DEGREE PROJECT IN COMPUTER SCIENCE AND ENGINEERING,
SECOND CYCLE, 30 CREDITS
STOCKHOLM, SWEDEN 2019

On the Structure of Resolution Refutations Generated by Modern CDCL Solvers

JOHAN LINDBLAD

KTH ROYAL INSTITUTE OF TECHNOLOGY
SCHOOL OF ELECTRICAL ENGINEERING AND COMPUTER SCIENCE

On the Structure of Resolution Refutations Generated by Modern CDCL Solvers

JOHAN LINDBLAD

Degree Project in Computer Science and Engineering
Date: April 2, 2019
Supervisor: Jakob Nordström
Examiner: Elena Troubitsyna
School of Electrical Engineering and Computer Science
Swedish title: Formen på resolutions-refutationer genererade av moderna CDCL-lösare

Abstract

Modern solvers for the Boolean satisfiability problem (SAT) that are based on conflict-driven clause learning (CDCL) are known to be able to solve some instances significantly more efficiently than older kinds of solvers such as ones using the Davis-Putnam-Logemann-Loveland (DPLL) algorithm.

In addition to solving instances that can be satisfied, SAT solvers will implicitly generate proofs of unsatisfiability for formulae that are unsatisfiable. Theoretical models of CDCL based solvers are known to have access to more powerful forms of reasoning compared to their DPLL counterparts and as a result, are able to generate proofs that are significantly shorter for certain kinds of formulae. Additionally, certain characteristics are expected when representing these proofs as graphs, such as them not being strictly tree-like in shape.

It is however less well known if these theoretical justifications are indeed the reason CDCL solvers are so successful in practice. This project attempts to answer this question by modifying a modern CDCL solver to output the proof and comparing these proofs to what theoretical results would predict.

Firstly, the results indicate that CDCL solvers generate significantly shorter proofs for all kinds of formulae that were investigated as compared to a DPLL solver. Furthermore, it appears that this is in large part due to the proof not being tree-like.

Secondly, utilizing restarts was found to make for significantly shorter proofs for most families of formulae but the effect was the opposite for formulas representing the relativized pigeonhole principle. The explanation for this is seemingly not clear.

Lastly, it appears that the Tseitin formulae used do not exhibit time-space trade-offs but instead simply require a large amount of space. This is indicated by the run time being significantly greater if clause erasure if more aggressive but the refutation being similar in both length and number of learned clauses.

To summarize, it has been found that modern CDCL solvers appear to result in significantly different proofs that largely mirror what one would expect. However, the results are unclear on the role of restarts and how their effect on the proof best can be explained.

Sammanfattning

Moderna lösare för *Boolean satisfiability problem* (SAT) baserade på konflikt-driven klausulinlärning (CDCL) har visats prestera väl och lösa vissa typer av formler mer effektivt än äldre varianter såsom Davis-Putnam-Logemann-Loveland-algoritmen (DPLL).

Förutom att lösa instanser som är lösbara så producerar SAT-lösare implicit bevis på olösbarhet för formler som är olösbara. Teoretiska modeller över CDCL-baserade lösare har visat på att mer kraftfulla former av resonemang är tillgängliga jämfört med DPLL-baserade motsvarigheter; som ett resultat kan CDCL-baserade lösare enligt dessa modeller producera kortare bevis. Vidare väntas dessa bevis ha vissa karaktärsdrag när de representeras som grafer som exempelvis att de inte är strikt trädformade.

Dock är det inte känt om dessa teoretiska förklaringar faktiskt korrekt beskriver anledningarna att CDCL-baserade lösare är så framgångsrika i praktiken. Detta projekt ämnar klargöra denna fråga genom att modifiera en CDCL-baserad lösare så att den producerar bevisen explicit och sedan jämföra dessa bevis med vad teoretiska resultat skulle förutspå.

För det första så visar resultaten att CDCL-baserade lösare genererar betydligt kortare bevis för alla sorters formler som undersöktes. Studier av små-skaliga probleminstanser visar att en del av förklaringen till detta är att beviset inte är strikt trädformat.

För det andra visar resultaten att omstarter gör bevisen betydligt kortare för nästan alla formler men att det motsatta är sant för så kallade *relativized pigeonhole principle*-formler. Förklaringen till detta är inte helt tydlig.

För det tredje sågs tendenser till tid-utrymmes-avvägningar för formler som var inspirerade av så kallade Tseitin-formler där dessa avvägningar är bevisade. Det antyder att även dessa inspirerade formler ger dessa avvägningar i praktiska implementationer av CDCL-lösare.

För att summera så visar resultaten att moderna CDCL-baserade lösare till stor del uppnår vad teoretiska modeller förutspår i termer av formen på deras bevis. Dock är resultaten mindre tydliga vad gäller omstarter och hur deras påverkan på bevisen bäst förklaras.

Contents

Chapter 1

Introduction

The Boolean satisfiability problem, SAT, is the seemingly simple problem of determining whether a given propositional formula, consisting of Boolean variables and logical operators (such as $(x\ OR\ y)\ AND\ (x\ OR\ z)$), can be satisfied by an assignment to its variables (satisfied meaning that it evaluates to *true*). It is a fundamental problem whose roots can be traced back to ancient Greece[24].

Algorithms to solve the SAT problem have been developed since the 1960s. The most notable early example is the DPLL solver[19], which introduces techniques that are still used in modern-day solvers. Common among DPLL and many modern-day solvers is the process of assigning a value to one variable at a time and undoing assignments when there is a *conflict* (meaning that the formula is found to evaluate to *false* given the current assignments). This process continues until either a solution is found or all possibilities have been explored (in the latter case, the formula is said to be *unsatisfiable*).

More recently, techniques have been proposed to improve this process. One such technique of particular interest is conflict-driven clause learning, or CDCL, which was introduced by Silva and Sakallah [39] in 1996. These solvers have been shown to be very good in real-life benchmarks, for example in the SAT Competition of 2018 where all of the top 3 solvers in all of the four tracks were based on CDCL[26, 27].

Compared to a DPLL solver, a CDCL solver will analyze the reasons behind each conflict. Information from the analysis is used to expand the formula by adding new clauses to explicitly state constraints that are implied by the formula. Furthermore, analyzing each conflict allows the solver to undo more than the last assignment, which means

that a greater number of unsatisfying assignments are discarded.

If a formula is unsatisfiable, the fact that it is unsatisfiable must be implied by the formula. However, it is not necessarily readily apparent that the formula is indeed unsatisfiable (except for trivial examples such as $((x) \wedge (\overline{x}))$). What the CDCL solver is doing then is to expand the formula with new clauses, step by step, until it is readily apparent that it is unsatisfiable. Eventually, it will either have found a satisfying assignment or it will have learned the empty clause; in the latter case the formula has been found to be unsatisfiable (recall that the the empty clause can not be satisfied).

Because solvers ensure that all possible assignments are exhausted before determining a formula to be unsatisfiable, they are implicitly building a proof of the unsatisfiability (referred to as a *refutation*). The solver combines clauses from the formula into new equally valid clauses until the empty clause has been derived; a full proof can be extracted by tracing all of the steps involved in these derivations.

Both DPLL and CDCL solvers can be seen as doing this using what is called the *resolution* operation. This is an operation that was introduced by Blake [12] and which combines (or *resolves*) two clauses into a new clause (the *resolvent*) which contains all literals from both clauses except the pair of literals (one from each clause) which represent the same variable but where one is negated (unless exactly one such pair of literals exists the resolution operation is not allowed). It is known that if (and only if) a formula is unsatisfiable, it is possible to use the resolution operation to derive the empty clause from the formula[38]. Such a derivation is referred to as a *resolution refutation*, because resolution is used to *refute* the existence of a satisfying assignment for the formula.

This refutation can be seen as a directed graph. Each resolution operation connects two vertices representing to the two ingoing clauses to a vertex representing the resolvent. When the result from one resolution operation is used as an input in another operation, both operations are seen as operating on the same vertex; thus such a vertex will have two ingoing edges from the vertices of the clauses it was resolved from and an outgoing edge to the vertex of the clause that resulted from it. Clauses that exist in the original formula are however copied each time they are used so that each such vertex only has one outgoing edge.

Both DPLL and CDCL solvers are either implicitly or explicitly us-

ing the resolution operation to derive the empty clause. Thus, their operations can be modeled as a graph as described above. However, because DPLL solvers do not learn clauses, any resolvent that is derived is only used as an ingoing clause once. This means that each vertex in the graph will be the parent of at most one vertex and thus, the graph is shaped like a tree.

CDCL solvers meanwhile are able to refer back to clauses that have already been derived which means that deriving the same clause multiple times can sometimes be avoided. Instead, the vertex representing learned clauses can be connected to multiple vertices whenever the learned clause is used to resolve a new clause. Because repeated derivations are avoided, the refutation may contain fewer steps (making it shorter) or equivalently, the graph can contain fewer vertices. However, because learned clauses can be the parents of multiple vertices, the graph may no longer be tree-like.

It is known that in theory the additional reasoning power that is enabled by this reuse of learned clauses makes it so that for certain kinds of formulae the refutation may be exponentially shorter[1, 9]. However, these results make assumptions of extensions that are not usually implemented in CDCL solvers in practice.

One additional classification of interest is related to the sequence of variables that are removed. As noted above, the resolution operation will remove one variable that appears in both of the ingoing clauses but that will not exist in the resulting clause. The refutation makes use of several resolution steps where the resulting clause from one application of resolution is used as an input for the next step. This means that when following a path from an arbitrary vertex to the empty clause, there is a sequence of variables that are removed at each step. If every such sequence contains no duplicates, the refutation is said to be *regular*. Such refutations are on the one hand sometimes exponentially longer than ones without this requirement[1] but on the other hand sometimes exponentially shorter than ones which are tree-like[9].

Furthermore, modern CDCL solvers make use of a technique called restarts[25], which entails periodically restarting the solver by undoing all assignments but keeping the learned clauses. This technique has similarly been shown to in theory allow for shorter refutations provided that the CDCL solver also utilizes the right heuristics[35]. Additionally, the solver may remove some of the learned clauses when restarting in order to save memory[7].

Thus, it is known that some theoretical models of CDCL solvers are capable of exponentially shorter refutations than DPLL solvers. Furthermore, as has been mentioned, it is known that in real-life benchmarks CDCL solvers seem to outperform DPLL solvers.

What is not known however is if the promising performance CDCL solvers have shown is because they utilize this additional reasoning power to actually refute the formula in the way that theoretical models would suggest or if they are performant for other reasons. Other possibilities are for example that learning clauses allows CDCL solvers to use better heuristics for things such as which variables to assign next.

A topic worth investigating then is whether CDCL solvers are successful in practice because they use more powerful forms of reasoning, allowing them to find to find shorter refutations, or if they are successful for other reasons. The goal of this project is to provide insight into this question by modifying a modern CDCL solver to explicitly generate resolution refutations and then to analyze their corresponding graphs.

1.1 Research question

As has been mentioned, there are both theoretical models that show that CDCL solvers have access to more powerful forms of reasoning than earlier solvers and practical results that show that they outperform these earlier solvers. However, it is not known if the theoretical justification is the reason for the practical success. This is the main focus of this research project.

Thus, the main research question is: "Do modern CDCL solvers appear to utilize the extra power theory has shown is afforded by clause learning and restarts or are other explanations for their success more reasonable?". To this end, we are interested in how clause learning affects the length of the refutation and certain aspects of its shape such as how much it resembles a tree, to what extent it is regular and how many literals are contained in the largest clause.

Three more specific subquestions have been identified:

1. A certain family of formulae known as *relativized pigeonhole principle*[3] is known to allow for short refutations even when the refutation is required to be tree-like. However, it is still conceivable that the more powerful forms of reasoning of a CDCL

solvers could allow it to achieve a shorter proof; this could for example be achieved by avoiding deriving the same clause twice. Does a modern CDCL solver still manage to arrive at an even shorter refutation and if so, how much does this refutation resemble a tree?

2. Families of formulae known as *ordering principle*[41] and *pebbling formulae*[30] are known to require relatively long refutations if the refutation is required to be tree-like[11, 13] compared to if no such restrictions are placed. DPLL-based solvers are restricted to tree-like resolution but a CDCL solver could potentially achieve shorter refutations that are not tree-like. Do CDCL solvers manage to find shorter refutations and if so, how much do these refutations resemble trees?

   Additionally, restarts have been shown to significantly affect the time required for these formulae[22]. Does the choice of restart policy significantly affect the refutation in length, tree-likeness or regularity?

3. Certain kinds of very large *Tseitin* formulae[42] are known to exhibit time-space trade-offs where theoretical models of solvers can finish in shorter time if more space is available[8]. Indications were found by Elffers et al. [22] that scaled-down versions of these formulae showed time-space trade-offs in practice, where a smaller clause database made the solver significantly slower. Is this because of time-space trade-offs or does the solver simply require more time to arrive at the same refutation?

1.2 Ethical implications and societal aspects

SAT solving is a general technique and as such does not have direct ethical implications. However, it has found uses in areas such as planning[37] and software verification[31]. Thus, improvements in SAT solving could bring improvements for whomever makes use of planning algorithms or the results thereof.

Furthermore, the SAT problem is of interest in complexity theory because it is relevant to one of the most fundamental open questions in computer science, the question of whether $\mathsf{P} = \mathsf{NP}$. The SAT problem

is known to be in NP and is the complement of the problem of determining whether a formula is unsatisfiable, which is in coNP[17]. If it were shown that there are formulae that can not be efficiently shown to be unsatisfiable, NP $\neq$ coNP and as a result, P $\neq$ NP. This would mean that there are problems which can not be efficiently solved even though the solutions themselves can be efficiently verified.

Chapter 2

Theory

This chapter aims to provide an overview of relevant theoretical background and previous work. The research question involves concepts from different areas. Because some of these may seem unrelated, an attempt is made below to summarize how these areas are related.

The research question is related to the Boolean Satisfiability Problem (SAT), described in 2.1. For instances of the problem that are unsatisfiable, different proof systems exist that formalize ways of proving this fact; a few are described in 2.2. The SAT problem can be solved by various methods, briefly summarized in 2.3. Some of these methods can also generate the kinds of proofs discussed in 2.2; this process is described in more detail in 2.4.

2.1 Boolean Satisfiability Problem (SAT)

The boolean satisfiability problem (SAT) consists of determining whether a given propositional logic formula has a satisfying assignment[16].

2.1.1 Conjunctive Normal Form (CNF)

While the SAT problem allows any well-formed, propositional formula, a formula is commonly converted into conjunctive normal form (CNF) before being used. Because this form is more restricted than propositional logic in general, it is easier to formulate solver algorithms for formulae on this form. Any valid propositional formula can however be converted into a formula on CNF that is *equisatisfiable*, meaning that the converted formula is satisfiable if and only if the

original formula is satisfiable; one method for doing this is described in Tseitin [42].

A CNF formula consists of the following parts:

Variable In this context, a boolean variable, i.e. a variable which takes on the values *true* or *false* (sometimes identified as 1 and 0). Written as x, y or z or in some cases x_1 or x_2. When clear from the context, they can also be referred to simply by their index; some examples in later chapters will use numbers like 0, 1, 2 and 3 to refer to x_0, x_1, x_2 and x_3.

Literal Either a variable, such as x, or its negation, $\overline{x}$. The value of a negation is the opposite of the value of the variable it refers to, so $\overline{x}$ evaluates to 1 if $x = 0$ and to 0 if $x = 1$. Alternatively, $\overline{x}$ is sometimes written as $\neg x$ or $\sim x$ when limited to plain text.

Clause A disjunction of literals, for example $C = x \vee \overline{y} \vee z$. A clause is *satisfied* if at least one of the literals evaluate to 1; in the example just given C is satisfied if at least of the following are true: $x = 1$, $y = 0$ or $z = 1$.

Formula A conjunction of clauses, for example $F = (x \vee \overline{y}) \wedge (y \vee \overline{z})$. A formula is *satisfied* if all clauses are satisfied; in the example just given both the clause $(x \vee \overline{y})$ and the clause $(y \vee \overline{z})$ are required to be satisfied.

Because any formula in propositional logic can be converted into an equisatisfiable formula on this form, we can refer only to formulae in this form with no loss of generality.

2.2 Proof systems

A proof system in propositional logic is usually defined as per Reckhow [36]. It consists of a proof-verifying function, P, which accepts valid proofs that a formula is tautological and rejects invalid proofs.

In this thesis we are instead interested in proving that a formula is unsatisfiable. The negation of a tautology is unsatisfiable because if F evaluates to *true* for all possible assignments then $\overline{F}$ by definition evaluates to *false* for all possible assignments. Thus, a proof system can equally be formulated to be used to show unsatisfiability. That is the

formulation that will be used in this thesis. Hence, proof systems in this context are used to prove unsatisfiability; such a proof is usually referred to as a *refutation*.

So, for any unsatisfiable formula, y and a valid refutation x, we require $P(x, y) = y$.

Furthermore, we require:

- For any unsatisfiable formula y there is a valid refutation x; this is known as *completeness*.
- If there is a valid refutation x of a formula, that formula is unsatisfiable; this is known as *soundness*.
- The proof-verifying function P runs in polynomial time in the length of its input; it is *efficient*.

Furthermore, it should be noted that the definition does not require an efficient algorithm for generating a proof, only that if given a proof it can efficiently be verified. If an algorithm for generating such a proof was known to exist, the proof system would be *automatizable*. As is discussed in Bonet, Pitassi, and Raz [14], this is usually defined as the algorithm being polynomial in time in the size of the shortest possible proof (given that a proof exists).

This section briefly mentions a naive attempt at a proof system and the cutting planes proof system. It then introduces the resolution proof system, which has a clear rationale in the light of these two aforementioned proof systems.

2.2.1 Truth tables proof system

One might imagine a naive proof system consisting of an exhaustive search of the input space, i.e. attempting all possible assignments to generate a truth table[36]. This table can then allege that the formula is falsified for all possible assignments, something that can be efficiently verified by simply verifying the truth table (recall that efficiency is measured in terms of proof size).

However, a refutation in this system will have a size exponential in the number of variables. There are other proof systems in which it is possible generate shorter refutations for certain families of formulae but it is still an open question whether there are proof systems that are

polynomially bounded (i.e. permit refutations polynomial in size for all formulae).

The existence of such a proof system would answer an open question in computational complexity, whether NP equals coNP. The SAT problem is in NP, which is the set of decision problems whose positive examples can be verified in polynomial time given the appropriate certificate[16]. In the case of SAT, a positive example (i.e. a formula that is satisfiable) can be verified using a satisfying assignment as a certificate.

If however a polynomially bounded proof system were to exist, a polynomially sized refutation exists for all unsatisfiable formulae. Furthermore, the proof verifying function runs in time polynomial in the size of the refutation. Thus, there is for all unsatisfiable formulae a refutation which can be verified in polynomial time in the size of the formulae. As a result, negative instances for SAT (i.e. formulae that are not satisfiable) can be verified in polynomial time using the refutation as a certificate. This would place the SAT problem in coNP, which is the set of all decision problems whose negative examples can be verified in polynomial time. Thus, because all problems in NP can be reduced to SAT and all problems in coNP to UNSAT (determining whether a formula is unsatisfiable), the existence of such a proof system would mean that NP $=$ coNP.

If on the other hand it could be shown that no such proof system exists, then NP $\neq$ coNP. This would have implications on another open question, whether P equals NP, where P is the set of all problems that can be solved in polynomial time. Because P is a subset of both NP and coNP, a result of NP $\neq$ coNP would be P $\neq$ NP. This would mean that there problems whose solutions can be verified in polynomial time but whose solutions cannot be generated in polynomial time.

2.2.2 Cutting-Planes Proof System

Cutting-planes is a proof system introduced by Cook, Coullard, and Turán [18] which uses the cutting-planes method used in mathematical optimization.

This system considers linear equalities on the form $(a_1x_1 + a_2x_2 + \cdots + a_nx_n) \geq a_0$, where $a_i \in \mathbb{Z}$ and where $0 \leq x_i \leq 1$ for all i. Formulae can be translated into this form. This is done by translating each clause into a linear equality by replacing literals on the form x into x, literals

on the form $\overline{x}$ into $1 - x$, the disjunction operator ($\vee$) by the addition operator ($+$) and demanding that this sum is greater than or equal to 1.

For example, the clause $(x \vee y \vee \overline{z})$ translates into $x + y + (1 - z) \geq 1$. If all constants are moved to the right hand side, it can be written in the standard form defined above as $(x + y - z) \geq 0$.

The set of inequalities representing a formula lacks integer solutions if and only if the formula is unsatisfiable.

The cutting-planes proof system consists of manipulating the inequalities corresponding to the clauses in the formula and deriving a contradiction. There are three valid ways in which the inequalities can be manipulated:

Addition Two inequalities can be added together, which results in a new inequality with all coefficients added (where variables missing from one inequality are treated as implicitly having the coefficient 0). For example:

$$\frac{(a_1x_1 + a_2x_2 + \cdots + a_nx_n) \geq a_0 \quad (b_1x_1 + b_2x_2 + \cdots + b_nx_n) \geq b_0}{(a_1 + b_1)x_1 + (a_2 + b_2)x_2 + \cdots + (a_n + b_n)x_n \geq a_0 + b_0}$$

Scalar multiplication An inequality can be multiplied by a positive integer b, which multiplies all coefficients by that number. For example:

$$\frac{(a_1x_1 + a_2x_2 + \cdots + a_nx_n) \geq a_0}{(a_1 \cdot b)x_1 + (a_2 \cdot b)x_2 + \cdots + (a_n \cdot b)x_n \geq a_0 \cdot b}$$

Integer division An inequality can be divided by a positive integer b, provided that all coefficients $a_1, a_2, \cdots a_n$ (but not necessarily a_0) are divisible by b. This divides all coefficients by that number and rounds up a_0 to the nearest integer. For example:

$$\frac{(a_1 \cdot b)x_1 + (a_2 \cdot b)x_2 + \cdots + (a_n \cdot b)x_n \geq a_0}{(a_1x_1 + a_2x_2 + \cdots + a_nx_n) \geq \lceil a_0 \,/\, b \rceil}$$

The reason that rounding up is performed is that we only permit integer coefficients. However, no valid solution is removed by this operation. The reason is that the left side of the equation

only contains integer coefficients and boolean variables and so it can only take on integer values.

As opposed to the naive proof system described above, it is possible to generate refutations which are not exponential in size. A small example follows below but a larger example can be found in A.1.

Consider the formula $(x_1 \vee x_2) \wedge (x_1 \vee x_3) \wedge (x_2 \vee x_3) \wedge (\overline{x_1} \vee \overline{x_2}) \wedge (\overline{x_1} \vee \overline{x_3}) \wedge (\overline{x_2} \vee \overline{x_3})$ (generated with the program `cnfgen`[32] and the parameters `bphp 3 2`).

In the cutting planes proof system, this would be represented as the following set of inequalities:

$$\begin{aligned} x_1 + x_2 &\geq 1 \\ x_1 + x_3 &\geq 1 \\ x_2 + x_3 &\geq 1 \\ -x_1 - x_2 &\geq -1 \\ -x_1 - x_3 &\geq -1 \\ -x_2 - x_3 &\geq -1 \end{aligned}$$

In order to construct a refutation, one can first use the addition operation on the first three inequalities. This gives the inequality $2x_1 + 2x_2 + 2x_3 \geq 3$, which can be divided by 2 to get the inequality (after rounding) $x_1 + x_2 + x_3 \geq 2$. If this is multiplied by 2 it gives the clause $2x_1 + 2x_2 + 2x_3 \geq 4$.

Using the addition operation on the last three inequalities gives the inequality $-2x_1 - 2x_2 - 2x_3 \geq -3$. Adding this to the inequality $2x_1 + 2x_2 + 2x_3 \geq 4$ gives $0 \geq 1$, which is a contradiction.

2.2.3 Resolution Proof System

Given the two proof systems now described there are characteristics of both that are advantageous and disadvantageous. Firstly, the naive system can easily be used to automatically generate a refutation for any formula, but this refutation is of exponential size in the size of the formula.

Secondly, the cutting planes system cannot be as easily automated as there are multiple choices to be made regarding which operations to perform on which clauses; given that one operation consists of multiplying by an arbitrary integer, the number of possible operations is infinite.

In contrast, the *resolution proof system* offers only one operation, that combines two clauses into a new clause[12]. This suggests that it is "morally easier" to generate a proof than in the cutting planes system. It is however not trivial to automatically generate such a proof and it is known that finding the shortest resolution refutation is NP-hard[29].

The one operation in the resolution proof system is called resolution and is used to combine two clauses into a new, resulting clause called a *resolvent*. The operation generates a new clause with all literals from both of the clauses being resolved, except the pair (one from each clause) of literals that represent the same variable but with opposite signs.

Informally, the rationale behind this operation is that if a variable appears in two clauses but with opposite signs, assigning a value to it will satisfy one of the two clauses but require the other clause to be satisfied by one of its other literals.

As an example, consider the two clauses $(x \vee \overline{y})$ and $(y \vee \overline{z})$. If one chooses $y = 1$, one satisfies the latter clause but then requires $x = 1$ to satisfy the former. Conversely, if one assigns $y = 0$ one also needs to assign $z = 0$. No matter the value of y then, one is forced to assign either $x = 1$ or $z = 0$, which can be expressed as the resolvent $(x \vee \overline{z})$. Thus, y and $\overline{y}$ were omitted from the resolvent because they represent the same variable but have opposite signs.

This is formalized as the resolution operator and is written as follows:

$$\frac{(C_1 \vee x) \;\; (C_2 \vee \overline{x})}{(C_1 \vee C_2)} \tag{2.1}$$

For the example given above, it is written as:

$$\frac{(x \vee \overline{y}) \;\; (y \vee \overline{z})}{(x \vee \overline{z})} \tag{2.2}$$

It is important to note that only one literal can be removed by the resolution, or invalid inferences could be drawn. For example, removing both x and y from the two clauses $(x \vee y)$ and $(\overline{x} \vee \overline{y})$ leaves the empty clause, suggesting that there is no assignment that satisfies both clauses simultaneously; however either of the two assignments where $x \neq y$ holds satisfy them.

As is discussed by Robinson [38], a formula is unsatisfiable if and only if a series of resolution steps can deduce the empty clause (con-

taining no literals) from the original clauses. These steps then constitute the actual refutation, which is called a *resolution refutation*.

This refutation can be seen as a directed, acyclic graph where each vertex is a clause with incoming edges from the clauses it was resolved from; clauses from the original formula are then necessarily sources and the empty clause the sole sink. We then refer to the number of vertices as the *length* of the refutation, which is a property whose relation to the input size is of special interest.

This kind of refutation is known as a *resolution refutation* and an example is given in figure 2.1. The example is for the formula $(x \vee y) \wedge (x \vee \overline{y}) \wedge (\overline{x} \vee \overline{y}) \wedge (\overline{x} \vee y)$; note that each of the four possible variable assignments make one clause falsified. Clauses from the original formula have been colored gray while clauses derived from these are white.

A larger example can be found in A.1 in the appendix.

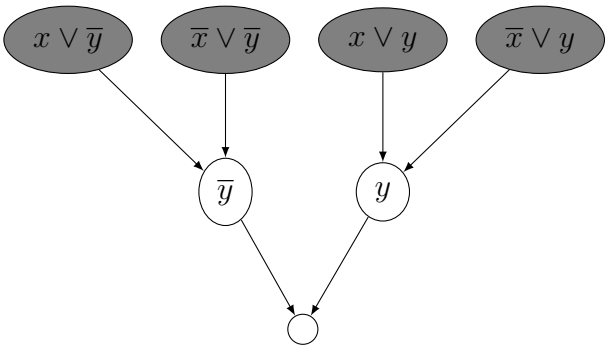

Figure 2.1: An example of a resolution refutation.

