Engineering SAT Applications

Dissertation

der Mathematisch-Naturwissenschaftlichen Fakultät
der Eberhard Karls Universität Tübingen
zur Erlangung des Grades eines
Doktors der Naturwissenschaften
(Dr. rer. nat.)

vorgelegt von
MSc. (Bioinf.) Christian Zielke
aus Rathenow

Tübingen
2015

Gedruckt mit Genehmigung der Mathematisch-Naturwissenschaftlichen Fakultät der Eberhard Karls Universität Tübingen.

Tag der mündlichen Qualifikation:	15.12.2015
Dekan:	Prof. Dr. Wolfgang Rosenstiel
1. Berichterstatter:	Prof. Dr. Michael Kaufmann
2. Berichterstatter:	Prof. Dr. Wolfgang Küchlin

Acknowledgements

First and foremost I want to thank my advisor, Prof. Dr. Michael Kaufmann, for giving me the opportunity to work in such an inspiring and light-hearted research group as his, for his support, for the freedom he gave me to follow my research ideas and last but not least, for the uncounted rides he gave me to the mensa.

I always enjoyed the easy, funny and friendly atmosphere at our group. I will surely miss the coffee breaks (although I did not drink a single coffee during all the time), the (active and passive) sports events we attended, the insightful discussions about research, life and weekend plans.

I am really grateful to my current and former colleagues, namely Michael Bekos, Till Bruckdorfer, Philip Effinger, Andreas Gerasch, Markus Geyer, Niklas Heinsohn, Stephan Kottler, Robert Krug, and Martin Siebenhaller for creating such a nice atmosphere. A special thank you has to be directed towards Renate Hallmayer, our secretary.

Furthermore, I want to thank all the students, that did their theses under my tutelage. It was a pleasure working with you. A special thanks has to be directed towards Johannes Dellert, since a lot of important ideas originate from the joint work with him.

Last but not least, I want to thank my family and friends, especially my parents, for their unwavering support.

Zusammenfassung

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) ist nicht nur in der theoretischen Informatik ein grundlegendes Problem, da alle NP-vollständigen Probleme auf SAT zurückgeführt werden können. Durch die Entwicklung von sehr effizienten SAT Lösern sind in den vergangenen 15 Jahren auch eine Vielzahl von praktischen Anwendungsmöglichkeiten entwickelt worden. Zu den bekanntesten gehört die Verifikation von Hardware- und Software-Bausteinen.

Bei der Berechnung von unerfüllbaren SAT-Problemen sind Entwickler und Anwender oftmals an einer Erklärung für die Unerfüllbarkeit interessiert. Eine Möglichkeit diese zu ermitteln ist die Berechnung von minimal unerfüllbaren Teilformeln. Es sind drei grundlegend verschiedene Strategien zur Berechnung dieser Teilformeln bekannt: mittels Einfügen von Klauseln in ein erfüllbares Teilproblem, durch Entfernen von Kauseln aus einem unerfüllbaren Teilproblem und eine Kombination der beiden erstgenannten Methoden.

In der vorliegenden Arbeit entwickeln wir zuerst eine interaktive Variante der Strategie, die auf Entfernen von Klauseln basiert. Sie ermöglicht es den Anwendern interessante Bereiche des Suchraumes manuell zu erschließen und aussagekräftige Erklärung für die Unerfüllbarkeit zu ermitteln. Der theoretische Hintergrund, der für die interaktive Berechnung von minimal unerfüllbaren Teilformeln entwickelt wurde, um dem Benutzer des Prototyps unnötige Schritte in der Berechnung der Teilformeln zu ersparen werden im Anschluss für die automatische Aufzählung von mehreren minimal unerfüllbaren Teilformeln verwendet, um dort die aktuell schnellsten Algorithmen weiter zu verbessern. Die Idee dabei ist mehrere Klauseln zu einem *Block* zusammenzufassen. Wir zeigen, wie diese Blöcke die Berechnungen von minimal unerfüllbaren Teilformeln positiv beeinflussen können. Durch die Implementierung eines Prototypen, der auf den aktuellen Methoden basiert, konnten wir die Effektivität unserer entwickelten Ideen belegen.

