Diese Arbeit wurde vorgelegt am LuFG Theorie hybrider Systeme

BACHELORARBEIT

COMPUTING MINIMAL INFEASIBLE SUBSETS FOR THE CYLINDRICAL ALGEBRAIC DECOMPOSITION

Wanja Hentze

Supervisors:
Prof. Dr. Erika Ábrahám
Prof. Dr. Jürgen Giesl

Advisor:
Gereon Kremer

27. März 2017

Abstract

In the field of satisfiability modulo theories (SMT), specifically for theories involving nonlinear constraints over real-valued variables, the cylindrical algebraic decomposition (CAD) is a fundamental algorithm. To use it efficiently in a lazy SMT solver, the CAD needs to provide the SAT solver with Boolean lemmas about its constraints. In this paper, we approach the problem of generating a specific kind of lemmas, namely infeasible subsets, by casting it as an instance of the set cover problem (SCP). A novel algorithm is introduced, comprising a preconditioning step and a hybrid approach switching between the greedy approximative solution to the SCP and an optimal exhaustive solution based on problem size. We embed an implementation of this algorithm into the SMT-RAT framework and judge its advantages over the greedy algorithm by comparing the two on a number of real-world and synthetic examples. We demonstrate that our algorithm allows SMT-RAT to reach a solution after significantly fewer CAD calls in a number of cases while incurring a negligible increase in cost over the greedy algorithm. However, we find that this only translates to a very slight increase in overall performance due to its benefits being limited to a very specific set of QF_NRA problems

Eidesstattliche Versicherung

Name, Vorname	Matrikelnummer (freiwillige Angabe)

Ich versichere hiermit an Eides Statt, dass ich die vorliegende Arbeit/Bachelorarbeit/ Masterarbeit* mit dem Titel

selbständig und ohne unzulässige fremde Hilfe erbracht habe. Ich habe keine anderen als die angegebenen Quellen und Hilfsmittel benutzt. Für den Fall, dass die Arbeit zusätzlich auf einem Datenträger eingereicht wird, erkläre ich, dass die schriftliche und die elektronische Form vollständig übereinstimmen. Die Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Ort, Datum	Unterschrift

*Nichtzutreffendes bitte streichen

Belehrung:

§ 156 StGB: Falsche Versicherung an Eides Statt

Wer vor einer zur Abnahme einer Versicherung an Eides Statt zuständigen Behörde eine solche Versicherung falsch abgibt oder unter Berufung auf eine solche Versicherung falsch aussagt, wird mit Freiheitsstrafe bis zu drei Jahren oder mit Geldstrafe bestraft.

§ 161 StGB: Fahrlässiger Falscheid; fahrlässige falsche Versicherung an Eides Statt

(1) Wenn eine der in den §§ 154 bis 156 bezeichneten Handlungen aus Fahrlässigkeit begangen worden ist, so tritt Freiheitsstrafe bis zu einem Jahr oder Geldstrafe ein.

(2) Straflosigkeit tritt ein, wenn der Täter die falsche Angabe rechtzeitig berichtigt. Die Vorschriften des § 158 Abs. 2 und 3 gelten entsprechend.

Die vorstehende Belehrung habe ich zur Kenntnis genommen:

Ort, Datum	Unterschrift

Acknowledgements

I would like to use this opportunity to express my gratitude towards all of the people that made it possible for me to write this thesis. First of all, I would like to thank everyone at the Computer Science 2 chair of the RWTH Aachen University. In particular, I thank my primary supervisor Prof. Dr. Erika Ábrahám for allowing me to write this thesis as part of the Theory of Hybrid Systems research group as well as Prof. Dr. Jürgen Giesl for volunteering as a secondary supervisor. In addition, I want to thank my advisor Gereon Kremer for tierelessly assisting me with both technical and academic advice at many points during my work. Finally, I wish to express my deep gratitude to my friends and family for their unrelenting moral support and to Joëlle for never leaving my side, reading through my work many times and helping me improve my writing.

Contents

Chapter 1

Introduction

Satisfiability modulo theories (SMT) is the problem class concerned with deciding the satisfiability of Boolean formulae over constraints from some algebraic theory. For its users, an SMT solver essentially combines the utility of a boolean satisfiability (SAT) solver and a computer algebra system. SMT solvers have found numerous applications in recent years, especially in formal verification of both software and hardware system designs.

Many SMT solvers today employ what is known as the lazy SMT algorithm, described in [Seb07]. A lazy SMT solver consists of a classical SAT solver and a theory solver working in an alternating fashion. The SAT solver only considers the Boolean structure of the formula, abstracting away theory constraints as Boolean literals, and tries to find satisfying assignments for it. The theory solver, in turn, is only concerned with the algebraic feasibility of those assignments and can ignore the Boolean structure of the greater problem at hand.

Although the SAT solver can never reason about anything but Boolean logic, it can still learn certain truths from the theory solver as long as they are encoded in purely Boolean logic. These pieces of Boolean information, called lemmas, are crucial to the efficiency of the SMT procedure. Without them, the SAT solver remains ignorant about any underlying relationships between its theory constraints. One kind of lemma that is particularly useful is the infeasible subset, as it tells the SAT solver a more precise reason why an assignment is inconsistent with the theory.

Lazy SMT is especially relevant to theories that allow for variables over an infinite domain or admit a complex algebraic structure. A theory with both of these properties is the logic of quantifier-free nonlinear real arithmetic (QF_NRA), which deals with equalities and inequalities involving polynomials over $\mathbb{R}^n$. The decidability of such a logic is implied by the results of Tarski and Seidenberg in [Tar51, Sei54]. However, solving any but the most trivial QF_NRA problems using the method presented in their proofs is computationally intractable. The cylindrical algebraic decomposition (CAD) is an improvement on that method which is intended specifically to be feasibly computable. This thesis concerns itself specifically with the problem of computing infeasible subsets using the information obtained from a CAD.

The modular SMT-solving framework SMT-RAT, described in detail in [CKJ$^+$15], will serve as a working basis for the implementation of the ideas presented here. It provides a number of modules representing either partial or complete solvers for various theories, including an implementation of the CAD. For computing infeasible subsets from a CAD, it currently employs a simple greedy algorithm.

In this thesis, we argue that this algorithm can be improved upon in several ways. Firstly, we introduce a preconditioning technique that solves many instances of the problem outright and simplifies most other instances to a small and difficult to solve core. Secondly, we hypothesize that, after preconditioning, most problem instances are too small to neccessitate the use of a polynomial-time approximative solution such as the greedy algorithm. We propose a hybrid algorithm that finds the optimal solution for small problem instances and falls back to a weighted version of the greedy algorithm for larger instances. In addition, we discuss how constraints should be weighed in this algorithm. Finally, we show how this hybrid algorithm can be adapted to produce not only one, but several infeasible subsets.

In the following chapter, the theoretical concepts and the terminology relevant for this thesis are introduced. Subsequently, in Chapter 2, we give a summary of the literature surrounding the topic of infeasible subsets, giving particular attention to one publication which approaches the issue from a linear programming viewpoint. In Chapter 3, the set cover problem (SCP) is introduced and its application to the topic is described. We then identify several theoretical weaknesses of the naive greedy solution to the SCP when it is used for finding infeasible subsets. In addition, we use data gathered from concrete solver runs to show how these weaknesses manifest in realistic scenarios. In Chapter 4, we present our novel algorithm and argue how it addresses these weaknesses. Chapter 5 briefly describes the implementation of the algorithm in the SMT-RAT toolbox and presents experimental results comparing its performance with that of the greedy algorithm. Finally, we give our conclusion and discuss possible further improvements upon the algorithm in Chapter 6.

Chapter 2

Background

To understand how infeasible subsets can speed up the lazy SMT procedure, some understanding of the inner workings of both modern SAT and SMT solvers is needed. In the following, SAT and SMT are introduced formally as problem classes. A brief overview of the central ideas in SAT solving, the lazy SMT algorithm and the CAD is given. Based on this, the idea of infeasible subsets is motivated and introduced formally. Finally, the existing approaches for generating infeasible subsets are reviewed.

In this paper, $\mathbb{R}$ is used to represent the set of real numbers, $\mathbb{B} = \{\mathit{true}, \mathit{false}\}$ is the set of Boolean truth values and $\mathbb{Z}$ is the set of integer numbers.

2.1 SAT Solving

SAT is the class of problems concerned with the satisfiability of boolean formulae. More precisely:

Definition 2.1.1. *(SAT Problem) Given a Boolean formula φ over a set of variables $X = x_1, \dots, x_n$, is there an assignment $X \to \mathbb{B}$ of truth values to these variables such that φ evaluates to true?*

In some cases, it is simpler to consider only formulae that are in the conjunctive normal form (CNF). A formula is said to be in CNF if it is of the form $(l_{1,1} \vee_{1,2} \vee \dots) \wedge (l_{2,1} \vee l_{2,2}) \wedge \dots$, where every $l_{i,j}$ is a *literal*, i.e. either a variable or the negation of a variable from X. Terms that are disjunctions of literals, such as $(l_{1,1} \vee {}_{1,2} \vee l_{1,3})$, are called *clauses*. A Boolean formula in CNF is therefore a conjunction of clauses. While all Boolean formulae can be converted to an equivalent Boolean formula in CNF, this transformation will in some cases result in an exponential blowup of formula size. However, for the SAT problem, a fully equivalent formula is not neeeded. In [Tse68], Tseitin presents a way to transform any Boolean formula into a SAT-equivalent one that is at most a constant factor larger than the initial one. This transformation can be computed in polynomial time, so it represents a reduction of SAT to CNF-SAT, the problem class concerned with deciding the satisfiability of a Boolean formula in CNF.