Various restricted subsystems have been introduced; many of these are summarized in Beame, Kautz, and Sabharwal [9]. The three of interest to this thesis are:

General resolution No restrictions other than all steps being valid resolution steps.

Despite this however, resolution is known to be weaker than the cutting-planes proof system[18].

Tree-like resolution The refutation forms a tree. Equivalently, each non-empty derived clause is used exactly once.

There are certain kinds of formulae which tree-like refutations are exponentially long in the size of the formula, while general

resolution refutations are of polynomial size; this means that tree-like resolution can be said to be *weaker* than general resolution.

When a refutation is not perfectly tree-like, this is because a non-empty derived clause has been used more than once. Whenever this occurs, this can be said to be a *tree violation*. The vertex of the clause that was used more than once can be said to be a tree-violating vertex and every instance of reuse corresponds to a tree-violating edge.

An example of a tree violation is displayed in figure 2.2.

Regular resolution If a variable x is removed by resolution, subsequent resolution steps will not reintroduce the variable [42].

More formally, for every sequence of clauses $C_1, \cdots, C_n$ such that C_{i+1} is derived from C_i and a variable, v, exists in both C_j and C_k, then it must also exist in C_x for all x, $j < x < k$.

This restriction allows for some reuse of intermediate steps, which allows regular resolution to be exponentially stronger than tree-like resolution[15], meaning that there are certain kinds of formulae where the proof is exponentially larger for tree-like resolution. General resolution is however also exponentially stronger than regular resolution [1]).

If there is a sequence of clauses $C_1, \cdots, C_n$ such that C_{i+1} is derived from C_i and a variable, v, exists in both C_j and C_k but not in C_l for some j, k and l such that $j < l < k$, the refutation is not regular. At some point C_x, where $l < x \leq k$ the variable is reintroduced; this can be said to be a *regularity violation* and the clause that reintroduces the variable is said to be a regularity-violating vertex.

An example of a regularity violation is displayed in figure 2.3.

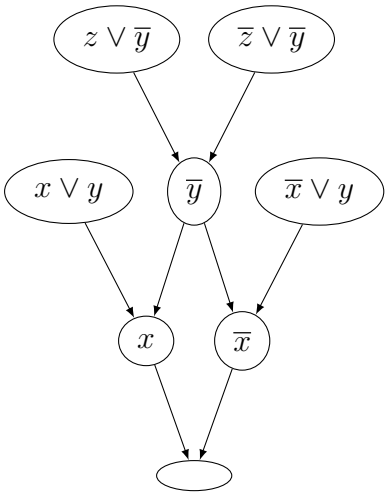

Figure 2.2: A resolution refutation which is not tree-like; the derived clause $\overline{y}$ is used multiple times and is said to be a *tree-violating vertex*. The edge from $\overline{y}$ that is added last is said to be a *tree-violating edge*.

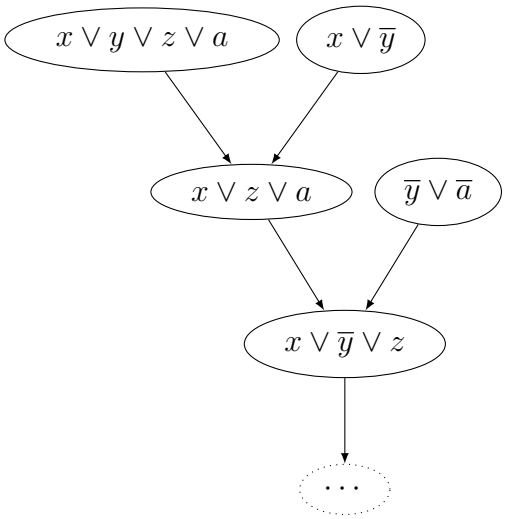

Figure 2.3: Part of a resolution refutation which is irregular; the variable y is removed at the first step and reintroduced in the second. $x \vee \overline{y} \vee z$ is said to be a *regularity-violating vertex* because it reintroduces a variable.

2.3 Methods for SAT solving

In the literature, a number of methods for solving the SAT problem have been proposed. This sections aims to introduce ones of particular interest to this thesis.

2.3.1 Exhaustive search with backtracking

One naive method of solving the SAT problem is to use exhaustive search with backtracking. Pseudo code can be found in algorithm 1. It consists of assigning variables until the formula is falsified and then attempting opposite assignments until it is no longer falsified. In the end either all options have been exhausted or a solution has been found.

Algorithm 1 Exhaustive search with backtracking

Input:
F - the formula to be solved
xs - the assignments that have been made as an ordered sequence of Boolean values, i.e. $[1, 0]$ if $x_1 = 1$ and $x_2 = 0$.

1: **function** EXHAUSTIVEWITHBACKTRACK($F, xs = [x_1, \cdots, x_n]$)
2: **for** $x_{n+1} \in \{0, 1\}$ **do**
3: $xs' \leftarrow [x_1, \cdots, x_{n+1}]$
4: **if** F is satisfied under xs' **then return** $\{SAT, xs'\}$
5: **else if** F is indeterminate under xs' **then**
6: **if** EXHAUSTIVEWITHBACKTRACK(F, xs') $= \{SAT, xs''\}$ **then return** $\{SAT, xs''\}$
return $UNSAT$

2.3.2 Davis-Putnam-Logemann-Loveland algorithm (DPLL)

The Davis-Putnam–Logemann–Loveland algorithm (DPLL) was introduced in 1962 as a method of theorem proving[19]. It is roughly based on the naive backtracking method described above but adds a number of improvements that can improve running time.

The first technique is known as *unit propagation*. In this case, a *unit* is a clause containing only one literal. If a formula contains a unit, that exact assignment has to be made in order to satisfy the formula. For

example, any satisfying assignment to the formula $(\overline{x}) \wedge (x \vee y)$ must have $x = 0$ because otherwise the first clause is falsified.

This is used as a technique during the run of the algorithms to avoid considering assignments that would never have been valid. As an example, consider the formula:

$$(x \vee \overline{y}) \wedge (y \vee \overline{z}) \tag{2.3}$$

Initially, the solver has not assigned values to any of the variables, so they are neither assigned 1 nor 0. If the solver chooses to assign $x = 0$, it has what is called a *partial assignment*, because only some of the variables have been assigned values.

With this partial assignment, we can remove all instances of x from our formula (because those literals can now not be satisfied). This results in:

$$(\overline{y}) \wedge (y \vee \overline{z}) \tag{2.4}$$

We now have a unit clause containing $\overline{y}$. Because we must satisfy this clause to satisfy the formula we immediately assign $y = 0$ and remove all instances of the literal y from our formula:

$$(\overline{y}) \wedge (\overline{z}) \tag{2.5}$$

This in turns forces us to assign $z = 0$. Because of these propagations, many assignments were never attempted because, as discovered by the propagations, the formula is not satisfied under them.

Furthermore the DPLL algorithm considers the number of times each variable appears with each respective sign and if a variable appears with only one sign throughout the whole formula, it can be immediately assigned as to satisfy them (i.e. if the variable x only appears as $\overline{x}$, $x = 0$ is assigned).

Lastly, it does not simply assign variables in an unspecified order but instead permits the next variable to assign to be chosen by some algorithm, that can then hopefully make a more informed decision.

This gives a pseudo code that is similar to the backtracking method but which differs in significant ways; it can be seen in algorithm 2. Two functions are used above:

- UNITPROPAGATE(F, x), which will investigate all clauses in F and assign variables to x from any unit clauses that appear (i.e.

Algorithm 2 DPLL algorithm

Input:
F - the formula to be solved
xs - a set of tuples representing assignments, with (m, v_m) indicating that the variable given by m is assigned the value v_m, i.e. $(2, 1)$ indicates $x_2 = 1$.

1: **function** DPLL($F, xs = \{(n, v_n), \cdots\}$)
2: $\{F, xs'\} \leftarrow$ UNITPROPAGATE(F, xs)
3: **if** F is satisfied under xs' **then return** $\{SAT, xs'\}$
4: **else if** F is falsified under xs' **then return** $UNSAT$
5: $i \leftarrow$ SELECTUNASSIGNEDLITERAL(F, xs')
6: **for** $v_i \in \{0, 1\}$ **do**
7: $xs'' \leftarrow xs' \cup (i, v_i)$
8: **if** F is satisfied under xs'' **then return** $\{SAT, xs''\}$
9: **else if** F is indeterminate under xs'' **then**
10: **if** DPLL(F, xs'') $= \{SAT, xs'''\}$ **then return** $\{SAT, xs'''\}$
return $UNSAT$

clauses where only one unassigned variable remains and the other literals are falsified).

- SELECTUNASSIGNEDLITERAL(F, x), which will investigate F and select the next variable to assign.

2.3.3 Conflict-Driven Clause Learning (CDCL) solvers

Building on the basic structure of the DPLL algorithm, a new method of SAT solving was introduced in 1996[39]; this is known as *Conflict-Driven Clause Learning (CDCL)*. As implied by its name, it is concerned with the conflicts encountered during the search, where a conflict is the point at which a clause is found to be unsatisfiable under the current assignment.

When these conflicts are encountered a CDCL solver will attempt to determine the cause of the conflict in order to avoid it in the future, instead of simply undoing the last assignment as a DPLL solver would. This cause is then formulated as a new clause that is added to the formula.

Concepts and Terminology

In order to explain the workings of a CDCL solver, a few concepts first have to be introduced.

Trail During the run of a CDCL solver, it will just like the DPLL solver make an assignment to an unassigned variable. Once the variable is assigned, unit propagation will be used to find assignments that the solver is now forced to make.

This means that the assignments of the solver are made either as explicit choices (referred to as *decisions*) or as the results of unit propagations. The current assignments along with annotations on why they were assigned is referred to as the trail.

As an example, the example formula in section 2.3.2 would result in the following trail:

1. $\overline{x}$ (decision)
2. $\overline{y}$ (unit propagation via $(x \vee \overline{y})$)
3. $\overline{z}$ (unit propagation via $(y \vee \overline{z})$)

Reason When a unit propagation occurs, this is because some clause has been found to only contain one literal that is satisfiable under the current partial assignment. This clause can then be said to be the *reason* the propagation occurred.

Decision levels Because the trail will contain a repeating sequence of decisions each possibly followed by unit propagations, we define the term *decision levels*. This is an incrementing number starting from 0 which is incremented before each decision (so the first decision is at level 1).

Furthermore, this can be referred to as a property of a variable at a particular point during the solver execution. Once again using the example from 2.3.2, we assign $\overline{x}$ at the first level so the decision level of x is 1. Furthermore, both $\overline{y}$ and $\overline{z}$ are propagated from decision 1 and so the decision levels of y and z are both 1 (if one more decision were to be made, it would be at level 2).

Implication graph As noted by Silva and Sakallah [39], the sequence of decisions and their implications (the trail) can be represented by an *implication graph* and can show how a conflict is analyzed by determining the reason for the conflict.

The implication graph is a directed graph where all vertices refer to an assignment (i.e. an item on the trail). Every decision is a source (i.e. a vertex with no ingoing edges) and has outgoing edges to all assignments that follow from unit propagation via a clause it is contained in.

For example, assume the following trail:

1. $x = 1$ by a decision (on level 1). This is represented by a vertex labeled "$x = 1 \ @ \ 1$", where "$@ \ 1$" refers to the fact that it was assigned on decision level 1.

2. $y = 1$ is propagated via the clause $(\overline{x} \vee y)$. This is represented by a vertex labeled "$y = 1 \ @ \ 1$", which has an incoming edge from $x = 1 \ @ \ 1$.

3. $z = 1$ by a decision (on level 2). This is represented by a vertex labeled "$z = 1 \ @ \ 2$".

4. $a = 1$ is propagated via $(\overline{z} \vee \overline{y} \vee a)$. This is represented by a vertex labeled "$a = 1 \ @ \ 2$" which has incoming edges from both "$x = 1 \ @ \ 1$" and "$y = 1 \ @ \ 1$".

Furthermore, assume that this leads to a conflict in the clause $(\overline{z} \vee \overline{a})$. This is represented by a vertex called the conflict vertex, which has incoming edges from all assignments which make it unsatisfiable, i.e. "$z = 1 \ @ \ 2$" and "$a = 1 \ @ \ 2$".

This makes for an implication graph as displayed in figure 2.4.

Traversing this graph is useful as it contains all relevant information on why the solver arrived at a conflict and analyzing this graph can give options for avoiding the same conflict in the future. From the example above, we can see that not only is the formula unsatisfied if $z = 0$ and $a = 0$ (which follows from the conflict clause) but also if $x = 1$ and $z = 1$ or if $y = 1$ and $z = 1$.

Dominator The implication graph may contain *dominators*. A vertex v is said to *dominate* a vertex w if all paths from w to the sink must pass via v. In figure 2.4, the vertex "$a = 1 \ @ \ 2$" dominates "$y = 1 \ @ \ 1$" but not "$z = 1 \ @ \ 2$".

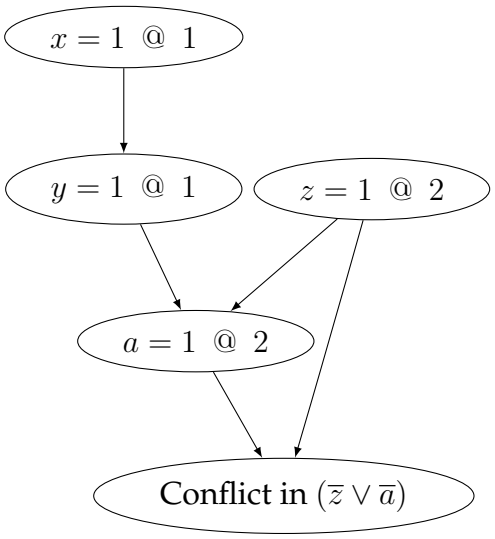

Figure 2.4: An example implication graph.

Description of algorithm

With the relevant concepts introduced, the basic algorithm of a CDCL solver can be described. The basic procedure is similar to that of a DPLL solver where variables are assigned followed by unit propagations from that assignment. What is novel in the CDCL method is what occurs when there is a conflict.

A rough description of the algorithm can be found in algorithm 3. Compared to algorithm 2, three new functions are used:

- LEARNFROMCONFLICT($F, trail$), which will analyze the conflict and determine both a new clause to learned (as described below) and an appropriate place to backtrack to (i.e. assignments to undo).
- BACKTRACKTO($F, trail, backtrackTo$), which will remove any assignments that were made on or after the given *decision level*.
- PICKVARIABLEANDVALUE($F, trail$), which selects a variable and a value (either 1 or 0) to assign using some heuristic.

Learning a clause When the conflict is encountered, the CDCL solver will learn a new clause from the information gathered via the implication graph; this clause will then ensure that the solver will not arrive at the same conflict again.

Algorithm 3 CDCL algorithm

Input:

F - the formula to be solved, containing a set of clauses and their indices. Variables are assumed to start at 0 and there are no gaps, i.e. if x_n exists in the formula then x_i exists for all i such that $i < n$.
trail - a set of tuples representing assignments, represented as (m, v_m, l_m, r_m). This indicates that the variable x_m is assigned the value v_m at decision level l_m. r_m refers to the index of the clause x_m was propagated from, or -1 if x_m was assigned by decision. For example, $(2, 1, 3, 4)$ indicates $x_2 = 1$ on decision level 3 by unit propagation from the clause with index 4. Can be the empty set if no initial assumptions are given.
numVariables - the number of variables in the formula

1: **function** CDCL($F, trail, numVariables$)
2: $decisionLevel \leftarrow 0$
3: $(state, trail) \leftarrow$ UNITPROPAGATE($F, trail$)
4: **if** $state = UNSAT$ **then return** $UNSAT$
5: **while** $|trail| < numVariables$ **do**
6: $decisionLevel \leftarrow decisionLevel + 1$
7: $(i, v_i) \leftarrow$ PICKVARIABLEANDVALUE($F, trail$)
8: $trail \leftarrow trail \cup (i, v_i, decisionLevel, -1)$
9: $(state, trail) \leftarrow$ UNITPROPAGATE($F, trail$)
10: **if** $state = UNSAT$ **then**
11: $(learnedClause, backtrackTo) \leftarrow$ LEARNFROMCONFLICT($F, trail$)
12: **if** $backtrackTo < 0$ **then return** $UNSAT$
13: **else**
14: $decisionLevel \leftarrow backtrackTo$
15: $F \leftarrow F \cup learnedClause$
16: $trail \leftarrow$ BACKTRACKTO($F, trail$)
return $(SAT, trail)$

There are different options for how to choose this learned clause. A naive version might be to simply learn a negation of all the currently assigned variables, as this prohibits at least one of the current assignments (that together falsify the formula).

As an example, assume that the three variables x, y and z have all

been assigned as $x = 1, y = 0$ and $z = 1$, and that this leads to a conflict. One possible clause to learn would simply be $(\overline{x} \vee y \vee \overline{z})$, corresponding to a disjunction of negations of the assignments.

This is however not necessarily enough to avoid the conflict; some of the variables might not be involved in the conflict and so setting them to their opposite value would not prevent the conflict from occurring.

What CDCL solvers do instead is to traverse the implication graph to ensure that only assignments that imply the conflict are part of the learned clause.

The first method of handling this traversal is described in Silva and Sakallah [39] and consists of initially considering the learned clause as containing only the literal of the last decision (i.e. if the last decision was $x = 0$, the learned clause is initially considered to be $(\overline{x})$. This can be justified by the two facts:

- The last decision made is relevant to the conflict, or else the conflict would have occurred earlier
- All other assignments made at the last level are implied by the last decision (via unit propagation)

Furthermore, the implication graph is traversed backwards from the conflict vertex until all vertices have been visited and all source nodes (i.e. decisions) are then added to the clause. Because they are visited during the implication graph traversal it is known that they contribute to the conflict.

Use of unique implication points As described in Silva and Sakallah [39], one can continue the traversal of the implication graph until only decisions remain in the clause (after which, no further traversal is possible). However, it is also possible to stop at an earlier point, which is often preferred. The main motivation for doing so is that this can lead to a smaller clause, which is then said to be a *stronger* implicate because it is a subset of the wider clause and as such more restrictive.

Stopping at an earlier point makes use of unique implication points, *UIPs*. These correspond to *dominators* of the implication graphs. If v dominates the decision vertex at the same decision level, it is said to be a UIP and can substitute the decision in the learned clause. For

exampple, in figure 2.4, "$y = 1 \ @ \ 1$" dominates the decision vertex "$x = 1 \ @ \ 1$" and so it is a UIP.

There have been different propositions for how to utilize unique implication points but the *1UIP* scheme[39] was found by Zhang et al. [44] to be robust and the most effective. It entails stopping the clause learning process as soon as a UIP has been found at the current decision level. This will then be the *first* UIP found and also the one closest to the conflict (as the search in the implication graph starts at the conflict). This first UIP is then the only reason for the conflict at the current decision level, as all other assignments that are involved in the conflict at the current decision level are implied by the first UIP and assignments on other decision levels.

It is somewhat arbitrary that the *first* unique implication point is used and indeed, many different UIPs may exist. There are however proposed reasons for preferring the first UIP. One such reason is the guaranteed longest backtrack[34], meaning that the greatest number of assignments can be undone, pruning more of the search tree. Furthermore, the first UIP is guaranteed to contain literals from the fewest decision levels, which has been proposed as a measure of quality of clauses[6].

In terms of resolution, the process for finding the first UIP clause consists of starting with the conflict clause, resolving with the reason clause for each variable (in reverse trail order) and continuing until only one variable from the last decision level remains.

Backtracking Once the clause is learned the next step is to perform backtracking. A DPLL solver would undo the last decision at each step; this is referred to as *chronological backtracking*. It can be noted however that it is not necessary for the traversal of the implication graph to encounter assignments made at all decision levels. If a certain decision level is never encountered, it follows that undoing the decision made at that level is of no consequence for the conflict that was encountered.

A CDCL solver will then instead choose to backtrack not to the last decision level, but to the last decision level *that was involved in the conflict* (except for the current decision level, which the solver is already at). This is referred to as *non-chronological backtracking*. If no other decision level is involved in the conflict, the backtrack is performed to level 0, meaning that all assignments are undone.

2.3.4 Common extensions

In addition to the CDCL method as described above, there are a number of extensions that are used by solvers that are successful in practice, such as Minisat[21] or Glucose[4].

Clause minimization

When learning a clause by 1UIP it is possible to further minimize the clause in order to both decrease the amount of memory required and to speed up the solving time; this has been implemented in the solver Minisat[21] and is described in Sörensson and Biere [40].

There are two variants of this specific minimization method, with one called *local minimization* and the other called *recursive minimization*. They both have in common that they aim to remove redundant literals from the learned clause.

Assume for example the following scenario:

1. The solver decides $x = 1$, which implies $y = 1$ via unit propagation from $(\overline{x} \vee y)$.
2. The solver decides $a = 1$, which implies $b = 1$ via unit propagation from $(\overline{a} \vee b)$
3. $x = 1$, $y = 1$ and $a = 1$ together imply $z = 1$ via unit propagation from $(\overline{x} \vee \overline{y} \vee \overline{a} \vee z)$
4. A conflict ensues in $(\overline{b} \vee \overline{z})$

In this case the 1-UIP clause found by the algorithm described in 2.3.3 would be the clause $(\overline{a} \vee \overline{x} \vee \overline{y})$. However, because x directly implies y, minimization allows this clause to be minimized to $(\overline{a} \vee \overline{x})$, via resolution of the 1-UIP clause with $(\overline{x} \vee y)$.

The local version would proceed as described above until there there are no literals remaining that are directly implied by the others. If the process is performed in two steps whereby literals are first marked for removal before removing all marked literals at once, the literals can be handled in an arbitrary order.

The recursive version builds on the local but also allows intermediate literals to be introduced if they are guaranteed to be removed eventually. This occurs when a newly introduced intermediate literal

is dominated in the implication graph by literals already in the 1UIP clause, meaning that the newly introduced literal is implied by literals already in the 1UIP clause.

As an isolated example of the above, consider the following:

1. There is a 1UIP clause containing $(\overline{a} \vee \overline{b} \vee \overline{c} \vee \overline{d})$
2. c is propagated via $(\overline{a} \vee \overline{x} \vee c)$
3. x is propagated via $(\overline{a} \vee x)$

In this case it is not possible to directly resolve to remove literals. However, we may perform the following two steps:

1. Resolving $(\overline{a} \vee \overline{b} \vee \overline{c} \vee \overline{d})$ with $(\overline{a} \vee \overline{x} \vee c)$ to get $(\overline{a} \vee \overline{b} \vee \overline{x} \vee \overline{d})$
2. Resolving $(\overline{a} \vee \overline{b} \vee \overline{x} \vee \overline{d})$ with $(\overline{a} \vee x)$ to get $(\overline{a} \vee \overline{b} \vee \overline{d})$

Note how the first step introduces a new literal, $\overline{x}$, which is later removed by the second step. Recursive minimization allows this temporary new literal in order to allow us to end up with a smaller final clause; these temporarily introduced literals are guaranteed to eventually be removed because they are dominated by literals in the 1UIP clause.

This process is repeated until the clause cannot be made any smaller. Unlike the local version, the order variables are handled is of consequence; unless they are handled in reverse assignment order, the derivation may have irregularities.

To summarize:

- *Local* minimization consists of resolution steps which removes literals but introduce no new variables, i.e. strictly reduce the width of the clause at each step, until no more such resolutions are possible.
- *Recursive* minimization consists of taking the 1UIP clause and performing resolution steps that remove literals but that may introduce new literals if these new literals correspond to variables are in turn implied by variables that are in the 1UIP clause. It continues in reverse trail order until it has considered all variables.

Restarts

A heuristic introduced by Gomes, Selman, and Kautz [25] and included in Minisat is the use of restarts to increase the speed of the solver. These restarts will discard all decisions that have been made but not the clauses that have been learned.

Historically, the theoretical justification was thought to be that in combinatorial search, certain parts of the search tree are often exponentially more expensive than others. This could for example correspond to some certain assignment of variables requiring an exceptional amount of cases to be explored. Restarting can then be performed in the hope that the program will choose more fruitful assignments afterwards.

However, this is not the right explanation in the case of CDCL solvers; instead restarts actually increase the reasoning power of the solver.

One justification for this is given by Huang [28]. The solver gets a more informed belief about the what its best next steps are as it is learning new clauses and assigning heuristic scores. While it gains a clearer view as it progresses, it cannot fully execute according to its current belief because it is still bound by its previous decisions. When the solver restarts it is allowed to execute fully according to its updated beliefs and so after a restart the search process is not restricted by previous beliefs that are no longer held. This means that variables can be assigned differently and in a different order.

Furthermore, it has been shown by Pipatsrisawat and Darwiche [35] that a CDCL solver that utilizes restarts is at most polynomially less efficient than general resolution, suggesting that restarts can increase the reasoning power provided that the restarts are frequent enough.

As described by Marques-Silva, Lynce, and Malik [34], there is a variety of options for when to restart, such as randomly or by some activity heuristic. In either case, the fact that clauses are learned is enough to guarantee completeness.

Learned Clause Removal

One issue with clause learning as noted by the Minisat authors Eén and Sörensson [20] is that the database of learned clauses eventually uses excessive space. While the learned clause avoids unfruitful branches in the search tree, they also necessarily use memory and time as they

are handled at each iteration. For this reason, CDCL solvers periodically remove learned clauses from the database. In the case of Minisat, this is by default done using an activity heuristic, meaning that clauses that are used often are more likely to be kept.

Other heuristics for selecting clauses to remove have been proposed. An example is the *literals blocks distance* (LBD) heuristic proposed by Audemard and Simon [6], which considers the number of different decision levels involved in the clause; a lower LBD is found by the authors to make a clause more likely to be used often. As opposed to the activity based heuristics, the LBD can be calculated only once.

Unit Elimination

One further strategy that is found in the code of Minisat but difficult to find described in the literature will be referred to as *unit elimination*. It occurs during the analysis stage where a conflict is analyzed in order to build a learned clause. When a literal is encountered that corresponds to a variable already assigned at decision level 0, Minisat will choose to ignore this literal and prevent it from appearing in the learned clause.

This is valid because assignments at level 0 are not propagations as the results of decisions but instead directly from the original formula and learned clauses. For this reason, the variables they involve can only be assigned that specific value, or the entire formula is necessarily unsatisfied.

Because they are assigned at level 0, the value assigned to the variable will never change. This means that including its negation in any learned clause provides no extra information to the solver and so it is redundant and unproductive use of memory to include it in any learned clause.

2.4 CDCL solvers and resolution refutations

Although CDCL solvers as they are described in 2.3.3 do not generate resolution refutations directly, the clauses learned by the solver are sufficient to generate such a refutation [43].

This section will bridge the gap between CDCL solver theory and resolution refutation theory and show how the execution of a CDCL solver can be framed as resolution. Furthermore, it will show how

common optimizations can be seen as being performed using resolution.

2.4.1 Resolution in a plain CDCL solver

As described earlier, a CDCL solver will assign variables values through some heuristic and use unit propagation to track the implications of these assignments. Once a conflict arises, it uses the implication graph to learn a clause that will avoid the same conflict from arising in the future.

This search through the implication graph can be mapped onto the concept of resolution. Initially, a conflict clause has been found (i.e. a clause in which all of the literals are falsified). This is called the "working clause". Next, the implication graph is used to replace literals in this clause with their reasons (the assignments that together necessarily led to this assignment). This process uses resolution over the current working clause and the reason clause of one of its literals. Once all resolution steps have been performed, the working clause is the new learned clause.

