KTH Computer Science
and Communication

On conflict driven clause learning – a comparison between theory and practice

Konfliktdriven klausulinlärning – en jämförelse mellan teori och praktik

GUSTAV SENNTON
gsennton@kth.se

Master's Thesis within theoretical computer science at KTH CSC
Supervisor: Jakob Nordström
Examiner: Johan Håstad
December 10, 2014

Abstract

The boolean satisfiability (SAT) problem is the canonical NP-complete problem and every other NP-complete problem can be reduced to the SAT problem. Since the SAT problem is NP-complete large instances of this problem should be difficult to solve. However, in practice conflict driven clause learning (CDCL) solvers solve large real-world instances of the SAT problem efficiently. Recently, a theoretical upper bound was shown for the efficiency of a certain model of such solvers, but that solver model differs in several ways from solvers used in practice. In this project experiments are used to investigate whether such a model is realistic and whether its theoretical bound holds for solvers used in practice. These experiments compare all the components that differ between the theoretical solver model and a real implementation of a solver. The experimental results show that the running times of the two solvers often differ substantially. At the same time, the theoretical running time bound seems to hold for the practical solvers. I.e. the running time of practical solvers seems to grow polynomially for formulas for which the theoretically predicted running time is polynomial. However, some of the formulas used should be tested further since not enough data points have been collected for these formulas. For these formulas we cannot rule out high asymptotic running times of real-world solvers.

Referat

Konfliktdriven klausulinlärning – en jämförelse mellan teori och praktik

Problemet där man ska avgöra huruvida en Boolesk formel är satisfierbar (SAT-problemet) är NP-fullständigt och därmed borde det vara svårt att lösa stora sådana probleminstanser. I praktiken finns det konfliktdrivna klausulinlärningslösare (CDCL-lösare) som löser stora verklighetsbaserade probleminstanser effektivt, vilket motiverar studier av sådana lösare. Nyligen visades en övre gräns på körtider för en specifik modell av en CDCL-lösare. Det finns dock viktiga skillnader mellan denna modell och lösare som används i praktiken. I detta projekt används experiment för att utforska hur realistisk den övre gränsen på körtider för CDCL-lösare är. Den lösarmodell som används för att visa den övre gränsen för körtider jämförs med en praktisk implementation av en lösare. Utifrån de resultat som produceras i detta projekt verkar det finnas stora skillnader i körtid mellan de två lösarna. Det finns heller inga bevis som pekar på att praktiska lösare inte skulle täckas av den teoretiska körningstidsgränsen. Flera av de formler som användes i detta projekt borde dock testas ytterligare. För dessa formler har de experiment som utförts inte producerat tillräckligt många datapunkter för att ge stabila körtidskurvor.

Preface

This is a Master's thesis within the area of theoretical computer science, the work described in this report was performed during the spring, summer and autumn of 2014.

I would like to thank Marc Vinyals for help regarding the experiment setup and Massimo Lauria for providing code necessary for running experiments. I would also like to thank my supervisor Jakob Nordström for introducing me to the area of computational complexity and for the helpful discussions and advice provided during this project.

Contents

Chapter 1

Introduction

1.1 General background

Within the area of computational complexity the difficulty of a problem is measured by how the running time of solving a problem instance scales with the size of the instance. Problems are divided into complexity classes depending on their difficulty. For example, the class P contains problems solvable in time scaling polynomially with instance size while the class NP contains problems which can be verified in time scaling polynomially with instance size. One of the goals of computational complexity is to figure out whether $\mathsf{P} = \mathsf{NP}$, i.e. whether problems verifiable in polynomial time are also solvable in polynomial time. The most difficult problems in NP are called NP-complete and we do not know whether such problems lie in P. In the real world there are many NP-complete problems. These seem difficult to solve and therefore many researchers think that $\mathsf{P} \neq \mathsf{NP}$.

The boolean satisfiability (SAT) problem (the problem of determining whether a boolean formula is satisfiable) is the canonical NP-complete problem [Coo71] and through the notion of NP-completeness each NP-complete problem can be reduced to the SAT problem. People have been working on SAT solvers since the 60s, the time of DPLL (Davis Putnam Logemann Loveland) solvers [DLL62]. A DPLL solver is a search algorithm which searches for a satisfying assignment for a boolean formula. The search is carried out through assigning values to variables in the formula. At each step in the algorithm the solver assigns some variable and by doing this splits the search space in two — one part where the variable is true and one where it is false. A conflict is when the current assignment of variables falsifies the given formula. When a conflict is reached the solver backtracks by negating the value of the variable which was assigned most recently (and which has not yet been negated). The solver terminates when either a satisfying assignment is found or when the entire search space has been explored. During the 90s DPLL solvers were extended with a feature called clause learning [MS99], [BS97]. Clause learning is the notion of learning from mistakes — whenever the solver reaches a conflict it will extend its input formula by adding the reason for reaching the current conflict. In such a way the input formula will grow until it becomes trivially unsatisfiable — then the solver terminates. The resulting solvers are called conflict driven clause learning (CDCL) SAT solvers. In 2001, an efficient implementation [MMZ$^+$01] of a CDCL solver showed a large improvement in performance compared to earlier solvers and because of such improvements CDCL solvers are today the best performing SAT solvers. Later, in 2003 MiniSat was created [ES04]. MiniSat is a simple and efficient implementation of a CDCL solver. MiniSat has been very important for adding and testing

extensions to CDCL solvers just because the implementation is both simple and efficient. Because of their great performance, CDCL solvers can today be used profitably in many different applications such as AI planning and software testing. More examples of applications can be seen in [Mar08].

A CDCL solver is performing a type of reasoning when running — whenever the solver is expanding its formula the initial formula must imply the expanded formula. This means that, by expanding a formula until it is trivially unsatisfiable a CDCL solver is essentially creating a proof for the unsatisfiability of that formula. In this way a CDCL solver can be seen as a proof system. A proof system is a verifier of proofs or a way of creating proofs which can be verified in polynomial time. I.e. a proof system which determines whether some object has a certain property is an algorithm which takes (a representation of) an object and a potential proof as input. This algorithm runs in time polynomial in the size of (the representation of) the object and the length of the potential proof. If the object indeed has the sought property there exists some proof for which the algorithm outputs 1, otherwise the algorithm outputs 0 for every potential proof. All proof systems studied in this report are systems showing that formulas are unsatisfiable. Showing that a formula is satisfiable can always be done using a small proof since a satisfying assignment of the formula can be used as such a proof. A proof of a formula being unsatisfiable could be much longer since it must show that no assignment can be satisfying. Therefore, when the efficiency of a CDCL solver is related to that of some proof system only unsatisfiable formulas are considered.

Both DPLL solvers and CDCL solvers can be seen as variations of a proof system called *resolution* (since DPLL solvers are based on resolution and CDCL solvers are based on DPLL solvers). Resolution was first presented in [Bla37] and is a (relatively simple) proof system. There is only one rule in the resolution proof system and that rule can be used to expand a formula until it is trivially unsatisfiable. The efficiency of a proof system for a formula can be measured as the relation between the length of the shortest proof for that formula and the size of the formula. The overall efficiency of a proof system is the worst-case efficiency of the proof system taken over all possible input formulas.

DPLL solvers correspond to a restricted version of resolution (called tree-like resolution) which for some formulas could produce proofs exponentially longer than the shortest proofs produced by the general resolution proof system [BEGJ00] and [BIW04]. Up until recently an important question regarding CDCL solvers was whether a similar restriction applies to such solvers or whether CDCL solvers produce proofs of size at most polynomially longer than those created by general resolution. Results in for example [HBPV08] and [BKS04] show that CDCL solvers with some extensions are as efficient as general resolution. However, these extensions seem difficult (if not impossible) to implement and are not part of regular CDCL solvers. Therefore, these results do not reach the goal of determining the efficiency of CDCL solvers. Recently, this goal was reached in [PD11] where CDCL solvers were shown to be as efficient as general resolution within a polynomial factor. However, that article uses a model of CDCL solvers where decisions are made in an optimal way — i.e. when searching through its search space the solver always picks the optimal path. In reality, picking the optimal path seems difficult — there is even a result [AR08] suggesting that the results of [PD11] cannot be recreated using a practical algorithm (unless certain computational complexity classes collapse). A more practical result is shown in [AFT11] where decisions are made randomly instead of optimally. There are however several aspects regarding these results that could be studied further. In both [AFT11] and [PD11]

certain assumptions are made about CDCL solvers and these assumptions do not seem to hold for real-world solvers. The model used in [AFT11] will hereby be called the AFT solver model. There are three major assumptions which seem to differ between the AFT model and real-world solvers. Firstly, practical solvers often use heuristics rather than random decisions when exploring their search space. Secondly, practical solvers sometimes remove parts of the current expanded formula to save space and speed up certain solver operations. Removing parts of the expanded formula is something the AFT solver never does. Finally, the AFT solver restarts very often while practical solvers restart less often. A restart is performed by removing all of the current decisions so that potentially "bad" decisions are removed.

1.2 Project goal

The goal of this project is to find out whether the results shown in [AFT11] are — or can be made — practical. Firstly, this involves investigating whether the assumptions made in that article (and in [PD11]) are needed, i.e. whether similar results can be reached when using other solver models than that of the AFT solver. Secondly, it would be interesting to know whether these assumptions are realistic, i.e. whether the efficiency of the AFT solver is comparable to that of a solver used in practice. These questions are not strict mathematical statements because these questions are about how a model used to produce theoretical results corresponds to reality. This report does not aim to fully answer the above questions — the method used in this project when trying to answer these questions is to run experiments, i.e. running different solvers and comparing their running times. Experiments cannot give a complete picture of the solvers used since only a fixed number of formulas can be tested. However, these experiments could be used to gain some insights regarding the differences between real-world solvers and the AFT solver. Similarly if the experiments would show that formulas which should be equally difficult to solve using the AFT solver actually differ in reality then some new conclusions could be drawn regarding CDCL running times and their dependence on different formula properties.

The major difference between the experiments of this project and earlier experiments performed on CDCL solvers is that many earlier experiments compare practical solvers in the hope of finding better solvers. This project instead involves relating practical solvers to theoretical results. Regarding the choice of method this project could have been more theoretical — instead of running experiments theoretical reasoning could have been used in an effort to answer the questions above. However, such methods often require (or are at least simplified by) some kind of intuition regarding their outcome, which experiments can provide. Therefore, as stated above, this project does not aim to fully answer the questions above, but to create intuition which can be helpful for future work within this area.

1.3 Course of action and report outline

The project explained in this report consists of two parts, one theoretical and one practical. In the theoretical part the most important papers relevant to this project were studied. This part resulted in expositions of [BW01], [AFT11] and [PD11]. These expositions form the theoretical background of this report which is presented in Chapters 2, 3 and 4. The aim of the theoretical part of this project is to get a better understanding of CDCL solvers in order to reason about their effi-

ciency. The practical part consists of running experiments using a CDCL solver with different combinations of settings on several different kinds of formulas. To be able to run such experiments the code of a modern implementation (MiniSat) of a CDCL solver was modified to add extra features used in the experiments. The implementation and the modifications are explained in Chapter 5. After making these modifications the actual experiments were set up, the details regarding these experiments are presented in Chapter 6. To set up the experiments, code for generating formulas in the DIMACS format (the format MiniSat uses) had to be written as a part of this project. Most of the code for running different solvers and saving their output was already written for use in similar earlier experiments. This code just needed to be modified to handle the new formulas tested and the different changes in solver settings. The results of the experiments are presented in Chapter 7 and some concluding remarks and suggestions for future work are discussed in Chapter 8.

Chapter 2

Theoretical background

In this chapter we survey some proof complexity background so that we can argue about the efficiency of proof systems related to CDCL solvers. CDCL solvers are descendants of the DPLL solver [DLL62] which is based on the Davis-Putnam solver [DP60] which in turn is based on the resolution proof system. As mentioned earlier, the efficiency of CDCL solvers can be studied through relating the running time of a solver to the efficiency of the resolution proof system. A CDCL solver takes a boolean formula as input and expands it — derives new formulas implied by the initial formula — until the expanded formula is trivially unsatisfiable. In this way a CDCL solver can be seen as a type of resolution proof system. If the solver instead of just storing the expanded formula in some database writes down each derivation step the solver will have produced a proof of the unsatisfiability of the formula when the solver terminates. We should note that the way we view a CDCL solver here is slightly simplified — for example we ignore preprocessing which is commonly used together with CDCL solvers in practice.

2.1 Basic terminology and notation

An important subclass of the SAT problem is the *conjunctive normal form* (CNF) SAT problem. The CNF SAT problem is NP-complete. For simplicity, we focus on the CNF SAT problem rather than the general SAT problem in this report. A CNF formula is a conjunction of *clauses*, where a clause is a disjunction of *literals*. A literal is a variable or the negation of a variable. A *unit clause* is a clause containing exactly one literal. An example of a CNF formula is

$$(\overline{a} \vee b \vee c) \wedge (\overline{a} \vee \overline{b} \vee c) \wedge (a) \wedge (b \vee \overline{c}) \wedge (a \vee c) \wedge (\overline{b} \vee \overline{c}).$$

Here $\overline{a}$ denotes the negation of the variable a. Sometimes, it is convenient to use the notation x^r for a variable x and a number $r \in \{0, 1\}$ to refer to the literal $x^r = x$ if $r = 1$ and $x^r = \overline{x}$ if $r = 0$.

A proof system for proving that some object has a certain property is an algorithm $PS(o, \pi)$ which takes as input (a representation of) an object o and a potential proof π. The algorithm runs in time polynomial in the size of its input $|o| + |\pi|$. The algorithm has the following two properties: if the input object does have the sought property there exists a proof π for which the algorithm outputs 1, i.e. $\exists \pi : PS(o, \pi) = 1$. If the input object does not have the sought property then for every potential proof the algorithm outputs 0, i.e. $\forall \pi : PS(o, \pi) = 0$. In this report we will only consider propositional proof systems, i.e. proof systems for showing the unsatisfiability of CNF formulas.

Definition 2.1 (P-simulation). A proof system A *p-simulates* a proof system B if for each formula F, the shortest proof in A refuting (showing the unsatisfiability of) F has length at most polynomial in the size of F or the length of the shortest proof in B refuting F.

P-simulation is a notion used to capture the relative efficiency of different proof systems. The definition of p-simulation depends not only on the relation between proof sizes in different proof systems but also on the relation between proof size and the size of the input formula. This latter dependency is used because there are formulas which can be refuted using constant size proofs. We are interested in proof systems based on CDCL solvers and such solvers should be able to read the entire input formula before refuting it (since this is what most solvers do in practice as long as the formula is not trivially refutable).

To relate the running time of a CDCL solver to some proof system (in this case resolution) we need to represent the run of a CDCL solver with some proof system. This is done more strictly later in this report (Definition 4.6). Less strictly, we could say that a CDCL solver works by expanding its input formula and thus deriving new clauses. A proof system representing a CDCL solver is then a system where clauses can be derived only if they can be derived by the CDCL solver. It can be shown [BKS03] that the run of a CDCL solver corresponds to the creation of a certain type of resolution proof. Therefore, any proof system representing a CDCL solver can be seen as a subsystem of resolution. I.e. a CDCL solver can be at most as efficient as the resolution proof system.

2.2 The resolution proof system

The resolution proof system can be used to refute CNF formulas. A resolution proof starting with a formula F is a sequence of clauses such that each clause is either a clause in F (an axiom) or is derived from clauses earlier in the sequence. A clause can be derived either by the resolution rule, shown in (2.1), or by the weakening rule, shown in (2.2). The resolution rule can be written as

$$\frac{A \vee x \quad B \vee \overline{x}}{A \vee B} \tag{2.1}$$

where A and B are clauses and x is a variable. Given the resolution rule and the clause $C = A \vee B$, the clauses $A' = (A \vee x)$ and $B' = (B \vee \overline{x})$ can be *resolved* to produce the resolvent C. We can also write $C = Res(A', B')$ if it is not necessary to show which variable is being resolved over. We use the term $F \vdash C$ to denote that the clause C can be derived from the formula F using resolution. A *resolution derivation* $\pi : F \vdash C$ of the clause C from the formula F is a resolution proof starting with F and ending in C. A *resolution refutation* $\pi : F \vdash \bot$ is a derivation of the empty clause $\bot$ (a clause with no literals). A resolution refutation is a proof showing that a formula is unsatisfiable since an empty clause cannot be satisfied (there are no literals to satisfy). More intuitively, the only way to reach the empty clause is to resolve two contradictory unit clauses, e.g. $(x) \wedge (\overline{x})$, and any formula containing such clauses is trivially unsatisfiable. An example of a resolution refutation can be seen in Figure 2.1. There the notation $Res(x, y)$ means that the clauses at lines x and y are resolved. The second rule that can be used in the resolution proof system, the weakening rule, is shown below.

$$\frac{A}{A \vee B} \tag{2.2}$$

Formula

$$(\overline{a} \vee b \vee c) \wedge (\overline{a} \vee \overline{b} \vee c) \wedge (a) \wedge (b \vee \overline{c}) \wedge (a \vee c) \wedge (\overline{b} \vee \overline{c})$$

Proof

(1)	$\overline{a} \vee \overline{b} \vee c$	Axiom
(2)	$\overline{a} \vee b \vee c$	Axiom
(3)	$\overline{a} \vee c$	$Res(1,2)$
(4)	a	Axiom
(5)	c	$Res(3,4)$
(6)	$b \vee \overline{c}$	Axiom
(7)	$\overline{b} \vee \overline{c}$	Axiom
(8)	$\overline{c}$	$Res(6,7)$
(9)	$\bot$	$Res(5,8)$

Figure 2.1. Resolution refutation example

The weakening rule is redundant — it is possible to show (using induction) that any resolution refutation using the weakening rule, can be rewritten as a resolution refutation of at most the same length where only the resolution rule is used. However, the weakening rule can be useful when arguing about resolution.

It is possible to show that resolution is sound, i.e. if A and B are clauses and x is a literal and $A \vee B = Res(A \vee x, B \vee \overline{x}, x)$ then $A \vee B$ is implied by $(A \vee x) \wedge (B \vee \overline{x})$. Resolution can be shown to be sound (by a case analysis on the variable which is being resolved) in the following way. Let us assume $A \vee x$ and $B \vee \overline{x}$ are both satisfied, then if $x = 0$, A must be satisfied and thus $A \vee B$ is satisfied. Otherwise, if $x = 1$ then B must be satisfied and again $A \vee B$ is satisfied. It is also possible to show that resolution is complete, i.e. each unsatisfiable formula can be refuted using resolution. Completeness can be shown by creating a decision tree for the given formula F— let each node be a variable and each edge represent an assignment to that node. Since F is unsatisfiable each path in such a decision tree must falsify some clause in F. Let each leaf node of the tree represent the clause which is falsified by the path from the root to that leaf. A resolution derivation can then be created from the decision tree by letting each node represent the clause resolved from its children nodes. The root node will then represent the empty clause.

A resolution derivation can be represented as a graph, where nodes represent clauses in the derivation and edges represent rule uses. Thus, in such a graph representation, for each rule usage in the represented derivation edges are drawn from each of the initial clauses in the rule to the resulting clause. Any graph representation of a resolution derivation is a directed acyclic graph (DAG) since each clause must either lie in the initial formula or be derived from earlier clauses. If the graph representation of a resolution derivation is a tree the derivation is called tree-like. The difference between such a derivation and a general resolution derivation is that in a tree-like derivation each time a clause is used in the derivation it must be re-derived from axiom clauses. In general resolution each clause need only be derived once and can then be used several times throughout the derivation.

Definition 2.2 (Trivial resolution). Trivial resolution is a restricted version of resolution where each variable can be resolved over at most once and where the resolution rule can only be used (except for the first usage) to resolve an axiom together with the latest resolved clause.

Definition 2.3 (Unit resolution). Unit resolution is resolution where whenever the resolution rule is used at least one of the clauses being resolved must be a unit clause.

2.2.1 Complexity measures

Just as the efficiency of a computer program is not only measured in terms of running time but also for example memory consumption, it is possible to argue about the efficiency of a proof system using different complexity measures.

