, ICALP (Laurent Kott, ed.), Lecture Notes in Computer Science, vol. 226, Springer, 1986, pp. 236–243. 4, 51, 54, 55

[78] Bertrand Mazure, Lakhdar Sais, and Éric Grégoire, *Tabu Search for SAT*, AAAI/IAAI, 1997, pp. 281–285. 2

[79] David A. Mcallester, *An Outlook on Truth Maintenance*, AI Memp (1980), no. 551. 17

[80] David Moews, *Pebbling Graphs*, J. Comb. Theory, Ser. B **55** (1992), no. 2, 244–252. 86

[81] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, *Chaff: Engineering an Efficient SAT Solver*, DAC, ACM, 2001, pp. 530–535. 4, 11, 13, 15, 16, 17, 19, 23, 35, 36, 93

[82] Ming Ouyang, *How Good Are Branching Rules in DPLL?*, Discrete Applied Mathematics **89** (1998), no. 1-3, 281–286. 11, 22

[83] ———, *Implementations of the DPLL Algorithm*, Ph.D. thesis, Rutgers University, 1999. 26

[84] Christof Paar and Jan Pelzl, *Understanding Cryptography - A Textbook for Students and Practitioners*, Springer, 2010. 85

[85] David J. Pearce and Paul H. J. Kelly, *A Dynamic Topological Sort Algorithm for Directed Acyclic Graphs*, J. Exp. Algorithmics **11** (2007). 57

[86] Natasa Przulj, Derek G. Corneil, and Igor Jurisica, *Efficient Estimation of Graphlet Frequency Distributions in Protein-Protein Interaction Networks*, Bioinformatics **22** (2006), no. 8, 974–980. 73

[87] Liam Roditty and Uri Zwick, *A Fully Dynamic Reachability Algorithm for Directed Graphs with an Almost Linear Update Time*, STOC (László Babai, ed.), ACM, 2004, pp. 184–191. 57

[88] Thomas J. Schaefer, *The Complexity of Satisfiability Problems*, STOC, ACM, 1978, pp. 216–226. 7

[89] Bart Selman and Henry A. Kautz, *An Empirical Study of Greedy Local Search for Satisfiability Testing*, AAAI, 1993, pp. 46–51. 2

[90] Merlijn Sevenster, *Battleships as a Decision Problem*, ICGA Journal **27** (2004), no. 3, 142–149. 86

[91] Mary Sheeran and Gunnar Stålmarck, *A Tutorial on Stålmarcks's Proof Procedure for Propositional Logic*, FMCAD (Ganesh Gopalakrishnan and Phillip J. Windley, eds.), Lecture Notes in Computer Science, vol. 1522, Springer, 1998, pp. 82–99. 2

[92] João P. Marques Silva and Karem A. Sakallah, *GRASP: A Search Algorithm for Propositional Satisfiability*, IEEE Trans. Computers **48** (1999), no. 5, 506–521. 4, 11, 13, 17, 21, 22, 93

[93] Laurent Simon, Daniel Le Berre, and Edward A. Hirsch, *The SAT 2002 competition*, Ann. Math. Artif. Intell. **43** (2005), no. 1, 307–342. 73

[94] Evgeny S. Skvortsov and Yulia Zaks, *Synchronizing Random Automata*, Discrete Mathematics & Theoretical Computer Science **12** (2010), no. 4, 95–108. 86

[95] Tom A. Snijders, Philippa E. Pattison, Garry L. Robins, and Mark S. Handcock, *New Specifications for Exponential Random Graph Models*, Sociological Methodology **36** (2006), no. 1, 99–153. 73

[96] Robert Endre Tarjan, *Depth-First Search and Linear Graph Algorithms*, SIAM J. Comput. **1** (1972), no. 2, 146–160. 3, 4, 27, 51, 52, 57

[97] Vijay V. Vazirani, *Approximation Algorithms*, Springer, 2001. 85

[98] Bartel L. Van Der Waerden, *Beweis Einer Baudetschen Vermutung*, Nieuw Archief voor Wiskunde **15** (1927), 212–216. 86

[99] Joost P. Warners and Hans van Maaren, *A Two-Phase Algorithm for Solving a Class of Hard Satisfiability Problems*, Oper. Res. Lett. **23** (1998), no. 3-5, 81–88. 2, 3

[100] Ke Xu, Frédéric Boussemart, Fred Hemery, and Christophe Lecoutre, *Random Constraint Satisfaction: Easy Generation of Hard (Satisfiable) Instances*, Artif. Intell. **171** (2007), no. 8-9, 514–534. 73

[101] H. Zhang and M. E. Stickel, *An Efficient Algorithm for Unit Propagation*, Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH'96) (Fort Lauderdale (Florida USA)), 1996. 36

[102] Hantao Zhang, *SATO: An Efficient Propositional Prover*, CADE (William McCune, ed.), Lecture Notes in Computer Science, vol. 1249, Springer, 1997, pp. 272–275. 14

[103] Hantao Zhang and Mark E. Stickel, *Implementing the Davis-Putnam Method*, J. Autom. Reasoning **24** (2000), no. 1/2, 277–296. 73

[104] Lintao Zhang, *Searching for Truth: Techniques for Satisfiability of Boolean Formulas*, Ph.D. thesis, Princeton University, Princeton, NJ, USA, 2003, AAI3102236. 13

[105] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik, *Efficient Conflict Driven Learning in Boolean Satisfiability Solver*, ICCAD, 2001, pp. 279–285. 21

Appendix A

Experimental Results

In this Appendix we presents the results for different families in the industrial and crafted categories for MiniSat and Eq-MiniSat. The instances are chosen from the SAT competition benchmarks [1]. All the experiments have been run on an AMD Opteron (tm) Processor 875, 2200 GHZ, 64GB RAM machine. The cut-off time is set to 7200 seconds. If both solvers are unable to solve an instance by the cut-off time, then the statistics for that instance is not shown in the results.