Consider the following example:

1. Assume a conflict arises in $(\overline{x} \vee \overline{y} \vee z)$
2. Furthermore, assume that x was propagated via $(a \vee b \vee x)$ because of the decisions $\overline{a}$ and $\overline{b}$.
3. We can choose to resolve with the reason clause of $\overline{x}$, $(a \vee b \vee x)$, corresponding to how the implication graph is used as described in 2.3.3. This removes the variable x as it appears twice with opposite signs.

$$\frac{(\overline{x} \vee \overline{y} \vee z) \;\; (a \vee b \vee x)}{(\overline{y} \vee \overline{z} \vee a \vee b)} \tag{2.6}$$

4. The resolvent of this operation can then be further refined by again resolving with the reason of one of the literals, until some heuristic indicates that the clause to learn is found.

As described earlier, this process is repeated and the learned clauses are added to the database. Eventually the formula is determined to be unsatisfiable, which occurs when a conflict resolution results in the

empty clause. This can only occur on decision level 0 (i.e. with no decisions made) because otherwise decisions can be reversed to generate new options.

The result of this is that once this final conflict occurs, a sequence of resolution applications has been found that begin with the conflict clause and result in the empty clause. If the derivations of all the learned clauses that are used in this sequence are also provided, a resolution refutation has been obtained.

Algorithm for building this refutation The algorithm starts with the conflict clause and considers this the ”working clause”. Then, the literal in the clause which is assigned last (i.e. occurs last on the trail) is determined. The working clause is then resolved with the reason clause for this assignment. This is repeated until the working clause is empty.

Thus, at each step the working clause is resolved with a reason clause to arrive at a new working clause. The resulting working clause is then resolved with the next reason clause. This can be represented as a graph as has been described earlier, by letting a vertex represent each clause and adding edges from the two ingoing clauses to the resolvent.

Furthermore, the first time a learned clause is used the full derivation of the learned clause is added to the graph (subsequent uses refer back to the same vertex). This means that all source vertices in the graph will represent clauses from the initial formula.

2.4.2 Resolution in CDCL solver extensions

In 2.3.4 a number of extensions to CDCL solvers were described. This section shows how they do not affect the soundness (validity) of the resolution refutation.

Restarts

This technique entails restarting the solver at certain intervals. Because it undoes decisions that have been made, it could cause the solver to arrive at a different refutation. It does however not affect the conflict analysis and so the solver will still arrive at a valid resolution refutation.

Learned Clause Removal

This extensions makes the solver remove some of the learned clauses at certain intervals, usually based on an activity-based heuristic (for example removing learned clauses that are rarely used). This is intended to improve performance, as there is a certain amount of book keeping required for all clauses in the database.

Because this extension only removes clauses from the database but does not introduce new clauses or modify existing ones, it does not affect the resolution refutation; the final refutation can still contain the clause that was removed.

Clause Minimization

The description in 2.3.4, is entirely in terms of resolution.

It should be noted that if a learned clause is used for minimization multiple times, this violates tree-likeness. Thus, a refutation that would otherwise be tree-like will no longer be.

Unit Elimination

Unit Elimination is as described in 2.3.4 the process of removing literals corresponding to variables that are assigned at level 0, i.e. the result of learned unit clauses.

The exact process of framing this as resolution is open to different interpretations. Three different ones have been considered in this thesis (and are demonstrated graphically in A.2 in the appendix):

Ignoring (mode 0) The simplest option is ignoring the elimination of literals when generating the refutation and pretend that these literals end up in the learned clause. As described above, the empty clause is arrived at by resolving with reason clauses in reverse trail order. If unit eliminations are ignored, the learned clause will then contain extra literals, corresponding to variables that are already assigned at level 0. However, once the trail traversal continues these literals will be removed by resolving with the correspoding learned unit clause. Thus, interpreting in this way does not prevent the resolution of the empty clause.

With this interpretation, the learned clauses will be different from the ones actually learned by the solver (as they contain the elimi-

nated literals as well). However, it does not disrupt tree-likeness or regularity.

Resolving with units (mode 1) A second option is using this unit clause at level 0 as any other learned clause and resolving with this in order to eliminate the unit. This means that every time a learned clause is derived and a unit eliminated, an extra resolution step will be performed with this unit clause.

The downside to this is that the same derivation can be performed multiple times, if different learned clause derivations all use the same initial clause and then eliminate the same units. This makes the refutation larger in the number of vertices (but of course smaller in the number of literals in some vertices).

Furthermore, because this learned unit clause is reused every time it is used to eliminate a unit, it introduces violations of tree-likeness; a refutation that would otherwise be tree-like could be made non-tree-like with this mode. Because units can potentially be eliminated multiple times, it can also introduce violations of regularity. To see this, consider the following sequence:

1. Using the initial clause $(x \vee y \vee z)$
2. Resolving with the learned unit clause $(\overline{x})$ to eliminate x and arrive at $(y \vee z)$
3. Resolving with the initial clause $(x \vee a)$ to arrive at $(y \vee z \vee x \vee a)$
4. Resolving with the same learned unit clause $(\overline{x})$ to eliminate x and arrive at $(y \vee z \vee a)$

With this sequence of events, the variable x has been introduced, resolved away and reintroduced; this is a regularity violation. If unit elimination would be ignored this violation would not have occurred.

Furthermore, it has introduced tree violations because the unit clause $\overline{x}$ has been used twice.

Learning intermediate steps (mode 2) The last option that was considered is similar to option 1 but sometimes avoids making the same derivation multiple times

In this interpretation mode, the result of eliminating a literal from a clause is considered to be a new learned clause that can then be reused in multiple places. If $x = 0$ and $y = 0$ are both assigned at decision level 0 and a certain clause $(C \vee x \vee y)$ is used in multiple places, the clause C is first derived once. Then, it can be used directly instead of using the full clause and resolving with the two units multiple times.

In order to ensure as much reuse as possible, units were eliminated in trail order when there were multiple eliminations in a row.

Like the unit resolving mode above, this mode can also introduce regularity and tree-likeness violations.

The reason the choice of interpretation mode is important is that it affects the refutation shape, both in its tree-likeness and its regularity. It does complicate both experimental setup and analysis but because the solver does not use resolution directly when eliminating units there is not one correct option.

Chapter 3

Methods

This chapter describes the methods that were used in this project, both the qualitative and quantitative analysis and their common preparations.

3.1 Preparations

In order to be able to analyze resolution refutations, two steps were required:

1. A CDCL solver was needed. The choice was made to use the solver used in Elffers et al. [22], which is based on the Glucose solver that is in turn based on Minisat, and had been further modified to have extra options and instrumentation capabilities. This particular solver was chosen because it had instrumentation capabilities, all the required parameters and source code that was found to be readable.

   This solver was then modified to output the information required for building resolution refutations.

2. An application was required to build the resolution refutation and generate relevant metrics from it.

While it would be possible to integrate these into the same application, the choice was made to have the solver output as text a detailed log of its actions that were relevant to the resolution refutation; this log could then be used by a separate application to build the actual proof.

This choice was made in order to allow the parts to be tested separately. In the end this ended up being crucial to allow complicated issues to be analyzed and solved.

3.1.1 Modification of solver

As mentioned above, the solver was modified to output a detailed log of all actions that were necessary to recreate the resolution refutation the solver was implicitly building. The source code is available, including these modifications, at `https://github.com/johanlindblad/minisat-instrumented`.

These actions were discovered by first reading the original report by Eén and Sörensson [20] to get an overview of its original structure. This was followed by reading the latest version of the code and attempting to reconcile this with the structure described in the report. Lastly, the solver was run with small example formulae while observing it in a debugger (or via log statements) until the structure was understood.

This process ended up with a set of different classes of instructions that were required to be printed to the log file in order to be able to reconstruct the resolution refutation. These were chosen to not only include what is strictly required, but also include certain verification information so that it could be verified that the solver and the analysis program shared a common view of the process.

The set of instruction is available in the appendix in A.4.

3.1.2 Implementation of resolution graph tool

Once the solver had been modified to output a log of relevant actions, a program was made to consume these logs. This program implemented the data structures required to not only keep the state needed by the solver, but also any extra information required that the solver can choose to only track implicitly. Source code for the program is available at `https://github.com/johanlindblad/resolution-graph-tool`.

As an example, the solver can (as discussed in 2.3.4) eliminate units at decision level 0 by simply noting that a variable is assigned at decision level 0 without being concerned with why that assignment was made. However, in order to make this a valid resolution step in the

resolution refutation, the resolution steps that led to the learned unit clause have to be attached at each point this unit is eliminated.

Furthermore, the solver can simply discard clauses that are removed during the removal of learned clauses. Because these clauses may have been used in the derivation of learned clauses that remain, the resolution graph tool must instead let these remain in memory.

For acceptable performance on large instances, the tool was implemented using C++. It mirrors the steps taken by the solver with the additional data storage that is required and once the solver log is complete, it performs a final analysis step which results in:

- The final resolution refutation as a graph, with labels for all clauses used, different colors depending on source of clauses (initial, learned or intermediate) and indications of the variables that were removed at each resolution step. This graph was output in the DOT format for use with the Graphviz package[23] and used in the qualitative analysis described in 3.2.
- A list of metrics generated from the resolution refutation. These are used in the quantitative analysis described in 3.3; the metrics used are also described there.

3.1.3 Selection of solver configurations

The research question is related to clause learning and restarts and different options are available in Minisat to tune these. The solver was run with different configurations for these in order to attempt to ascertain the effect they have.

In addition to clause learning and restarts, the solver was run with and without minimization since minimization can affect the tree-likeness and regularity.

Clause erasure policy The main option that affects clause learning is the clause erasure policy, which determines how many clauses can be kept in the database, in Minisat's case as a function of the number of conflicts that have occurred. All options available were used, which were

1. `off`, which turns off clause erasure, meaning that learned clauses are never removed

2. `min`, which makes database size scale by $\approx N^{0.25}$ (N being is the number of conflicts encountered)
3. `glu`, which makes the database scale by $\approx N^{0.5}$ (N being the number of conflicts encountered)
4. `lin`, which makes the database scale by $\approx N$ (N being the number of conflicts encountered)
5. `dpll`, which uses a DPLL-like mode described in Elffers et al. [22] that effectively turns clause learning off

It should be noted that the DPLL mode is implemented by performing conflict analysis as usual but effectively hiding the clauses from the solver afterwards. Thus, the solver will learn clauses but the learned clauses will never be reused afterwards.

Restarts This refers to when to restart

1. `off`, which turns off restarts
2. `luE1` and `luE3`, which uses two different frequencies of *luby* sequences related to the number of conflicts, as described by Luby, Sinclair, and Zuckerman [33] (the sequence $1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots$ multiplied by some constant). This *universal* strategy is known to be optimal (up to constant factors) for any randomized algorithm whose distribution of the running time is not known.

   When the number of conflicts encountered since the last restart is equal to the next number in the sequence, it will restart; this makes it a *static* restarting scheme, where the restart interval does not depend on any property of the clauses that are learned.
3. `lbd`, which uses adaptive restarts that depend on the *literals blocks distance* measure as described in Audemard and Simon [5]. This is a *dynamic* restart policy, which considers the clauses that are being learned. This specific policy restarts when the *lbd* measure does not decrease at a fast enough rate with the intention being to restart when the solver appears to be stalling.

Minimization mode This refers to the minimization mode chosen

0. Minimization is turned off

2. Recursive minimization is turned on

Furthermore, all experiments were run with the three different modes of interpreting the elimination of units, as described in 2.3.4 and further in 2.4.2.

3.1.4 Selection of problem instances

The analysis required the selection of suitable problem instances. These were chosen from the many different families used in Elffers et al. [22]. In order to make running the experiments feasible, a subset of families was selected for their different characteristics.

Furthermore, various sizes of these formulae (determined by their "scaling parameter") were selected in order to allow variety but keep the run-time feasible given the resources available. The exact meaning of the scaling parameter depends on the kind of formula but for in general a greater scaling parameter gives a larger formula.

The instances that remain are as follows:

Ordering Principle (OP) The Ordering Principle states that any nonempty set of integers contains a smallest member[2].

Negations of this principle, formulated as propositional formulae, were shown by Stålmarck [41] to require polynomial-sized proofs for general resolution but by Bonet and Galesi [13] to require exponential size for tree-like resolution. Thus they would be expected to benefit from clause learning and restarts.

Relativized Pigeonhole Formula (RPHP) The pigeonhole principle states the (false) claim that there is a way to pick k pigeons and place them in $k-1$ holes without two pigeons sharing a hole.

The *relativized* version of this formula was introduced in Atserias, Lauria, and Nordström [3] and adds the additional restriction that the pigeons fly via n resting places.

They have polynomially-sized proofs even with tree-like resolution (for fixed values of n); thus it is of interest whether clause learning manages to arrive at non-tree-like proofs.

These formulae were used with both 4 and 5 as the values for n, while k was adjusted as a scaling parameter.

Pebbling formulae Pebbling formulae are based on "pebbling games" played on graphs (further explained in the supplementary material to Järvisalo et al. [30]).

These require proofs of exponential size for tree-like resolution but polynomial for general resolution[11] and so they are interesting to study with and without clause learning and restarts. Furthermore, they are known to have large space requirements (i.e. they require a large database of clauses)[10].

Two variants were used in this experiment: pyr and pyrofpyr (meaning *pyramid* and *pyramid-of-pyramid* respectively).

Tseitin formulae These formulae encode systems of XOR equations based on graphs[42]. Some versions of these graphs have time-space trade-offs[8], meaning that the refutation can be shorter if more space is available. They do however quickly grow in size and so formulae were used that are inspired by these but that allow generating a wider variety of sizes. The formulae that were used are then not proven to have time-space trade-offs but if it was found that they do, this would encourage further study with the formulae that are proven to have them.

If there are time-space trade-offs, one would expect clause erasure heuristics to affect the size of the proof since these control the size of the clause database (i.e. the space that is allowed).

3.2 Qualitative analysis

A small-scale qualitative analysis was performed by using the tools mentioned above. Resolution refutations were graphed with the tools in Graphviz[23] and attempts were made to identify qualitative differences between the results depending on the configuration being used. When needed, the graph was also compared with the underlying trace output from the solver since the graph does not give as clear of a chronological view.

In order to arrive at graphs that could be meaningfully interpreted, only very small instances were used. For both the ordering principle and relativized pigeonhole principle it was possible to generate formulae that were small enough for this purpose. For the Tseitin and

pebbling families however this was not possible in the same way (because of the generator tools that were used).

For the two families that remained, small instances were generated to arrive at resolution refutations that were small enough. Because these instances are very small, the choice of restart policy does not affect the generated proof and all the clause erasure modes except the DPLL mode made for identical outputs. For this reason, the comparisons that could be made could be between:

- The $DPLL$ mode (which turns clause learning off) or any of the other clause erasure policies (which allow for clause learning)
- Minimization on or off
- The 3 different interpretation modes for unit elimination

3.3 Quantitative analysis

A larger scale quantitative analysis was run. This was done by using a modified version of the experimental setup used in Elffers et al. [22] on a cluster that was available at TCS. This made it possible to launch a larger number of instances and collect the results of all in a single place.

These resulted in metrics that were generated both by the instrumentation functionality in the modified Minisat solver and by the resolution refutation tool.

The metrics that were generated and used are:

Number of used and unused clauses During conflict analysis, the solver will perform resolution steps to arrive at a learned clause. The resolution graph tool represents these as a graph. Some of the vertices in this graph represent learned clauses and some represent clauses that are from the initial formula; the latter can be refered to as ”initial clauses”. All other vertices represent what can be refered to as ”intermediate clauses”, that are resolvents from two other clauses but not treated as learned clauses by the solver.

Furthermore, some of the resolution steps arrive at learned clauses that are not used in the final resolution refutation. The resolution

graph tool will however still generate vertices and edges to represent these steps. This means that the resolution graph tool generates a graph that contains the resolution refutation as a subset. Thus, all vertices are either in the resolution refutation (in which case they can be refered to as being ”used”) or they are not (in which case they can be refered to as being ”unused”).

This means that vertices can be classified according to both their kind (learned clause, initial clause or intermediate clause) and their usage (used or unused). These two classifications were combined and the number of vertices in each class was counted (so that there is, for example, a metric for the number of unused vertices representing learned clauses).

These can also be combined to count the total number of vertices in the unused portion of the graph.

It should be noted that these metrics also apply in DPLL mode, even though this mode represents clause learning being disabled. This is because the solver will, as described previously, still perform conflict analysis that results in a learned clause but then prevent this clause from being accessed afterwards. Thus, the metrics will indicate learned clauses even for the DPLL mode. This should be taken into consideration when analyzing runs made using this mode.

Number of tree-likeness-violating vertices This referred to the number of clauses (vertices in the refutation graph) that were used more than once, thus breaking tree-likeness of the proof.

Number of tree-likeness-violating edges This referred to the number of times clauses (vertices in the refutation graph) were used more than once. Along with the metric above, this was deemed to be an intuitively understandable metric of "tree-likeness" of the proof, especially when put in proportion to the size of the graph.

Tree copy cost The number of vertices that would need to be copied to turn the resolution refutation into a tree. This copying would have to be performed when a learned clause is used more than once, and would result in copying of the learned clause and all its parent derivation steps. Compared to the simple tree metrics mention above, this captures the *scale* of each tree violations in

that the power of reusing learned clauses is greater if the learned clause was more complex to derive.

Regularity-violating edges The number of times a clause (a vertex in the refutation graph) introduced a regularity violation. Such a violation is introduced when a clause is resolved from two other clauses so that a variable is introduced that has been introduced and later removed in the steps leading up to the two clauses being resolved.

Regularity-violating variables The number of distinct variables that have been involved in regularity violations.

Max width The max width (i.e. the number of literals) of any clause in the resolution refutation.

Chapter 4

Results

This chapter first summarizes the major findings and then presents them in more detail.

In short, the qualitative analysis served well to confirm that the methodology is sound and that the tools produce the correct results. Furthermore, it appears to show concrete small-scale examples of effects that can also be seen in the larger-scale quantitative experiments.

Firstly, the effects of the different unit elimination interpretations are clear to see when running the solver without clause learning (i.e. in the DPLL mode). As would be expected, the resolution refutation is perfectly tree-like when unit elimination is ignored; if this were not the case it would suggest that the generated resolution refutation is incorrect.

When unit elimination is used however, extra resolution steps are performed when building the learned clause, using learned unit clauses in order to remove literals that contradict them. In this case, the generated resolution refutation can be seen to no longer be perfectly tree-like because these learned unit clauses are reused every time such an extra resolution step is performed.

Furthermore, these extra resolution steps do as expected result in a refutation that is longer (i.e. a graph with more vertices) while at the same time making for a lesser maximum clause width (because learned unit clauses are used to remove literals as soon as they appear).

Secondly, running the solver with clause learning enabled shows improvements over running with clause learning disabled. The resolution refutations generated are significantly smaller and no longer perfectly tree-like even when not performing unit elimination. It can

be seen that this appears to be at least in part due to the DPLL mode needing to derive the same clause multiple times.

Lastly, the effects of recursive minimization appear to be to allow the solver to finish in fewer steps. Furthermore, the clauses that are learned appear to be smaller. On the other hand, the resolution refutation is longer, at least in part due to the extra resolution steps that minimization requires. Furthermore, the extra resolution steps make the refutation less tree-like because they potentially involve reuse of learned clauses.

The quantitative analysis uses a wide array of metrics in order to attempt to capture insights from a greater amount of larger formulae. Firstly, its most major finding is that clause learning significantly reduces refutation length for all families of formulae that were used, even for relativized pigeonhole principle formulae that are known to be easy without clause learning. Increasing the size of the database of learned clauses makes the refutations even smaller.

Secondly, restarts have an effect on refutation size although the effect is different depending on the family; for some families of formulae the refutation is longer when restarts are enabled while for others the opposite is true. There appears to be no clear explanation for what this effect results from.

Lastly, it appears that observed effects of clause erasure policies on run time on Tseitin formula are best explained by more aggressive policies requiring a greater amount of time to reach space needed to refute the formula. Because clause database size is a function of the number of conflicts encountered, a more aggressive policy requires more conflicts to reach the same size. This indicates that there are no time-space trade-offs with the Tseitin formulae and solver. However, some indication of time-space trade-offs was found for the pebbling formulae used.

4.1 Qualitative analysis

The findings from the qualitative analysis are described in more detail in this section.

4.1.1 The effects of the different unit elimination interpretations

When running the solver in DPLL mode (i.e. without clause learning), the effects of the different interpretation modes could be seen in isolation. It is known from the theoretic background that for a DPLL solver, the refutation is perfectly tree-like. Figure 4.1 shows an example for such a refutation (using a relativized pigeonhole principle formula).

As discussed earlier however, unit elimination can make the refutation no longer tree-like depending on how it is interpreted. Of the three different ways of interpreting unit elimination used in this report, the first ignores unit elimination altogether; this results in the perfectly tree-like graph referenced above.

The other two modes both involve resolving with learned unit clauses, which reuses the learned clause every time; this is the definition of a tree violation and makes the refutation no longer perfectly tree-like. Figure 4.2 shows the results of this on the same formula as was used above, using interpretation mode "1" (mode "2" leads to an almost identical graph). It can be seen how the units 0 and 15 (corresponding to x_0 and x_{15}) are reused multiple time in the refutation, with every reuse introducing a tree violation.

Furthermore, as has been shown earlier, each such unit elimination violates regularity. As an example, the variable 17 is removed in the derivation of the unit clause (0). Later on when the unit clause (17) is being derived, (0) is used in one of the steps. This is by definition a regularity violation because 17 is introduced and removed when deriving (0) and later on introduced again only to be removed later. Thus, the sequence of variables removed on the path from where 17 is removed when deriving (0) to the empty clause will contain the variable 17 twice.

It can also be seen how this affects the width of some clauses in the refutation. For example, the literal 15 is introduced in two branches at the bottom left hand side of the first graph; when resolving $(\overline{10} \vee \overline{14} \vee \overline{15} \vee \overline{17})$ and $(13 \vee 14 \vee \overline{17})$ to make $(\overline{10} \vee \overline{13} \vee \overline{15} \vee \overline{17})$ and when resolving $(\overline{9} \vee \overline{13} \vee \overline{15} \vee \overline{17})$ and $(13 \vee 14 \vee \overline{17})$ to make $(\overline{9} \vee 14 \vee \overline{15} \vee \overline{17})$. This literal then continues to exist in several resolution steps until it is removed as a result of a resolution with $(\overline{0} \vee 15)$.

In the second graph, it can instead be seen how $\overline{15}$ is removed as soon as it appears. On either side of the vertex containing 15, the same

two steps are represented. To the left, it can be seen how $(\overline{10} \vee \overline{14} \vee \overline{15} \vee \overline{17})$ is first resolved with (15) before resolving the result with $(13 \vee 14 \vee \overline{17})$ to get $(\overline{10} \vee \overline{13} \vee \overline{17})$, which is less wide than $(\overline{10} \vee \overline{13} \vee \overline{15} \vee \overline{17})$ as it does not contain 15. The same is done on the other side to arrive at $(\overline{9} \vee 14 \vee \overline{17})$ instead of $(\overline{9} \vee 14 \vee \overline{15} \vee \overline{17})$.

Thus, the interpretation with unit elimination avoids accumulating these literals that are already known to be unsatisfiable due to contradicting learned unit clauses, which reduces the width of several of the clauses.

However, this decrease in width comes at the cost of length because extra steps are introduced to remove the eliminated unit every time it is introduced. For this reason, the perfectly tree-like refutation contains 157 vertices while the one with unit elimination contains 167.

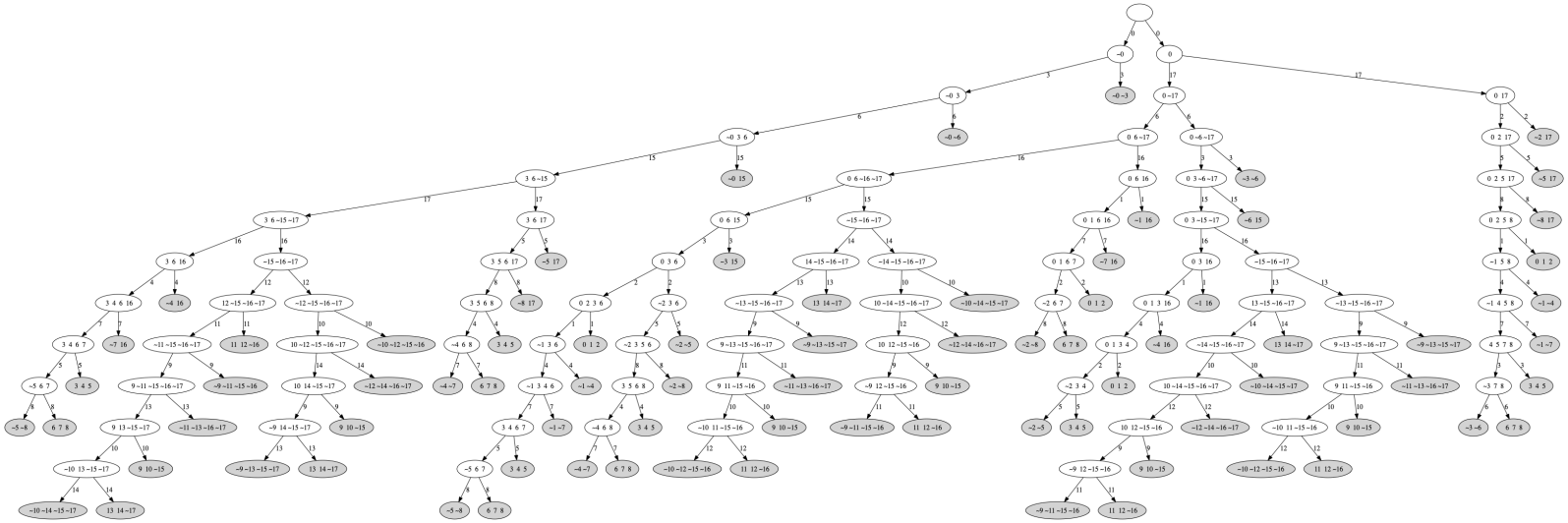

Figure 4.1: The resolution refutation generated for a relativized pigeonhole principle formula ($n = 3$, $k = 3$) in DPLL mode when ignoring unit elimination. *Note: An edge $A \rightarrow B$ means that A is derived from B and the labels on the edges indicate the variable that was removed as a result of the resolution.*

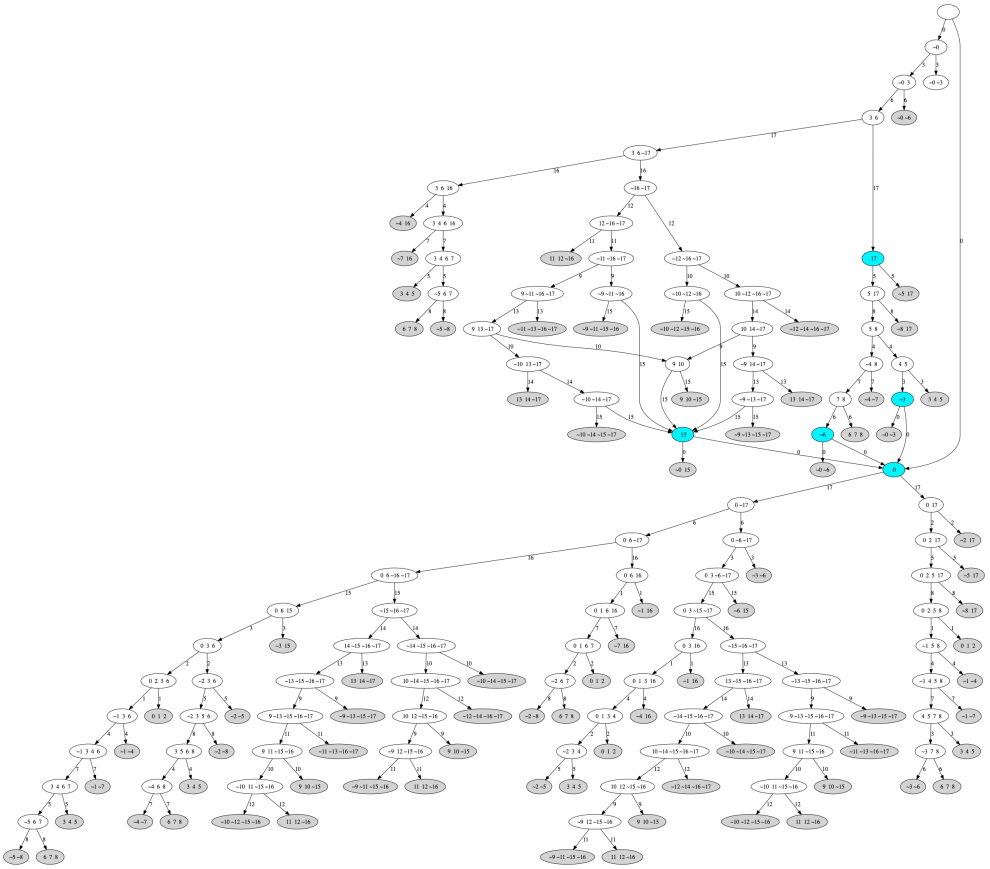

Figure 4.2: The resolution refutation generated for a relativized pigeonhole principle formula ($n = 3$, $k = 3$) in DPLL mode when unit eliminations are interpreted as resolutions with the learned unit clauses. *Note: Turquoise vertices indicate learned unit clauses.*

4.1.2 The effects of clause learning on refutation length and tree-likeness

The effects of clause learning can be seen using the same formula. The results when clause learning is turned off were displayed in 4.1. When instead running with clause learning turned on, the results are as displayed in 4.3 (the different clause erasure modes make no difference on such a small formula).