Definition 2.4 (Size). The size $S(\pi)$ of a resolution derivation π is the number of lines (clauses) in π. The size $S(F \vdash C)$ of deriving a clause C from a CNF formula F is the minimum size over all possible derivations of C from F: $S(F \vdash C) = \min_{\pi:F\vdash C}\{S(\pi)\}$. Similarly $S_{\mathcal{T}}(F \vdash C)$ is the minimum size over tree-like derivations of C from F.

Size essentially represents the running time of an algorithm implementing resolution, it is therefore natural to study such a complexity measure.

Definition 2.5 (Width). The width $W(C)$ of a clause C is the number of literals in the clause. The width $W(F)$ of a set of clauses F, such as a formula or a derivation, is the width of the largest clause in the set: $W(F) = \max\{W(C) : C \in F\}$. Finally, the width of deriving a clause C from a formula F, denoted $W(F \vdash C)$ is the minimum width taken over all possible derivations of C from F: $W(F \vdash C) = \min_{\pi:F\vdash C}\{W(\pi)\}$.

It is not obvious why the width of a resolution derivation should be studied. However, it turns out that there are strong relations between the size and width of resolution proofs (as we will see in Section 2.3). These relations essentially state that short resolution proofs can be used to create narrow proofs which means that if there are no narrow proofs for a certain formula then there are no short proofs either.

Another important complexity measure is space. Assume that whenever we have derived a clause and want to use it later (without deriving it again) we must save it to some memory. The space complexity measure then represents the minimum amount of memory needed to refute a formula. This notion is not defined in a strict way here because it is not used (in this report) to show any results regarding the running times of CDCL solvers.

2.2.2 Restrictions

We can restrict a formula through assigning values to some of its variables. This can be useful since restricting a formula makes it smaller and thus (possibly) easier to reason about. The restriction $x = a$ on the clause C denoted $C\restriction_{x=a}$ is 1 if x^a lies in C (the clause 1 represents a satisfied clause) and $C \setminus \{x^{1-a}\}$ otherwise. It is possible to restrict an entire derivation $\pi : F \vdash D$ (creating the restriction $\pi\restriction_{x=a}$), this can be done by simply restricting each clause in the derivation, i.e. if $\pi = (C_1, C_2, \ldots, C_m)$ then $\pi\restriction_{x=a} = (C_1\restriction_{x=a}, C_2\restriction_{x=a}, \ldots, C_m\restriction_{x=a})$.

Lemma 2.6. *Given a derivation $\pi : F \vdash D$, a restriction $\pi\restriction_{y=a}$ is a valid derivation of $D\restriction_{y=a}$ from $F\restriction_{y=a}$.*

Proof. To show that $\pi\restriction_{y=a}$ is a valid derivation of $D\restriction_{y=a}$ we simply show that each clause in $\pi\restriction_{y=a}$ that is not an axiom can be derived from some earlier clauses in $\pi\restriction_{y=a}$. Let $A' = (A \vee x)\restriction_{y=a}$, $B' = (B \vee \overline{x})\restriction_{y=a}$ and $C' = C\restriction_{y=a}$ where

$C = Res(A \vee x, B \vee \overline{x}, x)$. Let us now show that C' can be derived from A' and B'. Note that if C' cannot be resolved from A' and B' then either x is the variable being restricted, $A\restriction_{y=a} = 1$, or $B\restriction_{y=a} = 1$. In the latter case (when x is not being restricted) $C' = 1$ because C' contains the literal being satisfied in $A\restriction_{y=a}$ or $B\restriction_{y=a}$ since that literal is not being resolved over. The clause $C' = 1$ can always be derived through the weakening rule. If the restriction is $x = 0$, then $A \vee x$ is restricted to $A = A\restriction_{x=0}$ and by the weakening rule $(A \vee B)\restriction_{x=0}$ holds. The same applies for the restriction $x = 1$ except then $B \vee \overline{x}$ is restricted to $B = B\restriction_{x=1}$ which implies $(A \vee B)\restriction_{x=1}$. If instead $C = A \vee B$ is derived through the weakening rule from the clause A then $C' = A\restriction_{y=a} \vee B\restriction_{y=a}$ can be derived through the weakening rule from $A\restriction_{y=a}$. □

2.3 Some fundamental resolution results

In this section we discuss some important results regarding the efficiency of resolution — these results are important because CDCL solvers can be at most as efficient as resolution. Maybe the most important result is that there are exponential lower bounds for resolution, i.e. there are formulas for which any resolution refutation has size at least exponential in the size of the formula being refuted. Such results were initially presented in [Hak85] where the formula family used was an encoding of the pigeonhole principle (PHP). The pigeonhole principle states that if p pigeons are to be placed in h holes, where $p > h$, then some hole must contain at least two pigeons. This principle can be encoded by creating a matrix of variables where each variable represents a certain pigeon flying to a certain hole. The formula consists of two types of clauses — one which states that each pigeon must fly to some hole and one which states that no pair of pigeons can fly to the same hole. A more detailed explanation of PHP formulas can be seen in Section 6.1.2.

Results as that presented in [Hak85] were initially quite difficult to attain. However, in [BW01] a new simpler method for showing resolution size lower bounds was presented. The reason for this method being easier to use than earlier methods is that it uses width lower bounds to prove size lower bounds instead of proving size lower bounds directly. Width lower bounds can be used because the same article ([BW01]) shows a strong relation between the size and width of resolution proofs.

Theorem 2.7 ([BW01]). *For any unsatisfiable CNF formula F,*

$$W(F \vdash \bot) \leq W(F) + \log S_{\mathcal{T}}(F \vdash \bot)$$

where $S_{\mathcal{T}}(F \vdash \bot)$ is the size of the smallest tree-like resolution refutation of F.

Theorem 2.7 describes a relation between width and size of tree-like resolution proofs. This relation was originally shown in [BW01]. The relation can be rewritten to show how the size of refuting a formula in a tree-like manner grows exponentially with the width of refuting the same formula (if the formula has constant initial width), as can be seen in Corollary 2.8.

Corollary 2.8 ([BW01]). *For any unsatisfiable CNF formula F the following holds.*

$$2^{W(F\vdash\bot)-W(F)} \leq S_{\mathcal{T}}(F \vdash \bot)$$

Theorem 2.9 contains the slightly more complex version of the relation from Theorem 2.7. This more complex version holds for general resolution.

Theorem 2.9 ([BW01]). *For any unsatisfiable CNF formula F the following holds.*

$$W(F \vdash \bot) \leq W(F) + \mathrm{O}\left(\sqrt{n \ln S(F \vdash \bot)}\right)$$

Theorem 2.9 can also be rewritten to show how the size of refuting a formula grows exponentially with the width of refuting that same formula, as is shown below.

Corollary 2.10 ([BW01]). *For any unsatisfiable CNF formula F the following holds.*

$$\exp(\Omega(W(F \vdash \bot) - W(F))^2/n) \leq S(F \vdash \bot)$$

Together with lower bounds on width of resolution refutations Corollaries 2.8 and 2.10 imply size lower bounds of resolution refutations. It is important to note that, for these relations to be useful, the initial width of the formula F should be small. Typically, if we want exponential lower bounds on size of refutations we prove linear lower bounds on width and bound the width of the initial formula to a constant.

Below follow proofs of the width-size relations (Theorems 2.7 and 2.9). Both proofs are essentially the same as in [BW01]. These proof are shown (instead of only presenting the relations) to give the reader a view of how to create a narrow refutation given a short one. This view is important because short refutations are not necessarily narrow (as the title of [BW01] suggests) even though it is possible to create a narrow refutation given a short one. The proofs use induction on the size of a refutation and the number of fat (wide) clauses in a refutation respectively. In these proofs the initial resolution refutation is split into two smaller derivations which by themselves cannot be used to refute the initial formula. Then, by use of the induction assumption, these derivations are rewritten into new derivations of small width. Finally, Lemma 2.11 is used to merge these new narrow derivations into a derivation, of the target width, refuting the initial formula. To split the initial refutation into smaller derivations a restriction is used — one of the variables in the refutation is restricted first to 0 and then to 1 to create two smaller derivations. The difference between the two proofs is the goal when splitting the refutation into smaller derivations. The proofs use different induction assumptions which means that showing that a derivation has small width in the first proof can be done by showing that it has small size, while in the second proof this is done by showing that the derivation has a small number of fat clauses.

Lemma 2.11 ([BW01]). *Let F be a refutable CNF formula, and let x^a be a literal. If $W(F\restriction_{x=a} \vdash \bot) \leq k-1$ and $W(F\restriction_{x=1-a} \vdash \bot) \leq k$, then $W(F \vdash \bot) \leq \max(W(F), k)$.*

Proof. Pick a refutation of $F\restriction_{x=a}$ with width $k-1$ and add x^{1-a} to each axiom clause in that refutation which contained x^{1-a} before the use of the restriction $x = a$. Then modify the refutation so that the literal x^{1-a} is propagated accordingly. I.e. whenever at least one of the clauses being resolved in the refutation contains x^{1-a} the resulting resolvent should also contain x^{1-a}. The derivation created through modifying the aforementioned refutation is a valid derivation of x^{1-a} from F. The derivation is valid because each axiom in the derivation is a clause in F (and the derivation is based on a valid refutation to which we have added x^{1-a} without removing correctness). Furthermore, x^{1-a} is never resolved over in this derivation and thus instead of obtaining the empty clause the derivation ends up deriving x^{1-a}. Note that this derivation has width at most $k-1+1 = k$

since the original refutation of $F\restriction_{x=a}$ has width $k-1$ and we add at most one literal to each clause in that refutation to obtain our new derivation. Now, after obtaining x^{1-a} we can resolve x^{1-a} with each clause in F containing x^a. This step can be performed using width at most $W(F)$ and results in the restriction $F\restriction_{x=1-a}$ which can be refuted in width k (since this is what $W(F\restriction_{x=1-a}\vdash\bot)\leq k$ means). Thus, F can be refuted using width $\max(W(F),k)$. □

Proof of Theorem 2.7. This is a proof by induction over b and n, where n is the number of variables in the initial formula. The induction assumption is the following: if $S_\mathcal{T}(F\vdash\bot)\leq 2^b$ then $W(F\vdash\bot)\leq W(F)+b$. Let us start with the base cases. If $b=0$ then F must contain the empty clause. If $n=1$ then F contains at most one variable x and can be refuted using the resolution rule on x and $\overline{x}$. For the induction step, let us consider the last step in the tree-like refutation of F, which is

$$\frac{x \quad \overline{x}}{\bot}$$

for some variable x. Either the tree-like derivation of x or that of $\overline{x}$ must have size less than $S_\mathcal{T}(F\vdash\bot)/2$. Let us assume, without loss of generality, that x is the literal that can be derived from F using a derivation of size less than half of $S_\mathcal{T}(F\vdash\bot)$, i.e. $S_\mathcal{T}(F\vdash x)\leq 2^{b-1}$. By restricting such a derivation using the restriction $x=0$ we get a new derivation $\pi:F\restriction_{x=0}\vdash\bot$ (which is a valid refutation by Lemma 2.6), also of size less than 2^{b-1}. By induction on b we can refute $F\restriction_{x=0}$ in width $W(F\restriction_{x=0})+b-1\leq W(F)+b-1$. Furthermore, $W(F\restriction_{x=1}\vdash\bot)\leq W(F\restriction_{x=1})+b\leq W(F)+b$ by induction on n (since $F\restriction_{x=1}$ contains no instance of the variable x). Now, by Lemma 2.11, $W(F\vdash\bot)\leq W(F)+b$. □

Proof of Theorem 2.9. This proof is by induction over b and n where n is the number of variables in the initial formula F. Let π be a refutation of F of minimum size. The following is the induction assumption: if the number of fat clauses in π is strictly less than a^b then $W(F\vdash\bot)\leq W(F)+b+d$, where $a=\frac{1}{1-d/2n}$ and a fat clause is a clause of width at least $d=\sqrt{2n\ln S(F\vdash\bot)}$. The following are the induction base cases. If $b=0$ then π contains no fat clauses and the width of refuting F is less than d. If $n=1$ then F contains exactly one variable x and can be refuted by resolving x with $\overline{x}$, then $W(F\vdash\bot)\leq W(F)$.

The next step is to show that restricting a refutation of F by satisfying a literal contained in many fat clauses in the refutation will produce a restricted refutation with few enough fat clauses to use the induction assumption. Let f be the number of fat clauses in π. Then the number of literals contained in fat clauses is at least df and the number of distinct literals in total is $2n$. Therefore, by the pigeonhole principle there must be some literal x contained in at least $df/2n$ fat clauses in π. Now let $\pi'=\pi\restriction_{x=1}$ and note that this restriction π' is a refutation of $F\restriction_{x=1}$ (by Lemma 2.6). π' contains less than $(1-d/(2n))f<(1-d/(2n))a^b=a^{b-1}$ fat clauses since each clause containing the literal x will be satisfied by the restriction $x=1$, and can thus be removed. By induction on b, $F\restriction_{x=1}$ can be refuted in width $W(F\restriction_{x=1})+b+d-1$. Also, by applying the restriction $x=0$ to the original refutation π we get a refutation of $F\restriction_{x=0}$ with less than a^b fat clauses, and one less variable. Thus by induction on n, $F\restriction_{x=0}$ can be refuted in width $W(F)+b+d$. Then by Lemma 2.11, F can also be refuted in width $W(F)+b+d$. Now we have shown the induction assumption, but that assumption relates width and the number of fat clauses in a refutation. Let us instead relate width to size. The number of fat clauses of a refutation is at most the number of clauses in the refutation, i.e.

the size of the refutation. So letting $a^b > S(F \vdash \bot)$ will make the induction assumption hold for any refutable formula. Now let us calculate how this bounds the variable b (in the equations below, the actual equations are shown in the left column and non-trivial steps are explained in the right column)

$$
\begin{aligned}
a^b = (1 - d/2n)^{-b} &> S(F \vdash \bot) && [a = \frac{1}{1 - d/2n}] \\
b \ln (1 - d/2n) &< -\ln S(F \vdash \bot) && [\text{logarithm and negation}] \\
b \ln (1 - d/2n) \leq -bd/2n &< -\ln S(F \vdash \bot) && [\ln x \leq x - 1] \\
bd/2n &> \ln S(F \vdash \bot) \\
b &> \frac{2n \ln S(F \vdash \bot)}{d} \\
b &> \frac{2n \ln S(F \vdash \bot)}{\sqrt{2n \ln S(F \vdash \bot)}} && [d = \sqrt{2n \ln S(F \vdash \bot)}] \\
b &> \sqrt{2n \ln S(F \vdash \bot)}
\end{aligned}
$$

Thus, letting $b = \sqrt{4n \ln S(F \vdash \bot)}$ will make the induction assumption entail all refutable formulas and still fulfill $b + d \in \mathrm{O}\left(\sqrt{n \ln S(F \vdash \bot)}\right)$. □

Chapter 3

CDCL solvers and their ancestors

In this chapter we investigate the principles of how CDCL solvers work. We start by examining the ancestors of CDCL solvers called DPLL solvers.

3.1 Davis-Putnam-Logemann-Loveland solvers

Conflict driven clause learning (CDCL) solvers are based on the Davis-Putnam-Logemann-Loveland (DPLL) solver [DLL62] which in turn is based on the Davis-Putnam solver [DP60]. A DPLL solver is essentially a search algorithm that looks for satisfying assignments for a given formula. The search is carried out through restricting the given formula by assigning values to different variables. These assignments can be set through propagation or assumption — if some variable obviously must be assigned a certain value then the assignment of that variable is said to be propagated. Whenever the algorithm does not immediately see that a certain variable must be set to a certain value to make the formula satisfying the algorithm makes an assumption regarding the value of some variable. Therefore, essentially what the solver does is to divide the problem into subproblems, one where the latest assumption holds and one where it does not. The formula is satisfiable if a satisfying assignment can be found in at least one of these two subproblems. Whenever a conflict is reached (some clause is falsified by the current assignment) the algorithm backtracks to negate its latest assumption. Note that an assignment is not considered an assumption if it was derived through negating an assumption (i.e. we cannot negate the negation of an assumption). The algorithm terminates either when a satisfying assignment is found or when the algorithm reaches a conflict without any assumption being made (all assumptions have been disproved).

3.2 Conflict driven clause learning solvers

Algorithm 1 contains pseudocode for a CDCL solver. Just as a DPLL solver a CDCL solver searches for a satisfying assignment for its input formula. However, some extensions have been added that makes a CDCL solver more efficient than a DPLL solver. The most important extension is that of clause learning which is explained in Section 3.2.2. The solver contains a database of clauses which consists of the clauses in the initial formula and clauses that have been learnt by the solver during its run. At any time during a run the solver contains a sequence of assignments, these can be either decision (assumption) assignments (chosen through a decision scheme) or propagated assignments (as explained in Section 3.2.1). When arguing about a solver we usually argue about its database restricted by the variable assignments of the solver.

```
   input  : CNF formula
   output: SAT or UNSAT
 1 while true do
 2     while Propagation is possible do
 3         Propagate one variable
 4         if Conflict then
 5             if No variables are assigned by decisions then
 6                 return UNSAT
 7             end
 8             Learn conflict clause
 9             Backjump
10         end
11     end
12     if Time to restart then
13         Restart
14     end
15     if No unassigned variable left then
16         return SAT
17     else
18         Assign unassigned variable
19     end
20 end
```

Algorithm 1: CDCL solver pseudocode

3.2.1 Propagation

A CDCL solver only propagates assignments using unit propagation. A unit propagation is carried out when a clause in the database is reduced to a unit clause when restricted by the current variable assignments. A propagation is essentially a decision which is obvious to the solver — when one of the clauses in the clause database is reduced to a unit clause then the single literal in that clause must be satisfied. For example if we run a CDCL solver on the formula

$$(a \vee b \vee \overline{c}) \wedge (a \vee \overline{b})$$

and decide to assign the variable a to false, then by unit propagation b must be false because the clause $(a \vee \overline{b})$ is reduced to the clause $(\overline{b})$. Now the clause $(a \vee b \vee \overline{c})$ is reduced to the unit clause $(\overline{c})$ which means that c must also be false. Therefore by only assigning one variable by decision we have by propagation assigned two variables more. In the resolution proof system unit propagation is represented by unit resolution (Definition 2.3), i.e. we can reach a conflict using unit propagation on a formula F if and only if F can be refuted using unit resolution. This relation holds because propagating a literal x is equivalent to resolving all clauses (containing $\overline{x}$) in our formula with the unit clause (x).

3.2.2 Conflicts and clause learning

Whenever a conflict is reached, i.e. whenever a clause in the clause database of the solver is falsified by the current variable assignments, the CDCL solver will try to learn the reason for the conflict. Learning the reason for a conflict is done through clause learning — a clause is added to the clause database to make sure the solver cannot run into the same conflict again. By learning a clause any sequence

of decisions which include falsifying that clause are pruned from the search space. For example making the decision $a = 0$ in the formula

$$(a \vee b \vee \overline{c}) \wedge (a \vee \overline{b}) \wedge (a \vee b \vee c)$$

will propagate $b = 0$ and then $c = 0$ (or $c = 1$) which will falsify the clause $(a \vee b \vee c)$ (or $(a \vee b \vee \overline{c})$ if $c = 1$). Thus, all that needed to be done to reach a conflict was to set $a = 0$, therefore $a = 0$ is the reason for the conflict and a CDCL solver could learn the unit clause (a) after reaching such a conflict. If we now continue to run our solver the assignment $a = 1$ will be propagated (after backtracking) because of the learnt unit clause (a) which means that the solver cannot reach the same conflict again.

3.2.3 Backjumping and restarts

As mentioned earlier, whenever a DPLL solver reaches a conflicting state it will backtrack, i.e. negate the latest decision, to reach a non-conflicting state. A CDCL solver has more information than a DPLL solver after reaching a conflict — the CDCL solver has tried to understand the reason for the conflict through learning a clause. Because of this newly learnt clause the CDCL solver might, after backtracking, gain new propagations that the DPLL solver will not. Therefore backjumping is used instead of backtracking.