In the result tables throughout the chapter 'S', 'U' and 'I' in the result (RSL) column stand for 'Satisfiable', 'Unsatisfiable', and 'Unsolved', respectively. For every instance the following information is provided: number of variables (#Vars), number of original clauses (#Cls), number of branches (#Bran.), number of propagations (#Prop.), CPU runtime in seconds (RT), and the result (RSL) in two hours for both solvers MiniSat and Eq-MiniSat.

In Section A.1, the results for application based families are presented. Section A.2 presents the results for crafted instances.

A.1 Application Based Instances

The application based families include 2-dimensional strip packing or 2SPP, advanced encryption standard or AES, and equivalence checking multiplier.

Table A.1: 2-Dimensional Strip Packing instances.

Instance Name	#Vars	#Cls
E02F17	6664	69700
E03N17	6664	93544
E04F19	9044	295685
E04N18	7794	120068
E05F18	7794	126826
E05X15	4740	41379
korf-15	4740	45569
korf-17	6664	89966

Table A.2: Results for 2-Dimensional Strip Packing family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
E02F17	1.2E6	1.4E8	172.2	U	-	-	-	I
E03N17	6.8E4	6.9E6	3.5	U	3.2E4	4.5E6	18.8	U
E04F19	1.2E6	1.2E8	166.8	S	1.9E5	2.7E7	82	S
E04N18	5.2E3	5.3E5	0.2	U	4.4E3	6.6E5	12.8	U
E05F18	5.9E5	6.2E7	62.2	S	7.7E4	1.0E7	28.1	S
E05X15	1.8E5	1.9E7	15.7	U	1.9E5	2.8E7	90.8	U
korf-15	1.1E6	1.3E8	226.1	U	5.0E5	7.6E7	229.3	U
korf-17	1.5E7	2.3E9	5396.3	U	-	-	-	I
Number of solved for MiniSat: 8								
Number of solved for Eq-MiniSat: 6								

2-Dimensional Strip Packing or 2SPP In this family we have 11 instances. Two instances 'E07N15' and 'E15N15' are solved in less than a second and are not shown in the table. The instance 'korf-18' was not solved by either of the solvers in two hours. The instances information and results for the other instances in this family are shown in Table A.1 and Table A.2. MiniSat outperforms Eq-MiniSat by solving 10 instances versus 6 for Eq-MiniSat. The runtime for the instances solved by both solvers are 474.5 and 461.8 for MiniSat and Eq-MiniSat, respectively.

Advanced encryption standard There are five instances in this family. The unsolved instances in two hours are 'aes_32_4_keyfind_1', 'aes_32_5_keyfind_1'. MiniSat solves one instance more than Eq-MiniSat. The runtime for the instances solved by both are 3.2 and 14.3 for MiniSat

Table A.3: Advanced encryption standard or AES instances.

Instance Name	#Vars	#Cls
aes_32_1_keyfind_1	300	1016
aes_32_2_keyfind_1	504	1840
aes_32_3_keyfind_1	708	2664

Table A.4: Results for advanced encryption standard or AES family.

Instance	MiniSat				Eq-MiniSat			
Name	#Branches	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
aes_32_1_keyfind_1	9.8E2	4.7E4	0.0	S	3.4E3	1.2E5	0.1	S
aes_32_2_keyfind_1	1.0E5	1.2E7	3.2	S	2.3E5	1.5E7	14.3	S
aes_32_3_keyfind_1	1.2E7	1.5E9	564.8	S	-	-	-	I
number of solved for MiniSat: 3								
number of solved for Eq-MiniSat: 2								

and Eq-MiniSat, respectively. Table A.3 and Table A.4 presents the instance info and results for this family.

Equivalence checking multiplier design The equivalence checking multiplier design family has seven instances. MiniSat, and Eq-MiniSat solve 4 and 3 instances respectively. Unsolved instances are 'eq.atree.braun.11.unsat', 'eq.atree.braun.12.unsat', 'eq.atree.braun.13.unsat'. The runtime for the instances solved by both are 341.9 and 753.5 for MiniSat and Eq-MiniSat, respectively. The statistical results are shown in Table A.5 and A.6.

Table A.5: Equivalence checking multiplier instances.

Instance Name	#Vars	#Cls
eq.atree.braun.10.unsat	1111	3756
eq.atree.braun.7.unsat	505	1696
eq.atree.braun.8.unsat	684	2300
eq.atree.braun.9.unsat	892	3006

Table A.6: Results for equivalence checking multiplier family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
braun.10	2.1E7	3.6E9	2133.2	U	-	-	-	I
braun.7	2.4E5	2.3E7	8.7	U	9.4E4	8.8E6	11.7	U
braun.8	1.2E6	1.6E8	61.2	U	5.8E5	6.3E7	96.2	U
braun.9	3.7E6	5.6E8	272.0	U	2.7E6	3.6E8	645.6	U
number of solved for MiniSat: 4								
number of solved for Eq-MiniSat: 3								

Due to limited space, the instance names are shortened.