With the small formula that was used, only one learned clause is used more than once; that clause can be found in the middle of the picture, $\sim 15\ \sim 16\ \sim 17$ (corresponding to $(\overline{x_{15}} \vee \overline{x_{16}} \vee \overline{x_{17}})$). This clause is only ever derived once (the same clause is not found anywhere else in the refutation). In contrast, the non-clause learning refutation contains same clause derived in three places (albeit with slightly different derivations). This means that clause learning enabled the refutation to be smaller by twice the the number of vertices required to derive this clause.

These savings mean that the DPLL refutation contains 157 vertices while the clause learning refutation contains just 115.

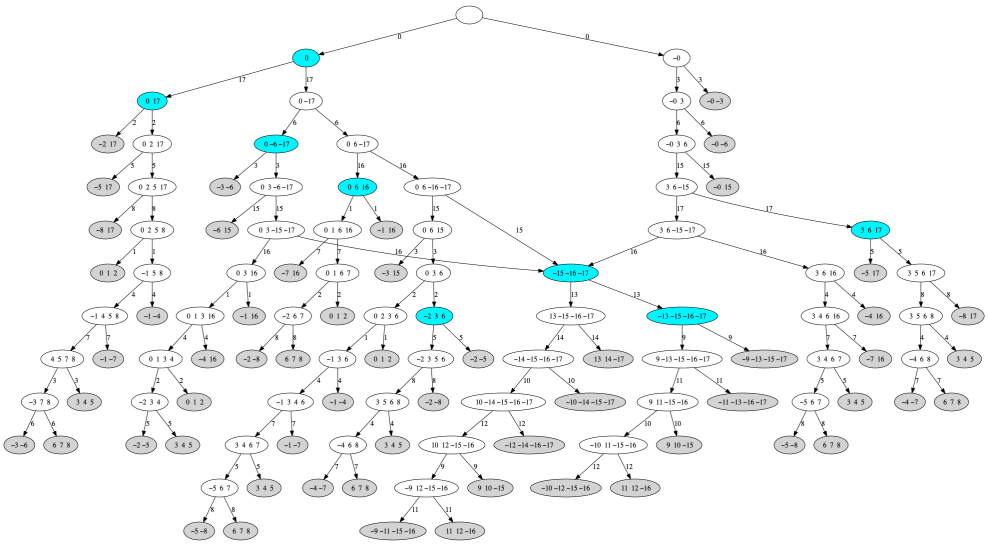

Figure 4.3: The resolution refutation generated for a relativized pigeonhole principle formula ($n = 3$, $k = 3$) when clause learning is enabled and unit elimination ignored. *Note: Turquoise vertices indicate learned clauses.*

4.1.3 The effects of minimization on refutation length and tree-likeness

Lastly, the effects of minimization become somewhat apparent even at this small scale. Figure 4.4 shows the refutation generated by running on an ordering principle formula with clause learning but without clause minimization and unit elimination. It can be seen that the refutation is largely tree-like, with only very few instances of reuse of learned clauses (which causes edges that cross what would otherwise be subtrees).

Figure 4.5 shows the results for the same formula but with clause minimization. The main difference can be seen on the right hand side, where the clause $(0 \vee 19)$ is reused multiple times (causing a tree violation). Investigating the trace of the solver shows that almost all of these uses are during minimization. There are also instances of minimization using initial clauses but these do not affect the tree-likeness because initial clauses are simply copied everywhere they are used.

Investigating the graph more closely together with analyzing the trace output from the solver shows that minimization does not only make the learned clauses less wide, but also allow the solver to finish earlier.

Without minimization, the solver generates 21 clauses before having a full refutation falsifying the formula. With minimization, the learned clauses are the exact same at first, but later on the solver starts learning clauses that are less wide. Finally, it finishes after just 17 learned clauses. This could suggest that because smaller clauses that are subsets of larger clauses eliminate a greater number of possibilities (are *stronger*), the space of possibilities is explored more quickly.

On the other hand, the refutation that is generated is larger, which appears to be because minimization introduces extra resolution steps. In this particular example, there are 238 vertices in the refutation with minimization and 226 without.

It should also be noted that minimization can be seen to introduce regularity violations. In the refutation with recursive minimization, the clause $(0 \vee \overline{9})$ is derived and the last literal that is removed is 19. Later on, $(0 \vee \overline{9})$ is used to minimize $(0 \vee 9 \vee \overline{19})$ into $(0 \vee \overline{19})$, so the variable 19 is introduced, removed and then introduced once more.

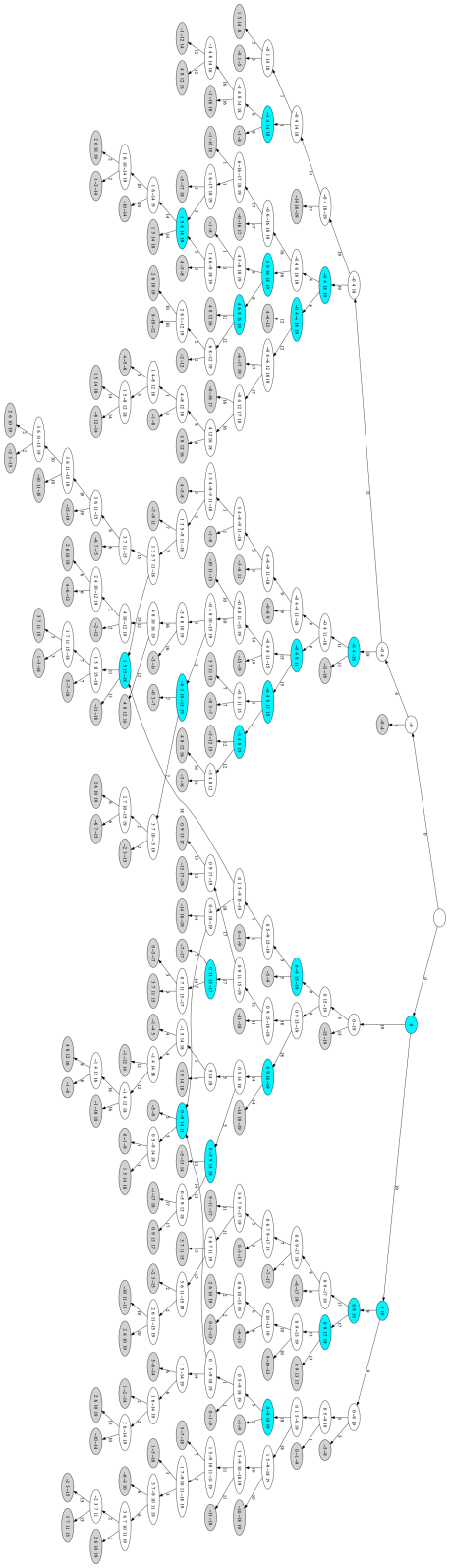

Figure 4.4: The resolution refutation generated for an ordering principle formula (partial, $n = 5$), without clause minimization and unit elimination.

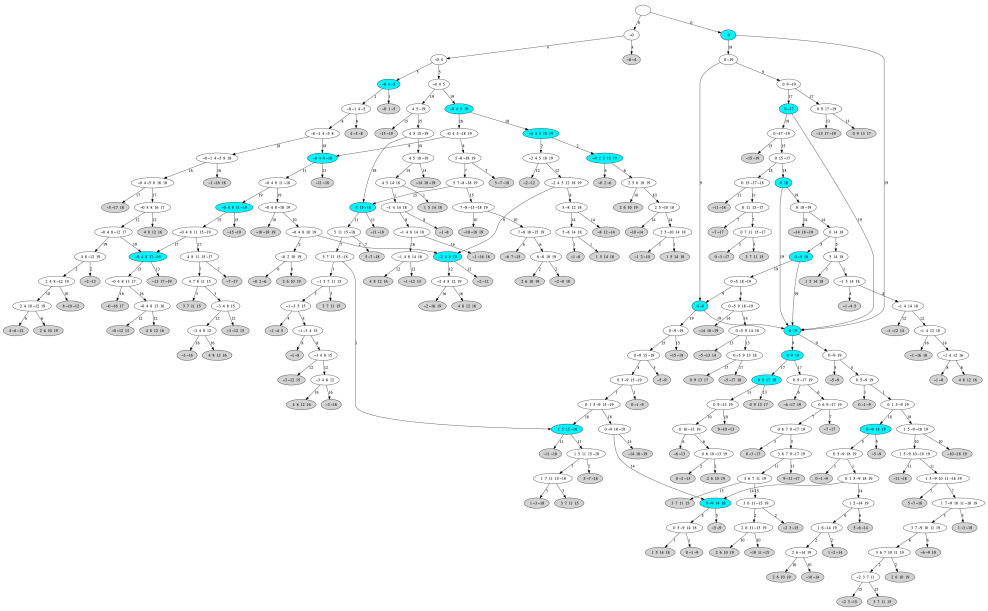

Figure 4.5: The resolution refutation generated for an ordering principle formula (partial, $n = 5$), with clause learning and recursive clause minimization but without unit elimination.

4.2 Quantitative analysis

The experiments collected 18960 samples. Due to the constraints set on both time and memory usage, a majority of samples were of incomplete runs, that failed due to either the solver or the refutation building tool not completing in the allotted time. These incomplete runs could not be used in the analysis as data was only available once the refutation building tool finished and this in turn required the solver to have finished.

Overall, the completion rate differed between different families of formulae. This is to be expected because the experiment was run with formulae of different sizes and no specific effort was made to adjust the sizes of the formulae to match run time or memory usage between families. However, this fact very much limits the strength of the conclusions that can be drawn for some of the families.

One more issue is that many of the larger formulae failed to finish. Because the completion time largely depends on the size of the input there is some cut-off after which the time allotted is no longer sufficient. This however means that for some families, only a handful of smaller inputs led to completed runs. A summary of families and sample status can be found in table 4.1.

Family	Kind	Count
op	Complete	957
	Solver incomplete	3000
	Resolution graph tool incomplete	123
peb	Complete	2367
	Solver incomplete	3624
	Resolution graph tool incomplete	1209
rphp	Complete	1833
	Solver incomplete	1207
	Resolution graph tool incomplete	560
tseitin	Complete	1329
	Solver incomplete	2588
	Resolution graph tool incomplete	163

Table 4.1: Status of collected samples for different formula families

Additionally, there is a difference between the different configurations. It is for example known from for example Elffers et al. [22]

that the min clause erasure mode has a much lower completion rate than the others. This means that comparisons between min and other modes are greatly limited in the amount of inputs that can be compared (and the same is true for the other configuration parameters).

Lastly, it is often of interest to compare with the DPLL mode when evaluating the different clause erasure options (because it effectively turns off clause learning). However, as would be expected from the theory, DPLL takes much longer to run and so there is a much smaller span of formulae it finishes on. This makes clause erasure comparisons more limited as one must either choose to not compare to the DPLL mode or to limit the comparison to only the very few smallest of formulae. A summary of completion rates for DPLL and the other policies can be found in 4.2.

Formula family	Success rate DPLL	Success rate clause learning
op	15%	26%
peb	3%	40%
rphp	29%	56%
tseitin	6%	39%

Table 4.2: Completion rate with and without the DPLL mode (completion meaning that both the solver and the resolution graph tool finished)

The completion rates differ between the families and largely mirror what would be expected given the theoretic background. For $rphp$, which is known to be easy for tree-like resolution, $DPLL$ manages to complete a relatively large number of instances although almost twice as many are completed with clause learning. Pebbling formulae meanwhile, which are known to be hard for tree-like resolution, have almost no completed instances for DPLL but almost half with clause learning.

On the whole, there are many instances where DPLL fails to finish but clause learning does. This needs to be kept in mind when drawing conclusions from the analysis in the following sections.

4.2.1 The effects of clause learning on refutation length

A very clear result that can be seen is that clause learning appears to significantly reduce the length of refutations for all families of formulae. This is true even for relativized pigeonhole principle formulae,

which are known to be easy for tree-like resolution (i.e. the DPLL mode).

Refutation length for the different clause erasure parameters is displayed below; figure 4.6 shows the results for $rphp\_4$ while the results for other families can be found in the appendix in figure A.5, A.6 and A.7.

In order to isolate the effects of clause erasure from the effects of all the other parameters, the data was grouped by instance family, minimization mode, unit elimination interpretation and restart policy so that difference between otherwise identical runs could be investigated. Due to the missing data as noted above, the amount of data was severely restricted and a full data set was only available for running with restarts off, with recursive minimization and with unit elimination mode "1". Furthermore, data was unavailable for the one of the two Tseitin subfamilies, $tseitin\_reggrid\_5$.

The results show that the refutation length is sometimes as little as $\frac{1}{5}$ as long with clause learning compared to the DPLL mode.

Furthermore, comparing the different clause erasure policies suggests that a greater database size makes for shorter refutations, with the refutation length for the min policy (allowing the smallest database) appears greater than the other policies. This is possibly because a larger database allows keeping track of more derived knowledge. The fact that there is less of a difference between glu and lin could suggest that a larger database is useful up to a certain threshold; with larger instances the differences might become more apparent.

Comparing the number of learned clauses in the refutations lends credence this view. Figure 4.7 shows the number of learned clauses in the refutations for the different clause erasure policies. Other than the DPLL and min modes, the number of learned clauses in the refutation are nearly identical, suggesting that a larger database size only made for a short refutation to a certain threshold.

There are learned clauses in the refutation even for the DPLL mode, even though this mode represents running without clause learning. This is due to how the DPLL mode is implemented in the solver; the conflict analysis process is the same and results in a clause being learned but this learned clause is inaccessible to the solver during any future analyses. Thus, the refutation will contain these learned clauses and indicate them as such but they will never be reused. The only exception to this is unit elimination, which will sometimes reuse learned

unit clauses as described in 4.1.1.

This means that it is expected that the refutations for the DPLL mode will contain learned clauses and the results suggest that refutations for the DPLL mode contain significantly more learned clauses than the other modes. However, this can be attributed to the fact that the refutations are significantly longer for the DPLL mode.

This view is strengthened by analyzing the number of times learned clauses are reused (displayed in figure 4.8 for $rphp\_4$). It shows that clauses are very rarely reused for the DPLL mode while there is significant reuse when clause learning is enabled. This shows that while the DPLL mode will sometimes reuse learned clauses for unit elimination, this is still significantly less common than for the modes with clause learning enabled, especially when considering that the refutation is significantly longer for the DPLL mode.

Altogether, these results show that refutation length is significantly reduced when clause learning is enabled. Furthermore, they show that one reasonable explanation is the fact that clauses can be reused instead of derived multiple times, as was shown to happen in the qualitative analysis.

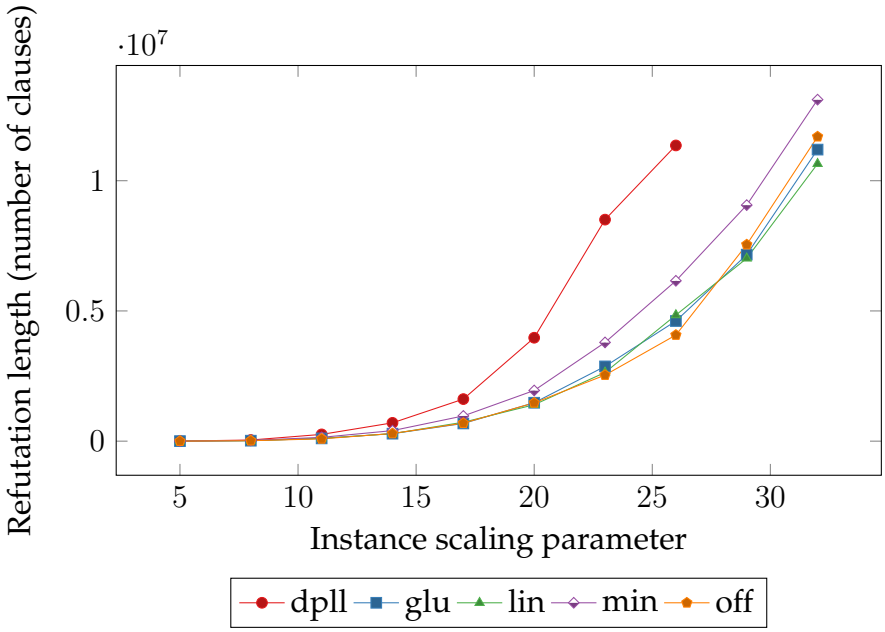

Figure 4.6: The effects of clause learning on refutation length, rphp_4 formulae (recursive minimization, unit elimination mode "1", no restarts)

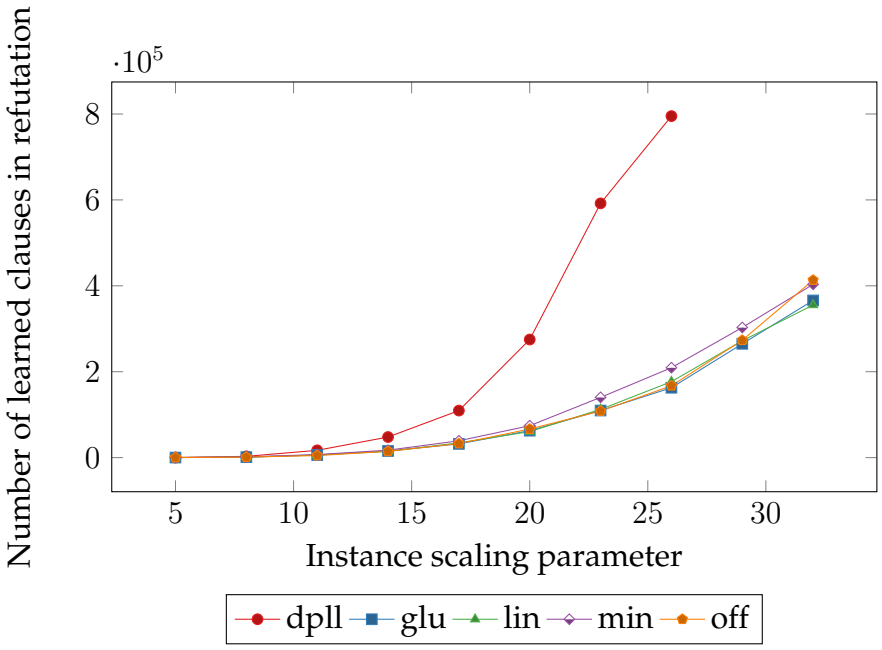

Figure 4.7: The effects of clause learning on number of learned clauses in refutation, rphp_4 formulae (recursive minimization, unit elimination mode "1", no restarts)

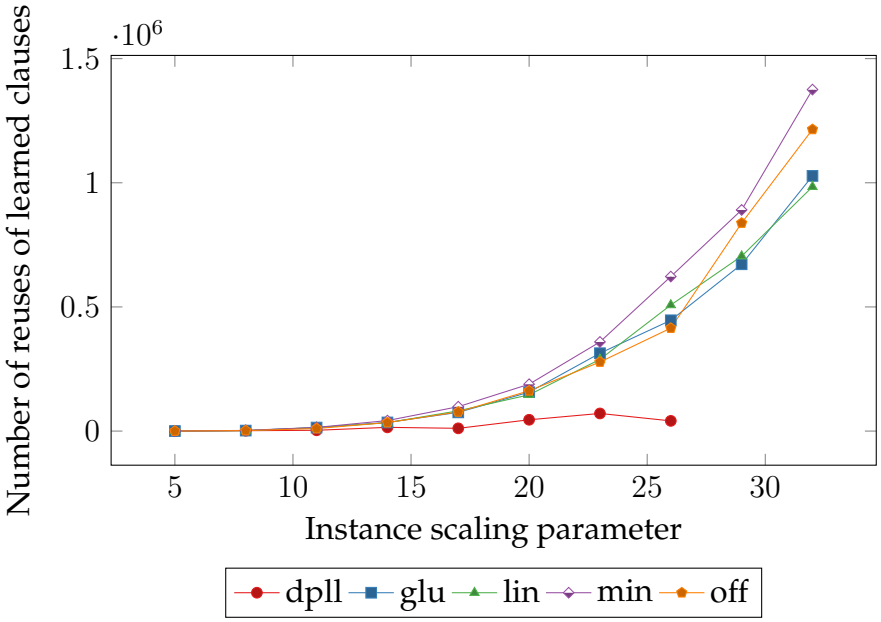

Figure 4.8: The effects of clause learning on number of times learned clauses are reused in refutation, rphp_4 formulae (recursive minimization, unit elimination mode "1", no restarts)

4.2.2 The effects of restarts on refutation length

The effects of restarts were analyzed as it is a known theoretic result that clause learning with restarts is more powerful than clause learning alone. In order to isolate the effects of restarts, configuration parameters were selected such that the same configurations existed for all different sizes of formulae. This limited the size of the formulae somewhat and some of the larger formulae are unused.

The effects for $rphp\_4$ formulae are shown in figure 4.9 while results for $tseitin\_diaggrid\_3$ are shown in 4.10. These two help demonstrate the two different kinds tendencies that were found. The results for the other families can be found in A.14, A.15, A.16, A.17 and A.18.

For $rphp\_4$ as well as $rphp\_5$ the results indicate that restarts are detrimental to refutation length, with the shortest refutations being found when restarts are turned off. For all the other families (such as $tseitin\_diaggrid\_3$) we instead find that restarts make for longer refutations and sometimes dramatically so. This could suggest that for some families, restarts do in fact increase the reasoning power and it seems that being allowed to discard earlier assumptions helps.

Attempts were made to make the difference clearer by looking at parameters other than refutation length, for example tree violations and regularity violations. However, all of these appear to be very well correlated with refutation length.

There were two places where differences could be found; these were number of conflicts encountered and the ratio of the total graph that was used in the refutation.

Firstly, the number of conflicts are displayed for $rphp\_4$ formulae in figure 4.11 while results for $tseitin\_diaggrid\_3$ are shown in 4.12. The results for the other families can be found in A.19, A.20, A.21 and A.28.

These show that for relativized pigeonhole principle formulae, restarts appear to not significantly affect the number of conflicts encountered before the solver finishes. For all other families however, restarts appear to make this number significantly smaller.

As a result of the differences in refutation length and number of conflicts there is a difference in how much of the generated graph is used. For relativized pigeonhole principle formulae, almost the entire graph is used, meaning that almost all clauses that were learned were also used in the refutation. The difference is small between the

different restart modes.

On the other hand, the other families show a clearer difference. There is a more dramatic effect of restarts on refutation length than on the number of conflicts encountered. This means that when restarts are not used, a significantly greater portion of the learned clauses are required to refute the formula.

This means that without restarts, the refutation may end up as much as 4 times longer despite the number of conflicts encountered being merely 2 times as large. Furthermore, almost all of the clauses that have been derived are also needed in the refutation.

One possible explanation for this is that the suggested explanations on the usefulness of restarts are correct; without restarts, one is locked in to earlier assumptions and as a result, one needs to do lots of work to arrive at a result because the initial path is not very fruitful.

If on the other hand one is allowed to restart, one can make a more informed choice on the path to take and so it is possible to arrive at a much shorter refutation quicker. However, this means that part of previous work is not used in the final refutation because it turned out to not be useful once more informed choices could be made.