Nachdem wir im ersten Teil der Arbeit grundlegende Algorithmen, die bei unerfüllbaren SAT-Problemen angewendet werden, verbessert haben, wenden wir uns im zweiten Teil der Arbeit neuen Anwendungsmöglichkeiten für SAT zu. Zuerst steht dabei ein Problem aus der Bioinformatik im Mittelpunkt. Wir lösen das sogenannte Kompatibilitätproblem für evolutionäre Bäume mittels einer Kodierung als Erfüllbarkeitsproblem und zeigen anschließend, wie wir mithilfe dieser neuen Kodierung ein nah verwandtes Optimierungsproblem lösen können. Den von uns neu entwickelten Ansatz vergleichen wir im Anschluss mit den bisher effektivsten Ansätzen das Optmierungsproblem zu lösen. Wir konnten zeigen, dass wir für den überwiegenden Teil der getesteten Instanzen neue Bestwerte in der Berechnungszeit

erreichen.

Die zweite neue Anwendung von SAT ist ein Problem aus der Graphentheorie, bzw. dem Graphenzeichen. Durch eine schlichte, intuitive, aber dennoch effektive Formulierung war es uns möglich neue Resultate für das *Book Embedding* Problem zu ermitteln. Zum einen konnten wir eine nicht triviale untere Schranke von vier für die benötigte Seitenzahl von 1-planaren Graphen ermitteln. Zum anderen konnten wir zeigen, dass es nicht für jeden planaren Graphen möglich ist, eine Einbettung in drei Seiten mittels einer sogenannten *Schnyder*-Aufteilung in drei verschiedene Bäume zu berechnen.

Contents

1 Introduction

1.1 Algorithms

"Before there were computers, there were algorithms. But now that there are computers, there are even more algorithms, and algorithms lie at the heart of computing."

These two sentences that were stated in the preface of the third edition of the book "Introduction to Algorithms" by Thomas H. Cormen et al. [33] summarize the current situation very well. The idea of designing machines which are capable of solving problems automatically, is highly appealing and fascinating for many people. Several hundred years ago scientists started to develop the theory and constructed some early automatic machines, whose applications were basic arithmetic operations. Blaise Pascal and Wilhelm Schickard, the latter a former professor of the University of Tübingen, are considered to be the first two inventors of the mechanical calculator in the early 17th century[1].

In fact, even the origin of the word "algorithm" has a mathematical background. The Persian mathematician, astronomer and geographer al-Khwarizmi wrote the book "On the Calculation with Hindu Numerals" around 825 AD. The Latin translation of the title is known as "Algoritmi de numero Indorum" and led to the term "algorithm", which nowadays refers to "a set of rules that precisely defines a sequence of operations" that terminates eventually [135].

The development of the computer in the last century was the major breakthrough in the creation process of a universal machine that is capable of systematically executing algorithms to solve problems that naturally arise from the most diverse applications in the growing areas of engineering, economy, science, and daily life. Although the vast majority of people have access to computers nowadays and although the development of faster and cheaper computation technology leads to the creation of record breaking super-computers every year, the need for optimizing algorithms is not diminishing. In fact, the need for sophisticated algorithms is bigger than ever. The application of an algorithm that searches in the huge data space of the internet is a prime example for that. The available data are enormous

[1] Further information about the debate Pascal vs Schickard is available at `http://metastudies.net/pmwiki/pmwiki.php?n=Site.SchicardvsPascal`

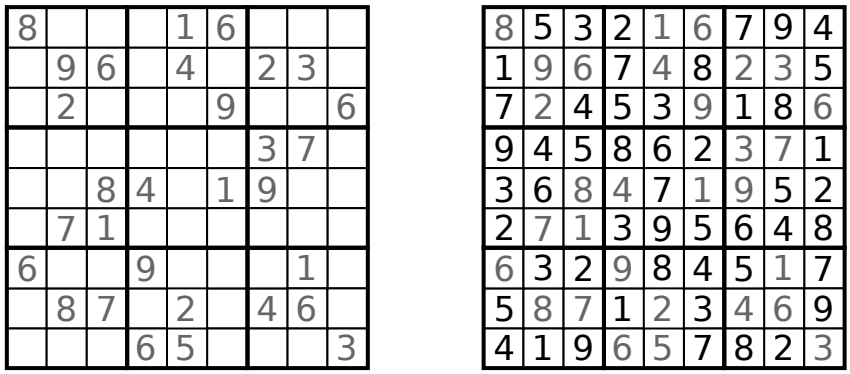


Figure 1.1. (a) A typical Sudoku puzzle. (b) The solution for the same puzzle, black numbers were added due to reasoning.

and rise rapidly every year[2]. With the help of algorithms that harvest information out of petabytes (10^{15} bytes) of data within fractions of a second and further algorithms that process the relations between different entities of the internet to filter relevant information out of the whole data set, the search engines changed the way we currently handle knowledge.