A.2 Crafted Instances

The crafted families in this section are SGI, FRB, QG, QWH, EZFACT, GT, RBSAT, VMPC, AUTOMATA, BATTLESHIP, PEBBLING, VDW, and MOD.

SGI There are 28 instances in the SGI family. While MiniSat is able to solve one of the instances in two hours, Eq-MiniSat solves 25 instances. The unsolved instance in two hours are 'srhd-sgi-m42-q585-n40-p15-s54275047', 'srhd-sgi-m52-q1041.25-n50-p15-s99099953', and '-srhd-sgi-m52-q918.75-n50-p30-s52376212'. Table A.8 and Table A.7 presents the statistical data for this family.

Forced RB Model The FRB family has 20 instances. None of the solvers are able to solve 'frb45-21-1' and 'frb45-21-2'. Eq-MiniSat solves the rest of 18 instances, while MiniSat solves 14 instances. The runtime for instances solved by both solvers are 6,340.1 and 5904.2 for MiniSat and Eq-MiniSat respectively. The instances information and results are shown in Table A.9 and Table A.10.

Quasigroup The QG family has twenty instances. The unsolved instances for both solvers are: 'QG7-gensys-icl100.sat05-3226.reshuffled-07', 'QG7-gensys-ukn003.sat05-3346.reshuffled-07', 'QG7-gensys-ukn003.sat05-3346.reshuffled-07' , 'QG7a-gensys-icl009.sat05-3830.reshuffled-07', 'QG7a-gensys-ukn002.sat05-3842.reshuffled-07', 'QG7a-gensys-ukn009.sat05-3849.reshuffled-07'. All of the remaining 14 instances are solved by Eq-MiniSat. MiniSat

Table A.7: Results for subgraph isomorphism (SGI) family.

Instance	#Vars	#Cls
srhd-sgi-m27-q225-n25-p15-s58217873	550	35586
srhd-sgi-m27-q225-n25-p30-s70617701	671	50773
srhd-sgi-m27-q255-n25-p15-s2076598	545	29734
srhd-sgi-m27-q255-n25-p30-s39712998	666	45238
srhd-sgi-m32-q326.25-n30-p15-s44266159	792	67279
srhd-sgi-m32-q326.25-n30-p30-s48700942	943	93431
srhd-sgi-m32-q369.75-n30-p15-s59317012	894	82294
srhd-sgi-m32-q369.75-n30-p30-s25693430	952	84809
srhd-sgi-m37-q446.25-n35-p15-s25120921	1106	129272
srhd-sgi-m37-q446.25-n35-p30-s33692332	1285	166723
srhd-sgi-m37-q505.75-n35-p15-s48276711	1244	150604
srhd-sgi-m37-q505.75-n35-p30-s59841049	1295	148400
srhd-sgi-m42-q585-n40-p30-s19690873	1638	253275
srhd-sgi-m42-q663-n40-p15-s72490337	1649	256038
srhd-sgi-m42-q663-n40-p30-s67876261	1634	212885
srhd-sgi-m47-q742.5-n45-p15-s28972035	1985	409171
srhd-sgi-m47-q742.5-n45-p30-s17570390	2088	405748
srhd-sgi-m47-q841.5-n45-p15-s16393788	1981	333838
srhd-sgi-m47-q841.5-n45-p30-s84954709	2115	356340
srhd-sgi-m52-q1041.25-n50-p30-s30907550	2561	490309
srhd-sgi-m52-q918.75-n50-p15-s98191766	2377	566749
srhd-sgi-m62-q1327.5-n60-p15-s1351253	3420	1120760
srhd-sgi-m62-q1327.5-n60-p30-s52708253	3672	1163952
srhd-sgi-m62-q1504.5-n60-p15-s80450670	3510	974465
srhd-sgi-m62-q1504.5-n60-p30-s88600538	3651	926696

Table A.8: Results for subgraph isomorphism (SGI) family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
s58217873	8.0E5	4.4E7	601.7	S	1.3E1	5.5E2	0.1	S
s70617701	-	-	-	I	2.6E5	1.5E7	113.3	S
s2076598	-	-	-	I	2.7E6	1.0E8	1092.2	S
s39712998	-	-	-	I	1.3E1	6.7E2	0.1	S
s44266159	-	-	-	I	9.0E0	7.9E2	0.1	S
s48700942	-	-	-	I	2.2E5	1.5E7	146.3	S
s59317012	-	-	-	I	1.7E1	1.1E3	0.1	S
s25693430	-	-	-	I	1.5E1	9.5E2	0.1	S
s25120921	-	-	-	I	1.3E1	1.1E3	0.2	S
s33692332	-	-	-	I	8.0E0	1.3E3	0.2	S
s48276711	-	-	-	I	1.2E1	1.2E3	0.2	S
s59841049	-	-	-	I	1.1E1	1.3E3	0.2	S
s19690873	-	-	-	I	1.2E1	1.6E3	0.4	S
s72490337	-	-	-	I	1.4E1	1.6E3	0.4	S
s67876261	-	-	-	I	1.7E1	1.6E3	0.4	S
s28972035	-	-	-	I	1.4E1	2.0E3	1.0	S
s17570390	-	-	-	I	1.5E1	2.1E3	0.7	S
s16393788	-	-	-	I	1.9E1	2.0E3	0.8	S
s84954709	-	-	-	I	1.5E1	2.1E3	0.6	S
s30907550	-	-	-	I	1.7E1	2.6E3	0.7	S
s98191766	-	-	-	I	1.4E1	2.4E3	0.7	S
s1351253	-	-	-	I	1.1E1	3.4E3	1.6	S
s52708253	-	-	-	I	2.0E1	3.7E3	1.6	S
s80450670	-	-	-	I	1.9E1	3.5E3	1.5	S
s88600538	-	-	-	I	1.8E1	3.7E3	1.5	S
number of solved for MiniSat: 1								
number of solved for Eq-MiniSat: 25								

Table A.9: FRB family instances.