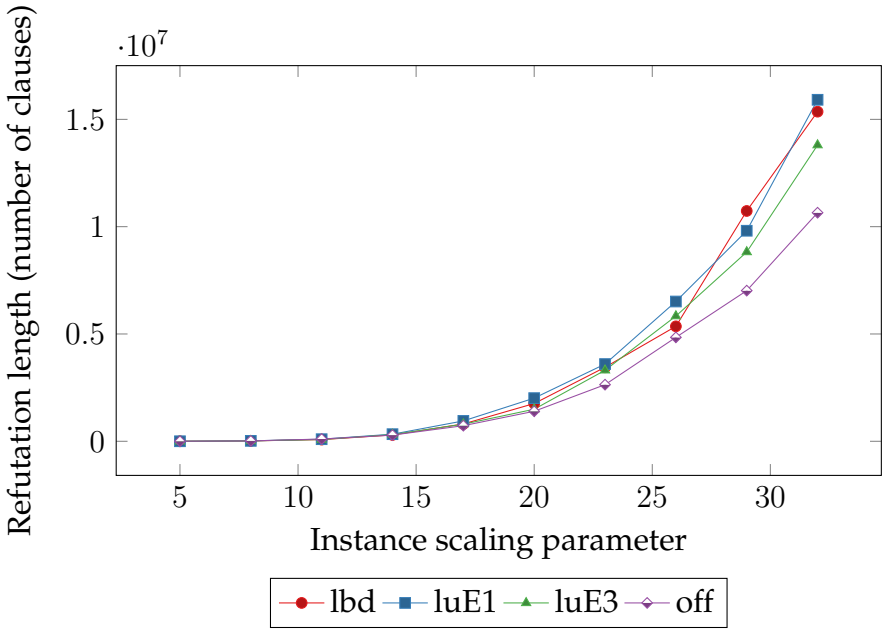

Figure 4.9: The effects of restarts on refutation length, rphp_4 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

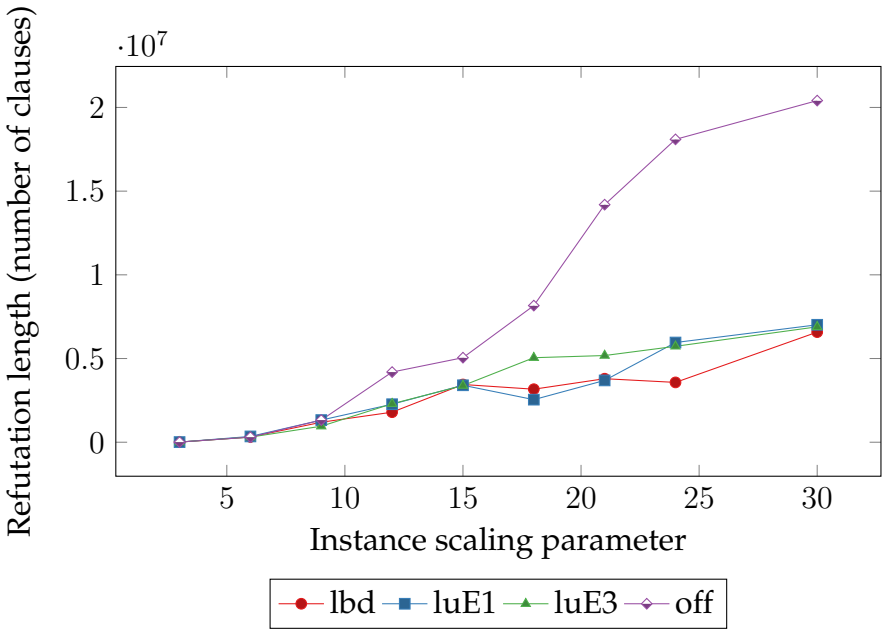

Figure 4.10: The effects of restarts on refutation length, tseitin_diaggrid_3 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

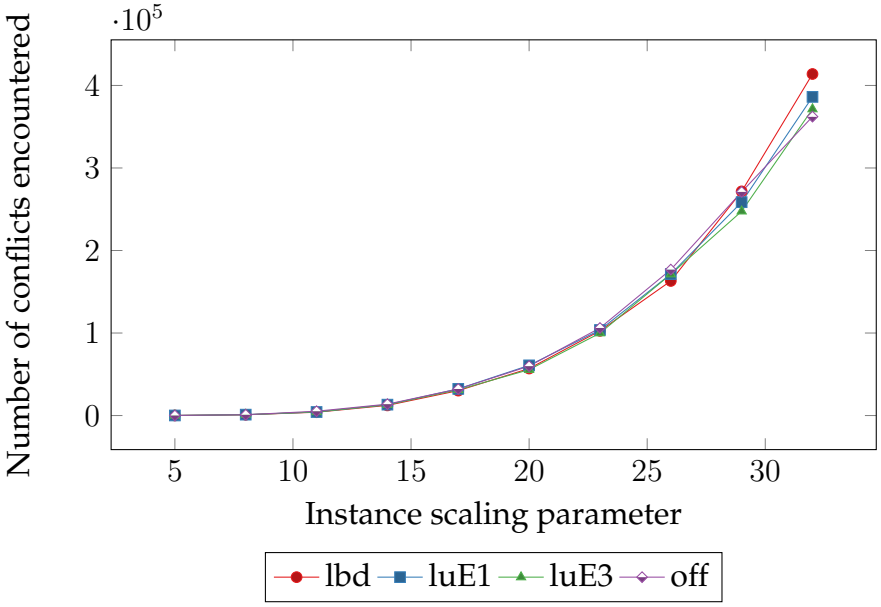

Figure 4.11: The effects of restarts on number of conflicts encountered, rphp_4 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

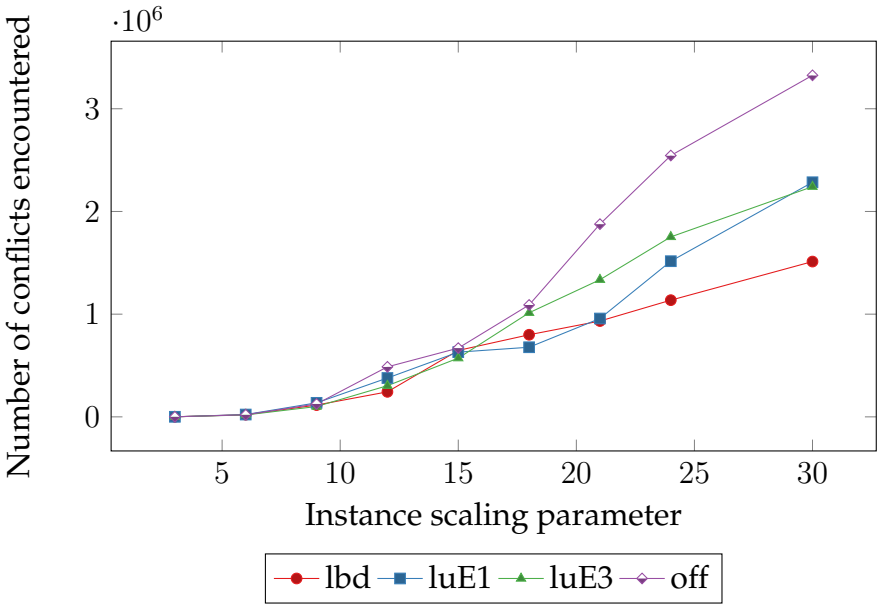

Figure 4.12: The effects of restarts on number of conflicts encountered, tseitin_diaggrid_3 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

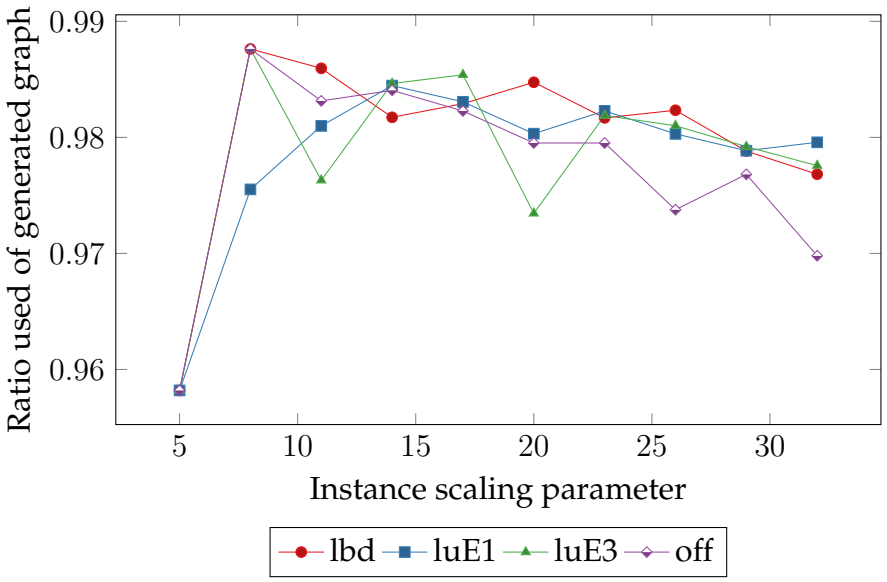

Figure 4.13: The effects of restarts on ratio of generated graph being used as proof, rphp_4 formulae (recursive minimization, unit elimination mode "1", lin clause erasure). The ratio refers to $\frac{\text{Number of vertices in refutation}}{\text{Number of vertices in full graph}}$

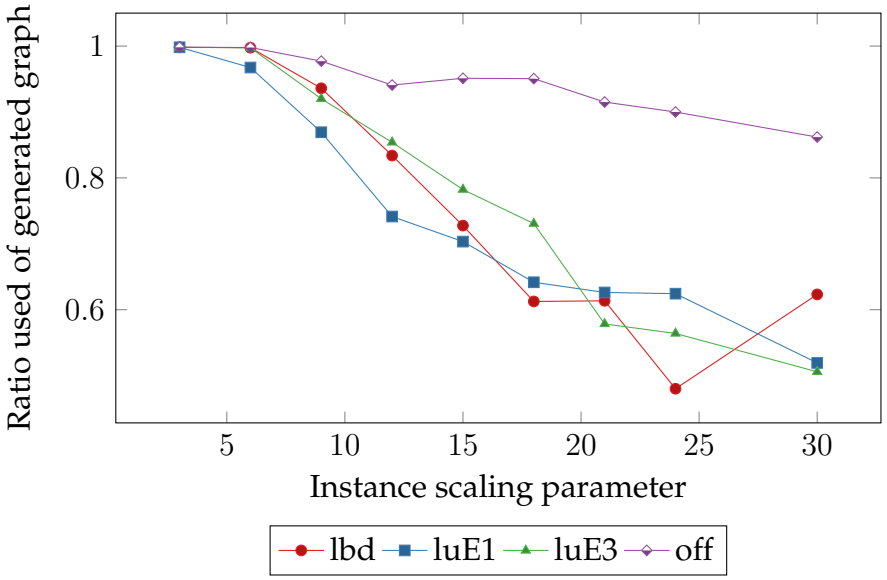

Figure 4.14: The effects of restarts on ratio of generated graph being used as proof, tseitin_diaggrid_3 formulae (recursive minimization, unit elimination mode "1", lin clause erasure). The ratio refers to $\frac{\text{Number of vertices in refutation}}{\text{Number of vertices in full graph}}$

4.2.3 The effects of clause learning on time-space trade-offs in Tseitin and pebbling formulae

As described earlier, there are time-space trade-offs for certain kinds Tseitin formulae, where theoretical models of solvers can finish in less time if they can keep a larger database of clauses. Because these formulae are difficult to scale down to the sizes required, other formulae were instead used which are inspired by these but that do not have the same proven trade-offs. Results indicating such a trade-off would however encourage future research with the original formulae.

These inspired formulae are known to be able to be solved in less time if clause erasure is less aggressive, i.e. if the clause database size is greater. There are two possible explanations for this:

- The larger database allows the solver to use more learned clauses in the refutation, making it shorter (for example, by avoiding deriving the same clause twice). This would be a time-space trade-off.
- The larger database allows the solver to find the exact same refutation quicker. This would not be a time-space trade-off, as clause space is the same. Instead, it would simply show that the formula requires large amounts of space.

In our solver, the database size is not constant but instead increases over time in relation to the number of conflicts encountered. For this reason, we are interested in finding out whether a larger database size compared to the number of conflicts allows a shorter refutation to be found or if the solver simply requires more time to reach the database size required for arriving at a similar refutation.

For this reason, the metrics chosen were refutation length and number of conflicts, which were compared between the different clause erasure policies. The reasoning is that this would contrast the size of the refutation with the number of learned clauses that were generated. Furthermore, the number of learned clauses in the proof was used to show the difference in tree-likeness.

In order to help make the results clearer, pebbling formulae were investigated in the same manner. These do not have time-space trade-offs but are known to have large space requirements. Thus, a similar results for Tseitin formulae and pebbling formulae would be reason to favor the latter of the two explanations.

Figure 4.15 and 4.16 show the results for number of conflicts for $tseitin\_diaggrid\_3$ and peb_pyr_neq3 formula while the appendix contains the results for $tseitin\_reggrid\_5$ and $peb\_pyrofpyr\_neq3$ formulae in figure A.29 and A.30. The results for refutation length are displayed in 4.17, 4.18, A.31 and A.32. The results for number of learned clauses in refutation are displayed in 4.19, 4.20, A.33 and A.34.

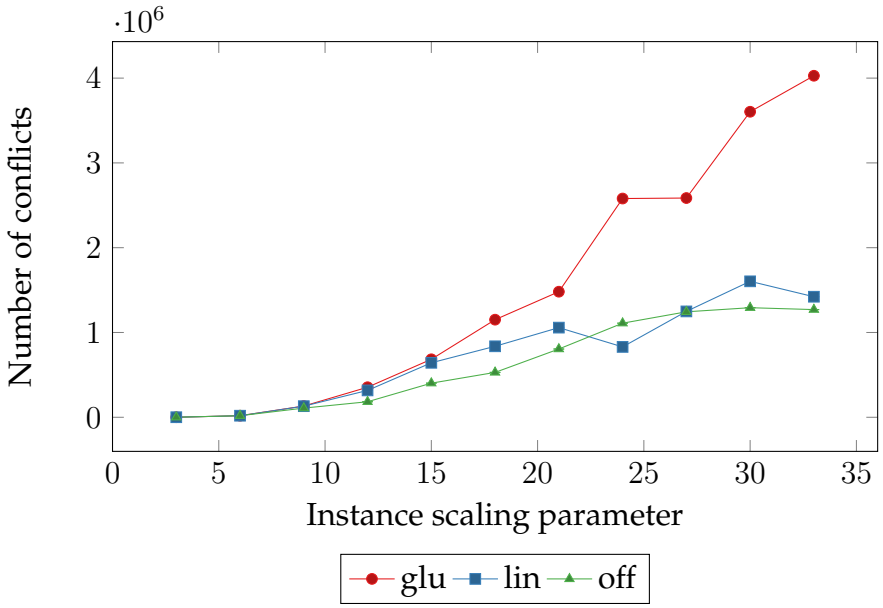

Figure 4.15: The effects of different clause erasure policies on number of conflicts, tseitin_diaggrid_3 formulae (lbd restarts, no minimization)

Unfortunately, no results are available for the min clause erasure policy, as it did not finish in time for any other than the smallest of formulae. However, all three unit elimination interpretation modes were available for the other policies, meaning that the effects of the interpretation modes can be seen. As the errors bars are next to invisible, it appears that the interpretation modes do not significantly affect this result.

The results show that the smaller database size for the glu mode appears to result in a significantly greater number of conflicts, i.e. a greater running time. This likely explains why the min policy did not finish, as it allows for the smallest database size; if the same pattern holds it would require even more conflicts before finishing.

For both families of formulae, there is a significant effect of clause

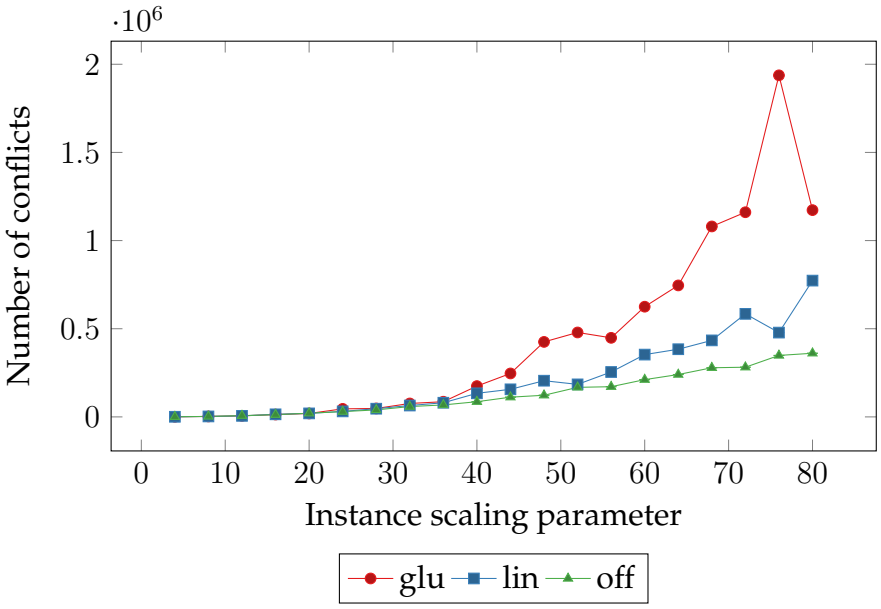

Figure 4.16: The effects of different clause erasure policies on number of conflicts, peb_pyr_neq3 formulae (lbd restarts, no minimization)

erasure policy on run time, where glu restarts in some cases required almost 4 times as many conflicts. However, for Tseitin formulae this effect can not be seen on the refutation. Instead, both the refutation length and the number of learned clauses in the refutation are almost identical.

For pebbling formulae the opposite appears to be true; in addition of number of conflicts being significantly different, the refutation length and number of learned clauses also differed significantly.

These results indicate that for Tseitin formulae, the best explanation for the increase in run time with aggressive clause erasure appears to be the latter of the two mentioned; with a more aggressive clause erasure policy the solver requires more time to reach the database size that is required for refuting the formula. Meanwhile, the pebbling formulae used appear to show some time-space trade-off.

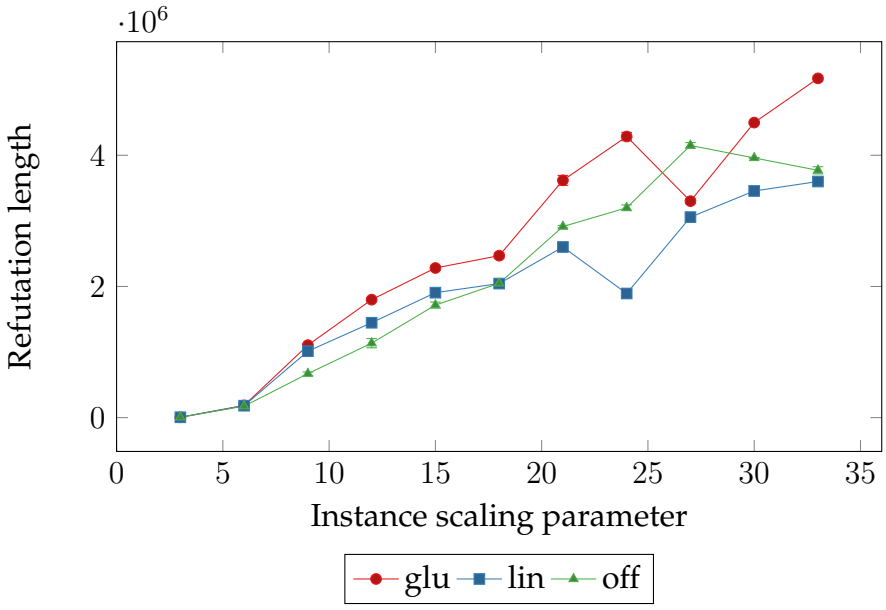

Figure 4.17: The effects of different clause erasure policies on refutation length for tseitin_diaggrid_3 formulae (lbd restarts, no minimization

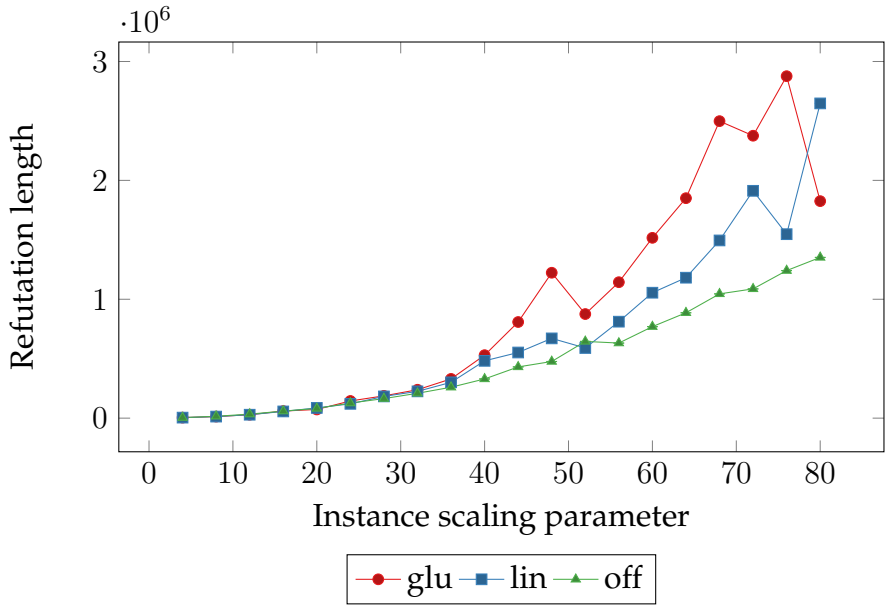

Figure 4.18: The effects of different clause erasure policies on refutation length for peb_pyr_neq3 formulae (lbd restarts, no minimization

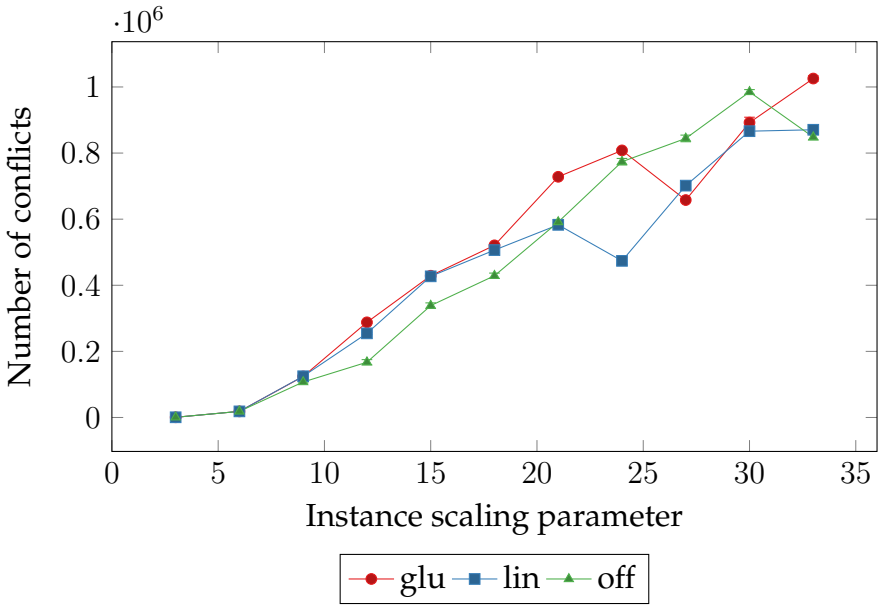

Figure 4.19: The effects of different clause erasure policies on number of learned clauses in refutation, tseitin_diaggrid_3 formulae (lbd restarts, no minimization)

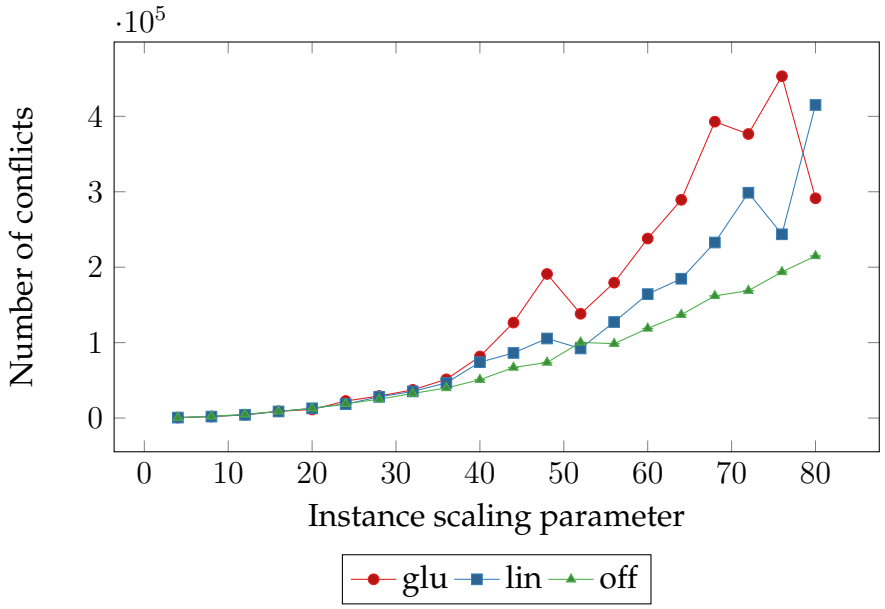

Figure 4.20: The effects of different clause erasure policies on number of learned clauses in refutation, peb_pyr_neq3 formulae (lbd restarts, no minimization)

Chapter 5

Conclusions

This work has aimed to study resolution refutations of real-life CDCL solvers in order to ascertain to what degree they utilize the reasoning power that theoretical results afford them. Furthermore, different configurations were used for clause learning, clause erasure and restarts in order to study whether they affected the refutations. If these make the solver utilize more powerful reasoning, one would expect to find significant differences in both the length and the shape.

In short, the results suggest that clear differences in the refutations can be found. Most notably, clause learning was found to allow for significantly shorter refutations on all families that were studied and a more lenient clause erasure policy was found to improve it further. Small-scale examples indicate that this is partly explained by reusing learned clauses instead of deriving the same clause multiple times.

Restarts were also found to affect the length of the proof although the results differed between different families of formulae; families that were easy for tree-like resolution seemed to require longer proofs when restarts were performed. It is not clear what the shorter length for the other families is best explained by. One possible explanation is that the solver can make better quality assumptions when no longer bound by previous ones; this would explain why the refutations is shorter but less of work performed actually being used in the refutation.

Lastly, the time-space trade-offs are known to exist for theoretical models of solvers for certain kinds of Tseitin formulae. Indications of these have been found in practice on formulae that are inspired for these, where a smaller clause database makes the solver require more

time to refute the formula. The results indicate that these are not due to time-space trade-offs; instead the solver appears to simply require more time to reach the database size required to refute the formula.

5.1 Future work

The experiments that were run were plagued by incomplete runs in which the solver or the analysis tools could not finish in time. This meant that the data that was collected was incomplete and in some cases significantly so. Furthermore, the formulae that could be analyzed were relatively small.

Thus, one area of future work would be to investigate the question further with more complete data from larger instances. Some of the results appear only above a certain size of formulae and it is possible that other effects would appear with larger formulae than the ones that were used in this work.

Furthermore, the results for restarts are not fully clear and the explanation for their significant effect on refutation length are difficult to explain with the other metrics that were collected. A more focused effort to identify a more well-motivated explanation would likely be of value. This could be done either by attempting to find metrics that manage to capture the differences or by studying the unused portions of resolution graphs in order to study the work that is discarded.

Lastly, the issue of unit elimination is briefly described in this work and it is clear that how this is interpreted significantly affects the shape of the resulting refutation. Although it is used as a memory saving technique, it also appears to mimic clause learning to a certain degree (though it is necessarily limited to unit clauses). A more thorough exploration of this topic could potentially investigate theoretical models of this behaviour or suggest alternative interpretations that more closely represent the reasoning the solver is actually performing.

Bibliography

[1] Michael Alekhnovich et al. "An Exponential Separation between Regular and General Resolution". In: *Theory of Computing* 3.5 (2007), pp. 81–102. DOI: 10.4086/toc.2007.v003a005. URL: http://www.theoryofcomputing.org/articles/v003a005.

[2] Tom M. Apostol. "The well-ordering principle". In: *Calculus, Vol. 1: One-Variable Calculus, with an Introduction to Linear Algebra*. 1967, pp. 34–41.

[3] Albert Atserias, Massimo Lauria, and Jakob Nordström. "Narrow Proofs May Be Maximally Long". eng. In: *ACM Transactions on Computational Logic (TOCL)* 17.3 (2016), pp. 1–30. ISSN: 1557-945X.

[4] Gilles Audemard and Laurent Simon. "Glucose and Syrup in the SAT'17". In: *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*. Ed. by Tomaáš Balyo, Marijn JH Heule, and Matti Järvisalo. Vol. B-2017-1. University of Helsinki, Department of Computer Science, 2017.

[5] Gilles Audemard and Laurent Simon. "GLUCOSE: a solver that predicts learnt clauses quality". In: *SAT Competition* (2009), pp. 7–8.

[6] Gilles Audemard and Laurent Simon. "Predicting Learnt Clauses Quality in Modern SAT Solvers". In: *Proceedings of the 21st International Jont Conference on Artifical Intelligence*. IJCAI'09. Pasadena, California, USA: Morgan Kaufmann Publishers Inc., 2009, pp. 399–404. URL: http://dl.acm.org/citation.cfm?id=1661445.1661509.

[7] Lus Baptista and JP Marques-Silva. "The interplay of randomization and learning on real-world instances of satisfiability". In: *Proceedings of AAAI Workshop on Leveraging Probability and Uncertainty in Computation.-July*. 2000.

[8] Paul Beame, Chris Beck, and Russell Impagliazzo. "Time-Space Trade-offs in Resolution: Superpolynomial Lower Bounds for Superlinear Space". In: *SIAM Journal on Computing* 45.4 (2016), pp. 1612–1645.