For a long time, SAT has been known to be NP-complete. This is known as the Cook-Levin theorem[1]. Therefore, all currently known general solutions exhibit at least exponential time complexity. However, it is possible to formulate algorithms that, while still exponential in the worst case, outperform the naive guessing approach by large margins on many real world problems. One of the simplest improvements over that brute-force approach is the backtracking SAT algorithm, which goes as follows: Pick any variable from the formula and guess a Boolean value for it. Then, replace every occurence of that variable by the guessed value and simplify the formula according to the rules of Boolean algebra. Repeat this step until the formula can be simplified to either *true* or *false*. If it simplifies to *true*, the current set of guesses is an assignment that satisfies the formula, so the algorithm returns *sat*. If it simplifies to *false*, backtrack to the last guess that has not already been inverted and invert it. If no more guesses can be inverted, return *unsat*.

An important milestone in the field of SAT solving was the work of Davis, Putnam, Logemann and Loveland in [DP60, DLL62]. Their approach, now known as the DPLL algorithm, still forms the basis of most SAT solvers today. It introduced two key ideas that set it apart from a simple backtracking algorithm: *unit propagation* and *pure literal elimination*. Unit propagation is concerned with *unit clauses*, i.e. clauses containing only a single literal. These clauses can only evaluate to *true* if that literal evaluates to *true*. Therefore, the variable involved in that literal has to be assigned *true* if it is a positive literal and *false* if it is a negative literal. Pure literal elimination is concerned with variables that appear either only in positive or only in negative literals. Variables that only appear in positive literals can always be assigned *true*, and variables that only appear in negative literals can always be assigned *false* without affecting the satisfiability of the formula. Applying unit propagation and pure literal elimination might lead to additional unit clauses and pure literals, so these steps can be repeated until no more unit clauses or pure literals remain. By applying these two steps after every decision in the backtracking procedure, the depth of the backtracking search tree can be reduced substantially.

There is another essential SAT algorithm called Conflict-Driven Clause Learning (CDCL). It expands upon DPLL by looking more closely at the conflicts it comes across, allowing it to backtrack several levels at a time to the last decision that impacted the conflicting clause. A more detailed description is given by Biere et al. in [BHvMW09]. CDCL is implemented in the popular SAT solver MiniSAT[ES03], which SMT-RAT's SAT solver is based on.

2.2 SMT Solving

The SMT problem, or Satisfiability Modulo Theories, is a generalisation of the SAT problem. Instead of boolean variables, the atoms comprising the formulae are constraints from a certain theory over a set of theory-variables. In the most general sense, a theory $\mathcal{T}$ is characterized by the its universe $U(\mathcal{T})$, a set of operators and relations specifying it algebraic structure and the set of $\mathcal{T}$-constraints that it allows. A $\mathcal{T}$-constraint is a formula connecting one or several variables and constants using the operators and relations from $\mathcal{T}$ that evaluates to either *true* or *false* for every assignment of $\mathcal{T}$-values to its variables. In other words, a $\mathcal{T}$-constraint can be thought of as defining a function $U(\mathcal{T}) \to \mathbb{B}$. Using this terminology, the SMT problem can be defined as follows:

Definition 2.2.1. *(SMT Problem) Given a Boolean formula φ over a set of $\mathcal{T}$-constraints $C = c_1, \ldots, c_n$ that involve $\mathcal{T}$-variables from a set $X = x_1, \ldots, x_m$, is*

[1]The theorem is attributed to both Stephen Cook and Leonid Levin, who discovered this property of SAT independently of each other.[Coo71, Lev73]

there an assignment $X \to U(\mathcal{T})$ of values to these variables such that φ evaluates to true?

Conventionally, the class of SMT problems over a theory $\mathcal{T}$ is referred to as SMT($\mathcal{T}$). For example, the class of SMT problems over the theory of quantifier-free linear integer arithmetic (QF_LIA) is called SMT(QF_LIA). The universe $U(QF\_LIA)$ of that theory is $\mathbb{Z}$ and its algebraic structure is that of regular integer arithmetic. The allowed constraints are all those of the form $\lambda_1 x_1 + \cdots + \lambda_n x_n \diamond c$, with $\lambda_1, \ldots, \lambda_n, c \in \mathbb{Z}$ and $\diamond \in \{=, <, \leq, >, \geq\}$.

Consider now the SMT(QF_LIA)-formula φ from Example 2.2.1.

Example 2.2.1.

$$\begin{aligned}\varphi(X) :=\ & 2x_1 - x_2 + x_3 > 3\\ \wedge\ & x_1 + x_3 = 2\\ \wedge\ & (x_1 < 0 \ \vee\ x_3 < 0)\\ \wedge\ & x_2 + x_3 > 0\end{aligned}$$

Even the satisfiability of a rather simple formula such as this, containing only 5 $\mathcal{T}$-constraints over 3 theory variables, can be difficult to determine without a structured approach.

Deciding the satisfiability of SMT formulae using computer programs is the subject of the field known as *SMT solving*, and the programs developed for this purpose are called *SMT solvers*. Fundamentally, most SMT solvers can be described as either a lazy or an eager solver. Both kinds of solvers are essentially a combination of a SAT solver and a theory solver. However, they differ in how these two are combined. Eager SMT solvers work by reducing the whole SMT problem to an instance of SAT and then consulting a traditional SAT solver to solve it. In the reduction, every theory constraint is encoded into a purely Boolean formula over Boolean variables. As this can result in a SAT formula containing a large number of Boolean variables, this technique is often referred to as bit-blasting. If the reduction step is formally correct, this approach is a complete decision procedure. For theories with an infinitely large universe, a semi-decision procedure can be obtained by bounding the domain of possible values and successively widening the bound until a solution is found. In [BDE+14], this is applied to the theory of quantifier free nonlinear integer arithmetic (QF_NIA).

The second fundamental approach to SMT solving is referred to as lazy SMT. It is based on the concept of the Boolean abstraction of an SMT($\mathcal{T}$)-formula, in which every unique $\mathcal{T}$-constraint is replaced by a unique Boolean variable. Abstracting away the example formula φ results in

$$\begin{aligned}\varphi^p(B) :=\ & b_1\\ \wedge\ & b_2\\ \wedge\ & (b_3 \ \vee\ b_4)\\ \wedge\ & b_5.\end{aligned}$$

The central idea underlying lazy SMT goes as follows: Every assignment of theory values to X can be associated with an assignment of Boolean values to B by substituting X into all constraints and evaluating them. Conversely, every assignment of

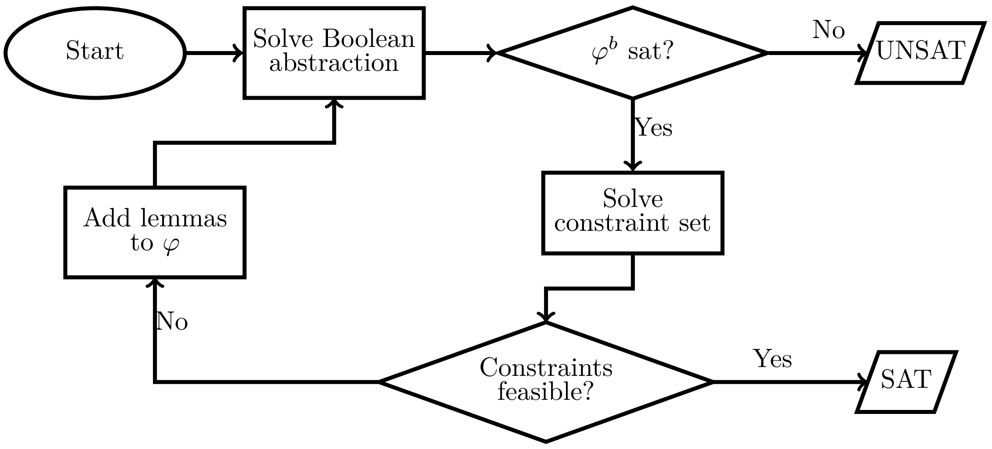

Figure 2.1: The basic full-lazy SMT procedure.

Boolean values to B represents a conjunction of $\mathcal{T}$-constraints, each of which is either a constraint from the original SMT($\mathcal{T}$)-formula or the negation of an constraint, depending on which value was assigned to the Boolean variable associated with that constraint. If there is an assignment of theory-values to X that satisfies $\varphi(X)$, the corresponding assignment of Boolean values to B must also satisfy $\varphi^p(B)$. Thus, enumerating all satisfying assignments of φ^p and using a $\mathcal{T}$-solver to decide the feasibility of the associated constraint set yields a complete solution to SMT($\mathcal{T}$) as long as the $\mathcal{T}$-solver is complete.

2.3 Cylindrical Algebraic Decomposition

One strategy to solve sets of inequalities over real numbers, introduced by Collins in [Col75], is the *Cylindrical Algebraic Decomposition*. Through successively projecting the solution space to lower-dimensional spaces, it yields a decomposition of $\mathbb{R}^n$ into cells such that each of a given set of polynomials has constant sign on every cell. Furthermore, it also gives a sample from each cell that is said to represent that cell. Using this decomposition, it is straightforward to check if a solution exists that satisfies every inequality: Evaluate every polynomial against every sample. If every sample results in the wrong sign for at least one polynomial, the set of inequalities is infeasible. Otherwise, any sample that satifies all inequalities is a feasible solution.

Although the CAD works for all such sets of inequalities, it is quite computationally complex in the worst case, taking time doubly exponential in the number of both constraints and variables. While it can still be feasibly computed in many cases, it is still the most computationally expensive part of SMT-RAT when used for QF_NRA problems. Therefore, minimizing the number of theory calls is highly desirable.

2.4 Infeasible Subsets

The classic lazy SMT($\mathcal{T}$) setup, sketched in Figure 2.1, consists of a SAT solver that continually presents proposed solutions to the $\mathcal{T}$-solver, which responds with $\mathcal{T}$-*sat* or $\mathcal{T}$-*unsat* until either SAT is returned or the SAT solver runs out of solutions.