Instance Name	#Vars	#Cls
frb30-15-1	450	19084
frb30-15-2	450	19084
frb30-15-3	450	19084
frb30-15-4	450	19084
frb30-15-5	450	19084
frb35-17-1	595	29707
frb35-17-2	595	29707
frb35-17-3	595	29707
frb35-17-4	595	29707
frb35-17-5	595	29707
frb40-19-1	760	43780
frb40-19-2	760	43780
frb40-19-3	760	43780
frb40-19-4	760	43780
frb40-19-5	760	43780
frb45-21-3	945	61855
frb45-21-4	945	61855
frb45-21-5	945	61855

solves 11 instances. The runtime for the instances solved by both solvers are 9,365.7 and 6,194.9 for MiniSat and Eq-MiniSat, respectively. Table A.11 and A.12 shows the statistical results.

Quasigroup With Holes The QWH family has five instances with less than 10,000 variables. Minisat solves four instances and Eq-MiniSat five instances. The runtime for the common solved instances are 3487.4 and 1788.6 for MiniSat and Eq-MiniSat, respectively. The statistical data for this family is presented in Table A.13 and Table A.14.

Factorization There are 39 instances in the family. Table A.15 and Table A.16 shows the statistics for the instances for which at least one of the solvers is able to solve the instance in less than two hours. Due to space concerns the instances with runtime less than a second are not shown in the table. Unsolved instance in two hours are 'ezfact64_3', 'ezfact64_4', 'ezfact64_7', 'ezfact64_8', 'ezfact64_9'. Eq-MiniSat solves 34 instances, while MiniSat solves 30 instances. The runtime for the instances solved by both solvers are 525.6 and 1846.3

Table A.10: Results for FRB family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
frb30-15-1	1.7E4	5.3E5	0.6	S	2.3E4	1.1E6	2.6	S
frb30-15-2	8.5E4	3.4E6	14.4	S	2.5E4	1.3E6	3.1	S
frb30-15-3	1.2E4	4.2E5	0.4	S	2.0E4	1.1E6	2.4	S
frb30-15-4	3.1E3	6.0E4	0.0	S	1.7E4	8.8E5	1.8	S
frb30-15-5	6.7E2	1.5E4	0.0	S	1.1E4	6.3E5	1.2	S
frb35-17-1	2.4E4	8.4E5	1.0	S	7.4E4	4.7E6	16.9	S
frb35-17-2	1.9E5	7.4E6	43.4	S	9.0E4	5.5E6	19.3	S
frb35-17-3	3.9E5	1.9E7	158.5	S	2.0E4	1.2E6	2.5	S
frb35-17-4	3.4E4	1.3E6	3.0	S	4.3E4	2.8E6	8.1	S
frb35-17-5	4.2E5	2.1E7	191.2	S	8.1E1	2.9E3	0.0	S
frb40-19-1	4.0E5	2.1E7	206.9	S	2.4E5	1.9E7	93.3	S
frb40-19-2	5.2E5	2.9E7	326.4	S	1.8E5	1.4E7	58.3	S
frb40-19-3	2.0E6	1.1E8	1938.2	S	5.3E6	4.1E8	5665.7	S
frb40-19-4	2.6E6	1.5E8	3456.1	S	1.1E5	7.8E6	29.3	S
frb40-19-5	-	-	-	I	1.2E6	8.9E7	663.5	S
frb45-21-3	-	-	-	I	2.8E6	2.4E8	2348.5	S
frb45-21-4	-	-	-	I	8.8E5	7.3E7	569.1	S
frb45-21-5	-	-	-	I	3.6E6	3.0E8	3742.8	S
number of solved for MiniSat: 14								
Number of cases solved by Eq-MiniSat: 18								

Table A.11: Quasigroup family instances.

Instance	#Vars	#Cls
QG7-dead-dnd001.sat05-3419.reshuffled-07	1040	13020
QG7-dead-dnd002.sat05-3108.reshuffled-07	1602	14784
QG7-dead-dnd005.sat05-3111.reshuffled-07	502	11816
QG7a-gensys-icl004.sat05-3825.reshuffled-07	2401	15960
QG-gensys-brn008.sat05-2685.reshuffled-07	1467	7521
QG-gensys-icl003.sat05-2715.reshuffled-07	1472	7737
QG7-gensys-icl001.sat05-2926.reshuffled-07	432	14889
QG7a-gensys-brn004.sat05-3669.reshuffled-07	2435	16016
QG7a-gensys-brn100.sat05-3765.reshuffled-07	2901	18243
QG7a-gensys-ukn001.sat05-3841.reshuffled-07	2737	18375
QG7a-gensys-ukn005.sat05-3845.reshuffled-07	2765	18046
QG8-gensys-ukn005.sat05-3584.reshuffled-07	1133	56210
gensys-ukn002.sat05-2744.reshuffled-07	2129	8961
pmg-11-U.sat05-3939.reshuffled-07	169	562