[9] Paul Beame, Henry Kautz, and Ashish Sabharwal. "Towards understanding and harnessing the potential of clause learning". In: *Journal of Artificial Intelligence Research* 22 (2004), pp. 319–351.

[10] E. Ben-Sasson and J. Nordström. "Short Proofs May Be Spacious: An Optimal Separation of Space and Length in Resolution". In: *2008 49th Annual IEEE Symposium on Foundations of Computer Science*. Oct. 2008, pp. 709–718. DOI: 10.1109/FOCS.2008.42.

[11] Eli Ben-Sasson, Russell Impagliazzo, and Avi Wigderson. "Near optimal separation of tree-like and general resolution". In: *Combinatorica* 24.4 (2004), pp. 585–603.

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

[13] Maria Luisa Bonet and Nicola Galesi. "Optimality of size-width tradeoffs for resolution". In: *Computational Complexity* 10.4 (2001), pp. 261–276.

[14] Maria Luisa Bonet, Toniann Pitassi, and Ran Raz. "On Interpolation and Automatization for Frege Systems". In: *SIAM J. Comput.* 29.6 (Apr. 2000), pp. 1939–1967. ISSN: 0097-5397. DOI: 10.1137/S0097539798353230. URL: https://doi.org/10.1137/S0097539798353230.

[15] Maria Luisa Bonet et al. "On the Relative Complexity of Resolution Refinements and Cutting Planes Proof Systems". In: *SIAM Journal on Computing* 30.5 (2000), pp. 1462–1484. DOI: 10.1137/S0097539799352474. eprint: https://doi.org/10.1137/S0097539799352474.

[16] Stephen A. Cook. "The Complexity of Theorem-proving Procedures". In: *Proceedings of the Third Annual ACM Symposium on Theory of Computing*. STOC '71. Shaker Heights, Ohio, USA: ACM, 1971, pp. 151–158. DOI: 10.1145/800157.805047.

[17] Stephen A. Cook and Robert A. Reckhow. "The Relative Efficiency of Propositional Proof Systems". In: *The Journal of Symbolic Logic* 44.1 (1979), pp. 36–50. ISSN: 0022-4812. URL: http://www.jstor.org/stable/2273702.

[18] W. Cook, C.R. Coullard, and Gy. Turán. "On the complexity of cutting-plane proofs". In: *Discrete Applied Mathematics* 18.1 (1987), pp. 25–38. ISSN: 0166-218X. DOI: 10.1016/0166-218X(87)90039-4. URL: http://www.sciencedirect.com/science/article/pii/0166218X87900394.

[19] Martin Davis, George Logemann, and Donald Loveland. "A Machine Program for Theorem-proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782. DOI: 10.1145/368273.368557.

[20] Niklas Eén and Niklas Sörensson. "An extensible SAT-solver". In: *International conference on theory and applications of satisfiability testing*. Springer. 2003, pp. 502–518.

[21] Niklas Eén and Niklas Sörensson. *Minisat 2.1 and minisat++ 1.0-sat race 2008 editions*. Tech. rep. Chalmers University of Technology, Sweden, 2008.

[22] Jan Elffers et al. "Seeking Practical CDCL Insights from Theoretical SAT Benchmarks". In: *Proceedings of the Twenty-Seventh International Joint Conference on Artificial Intelligence, IJCAI-18*. International Joint Conferences on Artificial Intelligence Organization, July 2018, pp. 1300–1308. DOI: 10.24963/ijcai.2018/181.

[23] John Ellson et al. "Graphviz and dynagraph – static and dynamic graph drawing tools". In: *GRAPH DRAWING SOFTWARE*. Springer-Verlag, 2003, pp. 127–148.

[24] John Franco and John Martin. "A History of Satisfiability". In: *Handbook of satisfiability*. Ed. by Armin Biere, Marijn Heule, and Hans van Maaren. Vol. 185. IOS press, 2009. Chap. 2, pp. 75–97.

[25] Carla P. Gomes, Bart Selman, and Henry Kautz. "Boosting Combinatorial Search Through Randomization". In: *Proceedings of the Fifteenth National/Tenth Conference on Artificial Intelligence/Innovative Applications of Artificial Intelligence*. AAAI '98/IAAI '98. Madison, Wisconsin, USA: American Association for Artificial Intelligence, 1998, pp. 431–437. ISBN: 0-262-51098-7. URL: http://dl.acm.org/citation.cfm?id=295240.295710.

[26] Marijn J. H. Heule, Matti Juhani Järvisalo, and Martin Suda. "Proceedings of SAT Competition 2018; Solver and Benchmark Descriptions". In: *Department of Computer Science Series of Publications B*. 2018.

[27] Marijn J. H. Heule, Matti Juhani Järvisalo, and Martin Suda. *SAT Competition 2018; Overview and Results*. SAT'18, Oxford, July 12, 2018. URL: http://sat2018.forsyte.tuwien.ac.at/downloads/satcomp18slides.pdf.

[28] Jinbo Huang. "The Effect of Restarts on the Efficiency of Clause Learning". In: *Proceedings of the 20th International Joint Conference on Artifical Intelligence*. IJCAI'07. Hyderabad, India: Morgan Kaufmann Publishers Inc., 2007, pp. 2318–2323. URL: http://dl.acm.org/citation.cfm?id=1625275.1625649.

[29] Kazuo Iwama. "Complexity of finding short resolution proofs". In: *International Symposium on Mathematical Foundations of Computer Science*. Springer. 1997, pp. 309–318.

[30] M. Järvisalo et al. "Relating proof complexity measures and practical hardness of SAT". eng. In: vol. 7514. 2012, pp. 316–331. ISBN: 978-364233557-0.

[31] Daniel Kroening. "Software Verification". In: *Handbook of satisfiability*. Ed. by Armin Biere, Marijn Heule, and Hans van Maaren. Vol. 185. IOS press, 2009. Chap. 16, pp. 505–532.

[32] M. Lauria et al. "CNFgen : A generator of crafted benchmarks". In: *20th International Conference on Theory and Applications of Satisfiability Testing, SAT 2017 :* vol. 10491. Lecture Notes in Computer Science (including subseries Lecture Notes in Artificial Intelligence and Lecture Notes in Bioinformatics) 10491. QC 20170919. 2017, pp. 464–473. ISBN: 9783319662626. DOI: 10.1007/978-3-319-66263-3_30.

[33] Michael Luby, Alistair Sinclair, and David Zuckerman. "Optimal speedup of Las Vegas algorithms". In: *Theory and Computing Systems, 1993., Proceedings of the 2nd Israel Symposium on the*. IEEE. 1993, pp. 128–133.

[34] Joao Marques-Silva, Ines Lynce, and Sharad Malik. "Conflict-Driven Clause Learning SAT Solvers". In: *Handbook of satisfiability*. Ed. by Armin Biere, Marijn Heule, and Hans van Maaren. Vol. 185. IOS press, 2009. Chap. 4, pp. 131–153.

[35] Knot Pipatsrisawat and Adnan Darwiche. “On the power of clause-learning SAT solvers as resolution engines”. In: vol. 175. 2. Elsevier, 2011, pp. 512–525.

[36] Robert A Reckhow. “On the lengths of proofs in the propositional calculus.” In: (1975).

[37] Jussi Rintanen. “Planning and SAT”. In: *Handbook of satisfiability*. Ed. by Armin Biere, Marijn Heule, and Hans van Maaren. Vol. 185. IOS press, 2009. Chap. 15, pp. 483–504.

[38] J. A. Robinson. “A Machine-Oriented Logic Based on the Resolution Principle”. In: *J. ACM* 12.1 (Jan. 1965), pp. 23–41. ISSN: 0004-5411. DOI: 10.1145/321250.321253.

[39] João P Marques Silva and Karem A Sakallah. “GRASP—a new search algorithm for satisfiability”. In: *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*. IEEE Computer Society. 1997, pp. 220–227.

[40] Niklas Sörensson and Armin Biere. “Minimizing learned clauses”. In: *International Conference on Theory and Applications of Satisfiability Testing*. Springer. 2009, pp. 237–243.

[41] Gunnar Stålmarck. “Short resolution proofs for a sequence of tricky formulas”. In: *Acta Informatica* 33.3 (May 1, 1996), pp. 277–280. ISSN: 1432-0525. DOI: 10.1007/s002360050044.

[42] Grigori Tseitin. “On the complexity of derivation in propositional calculus”. In: *Studies in constructive mathematics and mathematical logic* (1968), pp. 115–125.

[43] Lintao Zhang and Sharad Malik. “Validating SAT Solvers Using an Independent Resolution-Based Checker:Practical Implementations and Other Applications”. eng. In: *Design, Automation, and Test in Europe: Proceedings of the conference on Design, Automation and Test in Europe - Volume 1; 03-07 Mar. 2003*. Vol. 1. 2003. ISBN: 0769518702. URL: http://search.proquest.com/docview/31674065/.

[44] Lintao Zhang et al. “Efficient conflict driven learning in a boolean satisfiability solver”. In: *Proceedings of the 2001 IEEE/ACM international conference on Computer-aided design*. IEEE Press. 2001, pp. 279–285.

Appendix A

Unnecessary Appended Material

A.1 Comparison of cutting-planes proofs and resolution refutations

This section will compare proofs that the pigeonhole principle formula is unsolvable, using both the cutting-planes proof system and the resolution proof system. Specifically, the formula under question is one which claims that 3 pigeons cannot share 2 pigeon holes if we demand that each pigeon has its own hole.

This can be expressed in CNF form. Let $p_{x,hy}$ refer to pigeon x being in the pigeonhole y. Then the requirement that each pigeon is in at least one pigeonhole can be expressed by the three following clauses (one for each pigeon):

$$(p_{1,h1} \vee p_{1,h2})$$

$$(p_{2,h1} \vee p_{2,h2})$$

$$(p_{3,h1} \vee p_{3,h2})$$

Furthermore, we require that each hole only contains one pigeon. This demands the following two groups of clauses (one for each hole):

$$(\overline{p_{1,h1}} \vee \overline{p_{2,h1}})$$

$$(\overline{p_{1,h1}} \vee \overline{p_{3,h1}})$$

$$(\overline{p_{2,h1}} \vee \overline{p_{3,h1}})$$

$$(\overline{p_{1,h2}} \vee \overline{p_{2,h2}})$$
$$(\overline{p_{1,h2}} \vee \overline{p_{3,h2}})$$
$$(\overline{p_{2,h2}} \vee \overline{p_{3,h2}})$$

These 9 clauses together encode the pigeonhole principle.

A.1.1 Cutting-planes proof

The same formula expressed in the cutting-planes format is as follows:

$$p_{1,h1} + p_{1,h2} \geq 1$$
$$p_{2,h1} + p_{2,h2} \geq 1$$
$$p_{3,h1} + p_{3,h2} \geq 1$$

$$-p_{1,h1} - p_{2,h1} \geq -1$$
$$-p_{1,h1} - p_{3,h1} \geq -1$$
$$-p_{2,h1} - p_{3,h1} \geq -1$$

$$-p_{1,h2} - p_{2,h2} \geq -1$$
$$-p_{1,h2} - p_{3,h2} \geq -1$$
$$-p_{2,h2} - p_{3,h2} \geq -1$$

Using the addition operation on the first group yields the following:

$$p_{1,h1} + p_{1,h2} + p_{2,h1} + p_{2,h2} + p_{3,h1} + p_{3,h2} \geq 3 \tag{A.1}$$

Performing the same operation on the second group yields the following:

$$-2p_{1,h1} - 2p_{2,h1} - 2p_{3,h1} \geq -3$$

This can be integer divided to yield:

$$-p_{1,h1} - p_{2,h1} - p_{3,h1} \geq -1.5$$

Because the right hand side contains a fraction, it needs to be rounded up:

$$-p_{1,h1} - p_{2,h1} - p_{3,h1} \geq -1 \tag{A.2}$$

The same operations can symmetrically be performed for the second hole to yield:

$$-p_{1,h2} - p_{2,h2} - p_{3,h2} \geq -1 \tag{A.3}$$

If we now sum A.1, A.2 and A.3 we arrive at the following contradiction:

$$0p_{1,h1} + 0p_{1,h2} + 0p_{2,h1} + 0p_{2,h2} + 0p_{3,h1} + 0p_{3,h2} \geq 1$$

Because all coefficients on the left hand side are 0, we have arrived at the contradiction $0 \geq 1$; thus the formula is proven to be unsatisfiable.

The power in this proof comes from the integer division operation and how it allows for rounding; if rounding was not performed we would instead have arrived at $0 \geq 0$, which is clearly not contradictory. By comparison, the resolution proof system has no such operation and performing the equivalent steps in the resolution proof system would not arrive at a proof.

A.1.2 Resolution refutation

The proof is displayed in figure A.1.

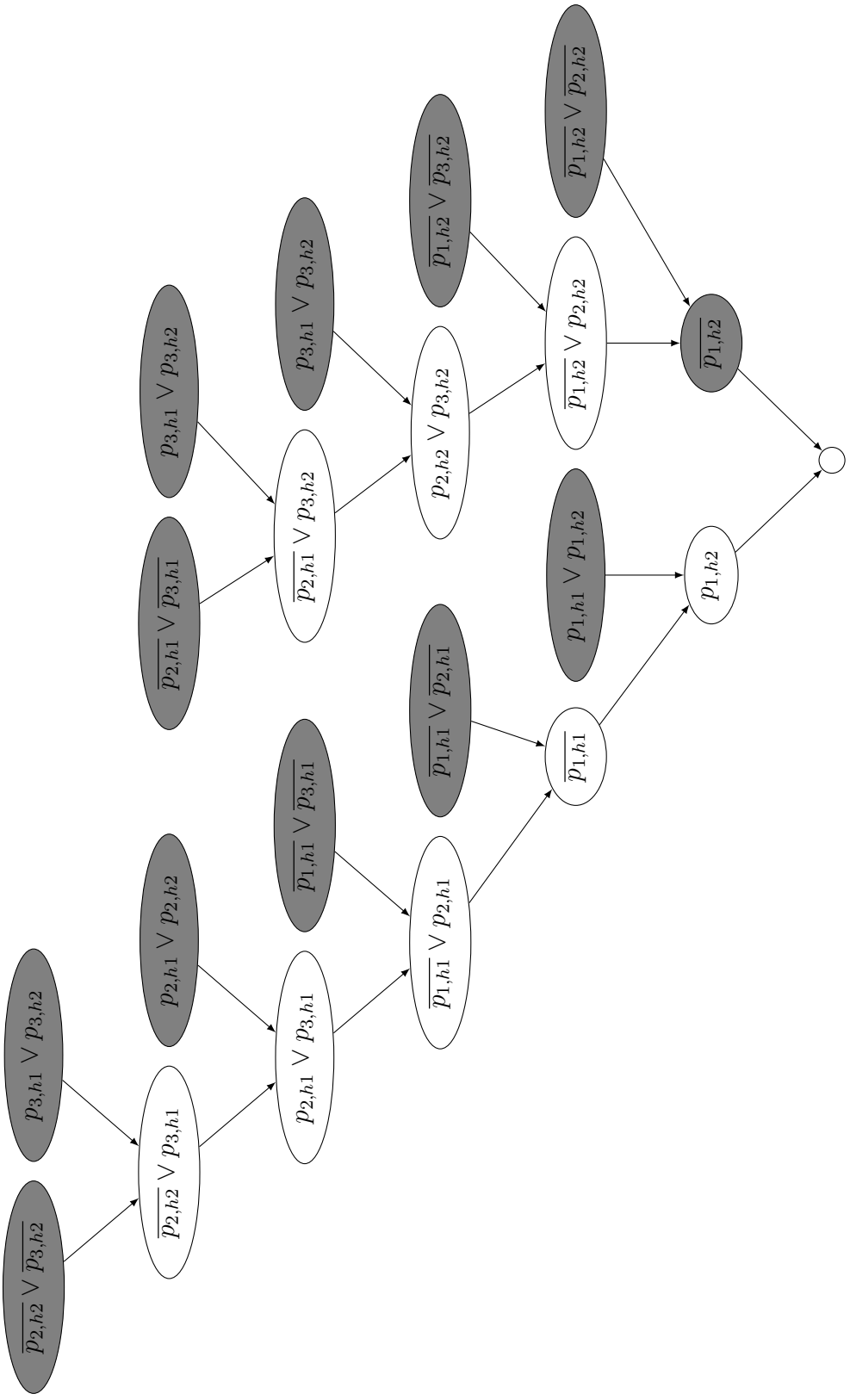

Figure A.1: Resolution refutation of the pigeonhole formula with 3 pigeons and two holes

A.2 Different interpretations of unit elimination

In order to demonstrate the differences between the three interpretations, a moderately complex example must be set up. Consider the following prerequisites:

1. $(x \vee \overline{y})$ and $(\overline{x} \vee \overline{y})$ have been resolved to learn the unit clause $\overline{y}$
2. $(x \vee \overline{z})$ and $(\overline{x} \vee \overline{z})$ have been resolved to learn the unit clause $\overline{z}$

Next, a conflict is encountered resulting in the following resolution steps for a learned clause:

1. $(a \vee y)$ is used, giving the possibility of eliminating y
2. $(c \vee y \vee z)$ is used, giving the possibility of eliminating y and z
3. From this, the clause $a \vee c$ is learned

Furthermore, the following similar steps lead to a second learned clause (with b instead of a:

1. $(b \vee y)$ is used, giving the possibility of eliminating y
2. $(b \vee y \vee z)$ is used, giving the possibility of eliminating y and z
3. From this, the clause $b \vee c$ is learned

With the first interpretation option, where no unit elimination is performed, the results are as shown in figure A.2 (gray background indicates initial clauses from the formula).

Note how no units are eliminated but instead the learned clauses are, in addition to the $\overline{y}$ and $\overline{z}$ clauses, the ones that would be expected except with the literals y and z still there. Furthermore, there four components are perfectly tree-like and contain no regularity violations.

With the second interpretation mode we resolve with unit clauses as soon as units can be eliminated. The results are as shown in figure A.3. The learned clauses (at the bottom) are as would be expected. However, note the clause $\overline{z}$ is reused two times and $\overline{y}$ four times, introducing tree violations. Furthermore, note how y is removed by the

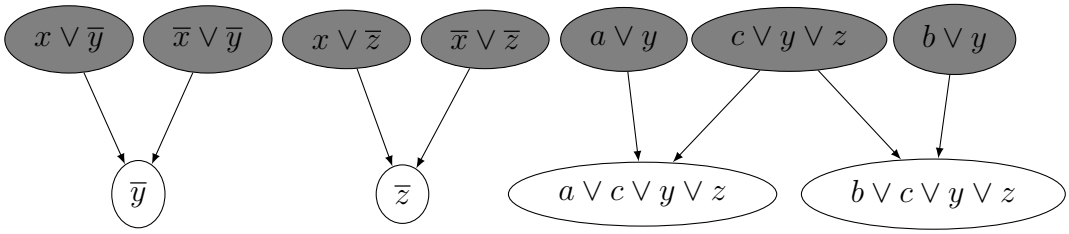

Figure A.2: The four learned clauses when no unit elimination is performed

step leading from $b \vee y$, only to be reintroduced in the next step and subsequently removed again; this is a regularity violation.

Lastly, with the third interpretation mode we remove literals from initial clauses and reuse the results in all places the same unit is eliminated from the same clause. The results are in figure A.4. Note how, compared to figure A.3, we only use $\overline{y}$ three times instead of four. This is because instead of using the clause $c \vee y \vee z$ and then eliminating y and z, we derive only c and use c directly.

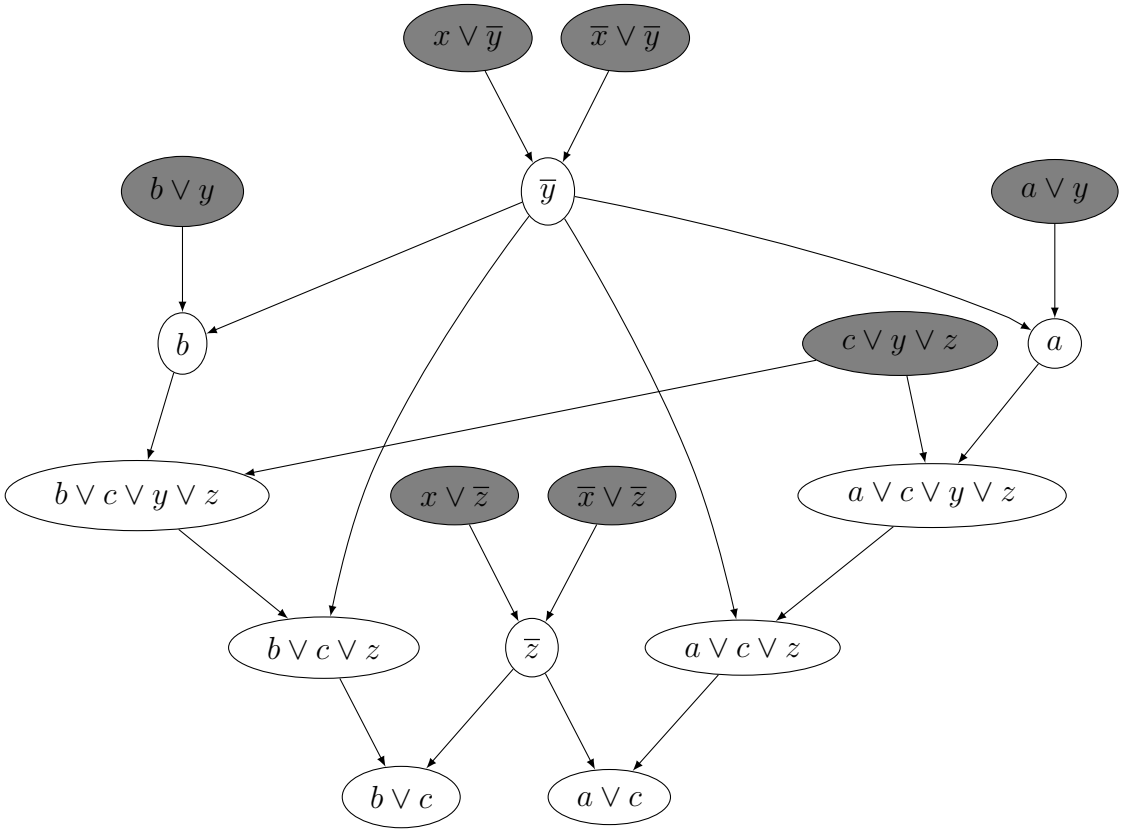

Figure A.3: The four learned clauses when learned units are directly resolved with

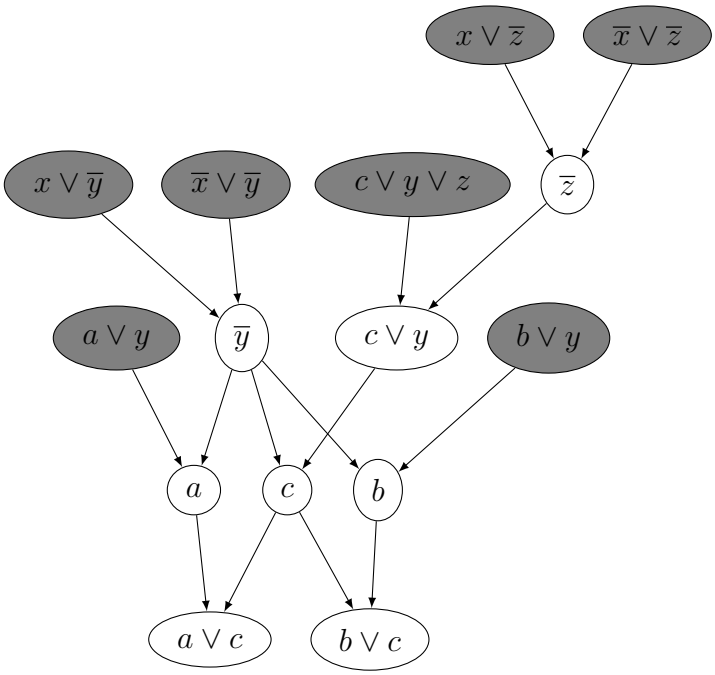

Figure A.4: The four learned clauses when units are eliminated from initial clauses only once

A.3 Additional graphs

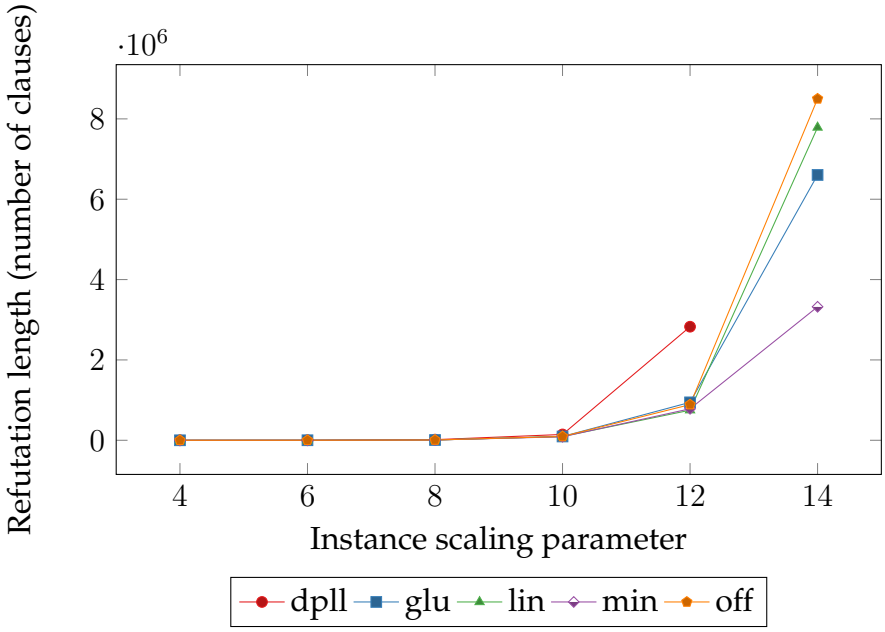

Figure A.5: The effects of clause learning on refutation length, op_partial formulae (recursive minimization, unit elimination mode "1", no restarts)

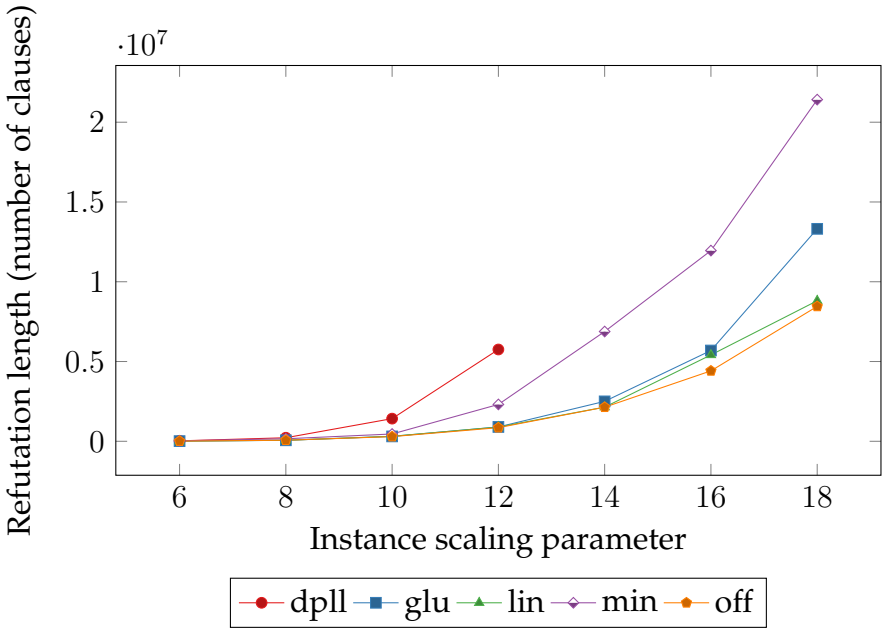

Figure A.6: The effects of clause learning on refutation length, rphp_5 formulae (recursive minimization, unit elimination mode "1", no restarts)