Every time the $\mathcal{T}$-solver returns $\mathcal{T}$-*unsat*, the SAT solver needs to learn some new information, otherwise it would continue making the exact same assignment again. As the SAT solver only works on Boolean variables and propositions, this information has to be encoded as Boolean clauses. These clauses, called $\mathcal{T}$-lemmas, can greatly reduce the search space the SAT solver has to traverse before either finding a satisfying assignment or returning *unsat*. The bare minimum of additional knowledge that the SAT solver can always learn is that its full assignment was $\mathcal{T}$-infeasible. Consider now a SMT(QF_NRA)-formula such as

$$\varphi(x_1, \ldots, x_k) := x_1 < 0 \ \wedge \ (p_1(x_1, \ldots, x_k) > 0 \ \vee \ \ldots \ \vee \ p_j(x_1, \ldots, x_k)) \ \wedge \ x_1 > 0$$

where $p_1, \ldots, p_j$ are polynomials in $x_1, \ldots, x_k$. The Boolean abstraction of φ that the SAT solver sees is

$$\varphi(c_0, \ldots, c_{j+1}) = c_0 \ \wedge \ (c_1 \ \vee \ \ldots \ \vee \ c_j) \ \wedge \ c_{j+1}.$$

This abstraction is satisfied as long as c_0, c_{j+1} and at least one other variable are true. Therefore. there exist $2^j - 1$ assignments satisfying it. However, from the perspective of the $\mathcal{T}$-solver, it is obvious that none of these assignments are feasible, as $x < 0$ and $x > 0$ can never hold at the same time. To convey this knowledge to the SAT solver, it can emmit the lemma $\neg(x < 0 \ \wedge \ x > 0)$, abstracted as $\neg(c_0 \ \wedge \ c_{j+1})$. Using this additional knowledge, the SAT solver can conclude *unsat* in the very next iteration, instead of having to search its way through all possible assingments. $\neg(c_0 \ \wedge \ c_{j+1})$ is a special kind of lemma, as it is a subset of the assignments passed to the $\mathcal{T}$-solver that is $\mathcal{T}$-inconsistent by itself. These lemmas are especially useful to CDCL SAT solvers, which rely on information gained from conflicts, because they give the SAT solver a precise reason why its assignments yielded a $\mathcal{T}$-conflict. This motivates the following definition:

Definition 2.4.1. *(Infeasible Subset) Given a set of $\mathcal{T}$-constraints P that is $\mathcal{T}$-inconsistent, a constraint set $I \subseteq P$ is called an* infeasible subset *if*

$$\mathcal{T} \not\models \bigwedge_{p \in I} p.$$

If P is $\mathcal{T}$-inconsistent, then it is always an infeasible subset of itself. This is called the trivial infeasible subset. It conveys no additional information to the SAT solver that it could not already deduce from the fact that the $\mathcal{T}$-solver returned $\mathcal{T}$-unsat. Furthermore, if $I \subseteq P$ is an infeasible subset of P, then any $I' \subset P$ with $I \subseteq I'$ must be an infeasible subset as well, because adding more constraints can never make an already infeasible constraint set feasible. Therefore, an infeasible subset always yields at least the same amount of information to the SAT solver as any of its supersets. This naturally leads to the notion of minimal and minimum infeasible subsets.

Definition 2.4.2. *(Minimal Infeasible Subset) An infeasible subset I of a set P of $\mathcal{T} -$ constraints is* minimal *if there exists no $I' \subset I$ that is itself an infeasible subset of P.*

Definition 2.4.3. *(Minimum Infeasible Subset) An infeasible subset I of a set P of $\mathcal{T} -$ constraints is a* minimum *infeasible subset if there exists no infeasible subset $I' \subset P$ with $|I'| < |I|$.*

In essence, a minimal infeasible subset represents a local minimum with respect to cardinality, while a minimum infeasible subset represents a global minimum. Note that it is entirely possible for the $\mathcal{T}$-solver to emmit multiple infeasible subsets for a single theory call. In that case, the SAT solver adds all of them to its clause set. However, simply emmiting all infeasible subsets would be suboptimal, because all the information the SAT solver would gain from them is already gained by only learning about all minimal infeasible subsets and too many redundant clauses can slow down the SAT procedure. Instead, the $\mathcal{T}$-solver should aim to emmit a moderate number of highly relevant infeasible subsets.

The question that remains then is how to find small or minimal infeasible subsets to speed up the SAT solver without incurring too high of a computational cost to the theory solver. Before introducing our own ideas, we will first take a look at existing research regarding infeasible subsets and the closely related topic of unsatisfiable cores.

2.5 Related Work

In this section, we shall examine the state of the literature on the topic of computing infeasible subsets. Although few have dealt with this problem explicitly in the context of the CAD, a closely related topic from the field of SAT Solving has received considerable attention, namely that of finding unsatisfiable cores of SAT clause sets.

2.5.1 Unsatisfiable Cores

Given an unsatisfiable SAT-formula $F = c_1 \wedge \cdots \wedge c_n$ in CNF, $U = c'_1 \wedge \cdots \wedge c'_k$ is said to be an unsatisfiable core of F if it is unsatisfiable itself and $c'_1, \dots c'_k \subseteq c_1, \dots c_n$. Finding these unsatisfiable cores has been a topic of interest for a while in the SAT community, as it allows an application making use of a SAT solver to receive more than a simple 'no' answer in the case of unsatisfiability. For example, an FPGA routing tool can use them to report exactly which wires are unrouteable. This proves to be important enough that today implementations of unsatisfiable core algorithms exist even in hardware.[GWKS08]

On the software side, several mature algorithms have been presented as well. Conceptually, they can be categorized into two groups: Those that follow a bottom-up approach start from an empty subset and add clauses until unsatisifiability is achieved. Those that follow a top-down approach conversely start with the entire formula and remove clauses until it is no longer unsatisfiable.

Another important distinction is whether an algorithm is monotonic. In a monotonic algorithm, the subset only ever grows or shrinks, for bottom-up and top-down approaches respectively. Non-monotonic algorithms may backtrack at times, i.e. a non-monotonic bottom-up algorithm may remove clauses it previously added and a non-monotonic top-down algorithm may add clauses it previously removed when neccessary. In the following, several distinct algorithms for finding unsatisfiable cores are presented briefly.

zChaff is an optimized SAT solver implementing the Chaff algorithm as presented in [MMZ$^+$01]. It includes a method to generate proofs of the unsatisfiability of a given SAT formula. In many cases, such a proof does not use all clauses comprising the formula, so those which are not mentioned are known to be redundant. Taking only the formulae included in the proof therefore yields an unsatisfiable core of that formula. In [ZM03], the authors show how, by applying this method repeatedly until no more clauses can be eliminated, a small but not neccessarily minimal unsatisfiable core can be obtained. This algorithm is therefore a monolithic top-down algorithm. zChaff also exposes this functionality as a stand-alone program called zCore.

A Minimally Unsatisfiable Subformula Extractor (AMUSE) is a monotonic top-down algorithm that sacrifices the guarantee of finding the minimum core every time in order to achieve faster runtimes and the possibility of computing several different unsatisfiable cores. It was introduced by Yoona et al. in [OMA$^+$04].

Compute All Minimal Unsatisfiable Subsets (CAMUS), presented in [LS08], is an algorithm that, as the name suggests, yields not only one minimal unsatisfiable subset, but all of them. It works by first computing all minimal correction subsets of the formula. A subset of an unsatisfiable formula is called correction subset if removing it would make the formula satisfiable. If it is also minimal in the sense that removing any element from it would make its complement unsatisfiable, it is an minimal correction subset. Using these minimal correction subsets, CAMUS iteratively constructs unsatisfiable cores beginning from the empty set, making it a bottom-up algorithm. Finding all unsatisfiable cores proves to be intractable in many cases, so Liffiton and Sakallah also included a relaxed variant of the procedure. This still produces only minimal subsets, albeit not neccessarily all of them, and does so a fraction of the computational cost.

While all these algorithms employ vastly different strategies, optimize for different metrics and yield different results, they still all solve the same problem and can, at least in theory, be used interchangeably. At the very least, they can be compared against each other on the same problem sets, and such comparisons were carried out successfully in [OMA$^+$04], [Hua05] and [GMP07].

2.5.2 Previous Work on Infeasible Subets

Algorithms for computing infeasible subsets of SMT($\mathcal{T}$) clause sets have not received nearly as much attention by researchers as algorithms for computing unsatisfiable cores. As the rules to what makes a set of $\mathcal{T}$-constraints infeasible are specific to the theory, this task is mostly only solved for some specific choice of $\mathcal{T}$.

Still, some research has gone into trying to develop a universal solution. In [CGS07], Cimatti et al. present a technique for generating infeasible subsets that is applicable to any theory solver as long as it has a way of generating other lemmas. It is based on the following observation:

Let $C = \{c_1, \dots, c_n\}$ be a set of $\mathcal{T}$-constraints, $X = \{x_1, \dots, x_n\}$ a set of Boolean variables representing the abstractions of those constraints and L a set of lemmas about these constraints, encoded as Boolean formulae over X. Then, consider the Boolean clause set $S := X \cup L$. If S is unsatisfiable and all the lemmas in L are valid, then C is infeasible in $\mathcal{T}$. Accordingly, if there is an unsatisfiable core $S' \subset S$, then the constraints represented by the Boolean variables in $X' := S' \cap X$ form an infeasible subset of C. Therefore, the algorithm only needs to compute unsatisfiable cores of the set of lemmas given by the $\mathcal{T}$-solver in order to find infeasible subsets. This allows the use of any of the existing solutions to the unsatisfiable cores problem, such as the ones presented in Section 2.5.1. However, the success of this algorithm crucially depends on the quality and quantity of lemmas provided by the $\mathcal{T}$-solver. When provided with no utile lemmas, its output is the trivial infeasible subset. There is no immediately obvious way to extract a large amount of lemmas from a CAD, and to the author's knowledge, there is no such method in the literature either. For these reasons, this solution is not further investigated here.