Consider now the well-known Sudoku puzzle as an application that can be solved by algorithms. It is a logic-based, combinatorial puzzle that is played on a partially filled 9×9 grid. The task is to complete the assignment using numbers from 1 to 9 such that the numbers in each row, each column and each major 3×3 block are pairwise different. See Figure 1.1 for a typical Sudoku puzzle and its solution. Sudoku is often referred to as the "21st century Rubik's cube". It is easy to understand, but the reasoning needed to reach completion may be difficult. Each puzzle is supposed to have a unique solution, which does not require the use of trial and error or guessing. Thus, each puzzle can be solved merely by reasoning.

Nevertheless, there is a wide range of algorithms to solve Sudoku puzzles. The easiest is probably just to iteratively create every possible combination of numbers for the 9×9 grid and to check whether that solution contains the partially filled assignments of the input puzzle. However, there are approximately $6.67 * 10^{21}$ possible final grids. Finding the single one that corresponds to the given puzzle might take a lot of time. Assuming that 10 million different possible final grids can be tested within 1 millisecond, this algorithm can take up to $20,819.5$ years to finish.

Another, more promising algorithm would be the idea to solve a Sudoku puzzle by placing the digit 1 in the first cell and checking if it is allowed to be there by checking the row, column, and box constraints. If it is allowed, the algorithm advances to the next empty cell and places a 1 in that cell. Assume that the

[2]In the first quarter of the year 2015 over 294 million top level domains were registered, an increase of 6.5% over the span of one year; numbers taken from `http://www.verisign.com/en\_US/innovation/dnib/index.xhtml` on September 3, 2015.

subsequent test discovers that the 1 is not allowed there. In that case, the value of that cell is increased to a 2. If a cell within the grid is discovered where none of the 9 digits is allowed, the algorithm leaves that cell blank and moves back to the previous cell. The value in that cell is then increased by 1. The algorithm is continued until a valid solution for all 81 cells is found.

However, there is also the possibility to translate the Sudoku puzzle into a so-called *constraint problem* (cf. [129, 95]) and to use several highly optimized algorithms that solve these problems. With the help of this approach, it is possible to solve even the hardest known instances [134] in less than a second on average [129].

1.2 Solving Problems by Translation into SAT

The *Boolean logic* is the underlying concept of this work. It uses exactly two values, `true` and `false`. With the help of special rules a *Boolean formula* can be created, which represents relations between basic atomic elements (called *variables*) in the domain of the Boolean logic. A real-world problem can be formally specified by encoding it as a logical formula in such a way that solutions of the problem correspond to *models* of the formula. In this context, a model is an assignment to the variables such that the formula evaluates to `true`. If the problem has no solution, then any possible assignment to the variables evaluates to `false`.

The problem that describes the possibility to evaluate a formula to `true` is the *Satisfiability problem* (SAT). It was the first problem known to be NP-complete [32]. Thus, all currently known algorithms for SAT, in the worst case, require a runtime that grows exponentially with the size of the formula. Whether there exist efficient (polynomial time) solutions to NP-complete problems is arguably the most famous open question in computer science. Although there is no definitive conclusion, most researchers are convinced that the answer is in the negative.

It follows from NP-completeness that many interesting problems can be solved by translating them into a SAT problem. Some of the first attempts to bring this result into practice were applications of the satisfiability technology to solve problems in the areas of planning [82, 60, 51] and scheduling [35].

The success in this field has aided the development of the most well-known application of SAT: hardware verification, special tasks are microprocessor verification [143], automated test pattern generation [128], equivalence checking of circuits [61], and bounded model checking [19].