The difference between an algorithm using simple backtracking and an algorithm using backjumping is that the one using backtracking only backtracks until it reaches a non-conflicting state. I.e. after a conflict the simpler backtracking algorithm backtracks until it has unassigned the latest assigned literal (out of those assigned by decision) of the newly learnt clause. A backjumping algorithm however, can continue backtracking as long as a propagation is gained from the newly learnt clause. I.e. a backjumping algorithm can backtrack until it reaches the assignment of the variable in the newly learnt clause which was assigned (by decision) second latest. This assignment is not undone since if it would be the algorithm would not be able to directly take advantage of the propagation produced by the newly learnt clause.

Let us consider an example where we have an unsatisfiable formula including the variables $a, b, \ldots, y, z$ and assign all those variables by decisions in the order they are presented and reach a conflict after assigning z. Then a DPLL solver would negate the latest decision, i.e. the assignment of z and continue running. This course of action is fine if all of the variables decided upon are part of the reason for the conflict. But what if the reason for the conflict is only the assignments of the variables a, b and z? Then the DPLL solver will perform many unnecessary operations by exploring the search space where a and b are assigned since a conflict will always be reached when z is assigned again. A CDCL solver implementing backjumping could instead learn the clause $(\overline{a} \vee \overline{b} \vee \overline{z})$ and undo all assignments until that of the second latest variable (b) which was assigned by decision in the newly learnt clause. In this case such a backjump results in undoing all assignments made after b was assigned. Then because of the newly learnt clause, the literal $\overline{z}$ will be propagated before making any new decisions — this propagation will remove a large part of search space that the DPLL solver is searching through.

The last major difference between DPLL solvers and CDCL solvers — restarts — removes all assignments made so far. Intuitively, restarts should be used to remove unwanted decisions, i.e. decisions that do not contribute to reaching conflicts. However, the theoretical understanding of how restarts work and how they affect the performance of CDCL solvers is limited.

3.2.4 CDCL components

In a CDCL solver there are several components which can be altered to tune the solver. These components can heavily alter the efficiency of a solver. The following are such components: *restart policy*, *learning scheme*, *decision scheme*, and *clause removal*. A restart policy determines how often a solver restarts. Usually a restart policy is expressed as a number of conflicts between restarts. In Algorithm 1 the restart policy is run at line 12.

The learning scheme determines what types of clauses will be learnt when reaching a conflict, i.e. it determines how the solver figures out the reason for a conflict. In Algorithm 1 the learning scheme is used at line 8. In this report, only asserting learning schemes are considered because they are the ones most commonly used in practice (for example in the solvers MiniSat [ES04] and Lingeling [Bie13]). That a solver is using an asserting learning scheme means that it learns only asserting clauses (defined in Definition 4.4). Such clauses will prove useful because they will make it simple to argue about what a solver gains from learning a new clause.

A decision scheme determines the order in which decision assignments are made. In Algorithm 1 a decision scheme is used at line 18. One example of a decision scheme is to randomly choose variables, another example is called Variable State Independent Decaying Sum (VSIDS). VSIDS is often used in practice, for example in the solvers MiniSat [ES04] and Lingeling [Bie13]. In VSIDS each variable is given a counter and the variable to be assigned is always the one with highest counter value (and that has not yet been assigned). This counter represents the relevance — or activity — of a variable, whenever a new clause is learnt the counters of the variables in that clause are incremented. The values of the counters decay by periodic multiplication with some value between 0 and 1 so that the counters are affected more by recent activity than older activity. If several variables are tied for the highest counter value one of those variables is chosen randomly.

The last CDCL component is clause removal which is the act of removing some of the clauses learnt during a CDCL run. The removal of clauses is done to save memory and speed up operations performed by the solver. Clause removal can be inserted into the CDCL algorithm shown in Algorithm 1 for example between lines 14 and 15.

Chapter 4

CDCL solvers efficiency results

This chapter contains explanations of results regarding the efficiency of CDCL solvers. These results relate the running times of CDCL solvers to the efficiency of the resolution proof system. The main goal of this project is to find out how the CDCL solver models used in this chapter, and the results acquired, are related to practical solvers and their running times. As mentioned earlier we can view a DPLL solver run as the creation of a proof in a restricted version of the resolution proof system (more specifically a tree-like resolution proof). For some formulas this restricted version creates proofs exponentially longer than the shortest proofs in general resolution. An important question regarding CDCL solvers is whether these only simulate some restricted version of the resolution proof system or whether CDCL solvers can simulate general resolution.

In [PD11], it was shown that a proof system representing a CDCL solver p-simulates general resolution. However, an optimal decision scheme is used which makes these results impractical. Such results seem to require an unrealistic scheme, since [AR08] indicate that similar results cannot be produced for realistic CDCL solvers. [AFT11] on the other hand uses a decision scheme which picks variables randomly. The use of such a scheme leads to a worse bound, but the model can be implemented in reality. These two results were obtained using a very similar proof strategy for expressing the efficiency of CDCL solvers in terms of resolution measures. This strategy is presented below, in the form of an exposition of [AFT11] and [PD11]. The strategy is based on an earlier strategy used in for example [BKS04] and [HBPV08]. The goal of that earlier strategy was to let the solver learn an entire resolution refutation one clause at a time. The strategy worked partially — the attempts were successful but the models used were not quite those of CDCL solvers. In [BKS04] the model used makes decisions on already assigned variables. In [HBPV08] it is instead shown that CDCL solvers *effectively p-simulate* resolution. This expression means that instead of directly comparing the length of the shortest proof in a CDCL proof system to that of resolution a preprocessing step is added when creating CDCL proofs. This preprocessing step maps the input formula to a new formula (which must be satisfiable if and only if the initial formula is satisfiable) and runs in polynomial time in the size of the initial input formula. So essentially what [HBPV08] shows is that there is some preprocessing step that together with a CDCL solver simulates resolution.

The reason for trying to learn an entire proof is that resolution refutations always end with two contradictory unit clauses (x) and $(\overline{x})$. When a CDCL solver learns two contradictory unit clauses it will terminate before making any more decisions since such clauses provide contradictory unit propagations. The earlier attempts to show that CDCL solvers simulate resolution needed artificial models

of CDCL solvers because learning an entire resolution refutation is difficult — at each step of a run the next clause in the refutation must be learnt within a reasonable amount of time. Both [AFT11] and [PD11] solve this problem by using an improved strategy where each clause in the resolution refutation is not learnt explicitly — instead each clause in the resolution refutation is learnt implicitly by learning some other set of clauses. Learning a clause implicitly can be done because the benefits a CDCL solver gains from learning a clause are unit propagations, and therefore, instead of learning a certain clause the solver can learn some other clauses providing the same unit propagations. Thus, instead of showing how long it takes for a CDCL solver to learn each clause in a resolution refutation we show how long it takes for a solver to acquire the same unit propagations as those provided by each of the clauses in a resolution refutation.

4.1 Notation and terminology

To be able to calculate the time it takes for a CDCL solver to acquire some unit propagations we need to define what happens during a CDCL run and how new clauses are learnt (or new propagations are acquired).

Definition 4.1 (Round [AFT11]). A round is a run of a CDCL solver, starting with a set of clauses D (the database of the solver) and an empty sequence of assignments $S_0 = \{\}$, until either $\bot \in D|_{S_r}$ or $D|_{S_r}$ does not contain any unit clauses, where S_r is the sequence of assignments made until the rth assignment of the round.

Definition 4.1 introduces the concept of a round. A round is a run of the CDCL algorithm, starting with an empty sequence of assignments, until either the algorithm reaches a conflict or a new decision must be made. We should note that an assignment can only set a variable which is not currently assigned. An assignment can be either a decision or a propagation. A propagation is carried out when some clause in the clause database is restricted to a unit clause. A decision assignment can only be carried out when there are no unit clauses in the database of the solver (otherwise a propagation assignment is carried out).

If the CDCL algorithm reaches a conflict the current round is conclusive. A round that is not conclusive is inconclusive. The notion of an inconclusive round is important because in such a round all possible unit propagations are carried out — in a conclusive round, on the other hand, there could be unit propagations which are never carried out before the algorithm reaches a conflict. Note that we are only interested in assignments made starting from an empty decision sequence until a conflict is reached. Even though the solver might backjump and continue making propagations after a conflict is reached we ignore those propagation assignments (and any potential new conflicts). Such propagations will not pose a problem since any efficiency bounds reached will be expressed using the number of restarts of the solver run. The number of restarts corresponds to the number of conclusive rounds of the run since the restart policy used will make the solver restart after each conflict. Furthermore, it can be shown (Lemma 4.10) that a CDCL solver cannot reverse any of its progress (progress here essentially means how large part of a resolution proof the solver has learnt implicitly) by learning new clauses. Therefore, letting the solver carry out propagations (and potentially reach new conflicts) after reaching a conflict will not affect the solver in a negative way.

Definition 4.2 (Decision level [AFT11]). The decision level of a variable x during a round R is the number of decisions performed until assigning x during the round R.

The following is an example of how decision levels work. If we decide to assign $a = 0$ and this yields the propagations $b = 1$ and $c = 0$ and finally decide to assign $d = 0$ then the decision level of a, b and c is 1 while the decision level of d is 2.

4.1.1 Clause learning

Now we study what kind of clauses a CDCL solver learns during a run so that we can reason about the progress the solver makes by learning clauses.

Definition 4.3 (Conflict clause). A conflict clause for a conclusive round R, starting from the set of clauses D, is a clause C which is falsified by R and such that any round starting with D which falsifies C is conclusive.

A conflict clause is meant to capture the reason for a conflict. Therefore, falsifying a conflict clause must be enough to reach a conflict. Furthermore, for a clause to be a potential reason for the current conflict it must have been falsified during the current round. All clauses learnt by a CDCL solver are conflict clauses for the round at which they are learnt. Therefore, a learnt clause is always implied by the clause database of the solver (at the time of the learning) since falsifying a conflict clause yields a contradiction. This implication is important because the database of the solver is then implied by the input formula, which means that refuting the database proves that the input formula is unsatisfiable. Thus, during a run a CDCL solver is essentially expanding its input formula (in the form of a clause database) until the formula is trivially refutable (through unit propagation) and then terminates.

Definition 4.4 (Asserting clause). An asserting clause of a round R is a conflict clause of R with exactly one literal at the last decision level of R.

Asserting clauses are especially useful when proving efficiency bounds on CDCL running times (and are also commonly used in practice, as explained in Section 3.2.4). There is a bound on the number of times one can perform a certain round (or rather make the same sequence of decisions) over and over again when using an asserting learning scheme (a scheme only choosing asserting clauses). This bound stems from the fact that there is a limited number of variables of the highest decision level for any round — and after each conflict one more such variable will be propagated at a lower level.

4.1.2 Optimal decision scheme

To take advantage of the full power of CDCL solvers an optimal decision scheme will be used when proving that CDCL solvers simulate resolution. Actually the scheme used is not necessarily optimal but we get to choose the order of decisions so we could use an optimal scheme.

Definition 4.5 (Branching sequence [PD11]). A branching sequence is a sequence of variable assignment decisions.

An example of a branching sequence is $\langle a = 1, b = 0, c = 1, a = 0, c = 0, b = 1\rangle$, where a, b, c are variables in the input formula.

It is possible to show (Lemma 4.14) that a conflict clause of a round starting from the database D can be derived from D using trivial resolution. Because conflict clauses (and thus learnt clauses) can be derived in such a way we can create a proof system using a CDCL solver where a proof is the concatenation of trivial resolution proofs deriving clauses learnt by the solver.

Definition 4.6. Let D be a set of clauses, let B be a branching sequence and let $C_1, \ldots, C_m$ be the clauses learnt by a CDCL solver running with D as initial clause database and B as branching sequence. Let $\pi_1, \ldots, \pi_m$ be trivial resolution proofs of $C_1, \ldots, C_m$ from the corresponding databases of the solver at the time the clauses are learnt. Then we define $Der(D, B)$ to be the concatenation of these resolution proofs: $Der(D, B) = \pi_1, \ldots, \pi_m$ and we let $|Der(D, B)|$ denote the size of $Der(D, B)$.

Definition 4.6 describes proofs from a proof system representing a CDCL solver. Such proofs will not end in the empty clause since that would imply that the solver learns the empty clause. However, since the database of the solver is conflicting under unit propagation when the solver terminates we can simply add a unit resolution proof (simulating the unit propagation) to reach the empty clause. This unit resolution proof will always be short (each variable need only be propagated at most once) and will therefore be ignored in the rest of this report.

4.1.3 Concepts for determining solver progress

This section contains concepts which are used when relating the efficiency of CDCL solvers and the resolution proof system. These concepts make the difference between the new successful efficiency results of [AFT11] and [PD11] and older results where artificial models of CDCL solvers were used. The first concept is called absorption and is meant to capture the benefits of learning a clause without actually learning it. As mentioned earlier learning a clause implicitly means that the solver gains the unit propagations the clause would have provided when learnt.

Definition 4.7 (Absorption [AFT11]). A clause C is absorbed at a literal $x^a \in C$ by a set of clauses D if for each inconclusive round R falsifying $C \setminus x^a$, the round R contains the assignment $x = a$. C is absorbed by D if C is absorbed by D at each literal x^a in C.

What Definition 4.7 says is that a clause C is absorbed at a certain literal l if falsifying every literal in C except l will always mean that l is set to true (unless a conflict is reached). It can be shown ([AFT11]) that any clause absorbed by a set of clauses is also implied by those clauses, however the other direction does not hold (as we will see shortly in Example 4.9). If all implied clauses would be absorbed then CDCL solvers would directly be able to determine the satisfiability of a formula since an unsatisfiable formula implies all possible clauses. For example the clauses (x) and $(\overline{x})$ are implied by such a formula, and if a solver absorbs (x) and $(\overline{x})$ it will run into a conflict immediately and return UNSAT. Let us further investigate the concept of absorption through a couple of examples.

Example 4.8 (Successful absorption). The formula $F = (a \vee b) \wedge (\overline{b} \vee c)$ absorbs the clause $(a \vee c)$, i.e. a CDCL solver can absorb the latter clause by learning the first two clauses. This absorption holds because falsifying a reduces $(a \vee b)$ to (b) which propagates the assignment $b = 1$ which in turn reduces $(\overline{b} \vee c)$ to (c), finally propagating $c = 1$. Thus $(a \vee c)$ is absorbed by F at c. Similarly if c is falsified

then $(\overline{b} \vee c)$ is reduced to $(\overline{b})$ which propagates $b = 0$ and reduces $(a \vee b)$ to (a) which propagates $a = 1$. Therefore $(a \vee c)$ is absorbed by F at a and then since $(a \vee c)$ is absorbed by F at both a and c, it is entirely absorbed by F.

Example 4.9 (Failed absorption). The formula $F = (a \vee b) \wedge (\overline{b} \vee c) \wedge (\overline{b} \vee \overline{c} \vee d)$ implies, but does not absorb, the clause $(a \vee d)$. This clause is not absorbed since falsifying d yields no unit propagations in F.

The only difference between a learnt clause and an absorbed (non-learnt) clause is that the learnt clause is contained in the memory of the solver. If someone would watch the solver without seeing its database that person would not be able to determine whether a clause is learnt or only absorbed by observing the running solver since the same propagations are carried out in both cases.

Lemma 4.10 presents several important properties of absorption. For brevity these properties are not proven in this report. The first property is arguably the most intuitive — absorption is supposed to capture the sense of learning a clause implicitly, therefore any learnt clause should also be absorbed. The second property can be useful in the same way the weakening rule can be useful when arguing about resolution. The third property implies that by learning clauses, a CDCL algorithm cannot "unabsorb" a clause. This is important because then we know that whichever new clauses the algorithm learns it cannot lose any of the progress made in terms of absorption.

Lemma 4.10 ([AFT11]). *For clauses A and B and sets of clauses D and E the following statements hold.*

1. *If $A \in D$ then A is absorbed by D.*
2. *If $A \subseteq B$ and D absorbs A then D absorbs B.*
3. *If $D \subseteq E$ and D absorbs A then E also absorbs A.*

Now since we have defined the notion of implicitly learning a clause we can implicitly learn — absorb — an entire resolution refutation step by step. To calculate the time it takes to absorb an entire resolution refutation we calculate the time it takes to absorb the next clause in the refutation given that the first part of the refutation is already absorbed. To make sure that the next clause can always be absorbed efficiently we are only interested in a certain type of clauses which are easily absorbed. In this report such clauses will be called semiabsorbed clauses. Those are clauses for which there exist so called *beneficial rounds*, as defined below.

Definition 4.11 (Beneficial round [AFT11]). An inconclusive round is beneficial for a clause C at a literal x, $x \in C$ if the round falsifies $C \backslash \{x\}$ but does not assign a value to x and extending that round through falsifying x through a decision creates a conclusive round. That conclusive round is also said to be beneficial for C at x. Also all decisions made during the round must assign values to variables contained in C, i.e. the round cannot decide on variables not contained in C.

A beneficial round for a clause C at the literal x indicates that the solver almost absorbs C at x. I.e. we could think of a beneficial round as a step towards absorption.

4.2 Absorbing a resolution refutation

We showed earlier that a CDCL solver terminates after absorbing an entire resolution refutation. In this chapter we investigate how a resolution refutation can be

absorbed so that we can then bound the running time of a CDCL solver. The first step towards showing that CDCL solvers can efficiently absorb resolution proofs is to show that in each proof that is not yet absorbed by a certain CDCL solver there is some clause which is semiabsorbed by that solver. We can then calculate how many conflicts and restarts it takes to absorb a semiabsorbed clause and finally (since a clause can never be "unabsorbed" by Lemma 4.10) we can calculate how many conflicts and restarts it takes to absorb an entire resolution refutation. Lemma 4.12 is used as a helper lemma for Lemma 4.13 which shows that any resolution refutation that is not fully absorbed contains a semiabsorbed clause.

Lemma 4.12 ([PD11]). *For any CNF formula F and clauses A and B absorbed by F, if $C = Res(A, B)$ and C is not absorbed by F then there exists a beneficial round for C, starting with F.*

Proof. Assume that x is the variable being resolved over to derive C from A and B. Then $A = (A' \vee x), B = (B' \vee \overline{x})$ and $C = (A' \vee B')$. Since C is not absorbed by F there must be some round which falsifies all literals in C except one, without the last literal being assigned. This round is beneficial for C since by falsifying the remaining literal in C we reach a conflict (after unit propagation). A conflict is reached because A and B are absorbed by F and therefore both x and $\overline{x}$ must be propagated when falsifying C. Propagating x and $\overline{x}$ yields a conflict since a variable cannot be assigned two distinct values simultaneously. Note that the round described here does not necessarily decide only upon variables in C (which is one of the criteria for a round being beneficial). However, it has been shown that given a round assigning a certain set of variables in C there exists a round deciding only upon variables in C that makes the same assignments on variables in C. There is a formal proof of this statement in [AFT11] but in this report it has been left out for brevity. Intuitively the statement holds because we can simply ignore decisions on variables not in C from the original round and instead decide only upon variables in C. In this way we can reproduce assignments of variables in C from the original round without producing any propagations that are not made in the original round since we are never assigning any variables that the original round is not assigning. □

Lemma 4.13 ([PD11]). *For any CNF formula F, which cannot be refuted using only unit resolution, and a refutation of that formula $\pi : F \vdash \bot$ there exists a clause in π which is semiabsorbed with respect to F.*

Proof. Simply pick the first non-absorbed clause in the refutation π. There is some such clause in the refutation since the refutation is not yet fully absorbed — if the refutation would be absorbed then F would be refutable using only unit resolution. The chosen clause cannot lie in the initial formula since by Lemma 4.10 every clause in the initial formula is absorbed. Therefore the chosen clause can be resolved from two earlier clauses in the proof and those clauses are absorbed by F since the chosen clause is the first non-absorbed clause in the proof. Thus by Lemma 4.12 there exists a beneficial round for the chosen clause and that clause is therefore semiabsorbed with respect to F. □

Now a refutation of the input formula is also a refutation of the clause database of the solver since that database contains the input formula. Therefore, as we run our CDCL solver its database can be plugged into Lemma 4.13 together with a refutation of the input formula. Thus, at every step of a CDCL run there is some clause in the given refutation that is semiabsorbed by the current CDCL database.

Now all we need to do is to show how efficiently a CDCL solver can absorb a semiabsorbed clause. Then at each step of the CDCL run the solver will work on absorbing the next semiabsorbed clause. When this clause has been absorbed there will be a new semiabsorbed clause waiting to be absorbed and this procedure will reiterate until the entire refutation is absorbed.