On the topic of CAD-specific algorithms for generating infeasible subsets, the literature appears to be rather sparse. An algorithm from [JDF15] by Jaroschek et al. approaches the issue from a linear programming perspective. The algorithm works by constructing the conflict matrix M. Each row in this matrix corresponds to a sample obtained from the CAD and each column corresponds to a $\mathcal{T}$-constraint. More specifically, let $C = \{c_1, \dots, c_n\}$ be the set of constraints passed to the CAD and $S = \{s_1, \dots, s_m\}$ be the set of sample points obtained it. Then,

$$M_{j,i} := \begin{cases} 1, & \text{if } c_i \text{ violates } s_j \\ 0, & \text{otherwise.} \end{cases}$$

The utility of this concept lies in the fact that it allows one to use linear algebra to find an infeasible subset. A vector $v \in \mathbb{B}^n$ can be thought of as a representation of a subset I of C, where $c_i \in I \Leftrightarrow v_i = 1$. Then, Mv is the vector whose entries state for every sample how many constraints in I violate it. Finding the minimum infeasible subset can then be stated as the following integer programming problem:

$$\begin{array}{ll} \textbf{minimise} & \displaystyle\sum_{i=0}^{m} v_i \\ \textbf{subject to} & Mw \geq (1, \dots, 1)^T \end{array}$$

This formulation, however, does not directly lead to an efficient solution, as integer programming is an NP-hard problem. In fact, integer programming was among the 21 original problems that Karp showed to be NP-complete in [Kar72]. Restating the problem as such still offers two advantages. Firstly, it allows the use of a state-of-the-art integer programming implementation. Although it is still bound to exhibit exponential complexity in the worst case, being highly optimized means it should perform well in many cases. Secondly, by relaxing the minimality condition of the optimization problem, the solution can be sped up considerably. In turn, the guarantee of finding the minimum infeasible subset every time has to be sacrificed. In the following chapter, we present a different technique, reducing the problem not to linear programming but to the set cover problem. However, we shall borrow the notion of the conflict matrix.

Chapter 3

The Set Cover Problem

Modelling infeasible subsets as solutions to an integer approximation problem is one possible way to approach the issue of computing infeasible subsets. There is another, arguably more natural abstraction that links infeasible subsets to set covers.

Intuitively, a set cover is a selection of sets from a given pool such that every element in a given universe is covered by at least one set in the selection. Formally, set covers can be defined as follows:

Definition 3.0.1. *(Set Cover)*
Given a set U, called universe, and a set $\mathcal{S}$ of subsets of U such that

$$\bigcup_{S \in \mathcal{S}} S = U,$$

$C \subseteq \mathcal{S}$ is called a set cover of U using $\mathcal{S}$ if

$$\bigcup_{S \in C} S = U.$$

The set cover problem (SCP) is concerned with finding the smallest set cover of a given universe using a given $\mathcal{S}$. It is commonly stated in its optimization problem version:

Definition 3.0.2. *(Set Cover Problem)[GN72]*
Given a set U and a set $\mathcal{S} \subseteq \mathcal{P}(U)$ such that

$$\bigcup_{S \in \mathcal{S}} S = U,$$

find a set cover C of U using $\mathcal{S}$ that minimizes $|C|$.

3.1 Infeasible Subsets as Set Covers

To draw the connection between set covers and infeasible subsets, one needs to understand exactly which information is known after the CAD is computed. Firstly, a set P of polynomial constraints and a set Q of sample points, each representing a cell of the CAD, can be assumed to be given. Secondly, by evaluating all constraints on all sample points, we can obtain the set $V_i \subseteq Q$ of samples violating it for every constraint p_i .

Due to the sign-invariance property of the CAD, if a constraint is not satisfied by a sample point, it cannot be satisfied by any point within that point's cell. Therefore, if a set $I \subseteq V = \{V_1, \dots, V_k\}$ contains at least one constraint for every sample point that violates that sample point, the conjunction of all constraints within I is unsatisfiable on all of $\mathbb{R}^n$. I is thus an infeasible subset. Intuitively, this is the case exactly if I represents a set cover of Q using V. We solidify our intuition in the following theorem:

Theorem 3.1.1. *Let $P = \{p_1, \dots, p_k\}$ be an infeasible set of polynomial constraints over $\mathbb{R}^n$ and let Q be the set of sample points obtained by applying the CAD to P. Furthermore, for all $i \in \{1, \dots, k\}$, let $V_i \subseteq Q$ be the set of sample points that violate p_i.*
Then, $I = \{p_{j_1}, \dots, p_{j_m}\}$, with $j_1, \dots, j_m \in 1, \dots, k$ is an infeasible subset of P if and only if $C = \{V_{j_1}, \dots, V_{j_m}\}$ is a set cover of Q using $\{V_1, \dots, V_k\}$.

To prove the forward implication, assume that C is not a set cover of Q. Then, there must be at least one $q \in Q$ not covered by C, that is to say

$$q \notin \bigcup_{i \in \{1,\dots,m\}} V_{j_i}.$$

This means that none of $p_{j_1}, \dots, p_{j_m}$ violate q. Therefore, I is satisfied by q and is not an infeasible subset.

Conversely, assume that C is a set cover of Q. Let $x \in \mathbb{R}^n$. As the CAD decomposes the entire solution space, it yields exactly one cell containing x. Let q be the sample point representing that cell. Since all the polynomials involved in $\{p_1, \dots, p_k\}$ are sign-invariant on that cell, $p_i \models x \iff p_i \models q$ for all $i \in \{1, \dots, k\}$. Because C is a set cover of Q, there is $l \in 1, \dots, m$ such that $q \in V_{j_l}$. Therefore, $p_{j_l} \not\models q$ and thus $p_{j_l} \not\models x$. As $p_{j_l} \in I$, it follows that I is not satisfied by any $x \in \mathbb{R}^n$. I is thus an infeasible subset.$\square$ As this theorem shows, infeasible subsets are equivalent to set covers in a natural way. Therefore, an efficient algorithm for the SCP would yield a fast way to find the minimum infeasible subset as well.

3.2 Greedy Algorithm for the SCP

Like integer programming, SCP was proven to be NP-complete by Karp's essential work in [Kar72]. Therefore, unless $P = NP$, any algorithm that finds the perfect solution to every SCP instance admits at least exponential time-complexity. Instead of aiming for the minimum solution, it is advisable to find an algorithm that runs fast in all cases and yields minimal infeasible subsets that provide useful information for the SAT solver. Thus, we relax the condition of the SCP so that instead of finding the smallest possible cover, the goal is to find a close approximation to the minimum solution. In [Joh73], Johnson presented a straightforward greedy solution, summarized in Algorithm 1, which runs in polynomial time and produces covers which are in the worst case larger than the minimum solution by a factor logarithmic in the problem size. In fact, as Ras and Safra show in [RS97], $P \neq NP$ implies that there can be no polynomial-time approximation algorithm for the SCP that has an approximation factor of better than $\Omega(\ln(n))$.

Therefore, by applying Theorem 3.1.1, the greedy SCP algorithm can be turned into an algorithm for finding infeasible subsets that is computationally inexpensive and yields satisfactory results in most cases.

Algorithm 1 Greedy algorithm for the SCP

```
1: $C \leftarrow \emptyset$
2: $R \leftarrow U$
3: **while** $R \neq \emptyset$ **do**
4:     $X \leftarrow \arg\max_{S \in \mathcal{S} \setminus C} (|S \cap R|)$
5:     $C \leftarrow C \cup \{X\}$
6:     $R \leftarrow R \setminus X$
   **return** $C$
```

3.3 Weighted SCP

Although the only thing that matters for solutions to the classical SCP is the cardinality of the set cover, the same is not necessarily true for infeasible subsets. A minimal infeasible subset is always more useful to the lazy SMT procedure than any of its supersets, but there is no reason why the most useful infeasible subset has to be a *minimum* infeasible subset. Some constraints are more useful to the SAT solver than others when included in infeasible subsets, and some constraints make the task of the CAD module harder whenever they appear in an assignment. Therefore, it is desirable to restate the problem in a way that allows one to optimize for a more sophisticated metric than just the size of the infeasible subsets. There is a generalization of the SCP that seems obvious for this purpose: the Weighted Set Cover Problem (WSCP).

Definition 3.3.1. *(Weighted Set Cover Problem) Given a universe U, a set of subsets $\mathcal{S} = \{S_1, \dots, S_n\} \subseteq \mathcal{P}(U)$ such that*

$$\bigcup_{S \in \mathcal{S}} S = U$$

and an assignment $w : \mathcal{S} \to \mathbb{R}$ of weights to subsets, a minimum weighted set cover is a set cover of U using $\mathcal{S}$ that minimizes

$$\sum_{S \in C} w(S).$$

More intuitively, every subset S is associated with a weight that expresses how expensive it is to include that subset in the set cover. The goal then is to find a cover such that the total weight of its elements is as small as possible. In the context of infeasible subsets, this allows one to assign a weight to every constraint expressing an estimate of how expensive that constraint is when included in an infeasible subset.

3.3.1 Adapting the Greedy Algorithm

WSCP is at least as hard as SCP. A simple argument showing this is that if w is constant, the WSCP becomes equivalent to its unweighted version. Therefore, WSCP itself has to be at least NP-hard and the same inapproximability results as for the SCP apply. However, it turns out that there is a natural way to generalize the greedy algorithm for the SCP into an algorithm that yields good results for the WSCP. Instead of picking the subset that covers the highest number of the remaining elements, pick the subset that maximizes the ratio between the number of remaining elements covered and its weight. A more formal description of the algorithm is given in Algorithm 2. As Chvatal proves in [Chv79], this algorithm, just like its unweighted counterpart, produces a polynomial-time approximation of the optimum solution that is, at worst, off by a factor logarithmic in the input problem size. This makes it an asymptotically optimal approximation among all polynomial-time algorithms for the WSCP.