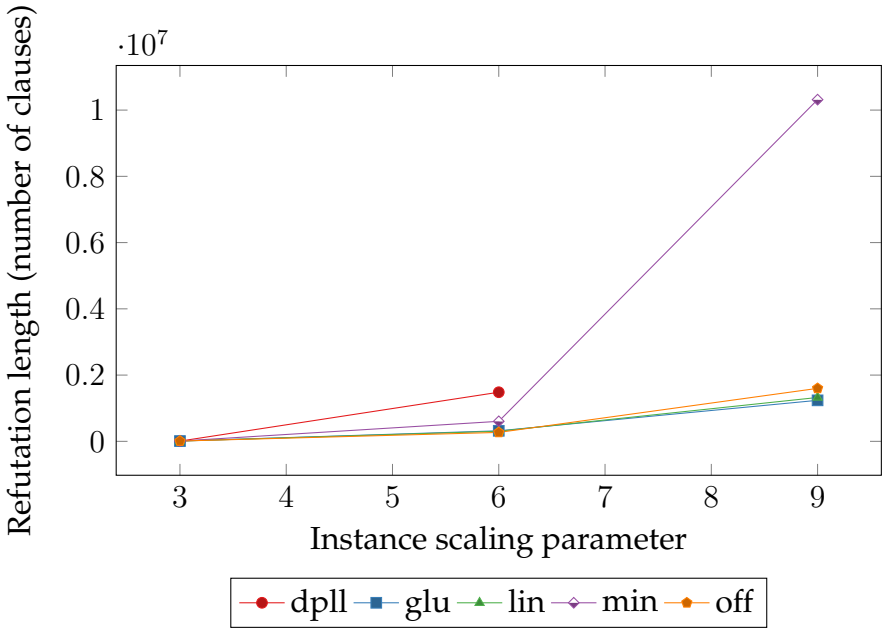

Figure A.7: The effects of clause learning on refutation length, tseitin_diaggrid_3 formulae (recursive minimization, unit elimination mode "1", no restarts)

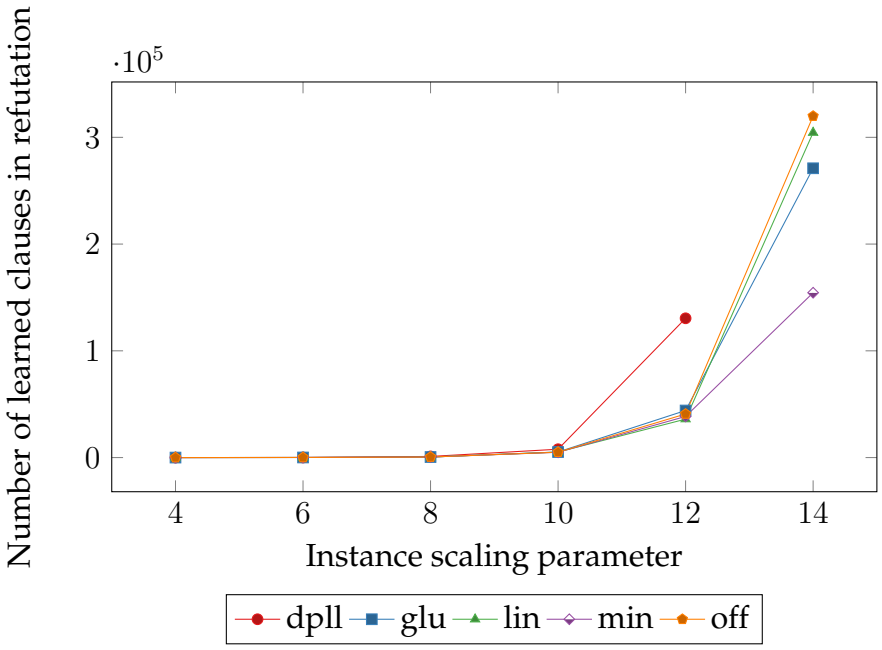

Figure A.8: The effects of clause learning on number of learned clauses in refutation, op_partial formulae (recursive minimization, unit elimination mode "1", no restarts)

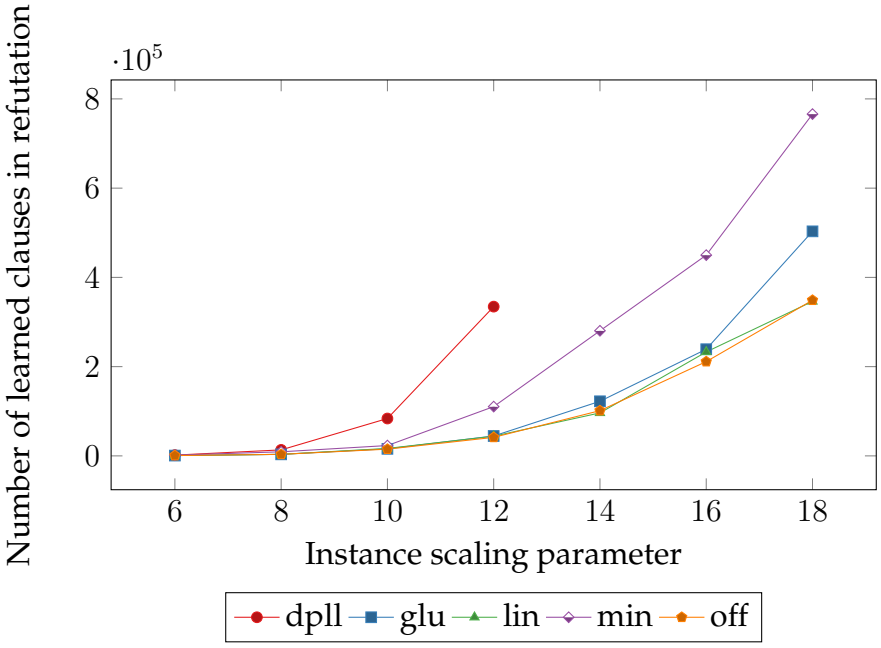

Figure A.9: The effects of clause learning on number of learned clauses in refutation, rphp_5 formulae (recursive minimization, unit elimination mode "1", no restarts)

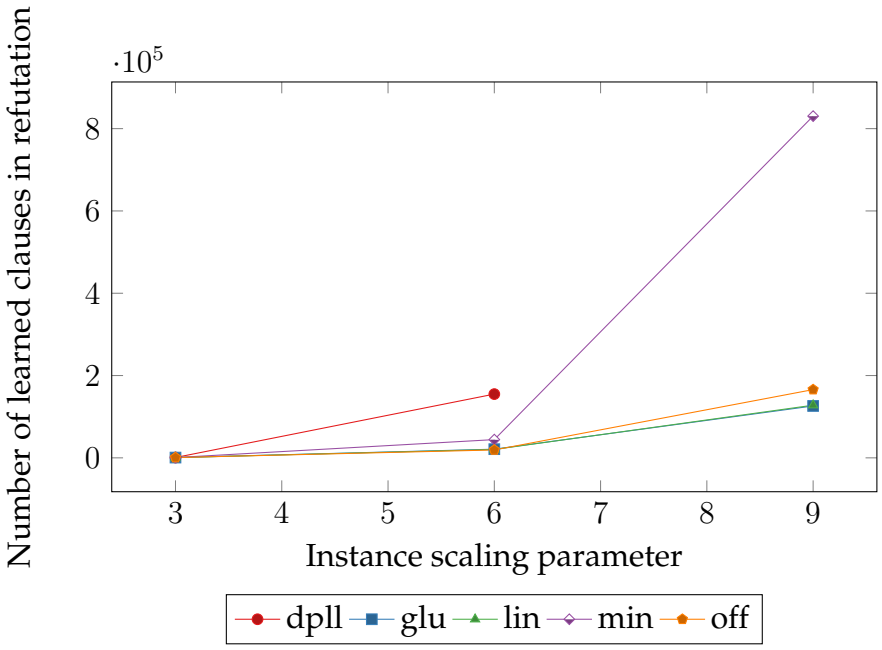

Figure A.10: The effects of clause learning on number of learned clauses in refutation, tseitin_diaggrid_3 formulae (recursive minimization, unit elimination mode "1", no restarts)

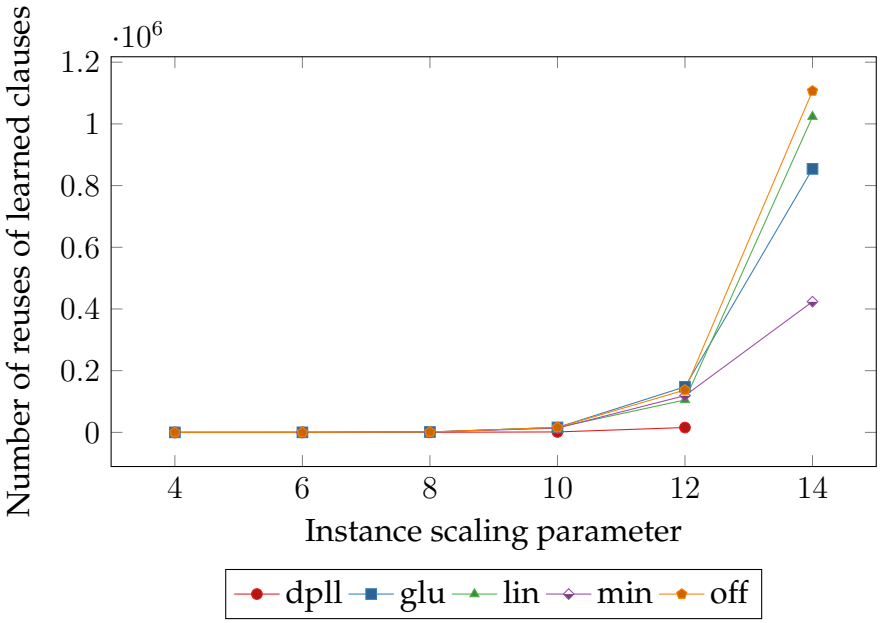

Figure A.11: The effects of clause learning on number of times learned clauses are reused in refutation, op_partial formulae (recursive minimization, unit elimination mode "1", no restarts)

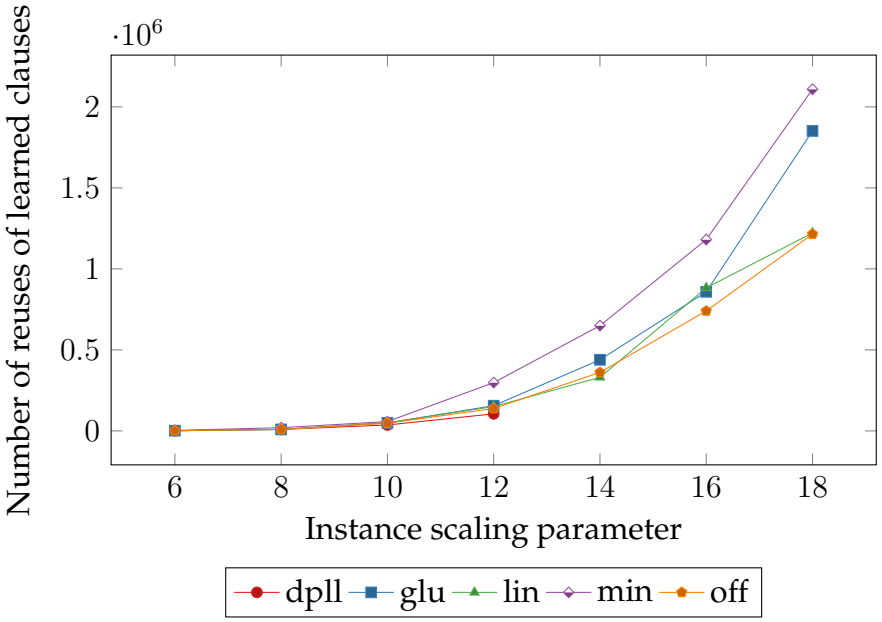

Figure A.12: The effects of clause learning on number of times learned clauses are reused in refutation, rphp_5 formulae (recursive minimization, unit elimination mode "1", no restarts)

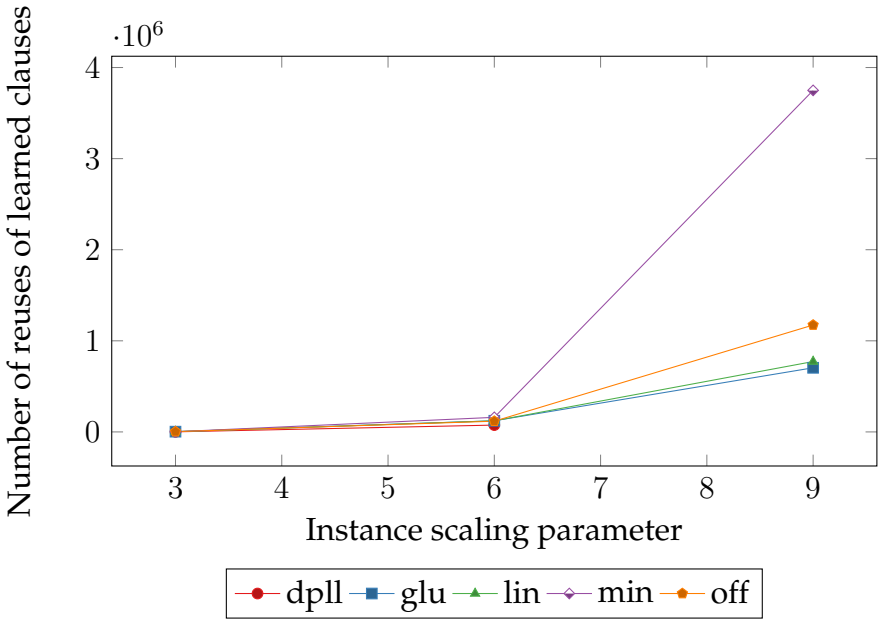

Figure A.13: The effects of clause learning on number of times learned clauses are reused in refutation, tseitin_diaggrid_3 formulae (recursive minimization, unit elimination mode "1", no restarts)

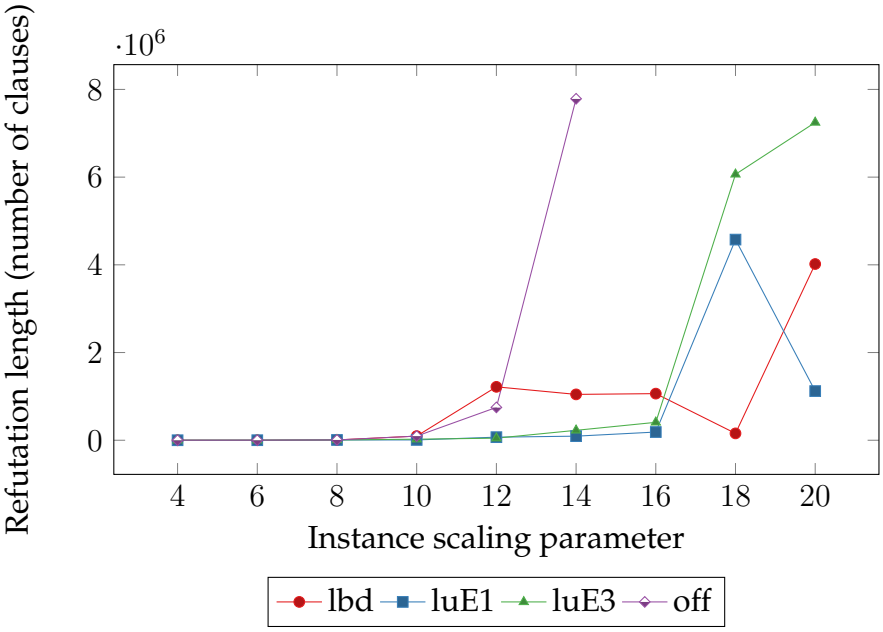

Figure A.14: The effects of restarts on refutation length, op_partial formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

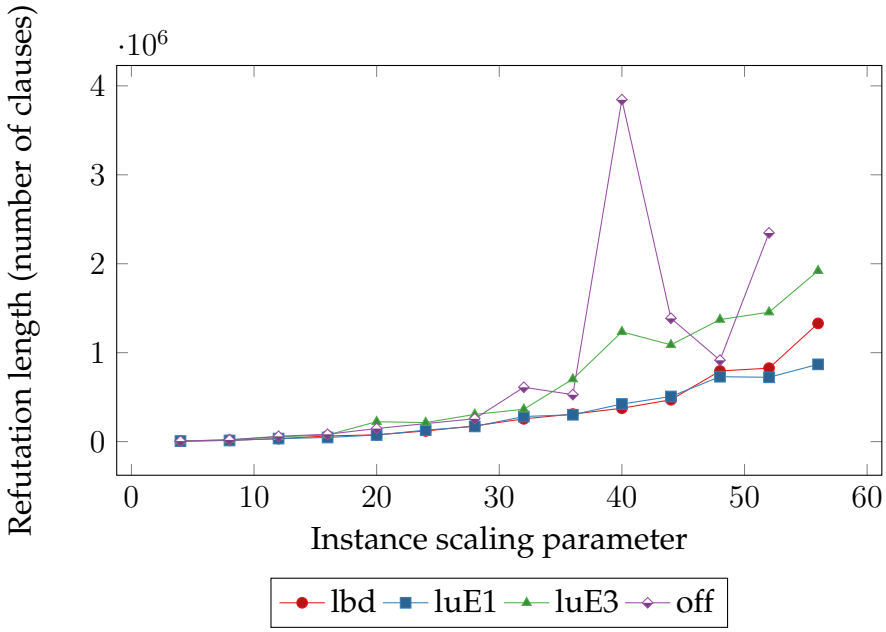

Figure A.15: The effects of restarts on refutation length, peb_pyr_neq3 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

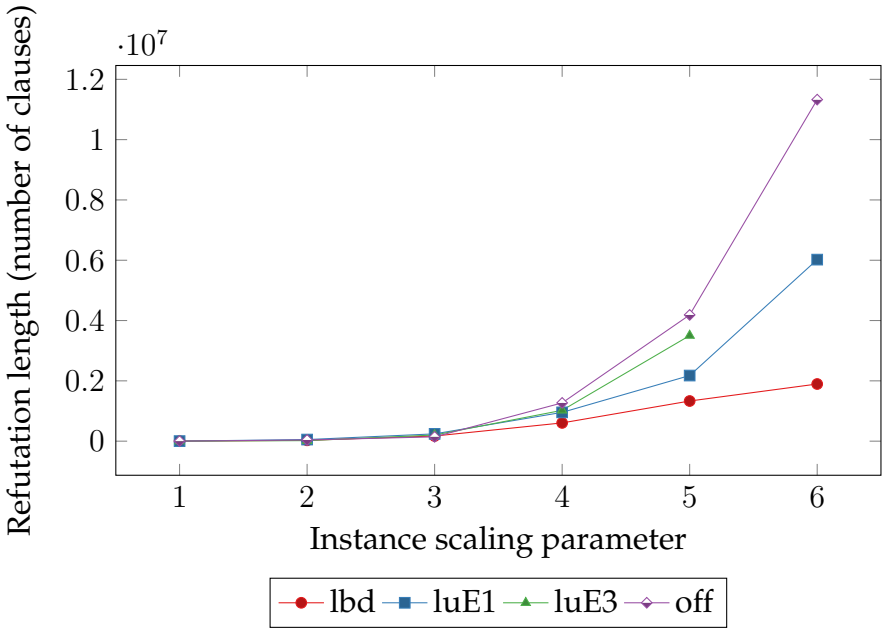

Figure A.16: The effects of restarts on refutation length, peb_pyrofpyr_neq3 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

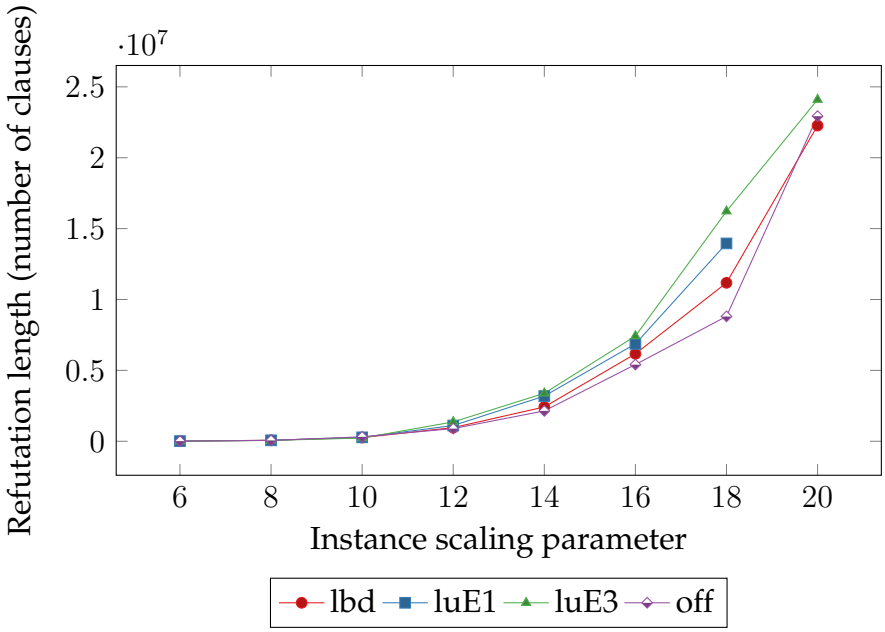

Figure A.17: The effects of restarts on refutation length, rphp_5 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

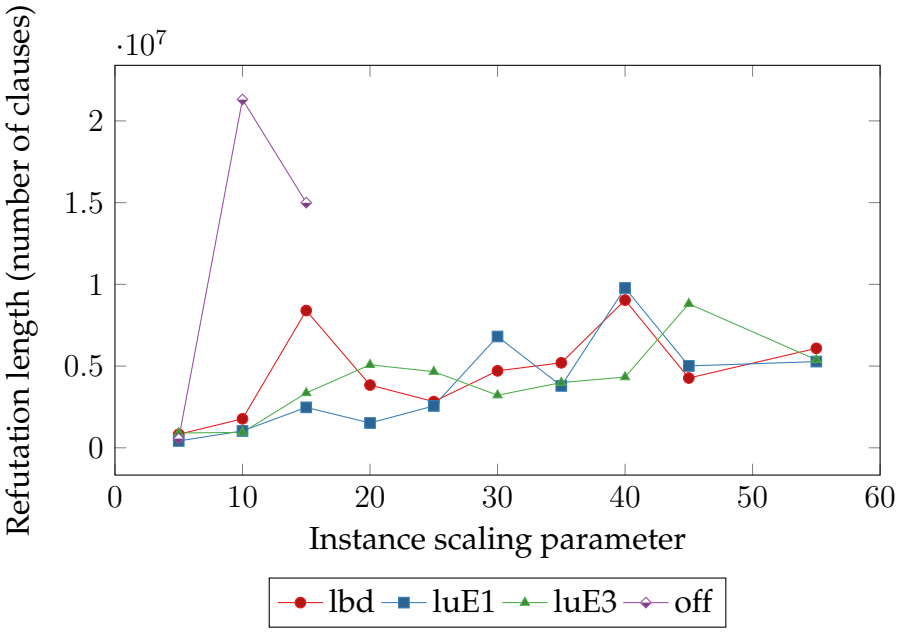

Figure A.18: The effects of restarts on refutation length, tseitin_reggrid_5 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

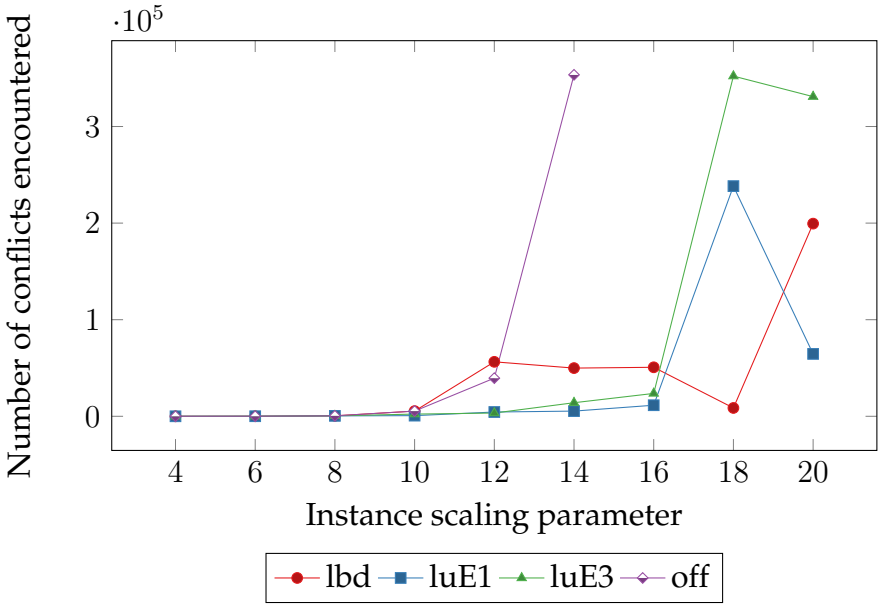

Figure A.19: The effects of restarts on number of conflicts encountered, op_partial formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

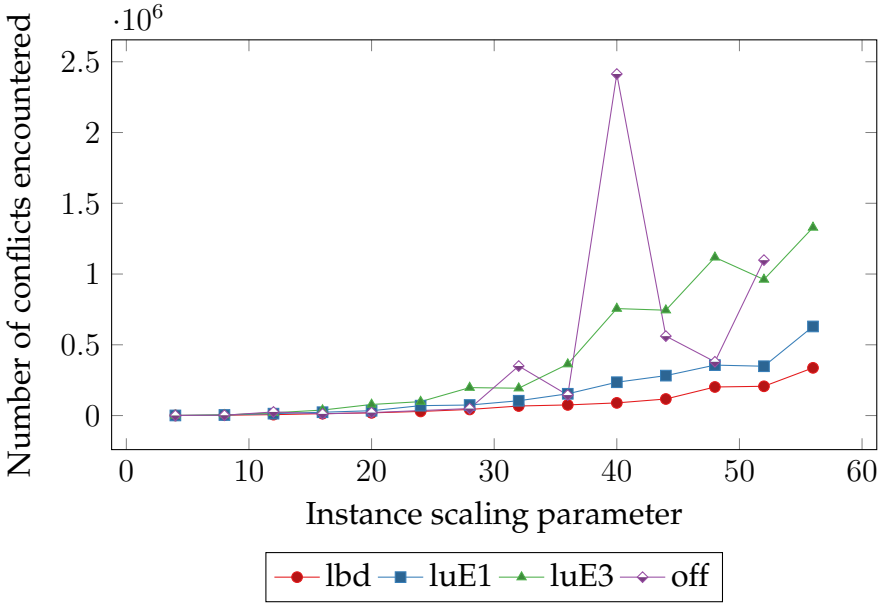

Figure A.20: The effects of restarts on number of conflicts encountered, peb_pyr_neq3 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