Lemma 4.14 ([PD11]). *Any conflict clause of a CDCL solver can be derived using a trivial resolution proof from the database of that solver at the time of the clause being conflicting.*

Lemma 4.14 was proven in [PD11] and will not be proven here. Essentially, Lemma 4.14 holds because conflict clauses (Definition 4.3) can be reached from the currently conflicting database clause C by replacing variables in C with the clauses from which those variables were propagated. I.e. if $C = (\overline{a} \vee b)$ is the conflicting database clause and a was propagated because of the clause $E = (x \vee \overline{y} \vee a)$ we can resolve C with E over a since we know that one clause contains $\overline{a}$ and the other contains a. This procedure corresponds to trivial resolution since one of the clauses resolved will always be from the database — the one which propagated the variable we want to resolve over. Moreover each variable can only be propagated once which means we will never resolve over the same variable twice.

The next step towards showing how to efficiently absorb an entire resolution refutation is to show what it takes to absorb a semiabsorbed clause. This step is partially performed in Lemma 4.15. When we know what it takes to absorb a single semiabsorbed clause we know that this procedure can be performed at most S times where S is the size of the resolution refutation we want to absorb. This limit exists because each clause in the refutation can be absorbed at most once as implied by Lemma 4.10. To show that CDCL solvers p-simulate resolution we can then simply bound the size of the proof created when absorbing a semiabsorbed clause. To instead bound the running time of a CDCL solver using a random decision scheme we need to calculate the probability of absorbing a semiabsorbed clause after a certain number of rounds.

Lemma 4.15. *Let D be a set of clauses and C be a clause semiabsorbed by D such that there exists a beneficial round for C at the literal x, starting from D. Then a CDCL solver using an asserting learning scheme and D as initial database absorbs C at x after at most n conclusive beneficial rounds for C at x, where n is the number of variables in D.*

Proof. When we perform a beneficial round for C at x we first falsify all literals in C except x. The next decision made is that to falsify x, which will make the solver reach a conflict and learn a new clause. Since we are using an asserting learning scheme (i.e. we only learn asserting clauses, described in Definition 4.4) the next time we perform exactly the same round there will be one more propagation just after falsifying $C \setminus \{x\}$. After at most n such rounds we will have absorbed C at x since then all possible propagations (including x) from $C \setminus \{x\}$ are contained in our database. Note that the negation of x is not a valid propagation since a clause learnt by the solver must be falsified at the round it is learnt (by the definition of a conflict clause). Therefore, a clause learnt during a beneficial round falsifying x cannot contain $\overline{x}$ since then the clause would be satisfied by that beneficial round. □

4.3 The CDCL proof system p-simulates resolution

This section contains the theoretical CDCL efficiency result showing that (a theoretical model of) CDCL solvers p-simulate resolution. We discuss this result before the one using an implementable solver because when deriving the more theoretical result we get to choose the order in which our solver makes decisions (i.e. we use a branching sequence, Definition 4.5). Later, when we want to show results regarding an implementable solver a random decision scheme will be used. The random decision scheme is more difficult to argue about since we cannot choose which decisions it will make.

Lemma 4.16 ([PD11]). *Let F be a set of clauses with n variables, and let C be a clause in $\pi : F \vdash \bot$ such that C is semiabsorbed by F. Let S be a CDCL solver with an asserting learning scheme and starting with clause database F. Then there exists a branching sequence B such that if D is the database of S after running with B as branching sequence, the following statements hold.*

- *D absorbs C.*
- *$|Der(F, B)| \in \mathrm{O}(n^4)$*

Proof. According to Lemma 4.15 it takes at most n conclusive beneficial rounds to absorb a semiabsorbed clause C at a certain literal. Since there are at most n literals in C it takes at most n^2 rounds to entirely absorb C. During each round the solver used learns one clause, meaning that the solver should need to learn at most n^2 clauses to absorb a semiabsorbed clause C. But during each such round the solver might backjump and then propagate some variables and reach a new conflict before restarting. There can be at most n such backjumps and conflicts per round since each backjump removes one decision (i.e. one decision level) and there are at most n decision levels per round. For each such backjump the solver might learn a new clause, therefore the solver learns at most n^3 clauses when absorbing a semiabsorbed clause. Finally, by Lemma 4.14 each clause learnt has a trivial proof, and each trivial proof has size at most n since all resolved variables in a trivial proof are distinct. Therefore absorbing a semiabsorbed clause yields a CDCL proof of size at most n^4. □

In Lemma 4.16 we obtain the length of a proof for absorbing a single clause in a resolution refutation. This result will now be used (Theorem 4.17) to calculate the length of a CDCL proof needed to absorb an entire resolution refutation and thus refute the given formula.

Theorem 4.17 (CDCL solvers p-simulate resolution [PD11]). *Let F be an unsatisfiable CNF formula and $\pi : F \vdash \bot$ be a resolution proof. Then there exists a branching sequence B such that for a CDCL solver, using an asserting learning scheme, with F as input and D as database after running with branching sequence B, D absorbs π and $|Der(F, B)| \leq n^4 S(\pi)$.*

Proof. By Lemma 4.13, in each resolution refutation of a formula F there exists some clause which is semiabsorbed by F. Therefore if we run a CDCL solver starting with the formula F, at each step of the run there will be some clause in π semiabsorbed by the database of the solver. Also by Lemma 4.16 a clause semiabsorbed by a certain formula can be absorbed by a solver starting with that formula using a CDCL proof of size at most n^4. Now all we need to do is run the CDCL algorithm and in each step absorb the next clause in π semiabsorbed from the current database. Since, by Lemma 4.10 each clause in a proof can be absorbed at most once this results in a CDCL proof of size at most $n^4 S(\pi)$. □

4.4 Running time of an implementable CDCL solver

Instead of calculating the proof size for a CDCL solver using a branching sequence to absorb a resolution proof we now calculate the running time of a solver using a random decision scheme. We should note that even though a random decision scheme is implementable in practice it is seldom used. It is more common to use heuristics such as the VSIDS decision scheme (described in Section 3.2.4). However, some of the proofs described in this section rely upon the use of a random decision scheme and it is not clear whether these proofs could be rewritten to incorporate other decision schemes.

To calculate the running time of a solver using a random decision scheme we start by looking at how many rounds of random decisions it takes to reach a beneficial round for a certain clause, this is done in Lemma 4.18. Using Lemma 4.15 we can then calculate how long it takes to absorb a semiabsorbed clause. Finally, we can simply apply this calculation to each clause in an entire resolution proof to find out how long it takes to absorb the proof.

Lemma 4.18 ([AFT11]). *Let F be a set of clauses and let C be a clause semiabsorbed by F. Let $D_1, \dots D_t$ be the different databases of a CDCL solver (with input F) at the beginnings of t consecutive rounds. The probability of none of the t rounds being beneficial for C and none of $D_1, \dots, D_t$ absorbing C is at most $e^{-t/(2n)^k}$, where k is the width of C, i.e. the number of literals in C, and n is the number of variables in the F.*

Proof. Let $\mathcal{R}_i$ be the probability that the ith round is beneficial for C and let $\mathcal{D}_i$ be the probability that D_i absorbs C. Then the probability we want to bound is the following.

$$\begin{aligned}\Pr\left[\bigcap_{i=1}^{t} \overline{\mathcal{R}}_i \cap \overline{\mathcal{D}}_i\right] &= \prod_{i=1}^{t} \Pr\left[\overline{\mathcal{R}}_i \cap \overline{\mathcal{D}}_i \;\middle|\; \overline{\mathcal{R}}_i \cap \overline{\mathcal{D}}_i\right] \\ &\leq \prod_{i=1}^{t} \Pr\left[\overline{\mathcal{R}}_i \;\middle|\; \overline{\mathcal{R}}_i\right]\end{aligned}$$

Thus, we reach an expression where all we need to calculate is the probability of not reaching a beneficial round given that none of the earlier rounds have been beneficial, and C not being absorbed. We can produce an upper bound for not reaching a beneficial round through calculating a lower bound for reaching a beneficial round. To calculate such a bound we will make use of the random decision scheme. The lower bound can be calculated by looking at the probability of randomly deciding on assignments such that the beneficial round is carried out. To obtain such a round we must falsify C in the same order as the beneficial round produced by Lemma 4.12:

$$\Pr\left[\mathcal{R}_i \mid \mathcal{R}_i\right] \geq \frac{1}{2n}\frac{1}{2(n-1)} \cdots \frac{1}{2(n-k+1)} \geq \frac{1}{(2n)^k}$$

Note that the bound shown above is a lower bound, it could be that some of the literals in the clause are falsified by unit propagation in which case the solver would not need to decide on falsifying assignments for those literals. Finally, we plug the bound on the probability of performing a beneficial round for C into the

bound on the probability of none of t rounds being beneficial for C.

$$\prod_{i=1}^{t} \Pr\left[\overline{\mathcal{R}}_i \,\middle|\, \overline{\mathcal{R}}_i\right] \leq \prod_{i=1}^{t} \left(1 - \frac{1}{(2n)^k}\right) = \left(1 - \frac{1}{(2n)^k}\right)^t \leq e^{-t/(2n)^k}$$

□

In [AFT11] the corresponding result of Lemma 4.18 yields a probability of $e^{-t/4n^k}$ instead of $e^{-t/(2n)^k}$. The result of Lemma 4.18 is worse because it does not use the property that the assignments of any inconclusive round can be reordered without changing the set of assignments carried out during the round (since this property is not proven in this report).

As mentioned earlier, now that we have a bound on the probability of reaching a beneficial round for a certain clause after a number of rounds we can calculate the number of rounds needed before absorbing that clause. Such a calculation can be carried out by declaring a new set of variables to represent the progress made when trying to absorb a clause. The variable u_{lCD} denotes the number of variables not being assigned during an inconclusive beneficial round for C at l, starting at D. If there is no such round u_{lCD} is 0. u_{CD} is defined in the following way.

$$u_{CD} = \sum_{l \in C} u_{lCD}$$

Thus, when u_{CD} is 0 the clause C is absorbed by the database D since then all variables are assigned during a beneficial round for C (or there is no beneficial round for C). It can be shown [AFT11] that the values of these progress variables never increase during a CDCL solver run. It can also be shown that when the solver, using an asserting learning scheme, reaches a beneficial round for a clause C and learns a conflict clause for that round the corresponding progress value will strictly decrease. This decrease stems from the use of an asserting learning scheme which ensures that some variable assigned at the highest level of the corresponding conclusive beneficial round instead will be propagated at some earlier level if the same round is carried out again.

Lemma 4.19 ([AFT11]). *Given a sequence of events* $e_0 \ldots e_i$ *for which* $\Pr[\overline{e}_0] = 0$ *and* $\Pr[\overline{e}_j \mid \overline{e}_j] \leq p$, $j \in \{1, \ldots, i\}$ *then* $\Pr[\overline{e}_i] \leq ip$.

Proof. By the law of total probability the following statement holds.

$$\Pr[\overline{e}_i] =$$
$$\Pr[\overline{e}_i \mid \overline{e}_i] \Pr[e_{i-1}] + \Pr[\overline{e}_i \mid \overline{e}_i] \Pr[\overline{e}_{i-1}] \leq$$
$$\Pr[\overline{e}_i \mid \overline{e}_i] + \Pr[\overline{e}_{i-1}]$$

Thus we have bounded $\Pr[\overline{e}_i]$ recursively, this bound can be written explicitly as is shown below.

$$\Pr[\overline{e}_i] \leq \Pr[\overline{e}_0] + \sum_{j=1}^{i} \Pr[\overline{e}_j \mid \overline{e}_j] \leq ip$$

□

Lemma 4.19 bounds the probability of the final event — in a sequence of events — happening given that each event in the sequence is dependent on the previous event. Below, we represent the progress made during a CDCL solver run by some sequence of events. We use this representation to bound the probability of making progress by using Lemma 4.19.

Lemma 4.20 ([AFT11]). *Let D be a set of clauses, C be a clause semiabsorbed with respect to D and S be CDCL solver with an asserting learning scheme, choosing decision variables randomly and using D as initial database. The probability of S not absorbing C after tnk rounds is at most $nke^{-t/(2n)^k}$, where k is the width of C and n is the number of variables in D.*

Proof. Let D_i be the database at the beginning of the ith round ($D_0 = D$). We want to bound the probability that C is not absorbed by D_{tnk}. To bound this probability we can define e_i to be the event that $u_{CD_{it}} \leq u_{CD} - i$. Because $u_{CD_0} = u_{CD}$ we have $\Pr[\overline{e}_0] = 0$ and because of Lemma 4.18 we get $\Pr[\overline{e}_j \mid \overline{e}_j] \leq e^{-t/(2n)^k}$ for $j > 0$. Therefore we can plug in the sequence $e_0, \ldots, e_{u_{CD}}$ into Lemma 4.19 to get

$$\Pr[\overline{e}_{u_{CD}}] \leq u_{CD} e^{-t/(2n)^k} \leq nke^{-t/(2n)^k}$$

where the last step holds since $u_{CD} \leq nk$ because $u_{lCD} \leq n$ and u_{CD} is the sum of u_{lCD} for k different literals $l \in C$. □

Lemma 4.20 bounds the probability of absorbing a semiabsorbed clause after a certain number of rounds. Now all we need to do to bound the running time of a CDCL solver is to apply Lemma 4.20 to absorb an entire resolution proof. This action is performed in Theorem 4.21 which is (a simplified version of) the main result of [AFT11].

Theorem 4.21 ([AFT11]). *Given a CNF formula F and a refutation $\pi : F \vdash \bot$, such that $W(\pi) = k$ and $S(\pi) = m$. With probability at least 1/2 a CDCL solver using an asserting learning scheme and picking decision variables randomly returns UNSAT after $mk \ln(2mnk)(2n)^{k+1}$ conflicts and restarts.*

Proof. As noted earlier if the solver absorbs an entire resolution proof (or at least every clause except the empty clause) then the solver terminates without making any more decisions. Let C_i be the ith clause of π and let D_{it} be the database of the solver after $itnk$ rounds. Let e_i be the event that D_{it} absorbs the clauses $C_1, \ldots, C_i$ (we can let e_0 be an event that is always true to be able to use Lemma 4.19). From Lemma 4.20 we know that $\Pr[\overline{e}_i \mid \overline{e}_i] \leq nke^{-t/(2n)^k}$ and by letting $t = \ln(2mnk)(2n)^k$ we get $\Pr[\overline{e}_i \mid \overline{e}_i] \leq 1/2m$. Then by using Lemma 4.19 we get the following statement.

$$\Pr[\overline{e}_m] \leq \frac{m}{2m} = \frac{1}{2}$$

Event e_m corresponds to the solver database absorbing π after $mtnk = mk \ln(2mnk)(2n)^{k+1}$ rounds. □

As mentioned earlier the bound produced in Lemma 4.18 is not as good a bound as that in [AFT11]. This bound also affects the results following that lemma. Therefore in Theorem 4.21 we reach the conclusion that a CDCL solver (with certain settings) terminates with probability at least 1/2 after $mk \ln(mnk)(2n)^{k+1}$ conflicts and restarts while in [AFT11] the corresponding number of conflicts and

restarts is $4km\ln(4knm)n^{k+1}$. It is also possible to show ([AFT11]) that a refutation of width k over n variables has length at most $4n^k$. Plugging this expression into the main result of [AFT11] yields the result that a CDCL solver, running on a formula with n variables and refutable in width k, terminates with probability at least 1/2 after $16k(k+1)\ln(16kn)n^{2k+1}$ conflicts and restarts.

Chapter 5

MiniSat

In order to perform experiments on a modern CDCL solver an already existing implementation of a CDCL solver was chosen. The solver used in this project is called MiniSat 2.2 and was chosen mainly because of its simplicity but also because MiniSat has performed well in several SAT solver competitions, e.g. SAT-Race 2008 [SATb] and SAT competition 2005 and 2007 [SATa]. MiniSat is a very small implementation of a CDCL solver and because of its simplicity it is easy to add new extensions to this solver. Because of both its simplicity and its performance MiniSat is one of the standard solvers to use when performing CDCL experiments and has therefore played an important role in the development of CDCL solvers. As an example of the importance and simplicity of MiniSat, recent SAT competitions have included a competition track called MiniSat hack track, where competitors make small adjustments to the MiniSat solver to try to improve its performance.

To be able to run experiments using the AFT model some changes had to be carried out in the code of the MiniSat solver. We survey the main parts of MiniSat in Section 5.1 and summarize the changes made to MiniSat in Section 5.2.

5.1 Important functions

This section presents code snippets representing the main parts of the MiniSat solver and also modifications made to the code. In some places the code has been removed or reformatted (when inserted into this report) to make it easier to read. For more information about the original MiniSat solver see e.g. [ES04]. MiniSat is written in C++. The parts of MiniSat presented in this section are, the main solver loop (Listing 5.1), the search function (Listing 5.2) called from the main loop, the clause learning function (Listing 5.3), the propagation function (Listing 5.4) and finally, the decision function (Listing 5.5).

The main solver loop of MiniSat is presented in Listing 5.1. A few adjustments have been made in that code to be able to run the AFT model. The code is essentially just calling the search function, shown in Listing 5.2, over and over again until the search function finds out whether the input formula is satisfiable. The parameter passed to the search function is the minimum number of conflicts before a restart should be triggered. When the solver wants to restart it will return from the search function to the solver loop and then rerun the search function, potentially using a new number of conflicts before the next restart. In the original MiniSat code the only way the search function was called in Listing 5.1 was through the call at line 20. In this project the calls at line 16 and line 18 were added to represent the cases where the solver restarts after each conflict and where the

```
1  lbool Solver::solve_()
2  {
3      lbool status = l_Undef;
4
5      // Search:
6      while (status == l_Undef){
7          random_round_index = random_round_freq ?
8              (random_round_index+1) % random_round_freq :
9              1;
10
11         double rest_base = luby_restart ?
12             luby(restart_inc, curr_restarts) :
13             pow(restart_inc, curr_restarts);
14
15         if( restart_mode == 2 ) // restart after each conflict
16             status = search(1);
17         else if( restart_mode == 0 ) // never restart
18             status = search(-1);
19         else // default restart policy
20             status = search(rest_base * restart_first);
21     }
22
23     return status;
24 }
```

Listing 5.1. MiniSat main solver loop

solver never restarts respectively. The default restarting policy in MiniSat is to use a Luby sequence (introduced in [LSZ93]) as can be seen at line 12. This sequence is multiplied by a factor of 100 (the variable `restart_first` at line 20), the first part of the MiniSat restarting sequence is $100, 100, 200, 100, 100, 200, 400$.

In Listing 5.1 the code at lines 7 to 9 has been added to update the variable `random_round_index`. That variable controls whether the search function is to pick decision variables randomly or through the MiniSat heuristic (VSIDS which is presented in Section 3.2.4). Whenever `random_round_index` is zero decision variables are picked randomly, otherwise the VSIDS heuristic is used.