Algorithm 2 Greedy algorithm for the WSCP

```
1: $C \leftarrow \emptyset$
2: $R \leftarrow U$
3: **while** $R \neq \emptyset$ **do**
4:     $X \leftarrow \arg\max_{S \in \mathcal{S} \setminus C} \left( \frac{|S \cap R|}{w(S)} \right)$
5:     $C \leftarrow C \cup \{X\}$
6:     $R \leftarrow R \setminus X$
   **return** $C$
```

3.4 Possible Metrics

The choice of w is critical for the success of the WSCP approach. In the worst case, a bad choice of w could lead to larger infeasible subsets that also contain less useful information for the SAT solver. In the following, we suggest three possible metrics and show how they can be unified into a single weighting function.

3.4.1 Algebraic complexity

One heuristic way of judging how useful a constraint is when contained in the infeasible subset is looking at its algebraic complexity. The runtime of the CAD is bounded above by a term depending both on the number of variables occurring in the inequality set as well as the polynomial degree of the inequalities. Therefore, one should try to minimize both of those numbers for all the constraints being passed to the CAD. By including a constraint in the infeasible subset, we increase the abilty of the SAT solver to reason about its truth value. Once it has deduced the truth value of a constraint completely, the SAT solver will include that constraint in every subsequent assignment. Hence, it is advisable not to include constraints with a high algebraic complexity in the infeasible subets, if possible.

3.4.2 SAT Activity

SMT-RAT keeps track of a number called "SAT Activity" for every literal that represents a guess towards how interesting that literal is for the SAT-solver. The heuristic currently employed for this is the Variable State Independent Decaying Sum (VSIDS), first introduced in [MMZ$^+$01] as part of the Chaff algorithm. In the beginning of the procedure, every literal has a VSIDS of 0. Every time a conflict occurs, all the literals occurring in the conflict clause have their VSIDS increased. The amount by which they are increased also increases with every conflict. This means that literals occurring in more recent conflict clauses tend to have higher VSIDSs. When assigning values to literals, the SAT solver will then pick the literal with the highest VSIDS. By doing so, the SAT solver favors using fresh information gained from recent conflicts over information from older conflicts which could be stale already. In order to provide even more information about high-activity literals, the theory module should try to pick constraints with a high SAT activity for inclusion in the infeasible subset.

3.4.3 Decision Level

SMT-RAT employs a backtracking SAT solver which tags literals with a decision level as it assigns truth values to them. The higher the decision level, the later a certain literal's value was assigned. If a guess for a literal that was made very early on in SAT procedure turns out to be false, a larger runtime penalty than in the case of a more recently guessed variable will be incurred, as the SAT solver has to backtrack further. When the SAT solver makes a wrong decision, it is therefore critical to recognize this error as early as possible. An infeasible subset containing literals with a low decision level can help with this. Thus, the theory module should aim to minimze the decision levels of the constraints included in the infeasible subsets.

3.4.4 Combining Heuristics

By picking a suitable weight function, all the metrics previously mentioned can be accounted for. How exactly the weight of a constraint is to be computed warrants some discussion. First, note that the aim is to minimize algebraic complexity and decision level, but to maximize SAT activity. Therefore, the weight function should have a negative derivative with respect to SAT activity. However, we still have to make sure that the weight stays positive, as a negative weight would indicate that including a constraint in the infeasible subset is favorable even if all samples are already covered. We propose the following function to assign weight to a constraint:

$$w(S) := w_0 + \lambda_c \; c(S) + \lambda_d \; d(S) + \frac{\lambda_a}{1 + a(S)}$$

where $c(S)$, $d(S)$ and $a(S)$ are the complexity, decision level and SAT activity of the constraint, respectively and w_0 λ_c, λ_d, λ_a are constants which can be adjusted to tune the weight function. By including a constant weight w_0, the algorithm can be instructed to always include constraints that violate a very high number of samples, even if they are considered expensive due to other metrics.

Chapter 4

Towards Smaller Infeasible Subsets

As seen before, finding a minimum infeasible subset is an NP-complete problem and therefore intractable in the general case. In contrast, the greedy algorithm concludes very quickly on most inputs but can yield subpar results. There is a trade-off here between computational complexity and solution quality. In the following, we show how the results can be improved considerably while by investing a small amount of additional work,

4.1 Preconditioning the Conflict Matrix

As the greedy SCP algorithm is already asymptotically optimal among polynomial-time approximations to the SCP, any optimization we can perform must be domain specific. Therefore, it is imperative to look at the actual SCP instances that occur when using the CAD for QF_NRA problems. By identifying certain regularities in those instances, we can develop heuristics to precondition most of them into a much smaller, equivalent SCP instances. To discover and quantify these regularities, a test run using standard benchmark sets was carried out. SMT-RAT was invoked on the full QF_NRA benchmark set provided by SMT-LIB consisting of 11540 problems and the resulting conflict matrices were captured. In total, 119453 conflict matrices were generated. In Figures 4.1, 4.2 and 4.3, a few of the conflict matrices thus obtained are pictured.

There are two common patterns present in all these matrices: Firstly, most of the columns are rather sparse. Many samples are even violated by only a single constraint. Secondly, many of the columns in each of the matrices are identical. We will discuss to what extent both of these phenomena are representative of the solver's behavior on the entire benchmark set and how to use them as heuristics to reduce the problem size.

4.1.1 Essential Constraints

In the example conflict matrices, most of the columns contain a single 1. The matrices from the full data set also exhibit this phenomenon: the average conflict matrix has

$$\begin{array}{c|ccccccccccc}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} & s_{11} \\
\hline
0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 0 \\
0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}$$

Figure 4.1: A conflict matrix from `hycomp/ball_count_1d_plain.03.qfree_global_10`.

$$\begin{array}{c|ccccccccccccccc}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} & s_{11} & s_{12} & s_{13} & s_{14} & s_{15} \\
\hline
c_1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
c_2 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 \\
c_3 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
c_4 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}$$

Figure 4.2: A conflict matrix from `kissing/kissing_2_4`.

$$\begin{array}{c|ccccccccccccccccccccccc}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} & s_{11} & s_{12} & s_{13} & s_{14} & s_{15} & s_{16} & s_{17} & s_{18} & s_{19} & s_{20} & s_{21} & s_{22} & s_{23} \\
\hline
c_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
c_2 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
c_3 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
c_4 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
c_5 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0
\end{array}$$

Figure 4.3: A conflict matrix from `meti-tarski/Arthan1A-chunk-0023`.

$$\begin{array}{c|cccccccccccccccccccc|}
 & s_1 & s_2 & s_3 & s_4 & s_5 & s_6 & s_7 & s_8 & s_9 & s_{10} & s_{11} & s_{12} & s_{13} & s_{14} & s_{15} & s_{16} & s_{17} & s_{18} & s_{19} & s_{20} \\
c_1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 \\
c_2 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 \\
c_3 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1 & 1 & 0 & 0 \\
c_4 & 1 & 1 & 1 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 1 & 1
\end{array}$$

Figure 4.4: A conflict matrix from `meti-tarski/atan-problem-1-chunk-0076`.

540.6 columns and 303.5 columns with only one 1. Such a column corresponds to a sample that satisfies all constraints except for one. Any set of constraints that does not include this constraint would be satisfied by that sample, so it cannot be an infeasible subset. We shall call constraints with this property *essential constraints*.

Once we have identified all the essential constraints and the corresponding columns, the first preconditioning step is to select all those constraints and to remove all columns covered by them. The order in which the essential constraints are selected is irrelevant to this algorithm, as selecting a constraint can never cause some other constraint to become essential. Therefore, it suffices to iterate through the entire matrix only once. This step's complexity is thus linear in both the number of constraints and samples, so its runtime is dominated by that of the greedy algorithm.

After selecting the essential constraints for inclusion in the infeasible subset, we can also drop all constraints that only cover samples that are already covered by essential constraints, since those can never appear in a minimal infeasible subset. Identifying them is simple: in the reduced conflict matrix, the rows corresponding to these constraints will only contain entries with a value of 0. Note that this preconditioning step not only reduces the size of the conflict matrix, but can at times also improve the quality of the solution when used with the greedy algorithm. Consider the conflict matrix in Figure 4.4, obtained from the first invocation of the CAD on a sample problem from the meti-tarski benchmark set.

Both c_1 and c_3 invalidate 10 samples each, while c_2 and c_4 invalidate 9 samples each. Therefore, the greedy algorithm could pick either c_1 or c_3 in its first iteration, depending on implementation. However, picking c_1 would result in a suboptimal solution: The samples s_9, s_6 and s_1 are only violated by c_2, c_3 and c_4, respectively, meaning c_2, c_3 and c_4 are essential constraints. Hence, the greedy algorithm has to include them all and returns $\{c_1, c_2, c_3, c_4\}$.

Our preconditioning step would identify the essential constraints and then drop c_1 as every sample it covers is also covered by an essential constraint. As $\{c_2, c_3, c_4\}$ is already an infeasible subset and consists only of essential constraints, it must be the unique minimum infeasible subset. Thus, the search for infeasible subsets is finished after this preconditioning step and any further computations can be foregone.

4.1.2 Identical Columns

The second phenomenon apparent in the example matrices is that there are large sets of columns that are identical. In the full data, there are on average 540.6 columns per matrix but only 6.2 unique columns. If several samples are violated by the exact same constraints, then it is unnecessary to consider all of them; all of them are satisfied if and only if any one of them is satisfied.