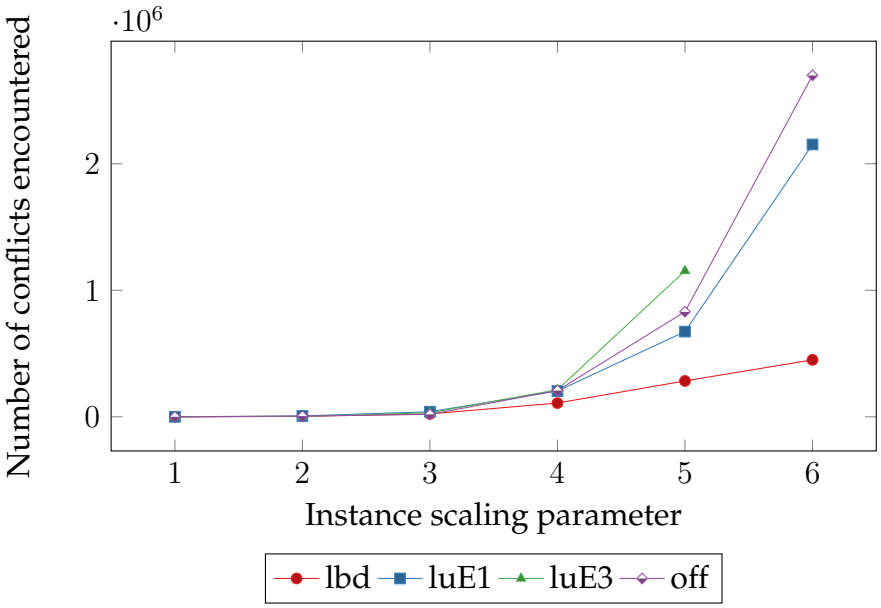

Figure A.21: The effects of restarts on number of conflicts encountered, peb_pyrofpyr_neq3 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

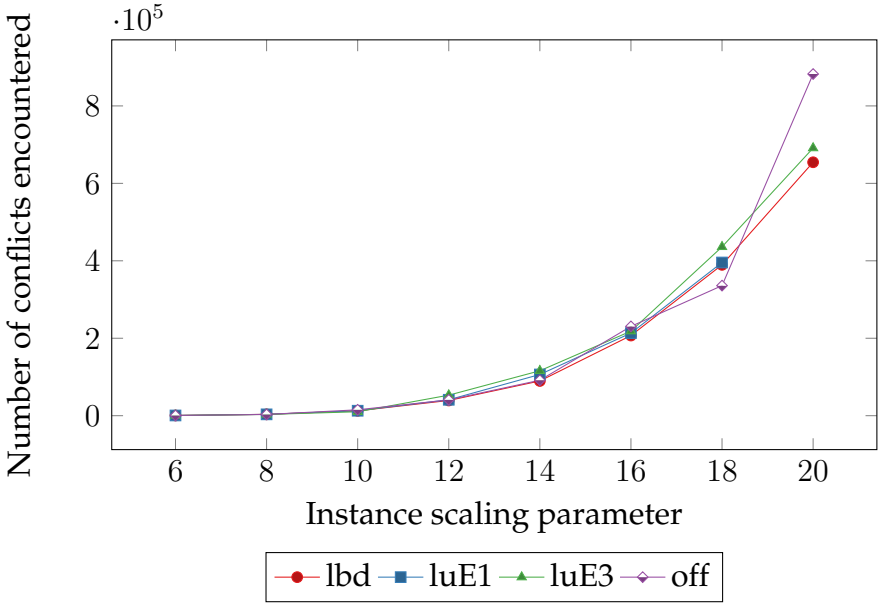

Figure A.22: The effects of restarts on number of conflicts encountered, rphp_5 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

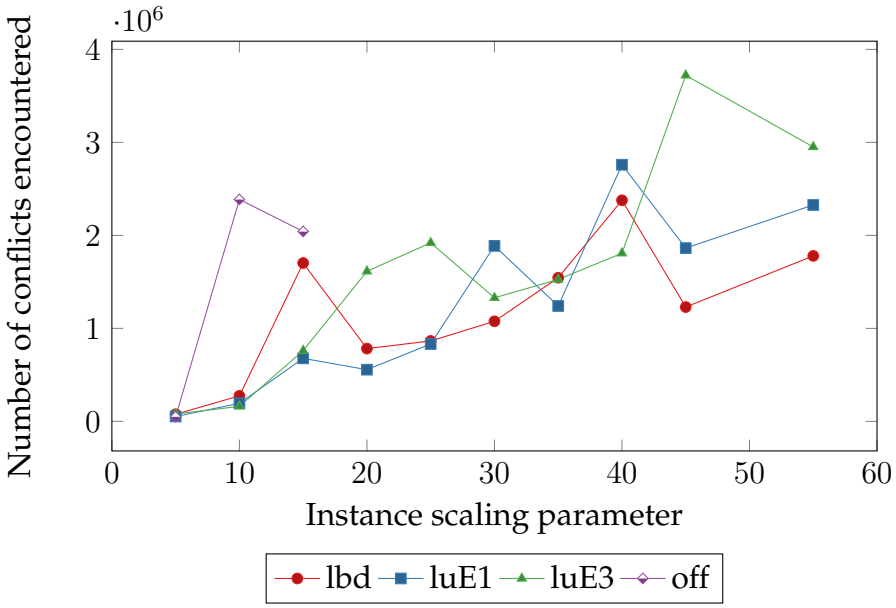

Figure A.23: The effects of restarts on number of conflicts encountered, tseitin_reggrid_5 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

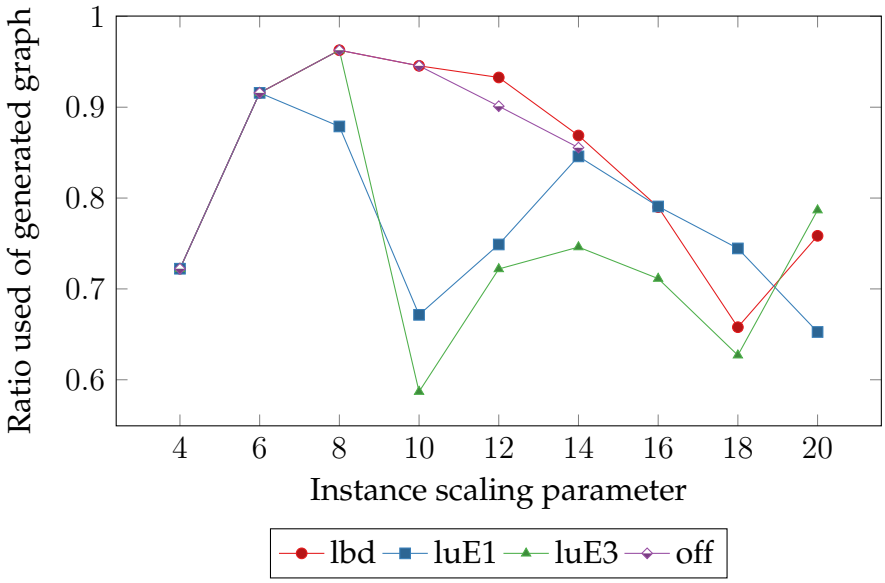

Figure A.24: The effects of restarts on ratio of generated graph being used as proof, op_partial formulae (recursive minimization, unit elimination mode "1", lin clause erasure). The ratio refers to $\frac{\text{Number of vertices in refutation}}{\text{Number of vertices in full graph}}$

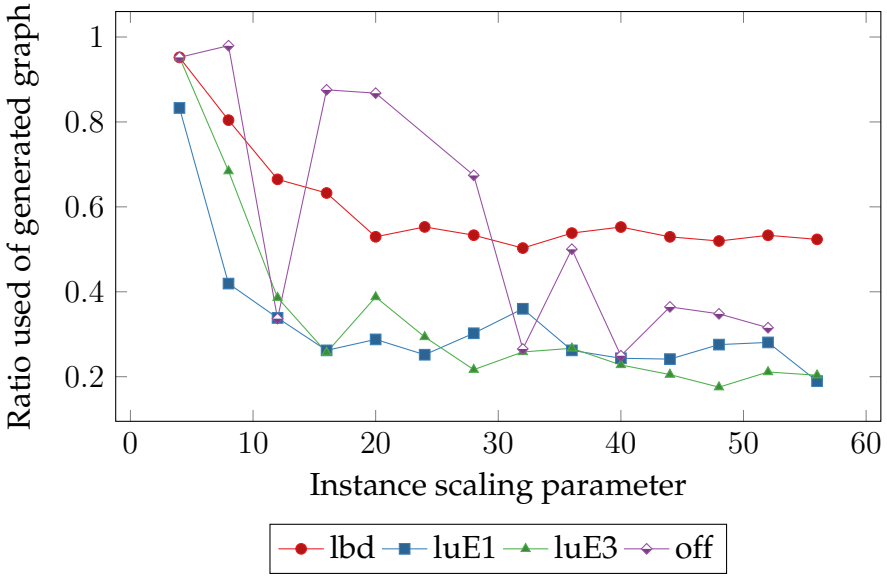

Figure A.25: The effects of restarts on ratio of generated graph being used as proof, peb_pyr_neq3 formulae (recursive minimization, unit elimination mode "1", lin clause erasure). The ratio refers to $\frac{\text{Number of vertices in refutation}}{\text{Number of vertices in full graph}}$

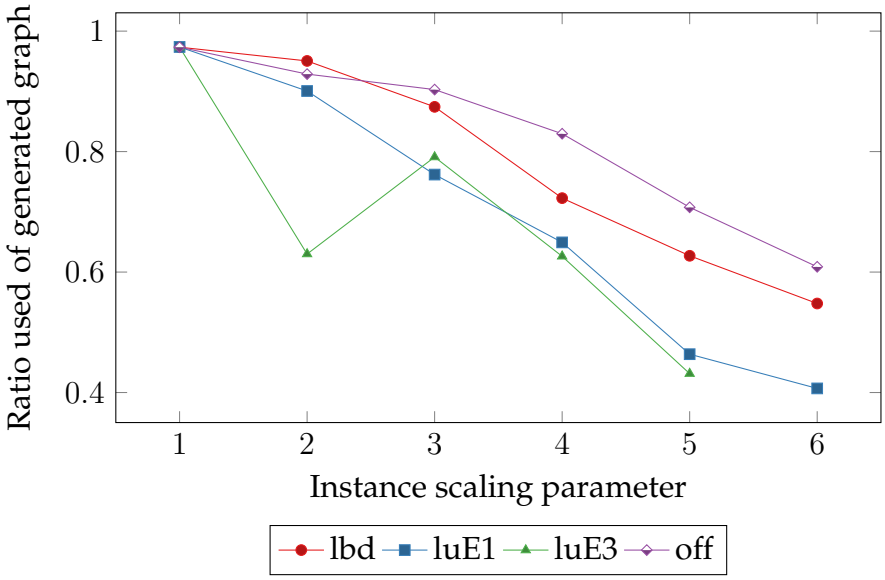

Figure A.26: The effects of restarts on ratio of generated graph being used as proof, peb_pyrofpyr_neq3 formulae (recursive minimization, unit elimination mode "1", lin clause erasure). The ratio refers to $\frac{\text{Number of vertices in refutation}}{\text{Number of vertices in full graph}}$

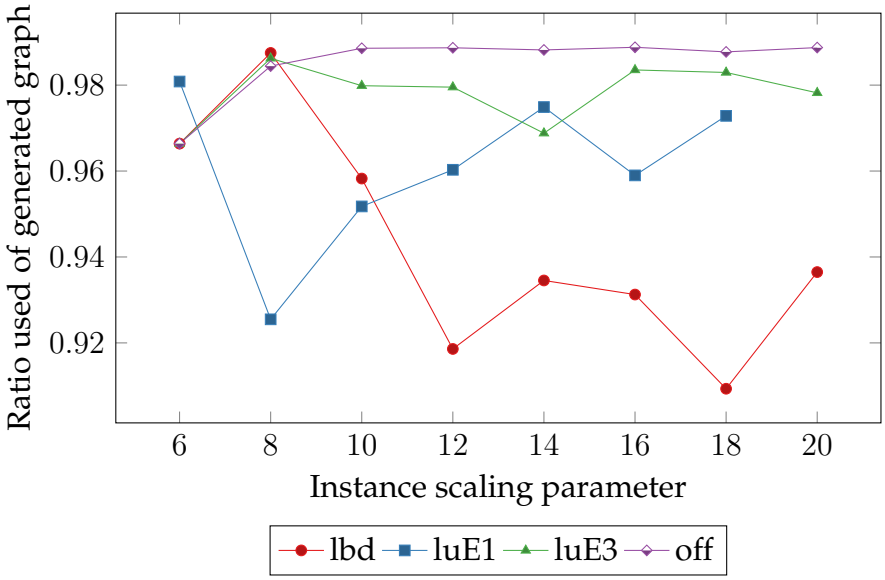

Figure A.27: The effects of restarts on ratio of generated graph being used as proof, rphp_5 formulae (recursive minimization, unit elimination mode "1", lin clause erasure). The ratio refers to $\frac{\text{Number of vertices in refutation}}{\text{Number of vertices in full graph}}$

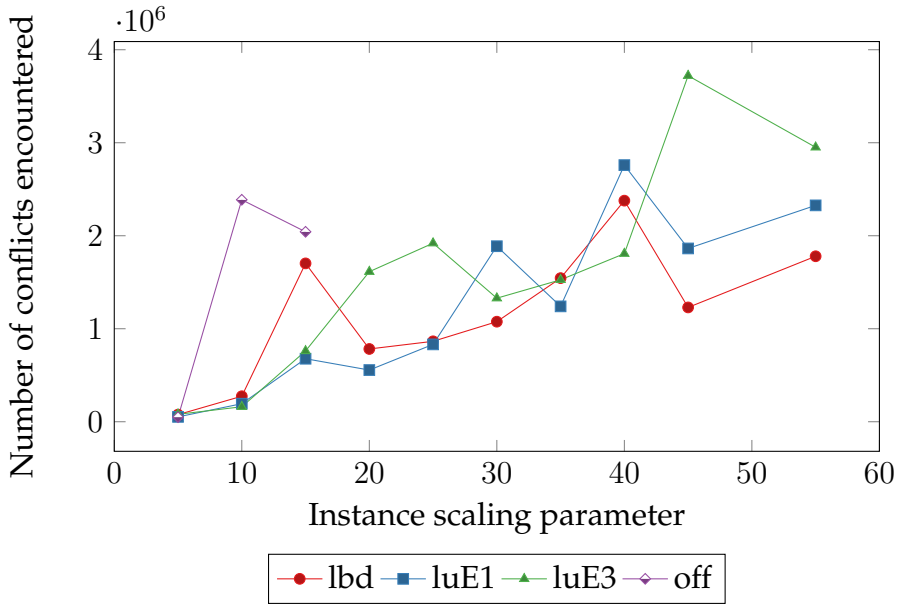

Figure A.28: The effects of restarts on number of conflicts encountered, tseitin_reggrid_5 formulae (recursive minimization, unit elimination mode "1", lin clause erasure)

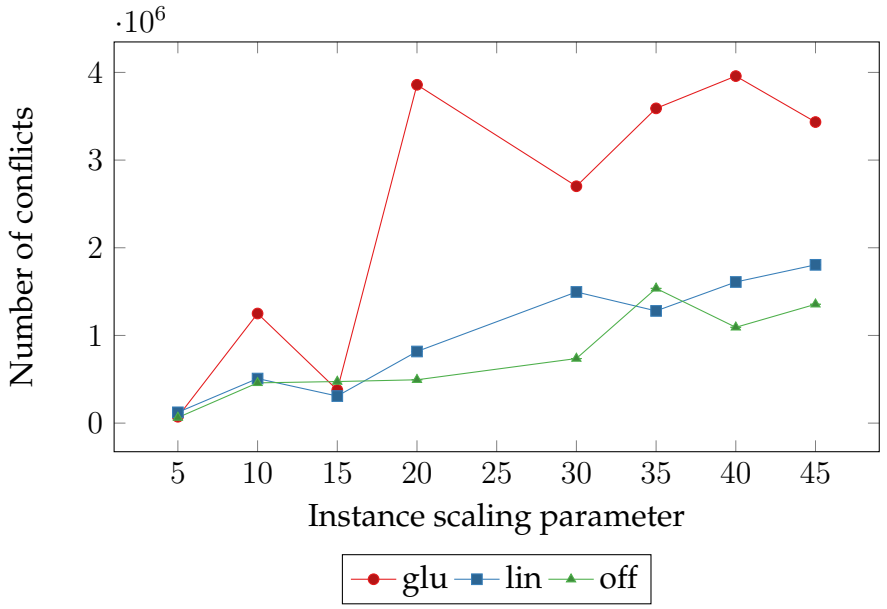

Figure A.29: The effects of different clause erasure policies on number of conflicts, tseitin_reggrid_5 formulae (lbd restarts, no minimization)

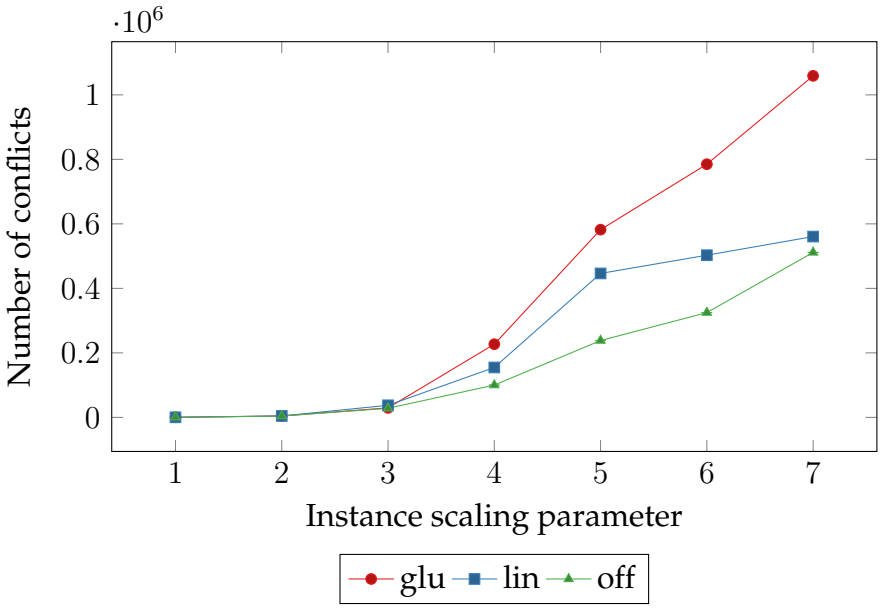

Figure A.30: The effects of different clause erasure policies on number of conflicts, peb_pyrofpyr_neq3 formulae (lbd restarts, no minimization)

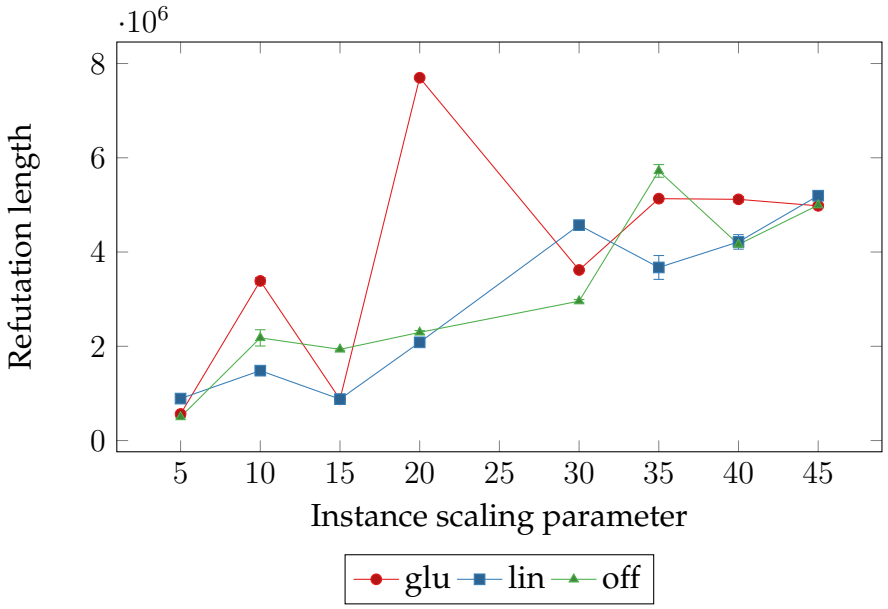

Figure A.31: The effects of different clause erasure policies on refutation length for tseitin_reggrid_5 formulae (lbd restarts, no minimization

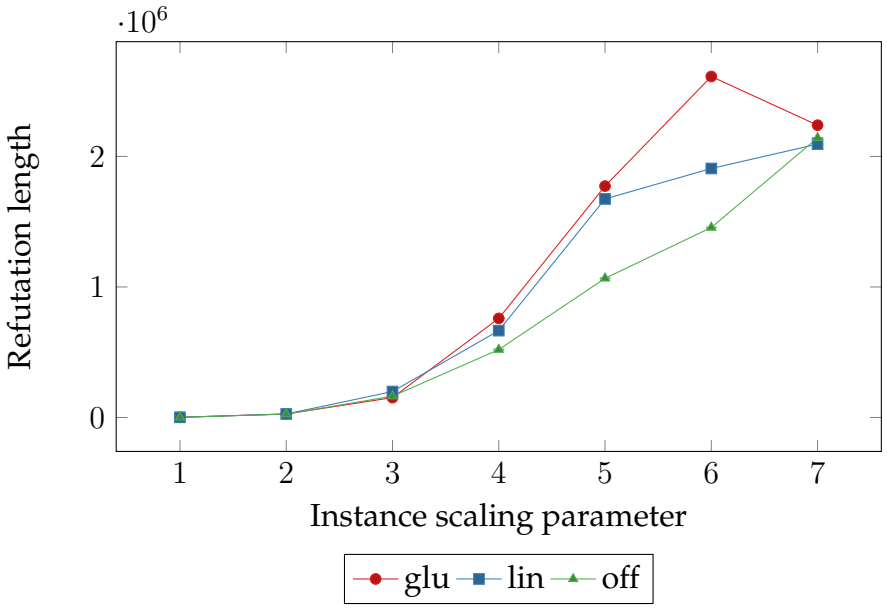

Figure A.32: The effects of different clause erasure policies on refutation length for peb_pyrofpyr_neq3 formulae (lbd restarts, no minimization

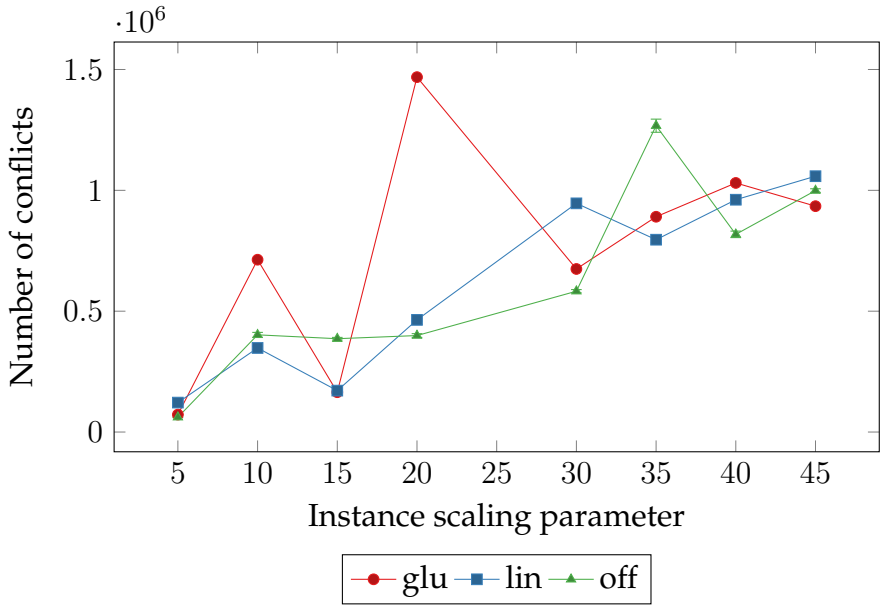

Figure A.33: The effects of different clause erasure policies on number of learned clauses in refutation, tseitin_reggrid_5 formulae (lbd restarts, no minimization)

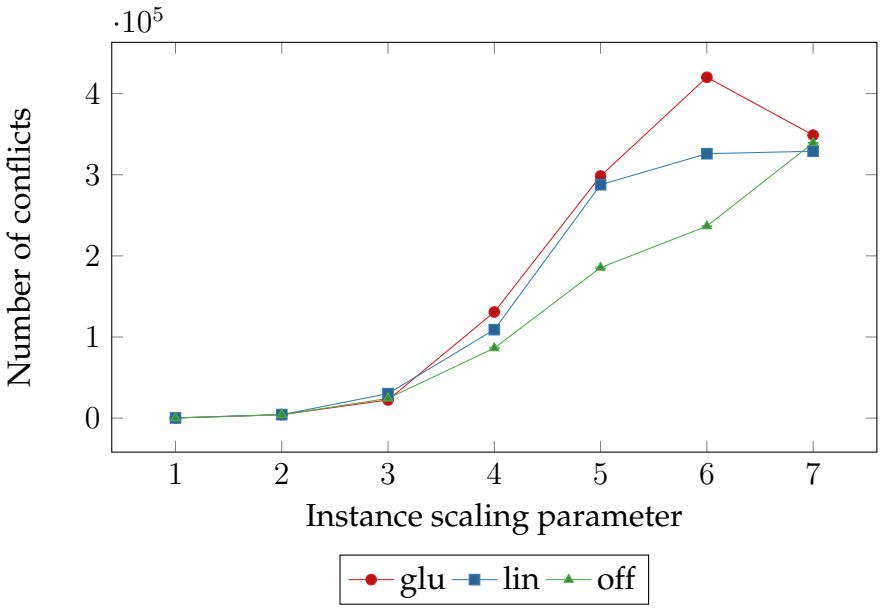

Figure A.34: The effects of different clause erasure policies on number of learned clauses in refutation, peb_pyrofpyr_neq3 formulae (lbd restarts, no minimization)

A.4 Instructions from solver

Number of variables The number of distinct variables that the formula has been found to contain.

Initial clauses A list of initial clauses that the solver has received, along with the unique ID number given to the clause.

Decision The assigned variable and its value every time a decision is made.

Propagation The propagated variable, its value and the ID of the clause that caused the propagation each time a unit propagation occurs.

Propagation from learned unit As a special case, the propagated variable whenever a unit propagation was caused by the solver learning a unit clause. This needs to be a separate instruction from the regular propagation above as Minisat does not assign an ID to learned unit clauses.

Conflict analysis steps When a conflict occurs, the clauses used in the resolution steps that arrives at the learned clause.

Eliminated units When a learned clause is built, Minisat ignores literals that correspond to variables assigned at decision level 0; this instruction lists these at the point they are eliminated (interleaved with the conflict analysis steps).

Learned clause When a clause is learned, its assigned clause ID is printed along with the contents of the clause. The latter is used to verify that the solver and the analysis program have arrived at the same clause.

Learned unit clause When a unit clause is learned, Minisat does not assign this an ID but will instead directly assign the variable at level 0. For this reason, the instruction above can not be used and this separate instruction is required instead.

Minimization When clause minimization is used in the regular (non-recursive) mode, the literals that are removed are printed. It is then up to the analysis program to generate the appropriate resolution steps.

Recursive minimization This is the same as the one above but for the recursive mode.

Backtrack When backtracking, the level that is backtracked to.

Restart When restarting.

Remove clause Minisat will remove learned clauses if this option is chosen. This instruction is then used to list the IDs of the clauses that are removed.

Moved clause Minisat will periodically move clauses to different IDs in order to more efficiently use memory (if learned clauses have been removed). This instructions lists the mappings between new and old IDs so that the analysis program can update its database accordingly.

Final conflict When a final conflict occurs and the formula is determined to be unsatisfiable, this instruction is used to signal the ID of the clause that the conflict occurred in.

TRITA -EECS-EX-2019:96