In Listing 5.2 we see the main search loop of the MiniSat solver. That code is essentially meant to implement the pseudocode presented in Algorithm 1, i.e. the main search procedure of a CDCL solver. Here, `nof_conflicts` (line 1) determines the number of conflicts between restarts. `conflictC` (defined on line 4) is a counter keeping track of the number of conflicts since the last restart. The function `propagate()` propagates variables which are waiting to be propagated and returns a conflicting clause if a conflict was reached while propagating. The code for propagating variables can be seen in Listing 5.4. The function on line 14 `analyze(conflict, learnt_clause, backtrack_level)` is the clause learning function of MiniSat. It takes a conflict in the form of a reference to a clause, and returns the learnt clause and the decision level to backtrack to after learning the clause, the code for this function can be seen in Listing 5.3. The `cancelUntil(level)` function (line 27) backtracks to the decision level `level`, i.e. it undoes all assignments above that level. The function `reduceDB()` (line 34) implements clause removal, i.e. it removes some clauses from the database. The `nAssigns()` function returns the number of assigned variables. Each assignment has a reason for being used, which takes the form of a clause. Therefore the clauses being reasons for assignments cannot be removed from the database. Because each assignment can prevent one clause from being

```
1  lbool Solver::search(int nof_conflicts)
2  {
3      int         backtrack_level;
4      int         conflictC = 0;
5      vec<Lit>    learnt_clause;
6
7      while(true){
8          CRef confl = propagate();
9          if (confl != CRef_Undef){
10             // CONFLICT
11             conflictC++;
12             if (decisionLevel() == 0) return l_False;
13
14             analyze(confl, learnt_clause, backtrack_level);
15             cancelUntil(backtrack_level);
16
17             if (learnt_clause.size() == 1)
18                 enqueueAssignment(learnt_clause[0]);
19             else
20                 add_to_database(learnt_clause);
21
22         }else{
23             // NO CONFLICT
24             if (nof_conflicts >= 0 &&
25                 conflictC >= nof_conflicts){
26                 // Reached bound on number of conflicts:
27                 cancelUntil(0);
28                 return l_Undef; }
29
30             if (learnts_removal &&
31                 learnts.size()-nAssigns() >=
32                 max_nof_learnts_clauses)
33                 // Reduce the set of learnt clauses:
34                 reduceDB();
35
36             Lit next = lit_Undef;
37
38             // New variable decision:
39             next = pickBranchLit();
40
41             if (next == lit_Undef)
42                 // No unassigned variables left
43                 return l_True;
44
45             // Increase decision level and enqueue 'next'
46             newDecisionLevel();
47             enqueueAssignment(next);
48         }
49     }
50 }
```

Listing 5.2. MiniSat search loop

deleted the number of assignments is subtracted from the number of learnt clauses when checking whether the database should be reduced (line 31). The variable `learnts_removal` (line 30) was added as a part of this project to be able to turn off clause removal. The function `pickBranchlit()` decides upon a variable to assign. The code for this function can be seen in Listing 5.5.

The code displayed in Listing 5.3 is the MiniSat clause learning algorithm. The algorithm implemented is called 1-UIP and applies trivial resolution starting with the current conflict clause until the resulting clause contains only one variable of the highest decision level (thus 1-UIP is an asserting learning scheme). The variable `pathC` (line 6) keeps track of the number of literals which need to be resolved over. The for-loop ranging from line 18 to line 28 essentially iterates through the newest clause that is being resolved to check which literals in that clause have not been resolved over (line 21). On line 23 the algorithm checks if the current literal is of the highest decision level — if so it should be resolved over, otherwise the literal is directly added to the output clause (line 26). At lines 31 to 35 the next literal to resolve over, and its reason clause (the next clause to resolve), are found. The algorithm resolves over literals in reverse order of the `trail` which is the sequence of assignments made during the current round. When there is only one literal of highest decision level left the variable `pathC` will be 0 and the while-loop at line 37 will terminate. The literal of highest decision level will then have been stored in the variable `p` which is added to the output clause at line 38. `simplify_conflict_clause(conflict_clause)` at line 40 represents a block of code simplifying the learnt clause by removing redundant literals.

Listing 5.4 shows the MiniSat code for propagating a literal. The most important concepts here are *watchers* and *blockers*. Watchers are intended to speed up the operation of checking whether a clause is reduced to a unit clause. There are two watchers per clause and each of these are connected to a literal. Whenever a literal becomes falsified, instead of restricting each clause containing that literal the solver restricts only the clauses containing watchers connected to that literal. The restriction is carried out through finding a new (non-falsified) literal to watch, if no such literal can be found the clause is a unit clause (under the restriction of the current assignments). However, if the second watcher for the current clause is also falsified (which could have happened while the first watcher was still waiting to be updated) the clause is conflicting. Aside from watchers each clause also has a literal called a blocker. The reason for having such a literal is to be able to skip examining a clause which is satisfied. If the blocker is satisfied then the clause is also satisfied and can be ignored when watchers are updated.

What the code in Listing 5.4 does is to propagate all literals currently on the trail (the current sequence of assignments) which have not yet been assigned. For each such literal, the solver goes through the watchers associated with that literal (line 11). For each watcher it checks whether the corresponding clause is satisfied by first checking its blocker literal (line 14) and then checking the other watcher of the clause (line 30). If the clause is satisfied it does not need to be inspected any further and the solver thus skips ahead to the next watcher. If the blocker and the first watcher of the given clause are not satisfied then the solver searches for a new watcher (a literal which is not falsified, line 34 to line 38), if one is found the current falsified watcher is replaced by the new one. If a new watcher is not found then the clause is either a unit clause under the reduction of the current assignments, or the solver has reached a conflict. If the clause is unit then the single literal of that unit clause is added to the trail (line 50), otherwise no more literals are propagated (line 44 to line 48) and the function returns a reference to

```
1  void Solver::analyze(
2        CRef confl,
3        vec<Lit>& out_learnt,
4        int& out_btlevel)
5  {
6      int pathC = 0;
7      Lit p     = lit_Undef;
8
9      // Generate conflict clause:
10     //
11     out_learnt.push(); // leave room for the asserting literal
12     int index = trail.size() - 1;
13
14     do{
15         Clause& c = ca[confl];
16
17         int start = (p == lit_Undef) ? 0 : 1;
18         for (int j = start; j < c.size(); j++){
19             Lit q = c[j];
20
21             if (!seen[var(q)] && level(var(q)) > 0){
22                 seen[var(q)] = 1;
23                 if (level(var(q)) >= decisionLevel())
24                     pathC++;
25                 else
26                     out_learnt.push(q);
27             }
28         }
29
30         // Select next clause to look at:
31         while (!seen[var(trail[index--])]);
32         p     = trail[index+1];
33         confl = reason(var(p));
34         seen[var(p)] = 0;
35         pathC--;
36
37     }while (pathC > 0);
38     out_learnt[0] = ~p;
39
40     simplify_conflict_clause(out_learnt);
41
42     out_btlevel = find_backtrack_level();
43
44     for (int j = 0; j < analyze_toclear.size(); j++)
45        seen[var(analyze_toclear[j])] = 0;    // clear seen[]
46 }
```

Listing 5.3. MiniSat clause learning

```
1  CRef Solver::propagate()
2  {
3      CRef    confl     = CRef_Undef;
4      while (qhead < trail.size()){
5          // 'p' is enqueued fact to propagate.
6          Lit            p   = trail[qhead++];
7          vec<Watcher>&  ws  = watches.lookup(p);
8          Watcher        *i, *j, *end;
9
10         i = j = (Watcher*)ws;
11         for (end = i + ws.size(); i != end;){
12             // Try to avoid inspecting the clause:
13             Lit blocker = i->blocker;
14             if (value(blocker) == l_True){
15                 *j++ = *i++; continue; }
16
17             // Make sure the false literal is data[1]:
18             CRef     cr        = i->cref;
19             Clause&  c         = ca[cr];
20             Lit      false_lit = ~p;
21             if (c[0] == false_lit)
22                 c[0] = c[1], c[1] = false_lit;
23             assert(c[1] == false_lit);
24             i++;
25
26             // If 0th watch is true,
27             // then clause is already satisfied.
28             Lit     first = c[0];
29             Watcher w     = Watcher(cr, first);
30             if (first != blocker && value(first) == l_True){
31                 *j++ = w; continue; }
32
33             // Look for new watch:
34             for (int k = 2; k < c.size(); k++)
35                 if (value(c[k]) != l_False){
36                     c[1] = c[k]; c[k] = false_lit;
37                     watches[~c[1]].push(w);
38                     goto NextClause; }
39
40             // Did not find watch --
41             // clause is unit under assignment:
42             *j++ = w;
43             if (value(first) == l_False){
44                 confl = cr;
45                 qhead = trail.size();
46                 // Copy the remaining watches:
47                 while (i < end)
48                     *j++ = *i++;
49             }else // enqueue assignment and save reason clause
50                 enqueueAssignment(first, cr);
51
52         NextClause:;
53         }
54         ws.shrink(i - j);
55     }
56     return confl;
57 }
```

Listing 5.4. MiniSat propagation

the conflicting clause.

The modified MiniSat code for making decisions on variables can be seen in Listing 5.5. The default mode for MiniSat is to make decisions using the VSIDS scheme (explained in Section 3.2.4). Actually, the VSIDS scheme used in MiniSat 2.2 is a variant of the original VSIDS scheme. In MiniSat, whenever a variable propagation is carried out the reason for that propagation is stored in the solver. The reason is a clause — the clause from which the propagation originated. Reason clauses are used when deriving clauses to be learnt from conflicts. In MiniSat 2.2 the activity of all variables in all reason clauses used when learning a new clause is increased. In the original VSIDS scheme only the activity of variables in the learnt clause is increased. A heap is used to store variable activity. The variable to be chosen is the one with the highest activity of all the unassigned variables. We should note that in the original VSIDS decision scheme whenever several variables share the highest activity one of those variables is chosen randomly. In MiniSat however, the order of the activity heap is used which means that when several variables share the highest activity randomness is not used to break the tie (though activities are represented by real values so ties should be rare).

When it comes to making only uniformly random decisions during an entire round the heap can easily be used to make random decisions efficiently. Such decisions can be made because the heap (used in MiniSat) provides a method for removing an arbitrary element from the heap, and also because the underlying array is packed (all elements lie within the same block, there are no holes in the array). Therefore all you need to do to pick an element randomly from the heap is to pick a random index in the underlying array and remove the corresponding element from the heap. This procedure is performed in Listing 5.5 on lines 14 to 16. Picking a variable based on activity is done on line 28. An alternative for using the heap to make random decisions is to add an additional data structure to the solver to store variables in. The use of an additional data structure could speed up the making of random decisions. However, the heap was chosen for making random decisions in order to make as small an impact as possible on the MiniSat solver. The while-loops at lines 6 and 21 make sure that the solver keeps picking variables from the heap until it has found one which has not yet been assigned a value. The choice of polarity (the value of an assignment, 0 or 1) should also be mentioned. In this project the rnd_pol variable seen on line 38 and line 39 is always set to 1 which means that during random rounds polarity is chosen randomly. During non-random rounds (VSIDS rounds) the polarity is chosen using a polarity vector which stores the latest used polarity for each variable.

5.2 Summary of code modifications

Here follows a summary of the changes made to MiniSat to be able to use the settings of the AFT model. The differences between the CDCL components used in the two articles [AFT11] and [PD11] and MiniSat can be seen in Table 5.1. The learning scheme of MiniSat was not modified since it was already asserting (1-UIP). Turning off clause removal was simply a matter of adding an option and checking the value of that option every time clause removal were to be used (as can be seen in Listing 5.2 on line 30). Similarly the number of conflicts between restarts can be passed as a parameter to the MiniSat search function (Listing 5.2) — this is done in Listing 5.1 from line 15 to line 20. To include entire rounds where decisions are made uniformly at random first the possibility to perform random

```
1  Lit Solver::pickBranchLit()
2  {
3      Var next = var_Undef;
4
5      if( rnd_round_index == 0 ){ // random round
6          while (next == var_Undef ||
7                  value(next) != l_Undef ||
8                  !decision[next]){
9              if( order_heap.empty() ){
10                 next = var_Undef;
11                 break;
12             }else{
13                 // choose randomly from order_heap
14                 next = order_heap[
15                     irand(random_seed,order_heap.size())];
16                 order_heap.remove(next);
17             }
18         }
19     }
20     else{ // activity based decision
21         while (next == var_Undef ||
22                 value(next) != l_Undef ||
23                 !decision[next])
24             if (order_heap.empty()){
25                 next = var_Undef;
26                 break;
27             }else
28                 next = order_heap.removeMin();
29     }
30
31     // Choose polarity
32     // user_pol is undefined by default
33     // rnd_pol is 1 by default
34     if (next == var_Undef)
35         return lit_Undef;
36     else if (user_pol[next] != l_Undef)
37         return mkLit(next, user_pol[next] == l_True);
38     else if (rnd_pol == 2 ||
39              (rnd_pol == 1 && rnd_round_index == 0) )
40         return mkLit(next, drand(random_seed) < 0.5);
41     else
42         return mkLit(next, polarity[next]);
43 }
```

Listing 5.5. MiniSat assignment decision

	[AFT11]	[PD11]	MiniSat
Clause removal	Off	Off	On
Conflicts between restarts	1	1	Luby $\times$ 100
Decision scheme	Random	Optimal	VSIDS
Learning scheme	Any asserting	Any asserting	1-UIP (asserting)

Table 5.1. Comparison of CDCL models

decisions was added (Listing 5.5 from line 14 to line 16). Then a counter was added that would determine what kind of round the solver is running. That counter is called `random_round_index` and is updated on lines 7 to 9 in Listing 5.1.

Chapter 6

Experiments

This chapter describes how the experiments performed during this project were set up. The goal of these experiments is to investigate how the theoretical results shown in Chapter 4 compare to practice. The formulas used during the experiments are presented in Section 6.1 and the different parameters used are presented in Section 6.2. These parameters affect the running time of the experiments and determine what kind of conclusions we can draw from the experiments (e.g. increasing the number of data points increases the reliability of the results). Finally, a specification of the hardware used for the experiments is presented in Section 6.3.

6.1 Experiment formulas

When choosing formulas an important constraint is for each formula to be refutable in low (constant) width. This constraint stems from the results of Chapter 4 which suggest that CDCL solvers should be able to solve formulas refutable in low width efficiently.

6.1.1 Tseitin formulas

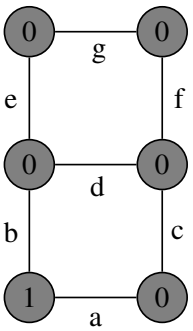
A Tseitin formula [Tse68] is a formula representing the fact that the sum of vertex degrees of a graph is even. This principle can be encoded in the following way. Given a graph G, a boolean function $f : V(G) \to \{0, 1\}$ over the vertices $V(G)$ of G, is odd-weight if the sum of the function's values over all vertices of G is odd, i.e.

$$\sum_{v \in V(G)} f(v) \equiv 1 \mod 2.$$

When creating a Tseitin formula from a certain graph, an odd-weight function is used to label the nodes of the graph. Given a graph G, let each edge e in G be represented by a variable x_e and then create a parity constraint

$$PARITY_v : \sum_{v \in e} x_e \equiv f(v) \mod 2$$

for each vertex v in G. The expression $v \in e$ means that e is an edge incident to the vertex v. A parity constraint for the vertex v can be written as a CNF formula of size at most $2^{d(v)-1}$, and width $d(v)$, where $d(v)$ is the degree of v. A CNF clause is falsified only by one specific assignment of its variables. Therefore, to create a CNF clause representing a parity constraint we can use exactly those clauses which are falsified by the assignments which falsifies the parity constraint. For example the constraint $a + b \equiv 1 \mod 2$ has two falsifying assignments: $a = 0, b = 0$ and



(a) Graph with labeled vertices

$$\begin{aligned} a+b &\equiv 1 \mod 2 \\ a+c &\equiv 0 \mod 2 \\ b+d+e &\equiv 0 \mod 2 \\ c+d+f &\equiv 0 \mod 2 \\ e+g &\equiv 0 \mod 2 \\ f+g &\equiv 0 \mod 2 \end{aligned}$$

(b) PARITY constraints

$$\begin{array}{ll} a \vee b & \overline{a} \vee \overline{b} \\ a \vee \overline{c} & \overline{a} \vee c \\ \overline{b} \vee d \vee e & b \vee \overline{d} \vee e \\ b \vee d \vee \overline{e} & \overline{b} \vee \overline{d} \vee \overline{e} \\ \overline{c} \vee d \vee f & c \vee \overline{d} \vee f \\ c \vee d \vee \overline{f} & \overline{c} \vee \overline{d} \vee \overline{f} \\ e \vee \overline{g} & \overline{e} \vee g \\ f \vee \overline{g} & \overline{f} \vee g \end{array}$$

(c) Tseitin formula

Figure 6.1. Tseitin formula example

$a = 1, b = 1$. Therefore, this parity constraint can be written as $(a \vee b) \wedge (\overline{a} \vee \overline{b})$ since $(a \vee b)$ is falsified by $a = 0, b = 0$ and $(\overline{a} \vee \overline{b})$ is falsified by $a = 1, b = 1$. The Tseitin formula

$$\tau(G, f) = \bigwedge_{v \in V(G)} PARITY_v$$

is a CNF formula of width at most $d(G)$ and size less than $|V(G)|2^{d(G)}$, where $d(G) = \max d(v) : v \in V(G)$. Such a formula can be formed by converting each parity constraint to a CNF formula and then taking the conjunction of those formulas. Thus, by choosing a graph of degree bounded to some constant, the size of our Tseitin formula $\tau(G, f)$ will be bounded linearly by the number of vertices in G and the width of the formula will be bounded to a constant. In Figure 6.1 we can see an example of a labeled graph together with the corresponding PARITY constraints and Tseitin formula. A Tseitin formula $\tau(G, f)$ is unsatisfiable if f is odd-weight with respect to G. This unsatisfiability holds because the parity constraints of the formula state that the sum of the values of the edge variables in the formula must be odd. But each edge variable is counted exactly two times (because each edge is connected to two vertices which together represent two parity constraints) and therefore the sum of their values must be even.

The graphs used to create Tseitin formulas during this project are narrow grids (i.e. grids of width between 2 and 5). We can see an example of a Tseitin formula representing a grid of width 2 and length 3 in Figure 6.1. The reason for using such graphs is that the width of the grid determines the width of refuting the corresponding Tseitin formula. A grid of width w can be refuted in resolution width $w + 1$ by refuting the formula layer by layer (in a straight-forward way) and only keeping track of variables connected to the current layer. It can also be shown that the width of refuting a Tseitin grid is at least the width of the grid. This result follows from a result in [BW01] which states that the width of refuting a Tseitin formula is at least the expansion of the corresponding graph (the expansion is essentially the size of the minimum cut in the graph such that the two vertex sets created by the cut have similar size). For a Tseitin grid the size of the expansion is at least the width of the grid. The size of refuting the Tseitin formula is at least the number of nodes in the grid — if we do not use this many clauses we cannot have "inspected" all nodes and thus we cannot know whether the sum of the node labels is odd. It can also be shown that the layer by layer refutation described above has

size linear in the number of nodes in the grid. This result holds because the width of the grid is constant and for each layer of the grid we need only resolve a certain number of clauses which is directly dependent on the width of the grid. Therefore, if we keep the grid width (and thus the refutation width) to a small constant CDCL solvers (or at least the AFT solver) should solve these formulas efficiently. Then it is interesting to see whether different solvers will run as efficiently as the AFT solver should do. Regarding the running time of the AFT solver, given a chain of width w, the results from Chapter 4 state that the running time should be at most $\mathrm{O}(mn^{w+1}\ln(nm)) = \mathrm{O}(n^{w+2}\ln(n))$ where m is the size of the refutation explained above and n is the number of variables in the formula.

6.1.2 Relativized pigeonhole principle formulas

Relativized pigeonhole principle formulas (RPHP) have been studied in [AMO13] and [ALN14]. RPHP formulas are based on pigeonhole principle formulas which are meant to represent the pigeonhole principle. The pigeonhole principle states that given p pigeons and h holes, $p > h$ if we fit each pigeon in a hole then there must be some hole containing at least two pigeons. A pigeonhole principle formula contains a matrix of variables $a_{i,j}$, $i \in \{1,\ldots,p\}$, $j \in \{1,\ldots,h\}$ where variable $a_{i,j}$ represents the ith pigeon staying in hole j. There are two types of clauses in such formulas. One represents the statement that a pigeon must fly to some hole, i.e.

$$\left(\bigvee_{j=1}^{h} a_{i,j}\right)$$

for each $i \in \{1,\ldots,p\}$. The other type states that no pair of pigeons can fly to the same hole, i.e.

$$(\overline{a}_{i,j} \vee \overline{a}_{i'j})$$

for $i, i' \in \{1,\ldots,p\}, i \neq i'$ and $j \in \{1,\ldots,h\}$. Pigeonhole principle formulas are unsatisfiable (if there are more pigeons than holes) since they essentially state that each pigeon must fly to a hole and no more than one pigeon can fit in each hole.