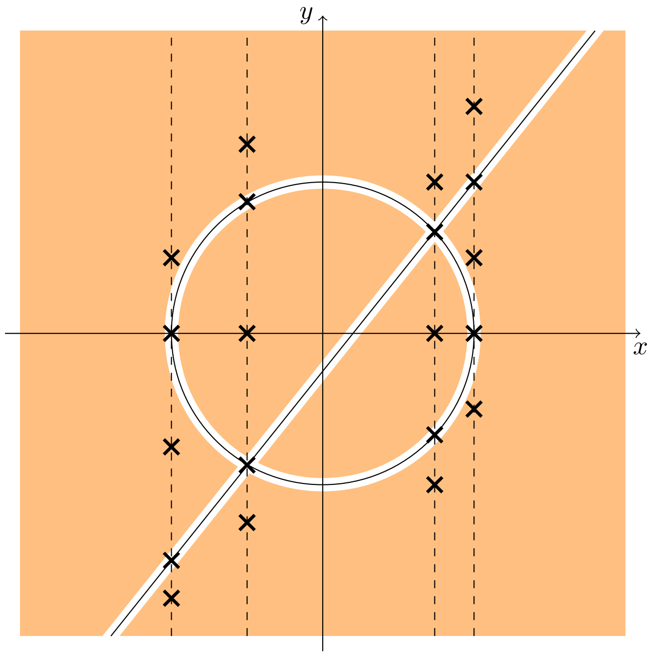

Figure 4.5: The CAD produces redundant samples even for very simple constraint sets

Why this occurs becomes apparent when one considers the limitations of the CAD. Given a set P of polynomials, there is an ideal decomposition of $\mathcal{R}^n$ into connected cells on which each polynomial is sign-invariant. While the CAD will produce at least one sample for every cell in that ideal decomposition, it will hardly ever produce exactly one. In most cases, the CAD yields many samples for every cell in the ideal decomposition. Consider for example the polynomials $p_1(x, y) = 5x - 4y - 2$ and $p_2(x, y) = x^2 + y^2 - 2$. Figure 4.5 visualizes both the cells in ideal decomposition for these polynomials as well as the samples obtained from the CAD. Notably, the CAD produces many samples per ideal cell even in such a simple case. These redundant samples in the CAD then result in duplicate columns in the conflict matrix, as all samples from the same ideal cell will result in the same signs for every polynomial.

This observation suggests another preconditioning step, namely dropping all the duplicates and keeping only the unique columns. Doing this in the naive manner, by comparing every pair of columns, takes $\mathcal{O}(m^2)$ comparisons. Comparing two columns takes $\mathcal{O}(n)$ elementary operations, which puts the overall complexity at $\mathcal{O}(m^2 n)$. A more efficient way to do this is to sort the columns first. This requires $\mathcal{O}(m \log(m))$ column comparisons. Once the columns are sorted, removing duplicates can be done by checking consecutive columns only and takes $\mathcal{O}(m)$ column operations. Alternatively, one can use an associative datastructure such as a hash set for storing unique columns, which results in an even better average-case performance.

At first, it is not clear whether the step of dropping duplicate columns should be carried out before or after selecting essential constraints. Removing duplicate columns has no influence on the number of essential constraints. In fact, that number is exactly equal to the number of unique columns in M that contain a single 1. Therefore, selecting all essential constraints first and the removing duplicate columns seems like the better choice. Two additional arguments support that choice: Firstly, the step of removing duplicate columns has a higher time complexity with respect to the number of columns in the conflict matrix. After selecting the essential constraints, the number of columns will have decreased, making this step less expensive. Furthermore, as seen above, there is the possibility of the essential constraints already forming an infeasible subset. In that case, the step of removing duplicate columns can be skipped. Thus, the right choice is to select the essential constraints first and to remove duplicate columns afterwards. The full preconditioning algorithm comprising both of these steps is described in Algorithm 3.

Algorithm 3 Preconditioning algorithm for reducing the size of the constraint matrix

```
 1: $E \leftarrow \emptyset$
 2: for all $i \in \{1, \dots, m\}$ do
 3:     count $\leftarrow 0$
 4:     for all $j \in \{1, \dots, n\}$ do
 5:         if $M_{i,j} = 1$ then
 6:             $\sigma \leftarrow$ count $+ 1$
 7:             candidate $\leftarrow j$
 8:     if $\sigma = 1$ then
 9:         $E \leftarrow E \cup \{$candidate$\}$
10:         for all $c \in \{1, \dots, m\}$ do
11:             if $M_{e,c} = 1$ then
12:                 $M$.markColumn(c)
13: uniqueColumns $\leftarrow \emptyset$
14: for all $i \in \{1, \dots, m\}$ do
15:     if $M_{\_,i} \in$ uniqueColumns then
16:         $M$.markColumn(i)
17:     else
18:         uniqueColumns $\leftarrow$ uniqueColumns $\cup \{M_{\_,i}\}$
    return $E, M$
```

The set E returned after preconditioning is the set of all row indices corresponding to essential constraints. Every column marked in M either corresponds to a sample that is covered by one of the essential constraints or is redundant because there is another unmarked column in M with the exact same entries. Either way, all marked columns are dropped after preconditioning. If no columns remain, then the set of essential constraints is already an infeasible subset. In that case, it is also the unqiue minimum and minimal infeasible subset and there is no need to proceed any further. If columns do remain, drop all the marked columns from M as well as all rows indexed by E and continue to the actual infeasible subset generation procedure.

4.2 Picking the Best Algorithm

After preconditioning, it is expected that many of the conflict matrices vanish completely, as is the case for the example matrix in Figure 4.4. Most other problem instances will likely be reduced to a small but possibly complicated core and the remaining procedure will only have to find a set cover of the samples that Previously, the greedy algorithm was chosen to guarantee polynomial run time on all problem sets, as any optimal solution would take exponential time. However, if the vast majority of input problems are very small in size, picking the greedy algorithm means trading solution quality in for a negligible gain in speed. Indeed, for very small instances of the SCP, even finding the optimal solution is computationally feasible. Therefore, we present a hybrid algorithm for computing infeasible subsets: If the number of constraints is below a certain threshold t, we compute the optimal solution. If it is not, we proceed with the greedy algorithm until the number of constraints that remain is below t.

The choice of algorithm for finding the optimal solution still remains. Solving the integer programming formulation of the problem using, for instance, a branch-and-cut algorithm, is an option. However, as we are only concerned with the smallest of problem instances, the overhead associated with such a sophisticated algorithm is likely to be too large to justify its use. Furthermore, as integer programming is only concerned with finding the optimum solution, it would yield all minimum infeasible subsets. By fully exhausting the search space, we can instead obtain a solution that is also guaranteed to yield all *minimal* infeasible subsets. Therefore, we opt for a simpler, hand-crafted exhaustive algorithm. Iterating over set cover size in increasing order, it enumerates all possible constraint sets of that size. If a constraint set forms a set cover and is not a superset of any set cover previously found, it must be a minimal set cover and is added to the set of known set covers. Because it visits all constraint sets in ascending order with respect to cardinality, every constraint set is visited by it after all of its subsets. Therefore, the algorithm is guaranteed to find all minimal set covers and none that are not minimal.

The final, full procedure is described in Algorithm 4. Note that there are three parts contributing to the return value: the set E of essential constraints, the set G of greedily selected constraints and the set $\mathcal{A}$ of all minimal infeasible subsets of the set of constraints left after the preconditioning and greedy steps. Because E is necessarily a subset of all infeasible subsets and $\mathcal{A}$ is computed exhaustively, only G can be suboptimal. Therefore, if $G = \emptyset$, the algorithm has found all minimal infeasible subsets.

As a final remark, it should be noted that the only part of the algorithm that has to be adapted to give consideration to weighted constraints is the `selectGreedily()` function. Essential constraints need to be included in every infeasible subset regardless of their weight, and the exhaustive step always has to consider all possible covers of the remaining sample set.

Algorithm 4 Hybrid algorithm to generate infeasible subsets

```
1: $E, M \leftarrow \texttt{precondition(M)}$
2: $M\texttt{.dropMarkedRowsAndColumns()}$
3: if $|M\texttt{.columns}| == 0$ then return $\{E\}$
4: $G \leftarrow \emptyset$
5: $c \leftarrow (0, \ldots, 0) \in \mathbb{B}^{|M\texttt{.columns}|}$
6: while $|M\texttt{.columns}| > t$ do
7:     $g \leftarrow M\texttt{.selectGreedily()}$
8:     $G \leftarrow G \cup \{g\}$
9:     $\texttt{cover} \leftarrow c + M_{g,\_}$
10:    if $\min(\texttt{cover}) \geq 1$ then return $\{E \cup G\}$
11: $\mathcal{A} \leftarrow \emptyset$
12: for all $\texttt{coverSize} \in \{1, \ldots, |M\texttt{.rows}|\}$ do
13:     for all $s \in \{b \subseteq \{1, \ldots, |M\texttt{.rows}|\} \mid |b| = \texttt{coverSize}\}$ do
14:         if $\exists\ A \in \mathcal{A}\ :\ A \subset s$ then
15:             continue
16:         $\texttt{cover} \leftarrow \sum_{j \in s} M_{j,\_}$
17:         if $\min(\texttt{cover}) \geq 1$ then
18:             $\mathcal{A} \leftarrow \mathcal{A} \cup \{s\}$
    return $\{E \cup G \cup A \mid A \in \mathcal{A}\}$
```

Chapter 5

Implementation and Evaluation

The previously desribed algorithm was implemented in the Satisfiability Modulo Theories Real Arithmetic Toolbox (SMT-RAT). SMT-RAT is written in C++ an maintained as an open source project by the Theory of Hybrid Systems research group of RWTH Aachen University. It is described in further detail in [CKJ+15]. SMT-RAT includes a fully functional CAD implementation, which was leveraged for this task. To improve the tractability of the strategy on a large number of problems, the CAD was combined with the virtual substition module also present in SMT-RAT.