Table A.12: Results for Quasigroup family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
dnd001	-	-	-	I	1.1E7	6.7E8	3367.8	U
dnd002	-	-	-	I	3.3E7	7.1E8	4686.9	U
dnd005	-	-	-	I	5.6E6	3.2E8	1623.5	U
icl004	2.1E7	5.9E8	3338.2	U	4.7E6	1.5E8	1061.9	U
QG-*-brn008	7.4E6	8.8E7	196.1	U	5.4E6	4.4E7	154.3	U
icl003	1.2E7	1.3E8	327.5	U	3.9E6	3.0E7	101.9	U
icl001	5.5E5	3.6E7	60.6	U	6.6E5	3.8E7	153.4	U
brn004	1.9E6	2.4E7	74.4	S	1.0E6	1.2E7	49.6	S
brn100	1.2E6	1.2E7	34.4	S	2.2E6	2.6E7	131.9	S
ukn001	2.3E6	3.6E7	122.9	S	8.9E5	1.2E7	54.6	S
QG7a-*-ukn005	1.3E6	1.4E7	39.4	S	2.1E6	2.9E7	145.4	S
QG8-*-ukn005	3.5E5	2.1E7	133.1	U	3.2E5	1.3E7	121.8	U
ukn002	7.1E6	7.3E7	171.9	U	3.0E6	2.2E7	79.6	U
pmg-11	1.8E8	7.3E9	4867.2	U	8.0E7	3.0E9	4140.5	U
Number of cases solved by MiniSat: 11								
Number of cases solved by Eq-MiniSat: 14								

Table A.13: Quasigroup With Holes Instances.

Instance	#Vars	#Cls
qwh.35.405.shuffled-as.sat03-1651	1597	10658
qwh.40.528.shuffled-as.sat03-1652	2511	18906
qwh.40.544.shuffled-as.sat03-1653	2843	22558
qwh.40.560.shuffled-as.sat03-1654	3100	26345
qwh.50.1250.shuffled-as.sat03-1655	16719	331272

Table A.14: Results for Quasigroup With Holes family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
qwh.35.405	5.4E4	1.0E7	5.1	S	3.0E4	6.2E6	19.6	S
qwh.40.528	-	-	-	I	6.2E5	1.2E8	472.9	S
qwh.40.544	1.3E7	2.0E9	3147.1	S	1.3E6	2.5E8	983.0	S
qwh.40.560	1.4E6	1.4E8	136.3	S	8.6E5	1.6E8	637.7	S
qwh.50.1250	3.5E6	9.8E7	198.9	S	2.3E5	2.2E7	148.3	S
Number of cases solved by MiniSat: 4								
Number of cases solved by Eq-MiniSat: 5								

Due to limited space, the instance names are shortened.

Table A.15: Factorization family instances.

Instance	#Vars	#Cls
ezfact48_1.shuffled	1729	11001
ezfact48_10.shuffled	1729	11001
ezfact48_2.shuffled	1729	11001
ezfact48_3.shuffled	1729	11001
ezfact48_4.shuffled	1729	11001
ezfact48_5.shuffled	1729	11001
ezfact48_6.shuffled	1729	11001
ezfact48_7.shuffled	1729	11001
ezfact48_8.shuffled	1729	11001
ezfact48_9.shuffled	1729	11001
ezfact64_1.shuffled	3073	19785
ezfact64_2.shuffled	3073	19785
ezfact64_5.shuffled	3073	19785
ezfact64_6.shuffled	3073	19785

seconds for MiniSat and Eq-MiniSat, respectively.

Ordering All the ten instances in the family are solved by Eq-MiniSat, while MiniSat solves 7 instances. Table A.17 and Table A.18 show the statistical results for this family. The runtime for MiniSat and Eq-MiniSat are 2591 and 1088.4, respectively.

Model RB There are 46 instances in the family. Eq-MiniSat and MiniSat, respectively, solve 6 and 2 instances in two hours. Table A.19 and Table A.20 present the statistical results for this family.

VMPC The VMPC family has six instances. The unsolved instances are 'vmpc_34.renamed-as.sat05-1926', 'vmpc_35.renamed-as.sat05-1921', 'vmpc_36.renamed-as.sat05-1922'. Eq-MiniSat is able to solve the remaining 3 instances while MiniSat solves 2 instances. Table A.21 and Table A.22 shows the statistical results for this family.