An RPHP formula is a variant of a pigeonhole principle formula where resting holes are added. In RPHP formulas pigeons first fly to resting holes and then from these resting holes fly to their final holes (there can be no more than one pigeon in each resting hole). As long as the number of destination holes is smaller than the number of pigeons these formulas are also unsatisfiable. These new formulas can essentially be encoded using a combination of two basic pigeonhole principle formulas — the first for sending pigeons to resting holes and the second for connecting occupied resting holes to destination holes. In this project the number of destination holes is always one less than the number of pigeons, i.e. $p - 1$. There are k resting holes. Thus, a matrix of variables $a_{i,j}, i \in \{1,\ldots,p\}, j \in \{1,\ldots,k\}$ can be used to encode pigeons flying to resting holes. For this matrix basic pigeonhole principle clauses are added to map pigeons to resting holes as described above. Furthermore, a sequence of variables $r_1,\ldots,r_k$ is used for representing each resting hole being occupied. To ensure that a resting hole is marked as occupied when a pigeon has flown there we can add the clauses

$$(\overline{a}_{i,j} \vee r_j)$$

for all $i \in \{1,\ldots,p\}$ and $j \in \{1,\ldots,k\}$. To represent a pigeon in resting hole j flying to destination hole d we can use a matrix containing variables $q_{j,d}$, for

$$
\begin{array}{rr}
(a_{1,1} \vee a_{1,2}) \wedge & (a_{2,1} \vee a_{2,2}) \wedge \\
(a_{3,1} \vee a_{3,2}) \wedge & (\overline{a}_{1,1} \vee \overline{a}_{2,1}) \wedge \\
(\overline{a}_{1,2} \vee \overline{a}_{2,2}) \wedge & (\overline{a}_{1,1} \vee \overline{a}_{3,1}) \wedge \\
(\overline{a}_{1,2} \vee \overline{a}_{3,2}) \wedge & (\overline{a}_{2,1} \vee \overline{a}_{3,1}) \wedge \\
(\overline{a}_{2,2} \vee \overline{a}_{3,2}) \wedge & (\overline{a}_{1,1} \vee r_1) \wedge \\
(\overline{a}_{1,2} \vee r_2) \wedge & (\overline{a}_{2,1} \vee r_1) \wedge \\
(\overline{a}_{2,2} \vee r_2) \wedge & (\overline{a}_{3,1} \vee r_1) \wedge \\
(\overline{a}_{3,2} \vee r_2) \wedge & (\overline{r}_1 \vee q_{1,1} \vee q_{1,2}) \wedge \\
(\overline{r}_2 \vee q_{2,1} \vee q_{2,2}) \wedge & (\overline{r}_1 \vee \overline{r}_2 \vee \overline{q}_{1,1} \vee \overline{q}_{2,1}) \wedge \\
(\overline{r}_1 \vee \overline{r}_2 \vee \overline{q}_{1,2} \vee \overline{q}_{2,2}) &
\end{array}
$$

Figure 6.2. RPHP formula example with 3 pigeons and 2 resting holes

$j \in \{1, \ldots, k\}$ and $d \in \{1, \ldots, p-1\}$. We then need to add some clauses to make sure occupied resting holes are connected to destination holes. Firstly, clauses

$$(\overline{r}_j \vee q_{j,1} \vee \cdots \vee q_{j,p-1})$$

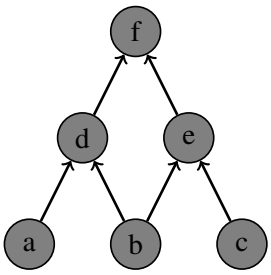
for $j \in \{1, \ldots, k\}$ ensure that a pigeon in a resting hole flies to a destination hole. Secondly, clauses

$$(\overline{r}_j \vee \overline{r}_{j'} \vee \overline{q}_{j,d} \vee \overline{q}_{j',d})$$

for $j, j' \in \{1, \ldots, k\}, j \neq j'$ and $d \in \{1, \ldots, p-1\}$ ensure that no two pigeons fly to the same destination hole. Figure 6.2 contains an example of an RPHP formula.

As is shown in [ALN14] the size of refuting a pigeonhole principle formula with resting holes using resolution is $n^{\Theta(w)}$, where n is the number of variables in the formula and w is the number of pigeons in the formula. In [ALN14], it is also shown that such a formula can be refuted in width $\mathrm{O}(w)$. Therefore, according to the results in Chapter 4, the running time of the AFT solver running on an RPHP formula should be similar to the size of the shortest resolution refutation for that formula. The reason for running experiments on these formulas is to see if practical solvers perform as well as the AFT solver should do for these formulas.

For the results of Chapter 4 to work (i.e. for an implementable CDCL solver to terminate within reasonable time) the width of the initial formula must be constant. The formula family described above does not fulfill this constraint — there are clauses of width equal to the number of pigeons in the formula (and also clauses of width equal to the number of resting holes in the formula). There is a standard way to solve such a problem — by introducing extension variables and splitting up wide clauses using these variables as "glue" to keep the original meaning of the clauses. If we have the clause $(\bigvee_{i=1}^{n} x_i)$ and then add extension variables $y_1, \ldots, y_{n-3}$ in the standard way (to create a 3-CNF formula) we get the formula $(x_1 \vee x_2 \vee y_1) \wedge (\overline{y}_1 \vee x_3 \vee y_2) \wedge \cdots \wedge (\overline{y}_{n-3} \vee x_{n-1} \vee x_n)$. Actually, the corresponding formulas used in this project are 4-CNF formulas. Such formulas can be created in a way similar to that in which 3-CNF formulas are created except one more original variable can be stored in each 4-CNF clause. I.e. using e extension variables the clause $(\bigvee_{i=1}^{n} x_i)$ would become $(x_1 \vee x_2 \vee x_3 \vee y_1) \wedge (\overline{y}_1 \vee x_4 \vee x_5 \vee y_2) \wedge \cdots \wedge (\overline{y}_e \vee x_{n-2} \vee x_{n-1} \vee x_n)$.



(a) Pebbling pyramid graph

$$
\begin{aligned}
(a)\wedge\\
(b)\wedge\\
(c)\wedge\\
(\overline{a}\vee\overline{b}\vee d)\wedge\\
(\overline{b}\vee\overline{c}\vee e)\wedge\\
(\overline{d}\vee\overline{e}\vee f)\wedge\\
(\overline{f})
\end{aligned}
$$

(b) Pebbling pyramid formula

Figure 6.3. Pebbling formula example

6.1.3 Pebbling formulas

Pebbling formulas [BW01] represent pebbling games which have been shown to be very useful when proving gaps between different proof systems (which is done for example in [BW01]). An important property of pebbling formulas is that it is possible to create different formulas which can be refuted in similar size and width but different space [Nor06, NH08, BN08].

A pebbling formula is created by taking a directed acyclic graph (DAG) with a single sink (a node without successors) and letting each node represent a variable or a small formula. Each node in the DAG must have either zero or two predecessors. The formula encodes the statement that each node in the graph is implied by its predecessors. Each source node (a node without predecessors) in the graph represents a positive variable or formula. For example if each node is represented by a single variable a source node represents a positive unit clause. The sink node represents a negative variable or formula. In the case where each node is represented by a single variable the sink represents a negative unit clause. The source nodes imply the sink node by transitivity since each node in the graph is implied by its predecessors. Since the sink is represented by a negative variable or formula each pebbling formula is unsatisfiable. See Figure 6.3 for an example of a pebbling formula where each node is represented by a single variable.

A pebbling formula where each node is represented by a single variable is refutable using only unit propagation. Such a refutation can be formed by simply propagating all source node variables (since the formula contains unit clauses for those variables). The restricted formula created through those propagations will then represent a graph where the successors of the previous source nodes are now the new source nodes. This procedure of propagating source node variables can then be repeated until the sink node is reached.

One way to create more difficult formulas is to substitute each variable in the pebbling graph using some function so that each node is instead represented by a formula. We can still refute a pebbling formula with substitutions in a similar way to that through which we refuted a formula with single variables, i.e. through recursively using source nodes to derive successors of source nodes. This procedure shows that if we can derive a certain node from its predecessors in a certain resolution width we can then use that width to bound the width of refuting the entire pebbling formula. The width of our refutation is the maximum width taken over the derivations of all nodes in the formula from their predecessors. Therefore the

structure of the pebbling graph used to create a pebbling formula does not affect the width in which the formula can be refuted (as long as the number of predecessors of each node is the same). The width of refuting a pebbling formula is determined by the substitution used. Similarly, the size of a derivation from a pair of nodes to their common successor will depend only on the substitution used and not on the size of the pebbling graph. Therefore, the size of the refutation explained above will grow linearly with the size of the given pebbling graph. Below, the pebbling graphs and substitutions used in this project are presented.

Depending on what kind of graphs you use when creating pebbling formulas you can produce formulas with different properties when it comes to refuting the formulas using resolution. The graphs used in this project are chains of width 2 and 5, pyramids and Gilbert-Tarjan graphs. Examples of such graphs can be seen in Figures 6.4 and 6.5. Gilbert-Tarjan graphs were created in [GT78] and are based on graphs shown in [PTC77]. The name "Gilbert-Tarjan graphs" is based on the names of the authors of [GT78]. The reason for using the four graph families presented above is that their corresponding pebbling formulas have different space complexities when refuted using resolution. Chain formulas have space complexity linear in the width of the chain, which in this case is constant (2 and 5). Pyramid formulas have space complexity linear in the height of the pyramid ($\Omega(\sqrt{n})$ for formulas of size n). The space complexity of Gilbert-Tarjan formulas is $\Omega(n/\log n)$ (actually $\Omega(n/\log^2 n)$ for the formulas used in this project) for formulas of size n. These space complexities are listed in [JMNŽ12] but were proven in [Nor06, NH08, BN08] by connecting space complexity of pebbling formulas to the pebbling price of the corresponding pebbling graphs.

Three different substitutions are used in this project, the first is *or* over two variables (denoted or of arity 2) which can simply be encoded as $(x_1 \vee x_2)$. An implication from the nodes $(x_1 \vee x_2)$ and $(y_1 \vee y_2)$ to the node $(z_1 \vee z_2)$ can be written as

$$(\overline{x}_1 \vee \overline{y}_1 \vee z_1 \vee z_2) \wedge (\overline{x}_1 \vee \overline{y}_2 \vee z_1 \vee z_2) \wedge (\overline{x}_2 \vee \overline{y}_1 \vee z_1 \vee z_2) \wedge (\overline{x}_2 \vee \overline{y}_2 \vee z_1 \vee z_2). \quad (6.1)$$

By using resolution, it is possible to derive $(z_1 \vee z_2)$ from $(x_1 \vee x_2)$, $(y_1 \vee y_2)$, and the clauses in (6.1) in width 4 (in a straight-forward way). Furthermore, since the clauses in (6.1) have width 4, $(z_1 \vee z_2)$ cannot be derived using width lower than 4. As described above the size of refuting a pebbling formula is linear in the size of the pebbling graph used. Therefore, a CDCL solver should be able to solve pebbling formulas with substitution or of arity two in running time $\mathrm{O}(mn^{k+1}\ln(nm)) = \mathrm{O}(n^6\ln(n))$. This calculation can be derived from the results in Chapter 4 by letting $k = 4$ be the width of refuting the formula, $m \in \mathrm{O}(n)$ the size of refuting the formula and n the number of variables in the formula.

The second substitution used in this project is *xor* over two variables, i.e. $x_1 \oplus x_2$ which can be encoded as $(x_1 \vee x_2) \wedge (\overline{x}_1 \vee \overline{x}_2)$. The encoding of an implication of a node in a pebbling formula of substitution xor of arity 2 has here been omitted for brevity. A node in such a formula can be derived from its predecessors using a resolution proof of width 6 (in a straight-forward way). This width is also the minimum width needed to derive a node from its predecessors (because clauses encoding the implication of a node have width 6). A formula using the substitution xor of arity 2 should be solvable in running time (or rather number of conflicts and restarts) $\mathrm{O}(n^8\ln(n))$ where n is the number of variables in the formula. This result can be reached by using a similar argument as for pebbling formulas with substitution or of arity 2.

The last substitution used is not-all-equal (*nae* or *neq*) of order 3 which means using three variables that cannot all be equal. A formula created from such a sub-

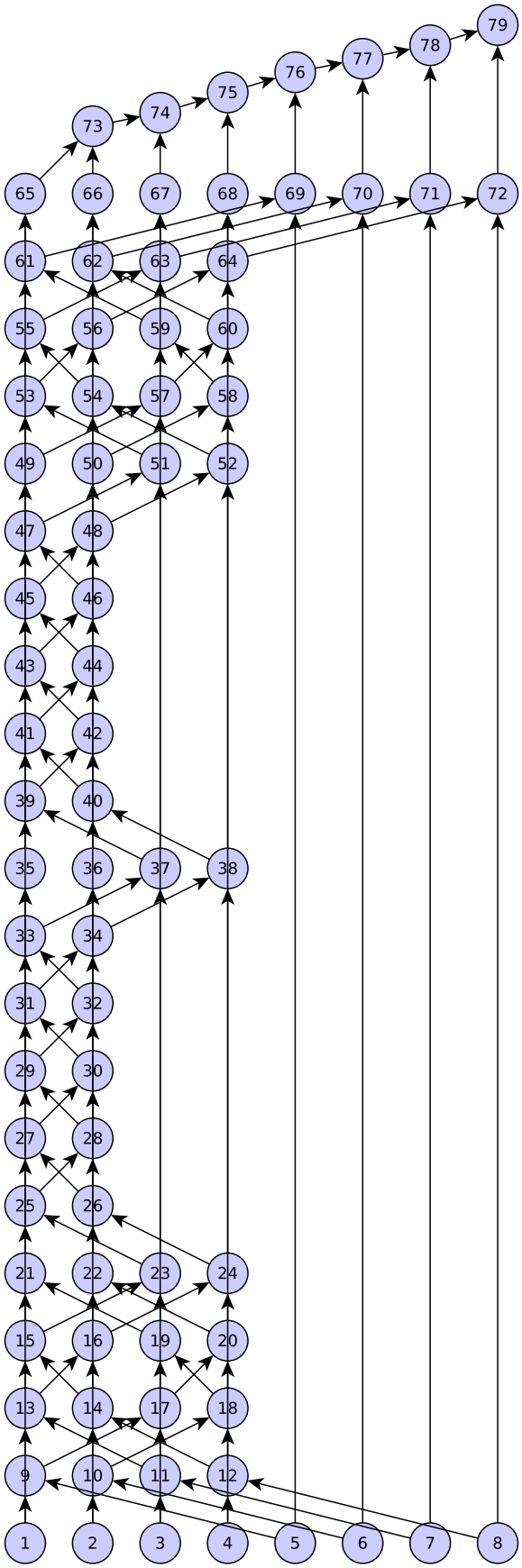
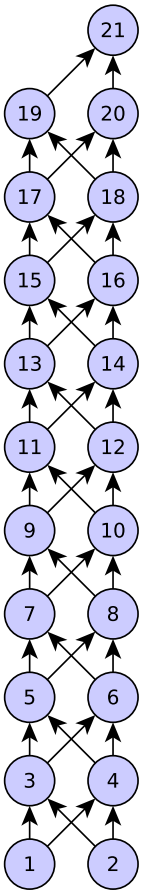
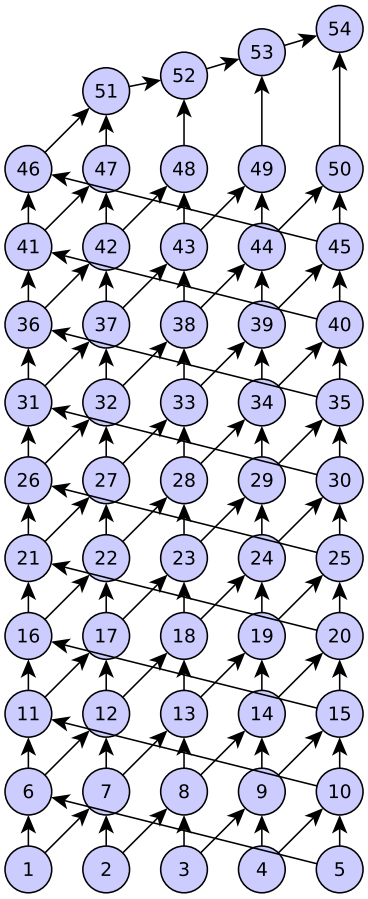


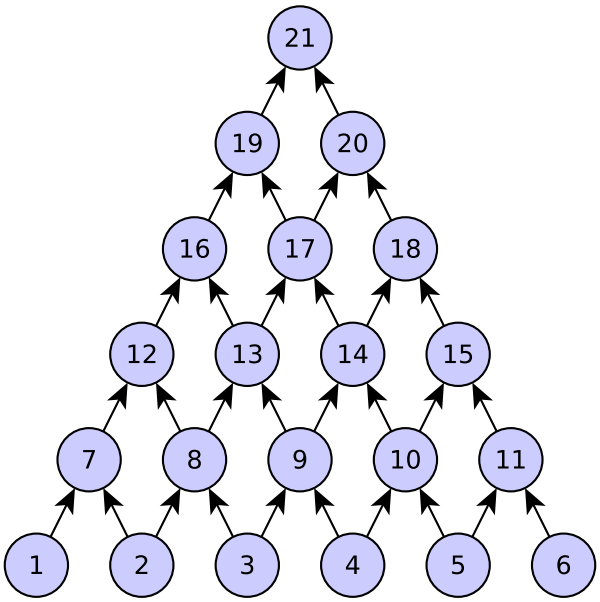
Figure 6.4. Gilbert-Tarjan graph with 8 source nodes



(a) Chain graph of width 2 and length 10



(b) Chain graph of width 5 and length 10



(c) Pyramid graph of height 5

Figure 6.5. Examples of pebbling graphs

stitution can be encoded as $(x_1 \vee x_2 \vee x_3) \wedge (\overline{x}_1 \vee \overline{x}_2 \vee \overline{x}_3)$. Just as for the substitution xor the encoding of the implication of one node from its two predecessors using the neq substitution is here omitted. A node in such a formula can be derived from its predecessors using a resolution proof of width 7 (in a straight-forward way). This width is also the minimum width needed to derive a node from its predecessors (because the clauses encoding the implication of a node have width 7). For pebbling formulas with substitution neq of arity 3 the CDCL solver running time should be at most $\mathrm{O}(n^9 \ln(n))$.

Initially a substitution called *lifting* of order 4 was also investigated, but this substitution was removed during pre-experiments because the results were very similar to those of or of order 2 and xor of order 2. The initial four substitutions were chosen based on suggestions from the supervisor of this project, which in turn were based on earlier experiments, e.g. those in [JMNŽ12].

6.1.4 Cartesian product formulas

We can define the Cartesian product of a pair of CNF formulas F and G as the formula where each clause of G has been added to each clause of F. For example if $F = (a \vee \overline{b}) \wedge (b)$ and $G = (d) \wedge (e \vee f)$ then the Cartesian product of F and G is $(a \vee \overline{b} \vee d) \wedge (a \vee \overline{b} \vee e \vee f) \wedge (b \vee d) \wedge (b \vee e \vee f)$.

Cartesian products of formulas are interesting because they can be refuted in width equal to the sum of the widths needed to refute the initial formulas. To refute the Cartesian product of a pair of formulas we can essentially use the refutation of one of the initial formulas to derive the other formula from their Cartesian product, and then refute that formula. Let us say that we have produced the Cartesian product P of the formulas F and G and want to derive the clause C that is a clause in G. Now assume we can refute F in width w then we can derive C from P in width $w + W(C)$ by simply using the refutation of F on the clauses $(A \vee C)$ for all $A \in F$. This procedure can be carried out for each clause C that is used as an axiom in a minimal width refutation of G. Finally we can simply refute G using that minimal width refutation. The entire refutation will have width at most $W(F \vdash \bot) + W(G \vdash \bot)$.

In this project formulas created using Cartesian products are created from pebbling formulas with no substitutions (i.e. each node in a pebbling graph is replaced with a single variable). Furthermore the Cartesian product formulas used in this project are always created through taking the Cartesian product of a formula with itself. The reason for using Cartesian product formulas is to try to "trick" the CDCL solvers used. The Cartesian product of a pair of formulas refutable in low width produces a formula also refutable in low width. Therefore according to the results in Chapter 4 the newly produced formula should still be easy to solve. However, the added complexity of "merging" two formulas might make the resulting formula more difficult to solve for some solvers. More specifically a pebbling formula with no substitution can be refuted in width 3 and therefore the Cartesian product of such a formula with itself can be refuted in width 6. The size of refuting a pebbling formula with no substitution grows linearly with the number of variables in the formula. However, when we want to refute the Cartesian product of such a formula with itself, using the refutation explained above, the size of our refutation will grow quadratically in the number of variables in the pebbling formula. The refutation size grows in this way because for each clause in the pebbling formula we will use the original pebbling formula refutation once. Thus, the worst-case bound for the running time of the AFT solver running on these formulas is $\mathrm{O}(mn^{w+1} \ln(n)) = \mathrm{O}(n^2 n^{6+1} \ln(n)) = \mathrm{O}(n^9 \ln(n))$ where n is the number of

variables in the formula, m is the size of refuting the formula and w is the width of refuting the formula.