SMT-RAT's CAD supplies an implementation of the greedy algorithm as presented, which serves as a reference point for judging the performance of our algorithm.

5.1 Gauging the Success of the Preconditioning Step

First, the effectiveness of the preconditioning step was analyzed. To measure this, our solver was run against the full SMT-LIB QF_NRA set with a timeout of 30 minutes. For every theory call, the number of constraints both before and after preconditioning were captured. This data is summarized by the density plots in Figures 5.1 and 5.2. All runs were carried out using an Intel Xeon X5675 CPU.

As the data shows, the constraint sets that ocurred were mostly rather small even before preconditioning and never exceeded 25 constraints. After preconditioning, 24871 out of 36926 cases were already solved completely. The majority of the remaining problem instances were shrunk considerably, with the largest containing only 17 constraints.

The average number of samples in the problem instances was reduced significantly as well, going from 1130.3 to 5.6 after preconditioning. The majority of this decrease in size is to be attributed to the deletion of duplicate samples, as the average conflict matrix already only contained 6.2 unique columns. The additional reduction in sample count beyond that is due to the deletion of samples that are covered by essential constraints. In the most extreme case, one problem instance contained 1 862 836 samples. After preconditioning, this number diminished down to 3.

We can thus conclude that the preconditioning performs as well as expected by our initial assumptions, solving over two thirds of all problem instances outright and

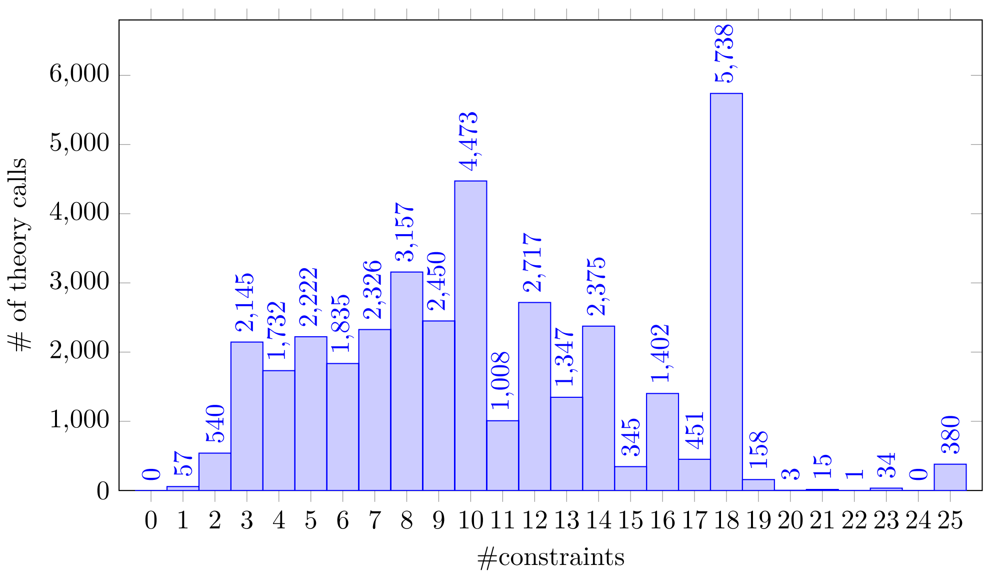

Figure 5.1: Constraint set sizes before preconditioning

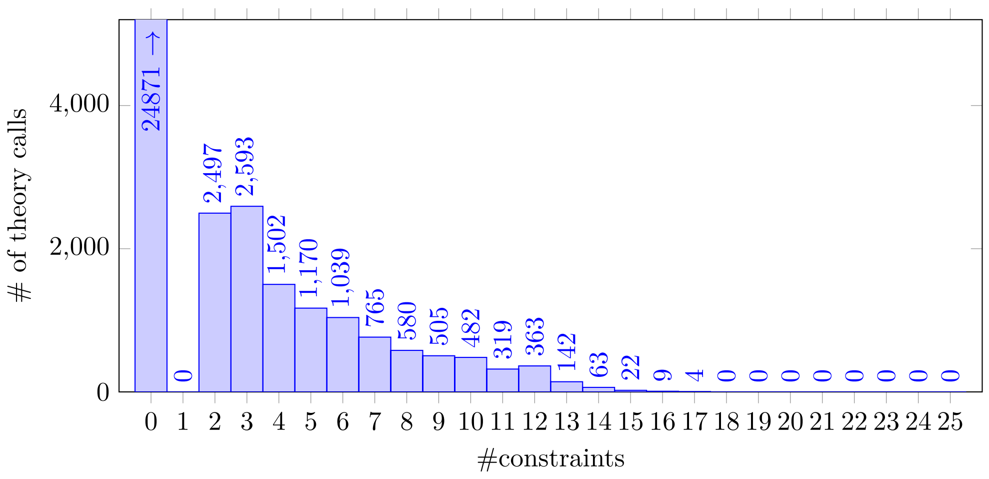

Figure 5.2: Constraint set sizes after preconditioning

reducing the rest substantially.

5.2 On the Choice of t

The central parameter impacting the performance of the hybrid algorithm is the threshold t at which the procedure switches from the greedy algorithm to the exhaustive algorithm. Picking a suitable value for t is therefore crucial. If it is too low, the procedure will not be able find all minimal infeasible subsets in most cases. If it is too high, the exponential asymptotic complexity of the exhaustive algorithm becomes a problem. To make an informed decision on the value of t, two metrics have to be evaluated: Firstly, it is important to know how many problem instances are reduced to t constraints or less by the preconditioning, as those instances are the ones for which our algorithm guarantees an optimal solution. Secondly, the computational cost of the exhaustive algorithm for problem instances containing exactly t constraints needs to be estimated.

Figure 5.3 displays for every value of t both the percentage of problem instances consisting of at most t constraints and the average computational cost of the full infeasible subset generation algorithm when invoked on a problem of size t. Computational cost was measured in CPU time. While for values of t up to 8, the runtime is mostly constant, it seems to grow exponentially after that, nearly doubling with every increase in t. This matches the theoretical asymptotic complexity of finding all minimal infeasible subsets. Adding to that, there appears to be a severe effect of diminishing returns on raising t: With $t = 5$, nearly 90% of the problem instances are solved perfectly. At $t = 12$, the fraction of problems solved perfectly surpasses 99%. Chosing a value of t above that is likely not worth the cost. We suggest a value of $t = 12$ and use that in further experiments.

5.3 Impact on Overall Solver Performance

As shown in the previous section, the algorithm worked out as intended, finding all minimal infeasible subsets in the majority of cases while incurring a negligible performance cost. However, it remains to be seen whether the infeasible subsets computed by it do in fact speed up the complete SMT procedure. Evaluating this proved to be complicated, as it became apparent during benchmarking that the solver only invoked the infeasible subset generation routine on a small portion of SMT-LIB's QF_NRA problem set. SMT-RAT was able to solve 8 147 problems out of the 11 540 without the CAD ever returning $\mathcal{T}$-*unsat*. This can happen due to one of several reasons. Firstly, for a few of the problems, even the Boolean abstraction is already unsatisfiable. To a lazy SMT solver, these problems appear trivial, as it can return *unsat* without ever consulting the theory solver. Secondly, some of the problems do require theory calls, but their algebraic constraints are limited to polynomials of degrees 1 and 2. In those cases, virtual substitution suffices to answer all theory calls and the CAD is never used. Lastly, on a number of problems, the solver was able to deduce *sat* after a single CAD call that returned $\mathcal{T}$-*sat*. In addition to the majority of the problems being solved without infeasible subsets being requested, another 2 181 problems caused the solver to hit the 30 minute timeout without requesting infeasible subsets. This is likely due to those problems containing very complex algebraic constraints, which results in the CAD becoming extremely computationally expensive. Thus, our

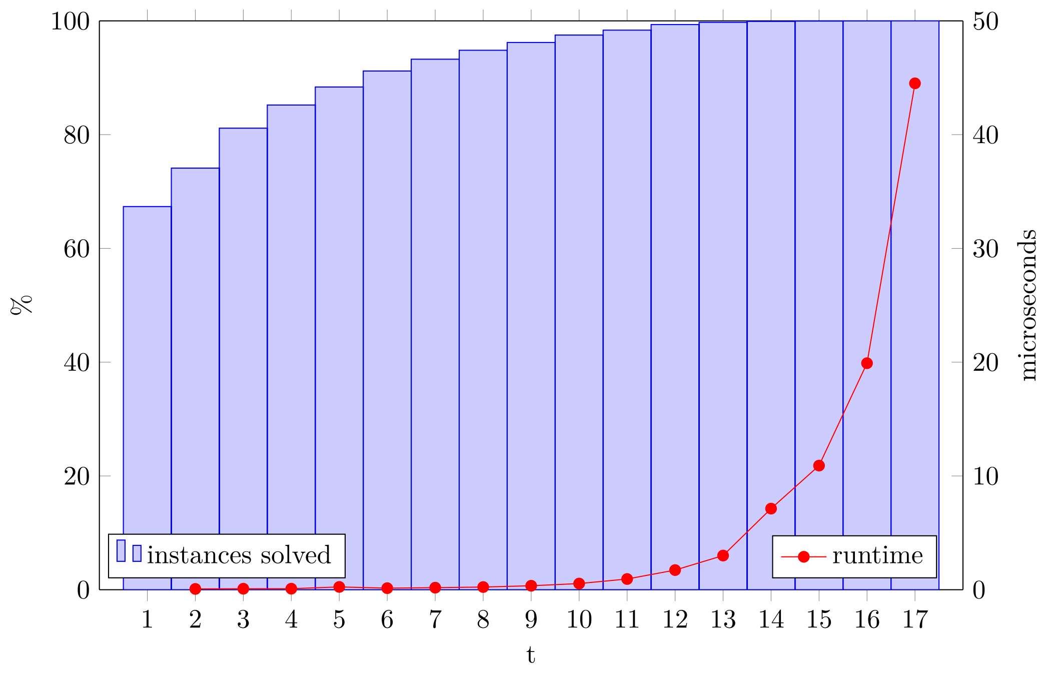

Figure 5.3: Impact of t on the number of perfect solutions and on worst-case runtime.

algorithm is limited in applicability to a very specific set of problems, namely those containing at least one constraint of polynomial degree 3 or higher but not containing constraints complex enough to make the CAD intractable.