MOD circuits The Mod circuits family has seven instances. There are four unsolved instances for both solvers in two hours: 'mod3block_2vars_9gates_u2_autoenc', 'mod3block_2vars_10gates_u2_autoenc', 'mod3block_2vars_11gates_u2_autoenc', 'mod3block_4vars_11gat-

Table A.16: Results for Factorization family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
ezfact48_1	2.9E4	1.4E7	3.7	U	3.7E4	1.3E7	14.3	U
ezfact48_10	4.1E5	1.3E8	53.3	U	7.9E5	2.7E8	354.5	U
ezfact48_2	7.3E4	2.1E7	7.1	U	5.2E4	1.8E7	19.2	U
ezfact48_3	9.6E4	4.1E7	12.5	U	1.4E5	4.0E7	53.3	U
ezfact48_4	1.7E5	7.6E7	24.2	U	3.1E5	8.9E7	117.9	U
ezfact48_5	3.9E5	1.4E8	52.3	U	3.7E5	1.2E8	151.1	U
ezfact48_6	2.2E5	7.3E7	26.7	U	8.2E5	2.5E8	361.8	U
ezfact48_7	5.9E5	2.2E8	93.9	U	8.0E5	2.6E8	355.2	U
ezfact48_8	7.2E5	2.6E8	115.5	U	5.5E5	1.7E8	223.9	U
ezfact48_9	8.6E5	3.2E8	136.4	U	4.5E5	1.4E8	195.1	U
ezfact64_1	-	-	-	I	1.4E6	6.7E8	1133.4	S
ezfact64_2	-	-	-	I	1.3E6	5.7E8	1024.3	S
ezfact64_5	-	-	-	I	3.1E6	1E9	2022.9	S
ezfact64_6	-	-	-	I	4.2E6	1.4E9	2022.9	S
Number of cases solved by MiniSat: 31								
Number of cases solved by Eq-MiniSat: 35								

Table A.17: Ordering family instances.

Instance	#Vars	#Cls
counting-clqcolor-unsat-set-b-clqcolor-08-06-07.sat05-1257.reshuffled-07	132	1527
counting-easier-fphp-012-010.sat05-1214.reshuffled-07	120	1212
counting-easier-php-012-010.sat05-1172.reshuffled-07	120	672
gt-ordering-sat-gt-030.sat05-1295.reshuffled-07	900	24824
gt-ordering-unsat-gt-025.sat05-1306.reshuffled-07	625	14125
gt-ordering-unsat-gt-030.sat05-1307.reshuffled-07	900	24825
gt-ordering-unsat-gt-035.sat05-1308.reshuffled-07	1225	39900
sat-strips-gripper-10t19.sat05-1143.reshuffled-07	3390	53225
sat-strips-gripper-12t23.sat05-1144.reshuffled-07	4940	93221
unsat-logistics-rotate-09t5.sat05-1139.reshuffled-07	4336	214585

Table A.18: Results for Ordering family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
sat05-1257	2.1E6	2.3E7	22.0	U	5.9E5	7.7E6	10.5	U
sat05-1214	2.8E7	3.8E8	1370.4	U	1.1E7	1.1E8	384.3	U
sat05-1172	2.9E7	3.5E8	831.3	U	1.4E7	1.7E8	444.4	U
sat05-1295	-	-	-	I	1.3E6	2.1E7	94.3	U
sat05-1306	1.2E7	1.4E8	251.8	U	9.2E5	1.3E7	52.4	U
sat05-1307	-	-	-	I	1.5E7	2.7E8	1399.2	U
sat05-1308	-	-	-	I	5.1E7	1.0E9	6659.5	U
sat05-1143	3.1E3	2.4E5	0.1	S	3.8E4	7.9E6	16.1	S
sat05-1144	1.7E4	1.8E6	0.4	S	1.8E5	4.9E7	143.2	S
sat05-1139	2.6E5	2.9E7	115.0	U	1.0E5	1.3E7	37.5	U
Number of cases solved by MiniSat: 7								
Number of cases solved by Eq-MiniSat: 10								

Due to limited space, the instance names are shortened.

Table A.19: Model RB family instances.

Instance	#Vars	#Cls
rbsat-v760c43649gyes8	760	43649
rbsat-v760c43649g7	760	43649
rbsat-v945c61409g3	945	61409
rbsat-v760c43649g5	760	43649
rbsat-v760c43649g7	760	43649
rbsat-v945c61409g3	945	61409
rbsat-v945c61409g4	945	61409

Table A.20: Results for Model RB family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
v760c43649gyes8	-	-	-	I	4.1E6	2.4E8	2345.6	S
v760c43649g7	-	-	-	I	2.4E5	1.6E7	84.4	S
v945c61409g3	-	-	-	I	6.5E6	4.4E8	4712.1	S
v760c43649g5	2.4E6	1.4E8	2620.9	S	1.9E5	1.3E7	67.0	S
v760c43649g7	8.6E5	4.3E7	509.4	S	-	-	-	I
v945c61409g3	-	-	-	I	6.5E6	4.4E8	4712.1	S
v945c61409g4	-	-	-	I	4.9E6	4.0E8	4503.5	S
Number of cases solved by MiniSat: 2								
Number of cases solved by Eq-MiniSat: 6								

Due to limited space, the instance names are shortened.

Table A.21: VMPC family instances.

Instance	#Vars	#Cls
vmpc_25.renamed-as.sat05-1913	625	76775
vmpc_29.renamed-as.sat05-1916	841	120147
vmpc_32.renamed-as.sat05-1919	1024	161664

Table A.22: Results for VMPC family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
vmpc_25	5.2E5	2.4E7	134.4	S	5.3E5	4.1E7	187.2	S
vmpc_29	7.8E6	4.5E8	5556.9	S	1.3E6	1.2E8	614.4	S
vmpc_32	-	-	-	I	3.9E6	4.3E8	2917.6	S
Number of cases solved by MiniSat: 2								
Number of cases solved by Eq-MiniSat: 3								

Due to limited space, the instance names are shortened.

Table A.23: MOD Circuits family instances.