6.2 Experiment parameters

Let us next discuss the different parameters used during the experiments. As mentioned earlier, we want to compare the components or settings that differ between the AFT solver and a practical solver (in this case the MiniSat solver using default settings). The different solver settings used during the experiments can be seen in Figure 6.6. All combinations of those settings are tested in this project, i.e. $2*3*2*2*2 = 48$ combinations in total. There are many more settings in MiniSat which could have been altered — regarding these settings only default values were used. These settings for example determine the aggressiveness of the clause removal used. It should be noted that changing the values of these settings could have a major impact on the outcome of the experiments, as is shown in [AS08]. However, performing an investigation of how these settings could affect the results of these experiments does not fall within the scope of this project (since doing so would increase the running time of the experiments).

Here follow descriptions of the solver settings shown in Figure 6.6. Regarding restarting policies, a solver either restarts as quickly as possible after a conflict or uses the default MiniSat restarting policy. When it comes to decision schemes, a solver uses either an entirely random scheme, the default MiniSat scheme, or a mix of the two. When using the mixed scheme, random decisions are performed every fourth round and during the rest of the rounds the VSIDS scheme is used. As described in Section 3.2.4 clause removal is the act of removing clauses from the clause database in order to save memory and speed up certain operations of the solver.

Neither the option of preprocessing nor shuffling a formula is included in the results of Chapter 4. However, these options are included in the experiments because they are often used together with CDCL solvers in practice. Preprocessing is simply the act of processing a formula in some way before sending it to the solver. The preprocessing algorithm used during these experiments is the one included in MiniSat 2.2. Shuffling is the act of reordering the clauses of a formula and also reordering the literals in each clause of the formula before letting the solver handle the formula. Shuffling could change the performance of a solver since it could change the order in which variables and clauses are stored in the solver.

The experiments described in this section were performed in the following way. The experiments were run for about a month on circa 30 computers simultaneously. Each solver (solver setting combination) was run 5 times on each formula in each formula family. The result of the runs of a solver running on a single formula is the median of those 5 running times. If a single run would take longer than 30 minutes, the run was interrupted. If such an interruption would occur for the same solver for each run on more than 3 consecutive formulas in a formula family, then the solver was not allowed to run on more formulas in that family. A summary of the parameters described above can be seen in Table 6.1.

The formula families used in these experiments are Tseitin grids of width 2, 3, 4, and 5 and RPHP formulas with 2, 3, 4, and 5 pigeons. Furthermore, the experiments include Cartesian products of pebbling formulas with no substitutions over the four pebbling graph families mentioned in Section 6.1.3. Finally, the experiments include pebbling formulas of all combinations of the four pebbling graph families and the three substitutions mentioned in Section 6.1.3.

1. Restarts
    a) As often as possible
    b) MiniSat default (Luby)
2. Decision scheme
    a) Random
    b) MiniSat default (VSIDS)
    c) Mixed (alternating between random and VSIDS)
3. Clause removal
4. Preprocessing
5. Shuffling

Figure 6.6. Experiment solver settings

Solver setting combinations	48
Number of formula families	24
Formulas per family	40
Runs/instances per formula	5
Maximum number of time-outs	3
Time-out	30 minutes

Table 6.1. Experiment parameter summary

The value of each parameter that highly affects the overall runtime of the experiments (e.g. time-out length and number of runs per formula) was chosen through running pre-experiments. One important parameter is the number of formulas per family. In order to fit the running times of the CDCL solvers to different functions this parameter should have a high value. Otherwise, the running times could be fit to functions not representing their actual growth rates. The value of this parameter was set to 40 so that the most important solvers would solve at least 20-30 formulas before reaching the time-out limit.

Here follow a couple of notes regarding the experiments. Firstly, the only difference between the 5 runs of a solver on a single formula is the random seed sent to the solver for use for example when making random decisions. After running the experiments the author of this report found that the solvers using the VSIDS decision scheme do not use any (pseudo-) randomness. This discovery implies that the 5 instances that should be different are exactly the same for solvers using the VSIDS decision scheme. Therefore, the results for solvers using the VSIDS scheme in this project are not as reliable as the results for other solvers. Secondly, the formula sizes used in these experiments sometimes differ between runs for solvers using clause removal and those not using clause removal. This difference was added so that the most efficient solvers would solve slightly larger input formulas while at the same time letting the most important solvers solve enough formulas before timing out.

6.3 Hardware

The experiments were run on a computer named Ferlin, consisting of a number of nodes where each node has two quad-core AMD Opteron 2,2 GHz CPUs (2374 HE) and 16 GB memory. However, each run was limited to 8 GB memory to make sure that the experiment computer would never run out of memory. At most one instance of solver setting combinations and formulas was run at a single node at any time during the experiments in order to avoid potential interference between different cores accessing the same memory simultaneously.

Chapter 7

Results

In this chapter we examine the results of the experiments performed in this project. The main goal of these experiments is to find out how the results presented in Chapter 4 correspond to practice. We know that a certain setting (e.g. random decisions) in the AFT solver is needed in the theoretical results if by using another setting we get running times exponential in the size of the input formula. To investigate which settings are needed for the theoretical results and which settings seem practical we simply compare running times of CDCL solvers containing different combinations of component settings.

7.1 Plots

In this section we discuss some of the plots produced when running the experiments of this project. These plots make up a small part of the plots created from the experiments. The rest of the plots can be found at `https://github.com/GSennton/thesis`. Many of the plots in this section are log-log plots. Such plots can be used to estimate whether a curve grows like an exponential or a polynomial function. A polynomial should look like a linear function when plotted on a log-log scale while an exponential function should look like it grows faster than linearly. We should note that performing a rigorous experimental investigation of the growth rate of algorithms is difficult, especially using only 30-40 data points when the curves investigated could be high degree polynomials. Therefore this section simply aims to provide some indications regarding the growth rates of the solvers tested.

As a general comment, none of the plots acquired during this project suggest that any of the solver versions run in time exponential in the size of their input formula. However, there are some formulas where we do not have enough data points to rule out exponential growth. These formulas are RPHP formulas and Tseitin grids, especially RPHP formulas with 4 or 5 pigeons and Tseitin grids of width 4 or 5. The plots for several of these formulas are not shown in this report but can be found together with the rest of the plots at the location specified above. As another general comment it seems like shuffling the formulas used in these experiments does not affect the running times of the solvers significantly. I.e. plots of running times including shuffling look very similar to those not including shuffling even though the difference between single data points sometimes vary much. Therefore the plots in this section do not include running times where formulas have been shuffled. As a final general comment, none of the solvers reached the memory limit given during any of the experiments.

Table 7.1 shows the abbreviations used to represent different solver settings in

Abbreviation	Description
ASAP	Restart policy: Restart after each conflict
Luby	Restart policy: Luby restart sequence
random	Decision scheme: random decisions
mixed	Decision scheme: random every 4th round, VSIDS otherwise
VSIDS	Decision scheme: VSIDS
remove	Clause removal: On
noremove	Clause removal: Off

Table 7.1. Descriptions of abbreviations used in experiment plots

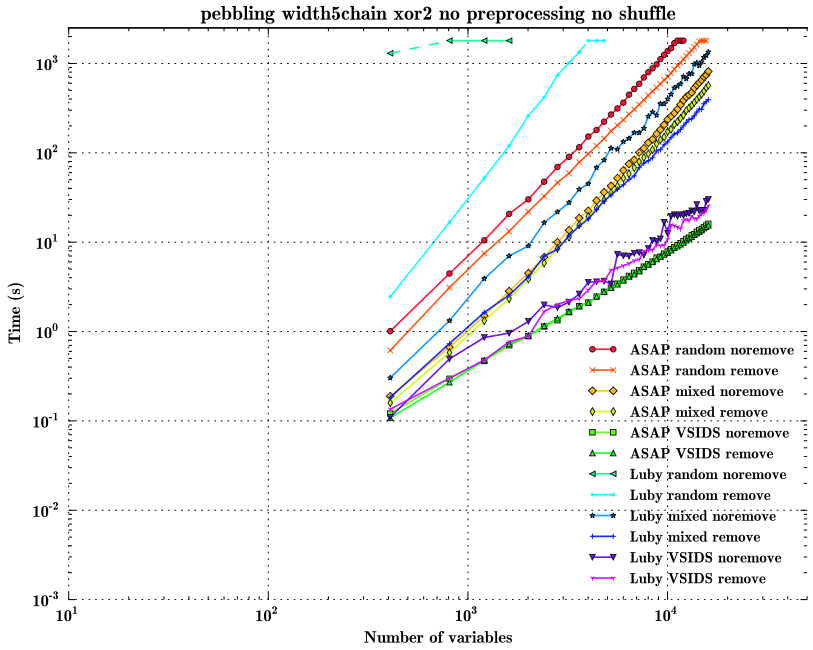


Figure 7.1. Running time comparison of solver settings (log-log scale), pebbling formulas, xor of arity 2, chains of width 5

the experiment plots of this chapter. Expressed in terms of these abbreviations the AFT solver uses the settings ASAP-random-noremove while the MiniSat default solver uses the settings Luby-VSIDS-remove.

Even though the Luby sequence is used as default restart policy in MiniSat there are cases where this is not the best choice. See e.g. Figure 7.1 where solvers are compared using pebbling formulas on chains of width 5 with substitution xor of arity 2. In this plot preprocessing is not used since preprocessing by itself seems to solve the formulas tested here (when using preprocessing all solvers have similar running times which implies that the formulas are solved by the preprocessor). The ASAP restart policy seems to perform slightly better than the Luby restart policy for these formulas. This difference is most clear when using random decisions (the Luby-random solver is clearly the worst version) and less clear when using VSIDS decisions. When it comes to mixed decisions there seems to be no major difference between the ASAP restart policy and the Luby restart policy for these formulas. Regarding decision schemes, the VSIDS decision scheme seems best for these formulas, followed by the mixed decision scheme (and then the random decision scheme).

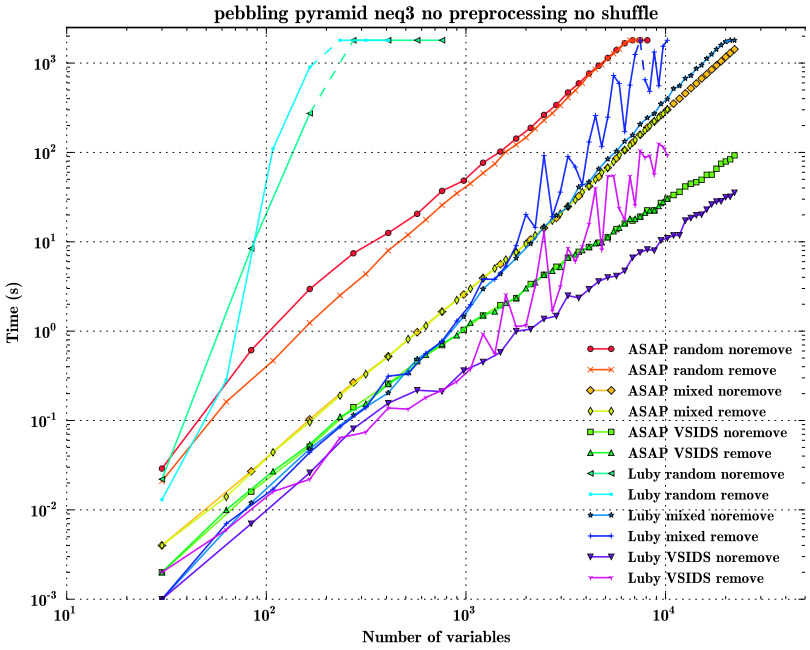
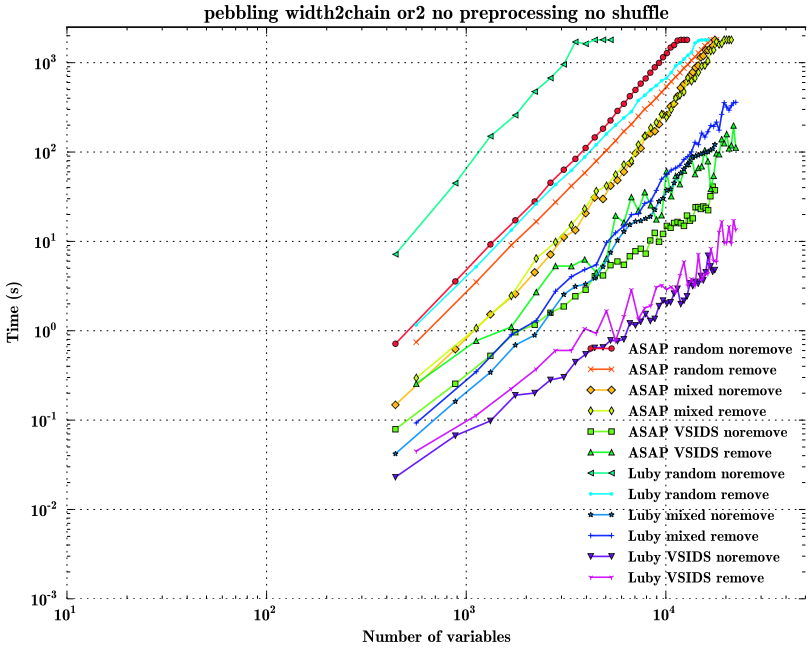


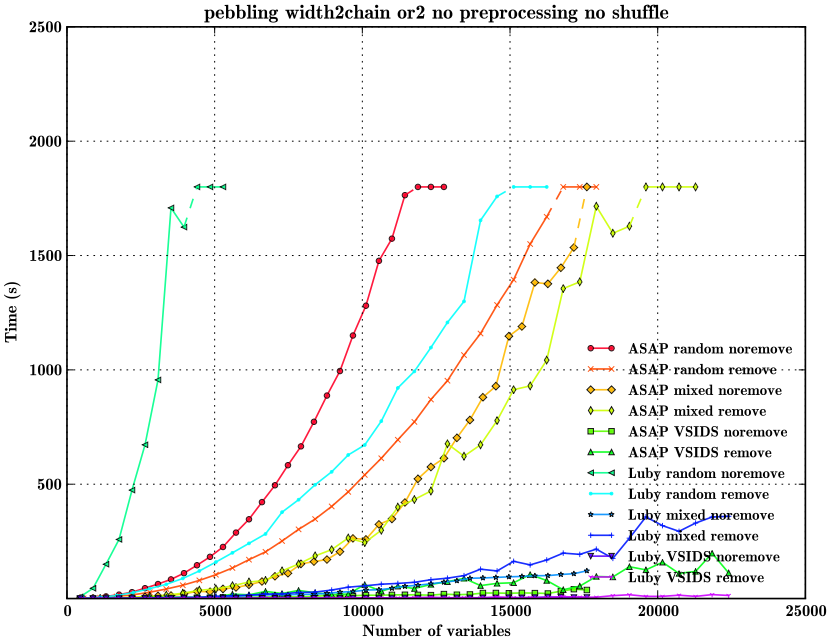
Figure 7.2. Log-log plot comparison of solver settings, pebbling formulas, neq of arity 3, pyramid graphs

Figure 7.2 shows comparisons between solvers running on pebbling formulas of pyramid graphs with the substitution neq of arity 3. Important to note is that pebbling formulas using this substitution do not seem to be solved by the MiniSat preprocessor (unlike the other pebbling formulas used in this project). That preprocessing does not solve these formulas cannot be seen by only studying Figure 7.2. However, comparing that plot to one including preprocessing shows that there is no major difference between the two. An interesting observation here is that the running time curve of the AFT solver seems to grow at a similar rate as that of the default MiniSat solver for these formulas (though the latter solver needs only a couple of seconds to solve formulas for which the AFT solver requires about 1000 seconds to solve). This observation is especially interesting because preprocessing does not simplify these formulas significantly. Therefore, there are formulas for which the AFT solver (without any extra extension such as preprocessing) seems to be as efficient (asymptotically) as the MiniSat model with preprocessing turned on. Clause removal does not seem to affect solvers using the ASAP restart policy in Figure 7.2 while the solvers using the Luby restart policy are negatively affected by clause removal. Just as in Figure 7.1 the VSIDS decision scheme seems to be the best decision scheme in Figure 7.2.

In Figure 7.3 we can see a comparison of solvers running on pebbling formulas of chains of width 2 with substitution or of arity 2. Figure 7.3a uses a log-log scale while Figure 7.3b uses a lin-lin scale. The former figure shows that the running times seem to grow polynomially while the latter figure is meant to give a picture of how the running time curves actually look. Regarding the decision scheme the VSIDS scheme seems best for these formulas, followed by the mixed decision scheme. The impact of clause removal seems to depend on the decision scheme used. When using random decisions clause removal has a significant positive impact while when using the VSIDS decision scheme clause removal seems to have a small negative impact.



(a) Log-log plot



(b) Lin-lin plot

Figure 7.3. Comparisons of solver settings, pebbling formulas, or of arity 2, chains of width 2

Figure 7.4 shows comparisons of different solver versions running on Cartesian products of pyramid pebbling formulas. Figure 7.4a does not include preprocessing but Figure 7.4b does. As can be seen in Figure 7.4b, the MiniSat preprocessor seems to solve these formulas by itself. However, the most efficient solvers in Figure 7.4 seem to be solving these formulas quicker than the MiniSat preprocessor. Actually all solvers except the ones using the Luby restart policy together with the random decision scheme seem to solve these formulas at least as efficiently as the MiniSat preprocessor. The ASAP restart policy seems to perform better than the Luby policy for all different solver versions in Figure 7.4. Clause removal seems to have a very small effect (or none at all) on running times. When it comes to the decision scheme the VSIDS decision scheme and the mixed decision scheme seem to perform better than the random decision scheme.

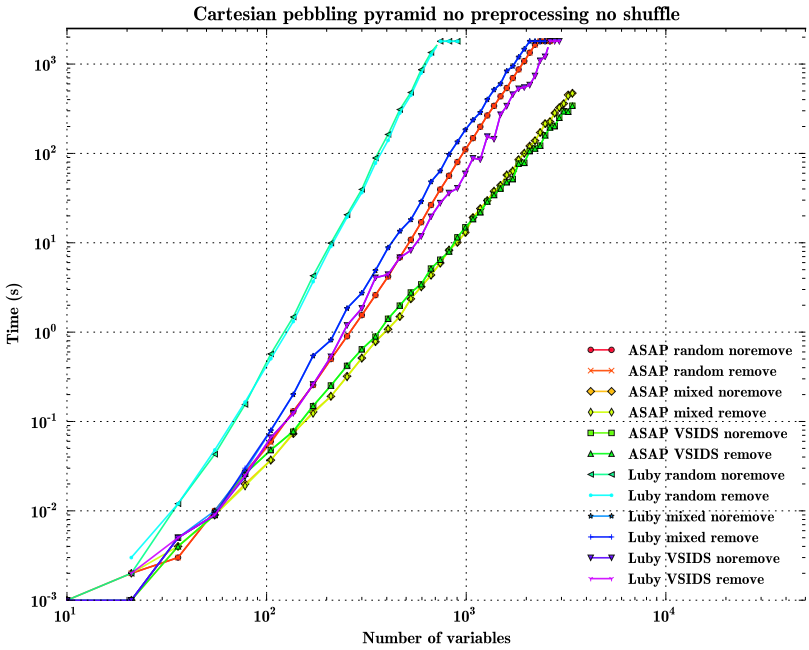
Figure 7.5 displays results from running CDCL solvers on Tseitin grids of width 4. This figure is interesting because here the AFT solver performs very badly compared to other solvers — the slope of the AFT solver running time curve seems to be greater than that of many other solvers. In this figure the Luby-mixed-noremove solver seems to perform very well. Aside from that the VSIDS decision scheme seems to be the most efficient decision scheme while clause removal affects solvers using the random decision scheme positively and all other solvers negatively. One problem with Figure 7.5 is that some of the fastest solvers finish solving the formulas given in less than one second. Having such small running times makes it difficult to say how the running times of those solvers actually grow — it would be interesting to see how the running times grow when larger formulas are used.