Other fields, where satisfiability was successfully applied are natural language processing [84], knowledge representation [93], security protocols [3], cryptanalysis [101], and even bioinformatics [94, 25]. A more complete list of possible applications can be found in the overview paper by Gu et al. [67].

The description as a Boolean formula allows to solve many further interesting problems using SAT technology. For example, (i) when solving the maximal satisfiability problem, an optimal solution for a formula is searched that satisfies the largest possible part of an unsatisfiable formula, (ii) a minimal unsatisfiable subformula can be extracted, which represents a subproblem that cannot be satisfied. We will cover both problems, as well as the basic SAT problem, within this work.

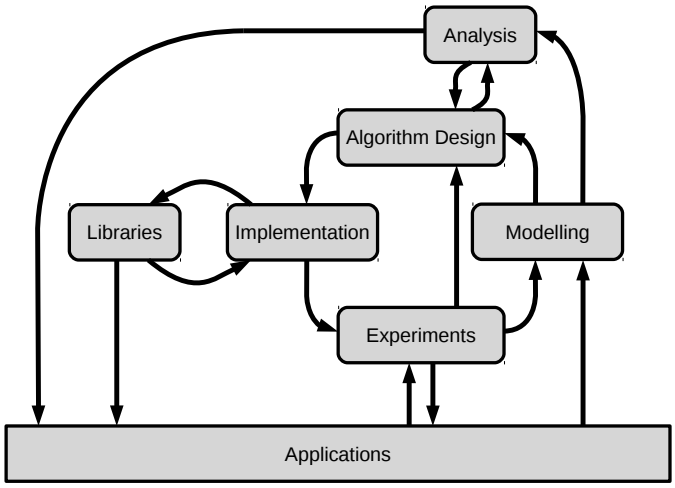


Figure 1.2. Methodological structure of algorithm engineering

1.3 Algorithm Engineering

Motivated by the paradigm of algorithm engineering, we will follow an experiment driven approach, as described in [121] and [105]. Over the last years the term algorithm engineering has become a wide spread synonym for experimental evaluation in the context of algorithm development. It is motivated by the fact that for a long time in classical algorithmics the analysis of algorithms focused exclusively on the theoretical analysis of asymptotic worst-case runtimes leading to the development of highly sophisticated data structures and algorithms.

However, theoretically efficient algorithms do not necessarily turn out to be fast in practice: the theoretic analysis may hide huge constants, the algorithm may perform poorly on typical real-world instances, where alternative methods may be much faster, or the algorithm and data structures may be too complex to be implemented for a specific task. On the other hand, theoretically slow algorithms may be efficient in practice, since the worst-case behavior can be seen only at some highly artificial inputs, and the main applications are of much better structure for the algorithm.

A classic example for this can be found in the field of mathematical programming: The most popular algorithm for solving linear programming problems, the simplex algorithm introduced by Dantzig in 1947, provides very good performance for most input instances occurring in practice but it has been shown to have exponential worst-case time complexity [86]. In the 1970s it was shown that linear programs are solvable in polynomial time using the ellipsoid method [85]. However, the simplex method and its variants are most widely used for decades due to their superior performance.

The general idea of the algorithm engineering approach is represented by the scheme shown in Figure 1.2. The algorithm engineering cycle can be interpreted in the following way (cf. [105]). It usually starts with some specific application in

mind, for which a realistic model has to be found, such that the solutions we will obtain match the requirements as well as possible. The main cycle starts with an initial algorithmic design. Based on this design we analyze the algorithm from a theoretical point of view to obtain performance guarantees like asymptotic runtime or approximation ratios. The next step of the algorithmic development is the implementation of the proposed algorithm, which is probably the most important step of algorithm engineering. We can only succeed in this step if the algorithm is reasonable in its implementation complexity. Since the next step is the experiments, the implementation has to be sufficiently effective regarding possible side effects, which are often underestimated. A very prominent example for this is the memory efficiency. In modern hardware architectures the memory is organized in a hierarchy, where the access costs may differ between different levels by several orders of magnitude. Cache misses should be avoided as much as possible, otherwise the algorithm has to load new data from memory levels that are much slower. Because of this, it may be beneficial to evaluate in particular early experiments not exclusively on the runtime of an approach, but on other quality measures. We will see an example for that approach in Chapter 4 of this thesis.