Instance	#Vars	#Cls
mod3_4vars_6gates	289	33900
mod3block_3vars_9gates_restr	784	209392
owp_4vars_5gates	267	23747

Table A.24: Results for MOD Circuits family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
mod3_4	1.5E7	3.8E8	1456.1	U	2.8E7	7.0E8	1500.6	U
mod3block	5.4E6	2.0E8	1098.9	S	1.4E5	5.1E6	4.4	S
owp_4	2.3E3	3.3E4	0.1	U	1.9E3	2.3E4	0.0	U
Number of cases solved by MiniSat: 3								
Number of cases solved by Eq-MiniSat: 3								

Due to limited space, the instance names are shortened.

es_b2'. The runtime for other instances are 2555.1 and 1505 for MiniSat and Eq-MiniSat, respectively. Table A.23 and Table A.24 shows the statistical results.

Automata synchronization This family has ten instances with number of variables less than 10,000. Unsolved instances in two hours are 'crn_20_360_u', and 'crn_20_361_s'. MiniSat solves all of the remaining instances, while Eq-MiniSat is not able to solve 'rnd_150_29_u'. The runtime for the instances solved by both solvers are 505.3 and 5109.8. Table A.25 and Table A.26 show the statistical data for this family.

Battleship This family has 24 instances. MiniSat outperforms Eq-MiniSat by solving 19 versus 2. The unsolved instances are 'battleship-13-13-unsat', 'battleship-14-14-unsat', ' battleship-15-15-unsat', 'battleship-16-16-unsat', 'battleship-17-33-sat'. Table A.27 and Table A.28 shows the statistical data.

Graph Pebbling There are ten instances in the family. MiniSat is able to solve all of them in two hours, while Eq-MiniSat solves seven instances. Table A.29 and Table A.30 shows the statistical results for these instances. The runtime for the instances solved by both solvers

Table A.25: Automata Synchronization family instances.

Instance	#Vars	#Cls
crn_11_100_s	1300	2355
crn_11_99_u	1287	2332
rnd_100_27_s	2754	10377
rnd_100_28_s	2856	10578
rnd_100_28_u	2856	10578
rnd_100_32_s	3264	11382
rnd_150_29_u	4408	19904
rnd_150_42_s	6384	23817

Table A.26: Results for Automata Synchronization family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
crn_11_100	1.6E6	3.1E7	9.4	S	4.2E6	1.1E8	77.4	S
crn_11_99	1.5E6	3.2E7	9.7	U	6.8E6	1.7E8	128.7	U
rnd_100_27	2.1E5	8.2E7	11.1	S	2.4E5	7.0E7	73.1	S
rnd_100_28	9.0E5	3.0E8	41.3	S	2.5E5	8.0E7	83.2	S
rnd_100_28	6.5E6	2.0E9	322.8	U	7.5E6	2.6E9	3007.9	U
rnd_100_32	1.1E6	3.6E8	52.6	S	2.6E6	9.1E8	1062.6	S
rnd_150_29	1.2E7	6.1E9	1031	U	-	-	-	I
rnd_150_42	8.7E5	3.1E8	58.4	S	2.0E6	4.0E8	676.9	S
Number of cases solved by MiniSat: 8								
Number of cases solved by Eq-MiniSat: 7								

Due to limited space, the instance names are shortened.

are 26.7 and 758.9 seconds for MiniSat and Eq-MiniSat, respectively.

Van Der Waerden numbers The VDW family has twelve instances. While MiniSat solves all the instances in two hours, Eq-MiniSat is not able to solve three instances. The statistical data is shown in Table A.31 and Table A.32. The runtime for the instances solved by both solvers are 14,921.8 and 16,289.4 seconds for MiniSat and Eq-MiniSat, respectively.

Table A.27: Battleship family instances.

Instance	#Vars	#Cls
battleship-10-10-unsat	100	550
battleship-10-17-sat	170	865
battleship-10-18-sat	180	910
battleship-10-19-sat	190	955
battleship-11-11-unsat	121	726
battleship-11-21-sat	231	1276
battleship-12-12-unsat	144	936
battleship-12-23-sat	276	1662
battleship-14-26-sat	364	2562
battleship-14-27-sat	378	2653
battleship-15-29-sat	435	3270
battleship-16-31-sat	496	4777
battleship-24-57-sat	1368	16308
battleship-5-8-unsat	40	105
battleship-6-9-unsat	54	171
battleship-7-12-unsat	84	301
battleship-7-13-sat	91	322
battleship-8-15-sat	120	484
battleship-9-17-sat	153	693

Table A.28: Results for battleship family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
10-10	2.3E5	4.1E6	2.5	U	4.9E5	6.5E6	9.9	U
10-17	3.3E5	3.8E6	4.7	S	-	-	-	I
10-18	7.8E2	6.6E3	0.0	S	-	-	-	I
10-19	4.9E2	3.7E3	0.0	S	-	-	-	I
11-11	1.5E7	2.5E8	334.2	U	6.7E6	8.9E7	182.9	U
11-21	3.0E6	3.1E7	75.0	S	-	-	-	I
12-12	3.1E7	5.8E8	726.5	U	-	-	-	I
12-23	2.4E3	1.6E4	0.0	S	-	-	-	I
14-26	2.0E7	2.3E8	1138.6	S	-	-	-	I
14-27	1.2E6	1.1E7	41.0	S	-	-	-	I
15-29	3.4E7	3.6E8	1982.4	S	-	-	-	I
16-31	1.2E6	9.6E6	43.2	S	-	-	-	I
24-57	4.1E3	1.2E4	0.0	S	-	-	-	I
5-8	8.7E3	8.7E4	0.0	U	-	-	-	I
6-9	5.9E4	6.7E5	0.3	U	-	-	-	I
7-12	6.2E7	6.9E8	817.4	U	-	-	-	I
7-13	2.1E2	1.5E3	0.0	S	-	-	-	I
8-15	7.5E2	6.8E3	0.0	S	-	-	-	I
9-17	4.1E3	3.6E4	0.0	S	-	-	-	I
Number of cases solved by MiniSat: 19								
Number of cases solved by Eq-MiniSat: 2								

Due to limited space, the instance names are shortened.