Note that in Figure 7.5 the AFT solver seems to time-out for formulas containing 100 variables. In Figure 7.2 the AFT solver instead solves formulas with up to approximately 6000 variables before reaching the time-out limit. This relation is interesting because the formulas used in Figure 7.5 are Tseitin grids of width 4 and can therefore be refuted in width 5. On the other hand, the formulas used in Figure 7.2 are pebbling formulas with substitution neq of arity 3 and have refutation width 7. This comparison could remind us that the refutation width of a formula is not the only property that determines the running time of a CDCL solver in the results of Chapter 4. Furthermore, those results are upper bounds which means that the actual running times of CDCL solvers could differ significantly from the theoretical results.

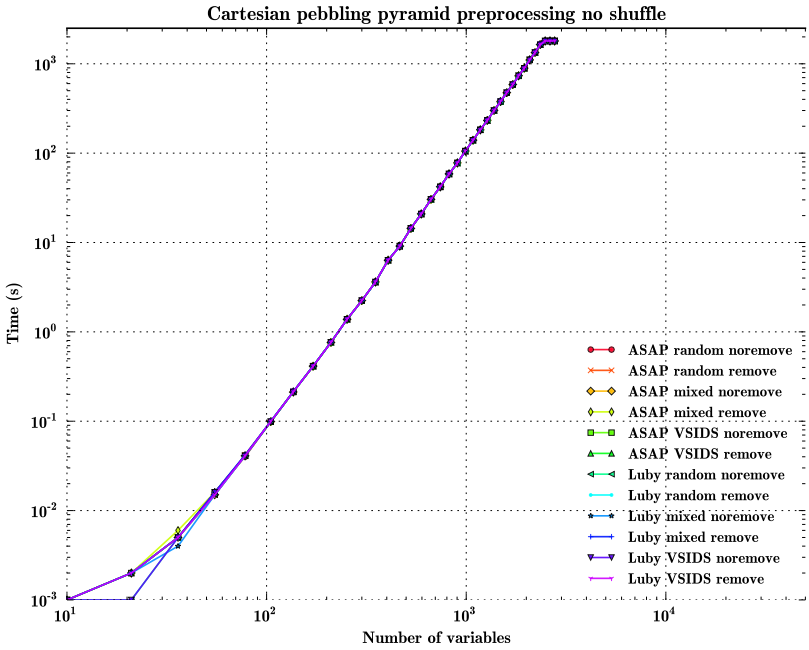
Figure 7.6 contains a comparison of solver versions running on RPHP formulas with 3 pigeons. Here, it seems most solvers have similar running times (or at least the running times grow at a similar pace). Clause removal seems to have a significant effect on some of the worst solver versions in Figure 7.6, for example it seems to have a strong negative effect on the AFT solver. The best solvers in that figure, including the default MiniSat solver, seem unaffected by clause removal.

7.2 Discussion

As mentioned earlier, it seems like the running times of most solvers used in this project grow polynomially in the number of variables of the given formula. However, for some of the plots (e.g. Figure 7.5) the most efficient solvers terminate very quickly. It is difficult to say whether the running times of such solvers actually grow polynomially and it would therefore be interesting to see how these solvers would perform on larger formulas. Running with such large formulas was not part of this project since that would increase the amount of time spent on the



(a) No preprocessing



(b) Preprocessing included

Figure 7.4. Log-log plot comparisons of solver settings, Cartesian products of pyramid pebbling formulas

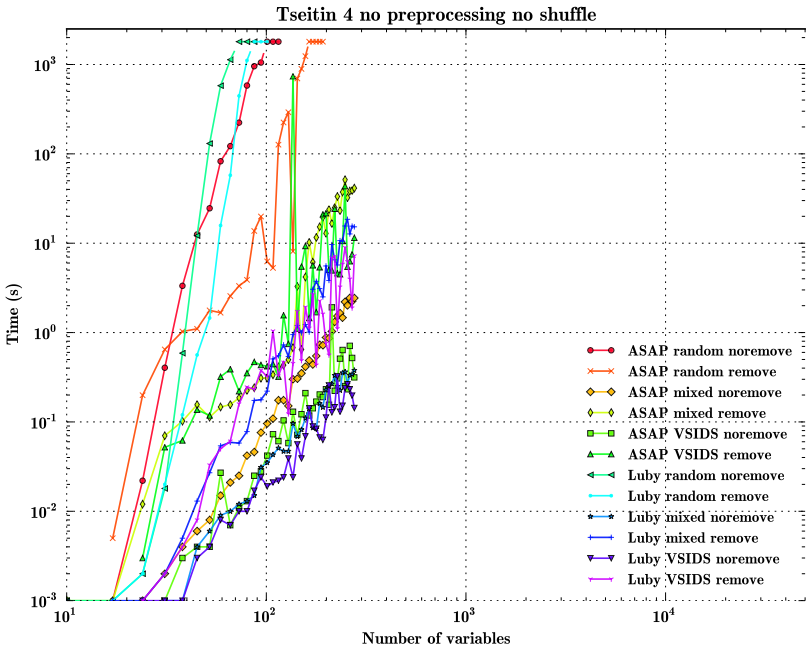


Figure 7.5. Log-log plot comparison of solver settings, Tseitin grids of width 4

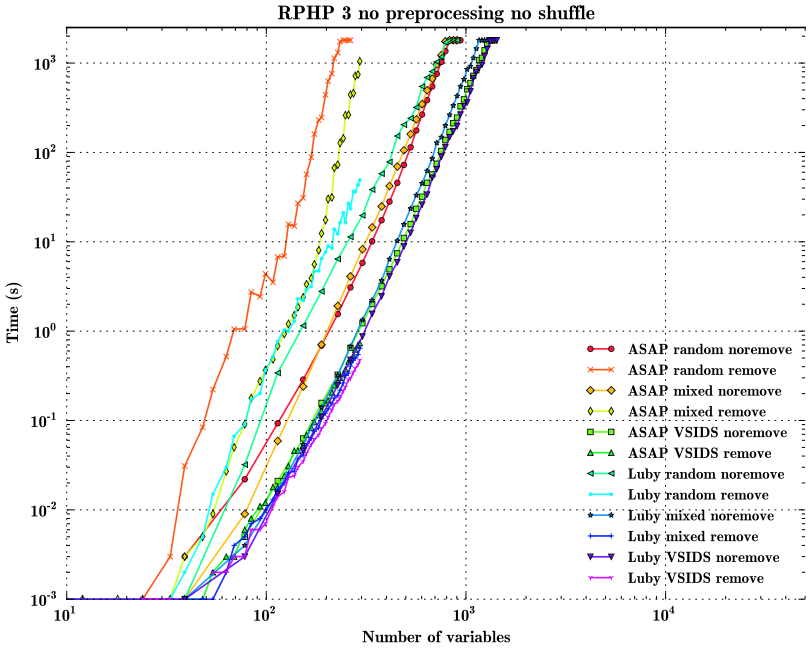


Figure 7.6. Log-log plot comparison of solver settings, RPHP formulas with 3 pigeons

experiments (which is already a month).

The AFT solver does not seem practical in terms of running time when compared to the default MiniSat solver — in several cases the AFT solver is up to a hundred times slower (Figures 7.1 and 7.2). This result is not very surprising since if the AFT solver would have been practical it would probably have been used more often in practice. On the other hand, the difference in growth rates between the running times of the AFT solver and the default MiniSat solver seems to be quite small for many formula families (see e.g. Figures 7.2 and 7.6). This result is more surprising since it suggests that the AFT solver and the MiniSat solver could have similar asymptotic running times (for some formula families). If we instead compare the components of the solvers the VSIDS decision scheme seems to outperform the random decision scheme for most formulas tested in this project. Interestingly, the solver created through changing the decision scheme of the AFT solver to the VSIDS scheme seems to perform just as well, or even better than the default MiniSat solver in many of the experiments performed in this project.

Intuitively clause removal should have a positive effect on solvers using bad decision schemes since such solvers should learn many unnecessary clauses. Since the VSIDS decision scheme seems to work better than the random scheme in the experiments of this project clause removal should be more beneficial for solvers using the random scheme. This theory seems to hold for several of the formulas tested. For example, clause removal has a positive impact on solvers using the random decision scheme and negative impact on solvers using the VSIDS decision scheme in Figure 7.3. However, for some other formulas clause removal has a negative impact on solvers using the random decision scheme (e.g. Figure 7.6). In the light of the discussion above this result is surprising — especially since the impact of clause removal on the AFT solver in Figure 7.6 is significantly negative.

Regarding restart policies the Luby restart policy often works well together with the VSIDS decision scheme (e.g. Figures 7.3 and 7.6). This correlation might hold because the order of the decisions made by the VSIDS decision scheme should not change much between consecutive rounds and because the Luby policy seldom restarts. Restarting often will only make a solver using VSIDS remake decisions similar to those made in previous rounds and therefore waste time. On the other hand, the ASAP restart policy should work well together with decision schemes which make many bad decisions. By restarting often those bad decisions will be removed. Therefore, it is interesting to see that for some formulas the ASAP restart policy works better than the Luby policy for both the random decision scheme and the VSIDS decision scheme (Figure 7.1). This observation suggests that when running on these formulas the VSIDS decision scheme is making many bad decisions since it is working better together with the ASAP restart policy than the Luby policy.

The effects of clause removal and restart policies seem to depend much on the current settings (CDCL components and formula used). Therefore, the greatest difference between the AFT solver and the MiniSat default solver (for the formulas used in this project) seems to be the decision scheme. Intuitively, the better the decision scheme is the less impact the other components will have. If the decision scheme would make many good decisions the restart policy could be less aggressive (restart less often) since we would not need to remove unwanted decisions as often as for a bad decision scheme. Similarly, if the decision scheme is good enough so that the solver learns very few irrelevant clauses the use of clause removal is not as beneficial as if the solver would learn many irrelevant clauses.

We should note that the experiments performed in this project use only a

few formula families. Therefore we should be careful when drawing conclusions regarding the importance of different solver components. It could be that the structure of the formulas used affect the performance of the solver components. Another practical aspect that might change the effect of the different solver components is the running time given for solving a formula. If a solver would run long enough for there to be a risk of running out of memory then the effect of clause removal should be far more positive than that seen in this report. It is also possible that tuning the parameters of the MiniSat solver could drastically change the effect of the different solver components.

As mentioned earlier, comparing Figures 7.5 and 7.2 it seems more difficult to solve Tseitin grids than pyramid pebbling formulas of substitution neq of arity 3 (for the AFT solver) even though the Tseitin grids have lower width. There could be several reasons for this relation. For example the results shown in Chapter 4 only handle the worst-case scenario i.e. the scenario where each clause in a proof has maximum width. In reality many of the clauses in a proof could have low width. I.e. it could be that the amount of clauses having low width is higher in pebbling formula proofs than in Tseitin grid proofs, making pebbling formulas easier to solve. Another possibility is that the number of ways in which a proof can be reordered differ between the two formula families. This difference would imply that for one family of formulas there are more clauses which could be absorbed at each step of a CDCL run. If more clauses can be absorbed then the probability of making progress at each step is increased and the running time of the CDCL solver is lowered.

Finally, each solver version runs five times on each formula (using different random seeds for each run) and the plots shown in this report show the median of those 5 values. Yet, solvers using the VSIDS decision scheme sometimes produce quite erratic plots. These plots are erratic because the solvers using the VSIDS decision scheme do not use the random seeds given to the solvers (those solvers do not use any pseudorandomness) as mentioned in Section 6.2. Because these solvers do not use the random seeds the five runs which should be different are actually similar runs. This problem could be solved by instead of using different random seeds for solvers using VSIDS for example shuffle the input formula using different shuffle seeds for each run.

Chapter 8

Final notes

In this report we have surveyed theoretical results regarding the running times of CDCL solvers. These results depend on some assumptions regarding the solver model used. The goal of this project is to determine how this model is related to solvers used in practice. Experiments have been used to compare the theoretical solver model with a real-world implementation of a solver. In this section we discuss the conclusions drawn from these experiments and the questions which are still left unanswered.

8.1 Conclusions

None of the experiments performed in this project suggest that CDCL solver running times grow exponentially with formula size for the formulas tested. These results are consistent with CDCL running times being polynomial for formulas of constant resolution refutation width not only for the theoretical AFT solver but also for the real-world MiniSat solver.

Regarding the question of whether the AFT solver is practical, the solver does not seem very practical when measuring running times. On the other hand, the growth rate of those running times often (but not always) seem similar to those of the more practical MiniSat solver. The majority of the difference in running time between the two solvers seems to stem from the use of different decision schemes. However, it should be noted that only a few formula families have been tested in this report and therefore it could be that the other solver components are far more important when running on other formulas.

The growth rate of solvers in practice do not seem to depend only on the width of refuting the input formula — sometimes formulas of small refutation width are more difficult to solve than formulas of higher refutation width. This shows that one should be careful when using the theoretical upper bound on CDCL running times shown in [AFT11] to estimate the efficiency of CDCL solvers in practice.

8.2 Future work

There are some formulas for which the experiments of this project do not rule out exponential running times and therefore it would interesting to further study those formulas and perform experiments using larger formula sizes and time-outs. The most interesting formulas to study further are RPHP formulas and Tseitin grids.

Because the most significant difference between the AFT solver and the default MiniSat solver seems to be their decision schemes it would be interesting to further investigate the VSIDS decision scheme. It would be especially interesting to see

whether the VSIDS decision scheme always performs as well (asymptotically) as the random decision scheme or if there are formulas which fool the VSIDS scheme into making too many bad decisions.

It would also be interesting to see how much the AFT solver can be improved without jeopardizing the assumptions which makes this solver a good model for deriving bounds on its running time. There are several cases in the experiments performed in this project where adding clause removal to the AFT solver makes the solver more efficient. One potential (practical) improvement to the model could be to remove clauses which are absorbed by the rest of the clause database. Removing only such clauses would never remove any progress made by the solver and thus the theoretical worst-case results would still hold.

Finally, the running times of solvers in practice do not seem similar to the theoretical upper bounds on CDCL running times. Therefore, it would be interesting to see which properties in a formula determine how long it takes before a CDCL solver terminates in the average case rather than the worst case.

Bibliography

[AFT11] Albert Atserias, Johannes Klaus Fichte, and Marc Thurley. Clause-learning algorithms with many restarts and bounded-width resolution. *Journal of Artificial Intelligence Research*, 40:353–373, January 2011. Preliminary version appeared in *SAT '09*.

[ALN14] Albert Atserias, Massimo Lauria, and Jakob Nordström. Narrow proofs may be maximally long. In *Proceedings of the 29th Annual IEEE Conference on Computational Complexity (CCC '14)*, pages 286–297, June 2014.

[AMO13] Albert Atserias, Moritz Müller, and Sergi Oliva. Lower bounds for DNF-refutations of a relativized weak pigeonhole principle. In *Proceedings of the 28th Annual IEEE Conference on Computational Complexity (CCC '13)*, pages 109–120, June 2013.

[AR08] Michael Alekhnovich and Alexander A. Razborov. Resolution is not automatizable unless W[P] is tractable. *SIAM Journal on Computing*, 38(4):1347–1363, October 2008. Preliminary version appeared in *FOCS '01*.

[AS08] Gilles Audemard and Laurent Simon. Experimenting with small changes in conflict-driven clause learning algorithms. In *Principles and Practice of Constraint Programming*, volume 5202 of *Lecture Notes in Computer Science*, pages 630–634. Springer, 2008.

[BEGJ00] María Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. On the relative complexity of resolution refinements and cutting planes proof systems. *SIAM Journal on Computing*, 30(5):1462–1484, 2000. Preliminary version appeared in *FOCS '98*.

[Bie13] Armin Biere. Lingeling, plingeling and treengeling entering the sat competition 2013. *Proceedings of SAT Competition*, pages 51–52, 2013.

[BIW04] Eli Ben-Sasson, Russell Impagliazzo, and Avi Wigderson. Near optimal separation of tree-like and general resolution. *Combinatorica*, 24(4):585–603, September 2004.

[BKS03] Paul Beame, Henry Kautz, and Ashish Sabharwal. Understanding the power of clause learning. In *Proceedings of the 18th International Joint Conference in Artificial Intelligence (IJCAI '03)*, pages 94–99, 2003.

[BKS04] Paul Beame, Henry Kautz, and Ashish Sabharwal. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research*, 22:319–351, December 2004.

[Bla37] Archie Blake. *Canonical Expressions in Boolean Algebra*. PhD thesis, University of Chicago, 1937.

[BN08] Eli Ben-Sasson and Jakob Nordström. Short proofs may be spacious: An optimal separation of space and length in resolution. In *Proceedings of the 49th Annual IEEE Symposium on Foundations of Computer Science (FOCS '08)*, pages 709–718, October 2008.

[BS97] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve real-world SAT instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI '97)*, pages 203–208, July 1997.

[BW01] Eli Ben-Sasson and Avi Wigderson. Short proofs are narrow—resolution made simple. *Journal of the ACM*, 48(2):149–169, March 2001. Preliminary version appeared in *STOC '99*.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC '71)*, pages 151–158, 1971.

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, July 1962.

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.

[ES04] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing (SAT '03), Selected Revised Papers*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518. Springer, 2004.

[GT78] John R. Gilbert and Robert Endre Tarjan. Variations of a pebble game on graphs. Technical Report STAN-CS-78-661, Stanford University, 1978.

[Hak85] Armin Haken. The intractability of resolution. *Theoretical Computer Science*, 39(2-3):297–308, August 1985.

[HBPV08] Philipp Hertel, Fahiem Bacchus, Toniann Pitassi, and Allen Van Gelder. Clause learning can effectively P-simulate general propositional resolution. In *Proceedings of the 23rd National Conference on Artificial Intelligence (AAAI '08)*, pages 283–290, July 2008.

[JMNŽ12] Matti Järvisalo, Arie Matsliah, Jakob Nordström, and Stanislav Živný. Relating proof complexity measures and practical hardness of SAT. In *Proceedings of the 18th International Conference on Principles and Practice of Constraint Programming (CP '12)*, volume 7514 of *Lecture Notes in Computer Science*, pages 316–331. Springer, October 2012.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas algorithms. *Information Processing Letters*, 47(4):173–180, September 1993.

[Mar08] João P. Marques-Silva. Practical applications of Boolean satisfiability. In *9th International Workshop on Discrete Event Systems (WODES '08)*, pages 74–80, May 2008.

[MMZ+01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC '01)*, pages 530–535, June 2001.

[MS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, May 1999. Preliminary version appeared in *ICCAD '96*.

[NH08] Jakob Nordström and Johan Håstad. Towards an optimal separation of space and length in resolution. Technical Report TR08-026, Electronic Colloquium on Computational Complexity (ECCC), February 2008.

[Nor06] Jakob Nordström. Narrow proofs may be spacious: Separating space and width in resolution (Extended abstract). In *Proceedings of the 38th Annual ACM Symposium on Theory of Computing (STOC '06)*, pages 507–516, May 2006.

[PD11] Knot Pipatsrisawat and Adnan Darwiche. On the power of clause-learning SAT solvers as resolution engines. *Artificial Intelligence*, 175:512–525, February 2011. Preliminary version appeared in *CP '09*.

[PTC77] Wolfgang J. Paul, Robert Endre Tarjan, and James R. Celoni. Space bounds for a game on graphs. *Mathematical Systems Theory*, 10:239–251, 1977.

[SATa] The international SAT Competitions web page. `http://www.satcompetition.org`. Accessed: 2 September 2014.

[SATb] Sat-race 2008. `http://baldur.iti.uka.de/sat-race-2008/`. Accessed: 2 September 2014.

[Tse68] Grigori Tseitin. On the complexity of derivation in propositional calculus. In A. O. Silenko, editor, *Structures in Constructive Mathematics and mathematical Logic, Part II*, pages 115–125. Consultants Bureau, New York-London, 1968.