Out of the remaining 1 212 problems, SMT-RAT was able to reach a solution in 782 cases, regardless of whether the greedy or our novel algorithm was used. This indicates that the improvement in overall solver performance was not substantial. The time taken by the solver on those 782 problems confirms this observation: On average, using the greedy algorithm, SMT-RAT ran for 49 385 milliseconds before returning a solution. With the hybrid algorithm, this number decreased negligibly to 49 154. However, the hybrid algorithm did manage to achieve a significant decrease in the number of CAD invocations in a number of cases. In total, SMT-RAT needed 5 764 CAD calls to solve the aforementioned 782 problems using the greedy infeasible subset generation algorithm, but only 4 941 when using the hybrid algorithm. The average size of the smallest infeasible subset found by the hybrid algorithm was 2.7, while the infeasible subsets found using the greedy algorithm contained an average of 2.9 constraints, so we met our goal of producing smaller infeasible subsets. The problem that took the most CAD calls, regardless of which algorithm was used, was `atan-vega-3-weak-chunk-0079` from the meti-tarski set. By switching to our hybrid algorithm, the number of CAD calls needed by SMT-RAT to decide *unsat* for this problem went down from 1738 to 997. However, this decrease of over 40% in the number of CAD calls only resulted in an improvement in performance of about 8%, as total runtime went down from 1 754 seconds to 1 622 seconds.

A possible explanation for this discrepancy could be that, while the reduction in CAD calls did benefit the overall runtime, this benefit was diminished by the additional time required for the more complicated infeasible subset computation procedure. By collecting call-graph data on SMT-RAT using the sampling profiler Callgrind from the Valgrind framework, described in [NS07], we were able to dismiss this concern; the computation of infeasible subsets using the hybrid algorithm only accounted for 0.08% of the total runtime when solving `atan-vega-3-weak-chunk-0079`.

Evidently, the time needed to carry out the hybrid algorithm is not the reason for the lack of a performance improvement. Instead, it seems likely that the CAD calls the solver was able to skip due to improved infeasible subsets were not the ones accounting for a majority of its runtime.

Chapter 6

Conclusion

In this paper, we gave an answer to the question of how to efficiently compute useful infeasible subsets from a CAD. First, we examined the existing research on both this topic and the closely related topic of unsatisfiable cores and discussed whether any of the solutions proposed therein could be applied to our exact problem statement. We concluded that due to the lack of a method to produce lemmas from a given CAD, the approach of reducing the problem to that of finding unsatisfiable cores was not suitable for our purposes. The integer programming perspective by Jaroschek was deemed to be a better fit, but we argued that it also had some crucial issues. We therefore introduced a third perspective based on the notion of the set cover and proved an important relationship between infeasible subsets and set covers to justify this.

In order to formulate an efficient algorithm optimized for the specific problem domain, we needed to grasp the nature of the problem instances our algorithm would be facing. To this end, we captured and analyzed a number of actual problem instances from solver runs. Based on this data, we presented two preconditioning heuristics to trim problem instances down to a much smaller but complex core. We argued that, due to this significant decrease in problem size, it is feasible to compute the optimal solution in the majority of cases instead of relying on a faster but suboptimal solution such as the greedy algorithm. Keeping this assumption in mind, we constructed a three-stage algorithm consisting of the preconditioning heuristics followed by an optional greedy stage and finally an exhaustive search. We make the observation that in the cases in which the greedy stage is skipped, our algorithm does indeed find all minimal infeasible subsets.

We implemented our algorithm in the context of the SMT-solving framework SMT-RAT. Using this implementation, we verified our assumption, showing that the preconditioning heuristics did indeed shrink the problem instances substantially. The heuristic of identifying and selecting essential constraints proved to be particularly successful, because it managed to find the unique minimum infeasible subset in two out of three instances. We also provided profiling data supporting our choice of $t = 12$ for the hybrid threshold parameter.

Unfortunately, our hybrid algorithm was not able to make a meaningful impact on SMT-RAT's overall performance. We argue that this is due to the narrow spectrum of QF_NRA problems to which it is currently applicable. If a problem is algebraically simple enough to allow the use of virtual substitution, then that solution is much more preferable in terms of perfomance than the CAD. However, if a problem is too algebraically complex, the CAD will become intractable. Still, we were able to demonstrate a reduction of 14.2% in the number of CAD invocations needed by SMT-RAT. In the future, improvements to the CAD algorithm could open up more problems to its use, possibly increasing the necessity of a sophisticated infeasible subset generation procedure.

In this work, we also described a method to tune the greedy stage of the algorithm by attaching weights to constraints. In addition, we suggested several possible heuristics for picking weights. It remains to be seen which of these heuristics are advantageous to the SMT solving process. This could be analyzed in further research. A different venue for improvement could be the modification of our algorithm to produce not only infeasible subsets but also more general lemmas. For example, if the samples that violate some constraint c_1 form a subset of the samples that violate another constraint c_2, the lemma $c_2 \Rightarrow c_1$ could be inferred.

Bibliography

[BDE$^+$14] Karsten Behrmann, Andrej Dyck, Fabian Emmes, Carsten Fuhs, Jürgen Giesl, Patrick Kabasci, Peter Schneider-Kamp, and René Thiemann. Bit-blasting for SMT-NIA with AProVE. *Proc. SMT-COMP*, 14, 2014.

[BHvMW09] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. Conflict-driven clause learning SAT solvers. *Handbook of Satisfiability, Frontiers in Artificial Intelligence and Applications*, pages 131–153, 2009.

[CGS07] Alessandro Cimatti, Alberto Griggio, and Roberto Sebastiani. A simple and flexible way of computing small unsatisfiable cores in SAT modulo theories. In *Proceedings of the 10th International Conference on Theory and Applications of Satisfiability Testing (SAT-2007)*, pages 334–339, 2007.

[Chv79] V. Chvatal. A greedy heuristic for the set-covering problem. *Mathematics of Operations Research*, 4(3):233–235, 1979.

[CKJ$^+$15] Florian Corzilius, Gereon Kremer, Sebastian Junges, Stefan Schupp, and Erika Ábrahám. Smt-rat: an open source c++ toolbox for strategic and parallel smt solving. In *International Conference on Theory and Applications of Satisfiability Testing*, pages 360–368. Springer, 2015.

[Col75] George E Collins. Quantifier elimination for real closed fields by cylindrical algebraic decompostion. In *Automata Theory and Formal Languages 2nd GI Conference Kaiserslautern, May 20–23, 1975*, pages 134–183. Springer, 1975.

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

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

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

[ES03] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Selected Revised Papers of the Sixth International Conference on Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in*

Computer Science, pages 502–518. Springer International Publishing, 2003.

[GMP07] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Local-search extraction of MUSes. *Constraints*, 12(3):325–344, 2007.

[GN72] Robert S Garfinkel and George L Nemhauser. *Integer programming*, volume 4, pages 298–300. Wiley New York, 1972.

[GWKS08] K. Gulati, M. Waghmode, S. P. Khatri, and W. Shi. Efficient, scalable hardware engine for boolean satisfiability and unsatisfiable core extraction. *IET Computers Digital Techniques*, 2(3):214–229, May 2008.

[Hua05] Jinbo Huang. MUP: A minimal unsatisfiability prover, 2005.

[JDF15] Maximilian Jaroschek, Pablo Federico Dobal, and Pascal Fontaine. Adapting real quantifier elimination methods for conflict set computation. In *Frontiers of Combining Systems - 10th International Symposium, FroCoS 2015, Wroclaw, Poland, September 21-24, 2015. Proceedings*, pages 151–166, 2015.

[Joh73] David S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the Fifth Annual ACM Symposium on Theory of Computing*, STOC '73, pages 38–49, New York, NY, USA, 1973. ACM.

[Kar72] Richard M. Karp. *Reducibility among Combinatorial Problems*, pages 85–103. Springer US, Boston, MA, 1972.

[Lev73] L. A. Levin. Universal sequential search problems. *Probl. Peredachi Inf.*, 9(3):115–116, 1973.

[LS08] Mark H. Liffiton and Karem A. Sakallah. Algorithms for computing minimal unsatisfiable subsets of constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver, 2001.

[NS07] Nicholas Nethercote and Julian Seward. Valgrind: a framework for heavyweight dynamic binary instrumentation. In *ACM Sigplan notices*, volume 42, pages 89–100. ACM, 2007.

[OMA$^+$04] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: A minimally-unsatisfiable subformula extractor, 2004.

[RS97] Ran Raz and Shmuel Safra. A sub-constant error-probability low-degree test, and a sub-constant error-probability PCP characterization of NP. In *Proceedings of the Twenty-ninth Annual ACM Symposium on Theory of Computing*, STOC '97, pages 475–484, New York, NY, USA, 1997. ACM.

[Seb07] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation*, 2007.

[Sei54] Abraham Seidenberg. A new decision method for elementary algebra. *Annals of Mathematics*, pages 365–374, 1954.

[Tar51] Alfred Tarski. A decision method for elementary algebra and geometry. 1951.

[Tse68] G Tseitin. On the complexity of derivation in propositional calculus. *Studies in Constrained Mathematics and Mathematical Logic*, 1968.

[ZM03] Lintao Zhang and S. Malik. Validating sat solvers using an independent resolution-based checker: practical implementations and other applications. In *2003 Design, Automation and Test in Europe Conference and Exhibition*, pages 880–885, 2003.