Table A.29: Graph pebbling family instances.

Instance	#Vars	#Cls
sat-grid-pbl-0070.sat05-1334.reshuffled-07	4970	9731
sat-grid-pbl-0200.sat05-1339.reshuffled-07	40200	79801
sat-pbl-00400.sat05-1322.reshuffled-07	1497	7374
unsat-grid-pbl-0080.sat05-1344.reshuffled-07	6480	12722
unsat-pbl-00070.sat05-1324.reshuffled-07	257	7375
unsat-pbl-00080.sat05-1325.reshuffled-07	288	11604
unsat-pbl-00090.sat05-1326.reshuffled-07	322	9322
unsat-pbl-00150.sat05-1328.reshuffled-07	552	30977
unsat-pbl-00200.sat05-1329.reshuffled-07	725	22684
unsat-pbl-00250.sat05-1330.reshuffled-07	864	32700

Table A.30: Results for graph pebbling family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
sat-grid-pbl-0070	3.3E5	5.6E5	0.2	S	7.8E5	1.5E6	3.4	S
sat-grid-pbl-0200	1E7	4.1E7	22.0	S	7.5E7	1.1E8	377.0	S
sat-pbl-00400	2.8E5	5.5E5	1.1	S	5.8E6	1.4E7	181.3	S
unsat-grid-pbl	2.3E6	4.5E6	2.7	U	2.9E6	6.5E6	17.2	U
unsat-pbl-00070	5.4E4	1.4E5	0.2	U	5.9E6	2.1E7	65.3	U
unsat-pbl-00080	7.3E4	2.0E5	0.3	U	1.7E6	6.4E6	27.8	U
unsat-pbl-00090	7.7E4	1.8E5	0.2	U	7.3E6	2.5E7	86.9	U
unsat-pbl-00150	1.2E5	2.9E5	0.7	U	-	-	-	I
unsat-pbl-00200	1.5E5	3.2E5	0.5	U	-	-	-	I
unsat-pbl-00250	2.4E5	5.1E5	0.9	U	-	-	-	I
Number of cases solved by MiniSat: 10								
Number of cases solved by Eq-MiniSat: 7								

Table A.31: Van Der Waerden family instances.

Instance	#Vars	#Cls
VanDerWaerden_2-3-12_135	135	5251
VanDerWaerden_2-3-13_160	160	7038
VanDerWaerden_pd_2-3-19_348	174	16458
VanDerWaerden_pd_2-3-20_381	191	19482
VanDerWaerden_pd_2-3-20_390	195	20607
VanDerWaerden_pd_2-3-21_399	200	21294
VanDerWaerden_pd_2-3-21_401	201	21509
VanDerWaerden_pd_2-3-21_404	202	22023
VanDerWaerden_pd_2-3-22_443	222	26201
VanDerWaerden_pd_2-3-22_462	231	28738
VanDerWaerden_pd_2-3-23_505	253	34014
VanDerWaerden_pd_2-3-23_506	253	34386

Table A.32: Results for Van Der Waerden family.

Instance	MiniSat				Eq-MiniSat			
Name	#Bran.	#Prop.	RT	RSL	#Bran.	#Prop.	RT	RSL
2-3-12_135	1.3E7	2.5E8	624.3	U	5.2E6	1.1E8	353.7	U
2-3-13_160	6.3E7	1.3E9	4915.2	U	5.0E7	1.2E9	5847.5	U
pd_2-3-19_348	7.3E6	1.6E8	752.3	U	3.0E6	7.2E7	413.2	U
pd_2-3-20_381	6.0E6	1.4E8	625.6	U	3.0E6	7.9E7	454.0	U
pd_2-3-20_390	1.5E7	3.5E8	2009.7	U	8.4E6	2.2E8	1546.1	U
pd_2-3-21_399	8.2E6	2.0E8	955.3	S	9.3E5	2.5E7	135.1	S
pd_2-3-21_401	2.1E7	5.2E8	2833.5	U	8.2E6	2.3E8	1470.2	U
pd_2-3-21_404	5.0E6	1.2E8	587.8	S	7.8E6	2.0E8	1435.5	S
pd_2-3-22_443	1.1E7	2.9E8	1618.1	S	2.0E7	5.9E8	4635.3	S
pd_2-3-22_462	9.4E6	2.4E8	1474.8	S	-	-	-	I
pd_2-3-23_505	1.0E7	2.7E8	1681.0	S	-	-	-	I
pd_2-3-23_506	1.5E7	4.0E8	2808.6	S	-	-	-	I
Number of cases solved by MiniSat: 12								
Number of cases solved by Eq-MiniSat: 9								