It remains to close the main cycle of the algorithm engineering scheme and start a new iteration by finding an improved algorithmic design using the knowledge obtained during the steps of analysis, implementation, and experimentation.

A final by-product of the algorithm engineering cycle should be some kind of algorithm library that allows simpler verification of the findings and allows other groups to compare their new results against preceding algorithms.

The SAT community can be seen as a key example for the paradigm of algorithm engineering, due to several observations:

- Satisfiability problems are solved in a broad range of applications. This led to the creation of different benchmark sets. In its entirety, these sets cover a wide range of different applications, as well as several possible sizes of the formulae.

- Any common SAT solver is publicly available, often even with source code.

- An extensive practical analysis is the basis for determining the most performant SAT solver. Therefore, the SAT competition/SAT race[3] is organised every year.

- The basic branch-and-bound algorithm to solve SAT, which was published in 1962 [36], and which can still be identified in nearly any state-of-the-art SAT solver is, in fact, an extension to an algorithm from 1960. The original DP algorithm [37] was found to be too memory consuming in practice, leading to the development of the much more memory friendly DPLL algorithm.

[3] cf. `www.satcompetition.org` / `baldur.iti.kit.edu/sat-race-2015`

1.4 Outline

The basic preliminaries are introduced and defined in Chapter 2. We focus not only on the DPLL algorithm and the translation into SAT, but also on interesting problems that can be tackled when operating on unsatisfiable instances: the extraction of a Minimal Unsatisfiable Subset (MUS), and its dual, the extraction of a Maximal Satisfiable Subset (MSS).

The MUS extraction will play a central role in the next two chapters. Chapter 3 introduces a tool that enables the user to interactively extract different MUSes. Since all the automatic MUS extraction algorithms do not have any idea about the search space and the underlying application domain we give the user full control of the MUS extraction mechanism by interactively focusing on, or eliminating parts of, the search space. That way, we will not only help domain experts, who usually have good intuitions about the clauses relevant for good explanations, to find meaningful explanations of unsatisfiability, but also researchers and students in analyzing unsatisfiable instances and evaluating the effects of different heuristics for selecting deletion candidates. However, the main result of this tool is the development of a *meta instance* that speeds up the extraction algorithm by enabling the user to save unnecessary steps in the MUS extraction algorithms. The chapter is based on the published work in [45]. The idea of using a meta instance that is an extra SAT formula representing additional information about the search space, will be reoccurring through the thesis in multiple ways.

In Chapter 4 we introduce a technique to speed up the extraction of multiple MUSes. The improvements are based on techniques from the interactive MUS extraction, which are modified to fit the new purpose. We will show how to identify a set of clauses that exhibit a very important *block* property: either none or all members of the set are present in every MUS. With the help of this information, we put the focus on the computation of an often undesired by-product of the extraction of multiple MUSes: Minimal Correction Sets (MCSes). We will use these MCSes and a meta instance to speed up a single MUS extraction by reducing the search space considerably. We will incorporate the ideas into a state-of-the-art MUS enumerator and analyze the effect of exploring the *block* property in detail. The main results of this chapter are also presented in [156].

The remaining parts of the thesis will introduce two new applications to the satisfiability problem. Chapter 5 focuses on a SAT formulation of a problem from bioinformatics. The so-called Quartet Compatibility problem is a decision problem that describes whether the relationship information given in small trees can be preserved in a big tree. We introduce a novel SAT formulation for this problem, and show how to solve the related optimization problem of maximizing the amount of relationship information that can be preserved. Motivated by the earlier chapters, we will again use a meta instance to solve the optimization problem in one of the approaches. The practical analysis will cover a well-established benchmark set and compares our SAT formulation to state-of-the-art approaches from other optimization domains.

The second new application comes from the field of graph theory and graph drawing. Chapter 6 introduces a novel SAT formulation for the Book Embedding problem. In a book embedding, the vertices of a graph are placed on the spine of a book and the edges are assigned to pages, so that edges on the same page do not cross. We approach this problem of determining whether a graph can be embedded in a book of a certain number of pages from a SAT solving perspective by encoding it as a satisfiability problem. Since this is a problem that was not tackled by SAT before, we are required to create benchmark sets that model the possible inputs properly. We will show how to generate different benchmark sets based on underlying hypotheses we want to prove or disprove for particular graph classes. The material of this chapter is published in [153].

The thesis will finish with a short summary and conclusion in Chapter 7.

2 Preliminaries

This chapter defines the basic notation used in this thesis and introduces the most relevant concepts. Some additional (mostly application-based) definitions and algorithms will be introduced within later chapters. For basic concepts from graph theory and algorithms we refer the reader to comprehensive textbooks, like [33].

2.1 Propositional Logic

The SAT problem describes the problem, whether a given Boolean formula can be evaluated to `true`. To understand this problem in its full quality, we have to define what a Boolean formula is, and how to evaluate it.

A Boolean formula consists of a countably infinite set of variables $\mathcal{V}$. Together with the two constants `true` ($\top$) and `false` ($\bot$) the set of atoms $\mathcal{A} = \mathcal{V} \cup \{\texttt{true}, \texttt{false}\}$ is built. A propositional formula is constructed from the set of atoms $\mathcal{A}$, the unary operator $\neg$ for negation, and the binary connectives $\vee$ for disjunction, and $\wedge$ for conjunction and the parentheses (and).

Definition 2.1 (Syntax of Propositional Logic)**.**

- *The constants* $\top$ *and* $\bot$ *are formulae.*
- *Each variable* $v \in \mathcal{V}$ *is a formula.*
- *If* φ *is a formula,* $\neg\varphi$ *is a formula.*
- *If* φ *and* ψ *are formulae, then* $(\varphi \square \psi) : \square \in \{\vee, \wedge\}$ *is a formula.*

Further operations can be derived, e.g. implication ($\rightarrow$), equivalence ($\leftrightarrow$), and exclusive-or ($\oplus$).

If a is an atom, then a and $\neg a$ are literals, where a is a positive literal and $\neg a$ is a negative literal. The set of literals of a formula $\mathcal{F}$ is $\mathcal{L}(\mathcal{F})$. The polarity of a literal λ_a is positive if a is a positive literal. Otherwise, the polarity of λ_a is negative.

The semantics of propositional logic is defined relative to an assignment $\tau : \mathcal{V} \mapsto 0, 1$ of variables to the numbers 0 and 1, where 0 represents the truth value `false`, and 1 `true`. We will encounter both partial and complete assignments in

this thesis. The truth value of a formula φ under a complete assignment τ is defined recursively via an evaluation function $eval(\varphi, \tau)$:

Definition 2.2 (Semantics of Propositional Logic)**.** *For a complete assignment τ and formulae φ, ψ, we define the evaluation function $eval(\varphi, \tau)$ by*

- $eval(\bot, \tau) = 0$ *and* $eval(\top, \tau) = 1$
- $\forall v \in \mathcal{V} : eval(v, \tau) = \tau(v)$
- $eval(\neg\varphi, \tau) = 1eval(\varphi, \tau)$
- $eval(\varphi \vee \psi, \tau) = \max\{eval(\varphi, \tau), eval(\varphi, \tau)\}$
- $eval(\varphi \vee \psi, \tau) = \min\{eval(\varphi, \tau), eval(\varphi, \tau)\}$

The ground-breaking work by Davis and Putnam [37] started to consider Boolean formulae in conjunctive normal form (CNF). Any Boolean expression can be transformed into CNF [115].

2.1.1 Conjunctive normal form

A formula $\mathcal{F}$ in CNF is a set of clauses that are connected as conjunctions. Let $\mathcal{V}$ be the set of Boolean variables of $\mathcal{F}$. A clause $c \in \mathcal{F}$ is a disjunction of $|c|$ literals, whereas each literal is either a variable or its negation. A clause c is called unit if it contains only one literal ($|c| = 1$), binary if $|c| = 2$ and ternary if $|c| = 3$.

The formula $\mathcal{F}$ is satisfied by an assignment τ, if and only if every single clause $c \in F$ is evaluated to `true`. A complete assignment τ that evaluates the formula to 1 (`true`) is called *model*. A formula $\mathcal{F}$ is satisfiable if and only if at least one model exists; it is unsatisfiable if no such assignment exists.

As already stated, the satisfiability problem is one of the original NP-complete problems [32]. In general it suffices, that the formula $\mathcal{F}$ contains ternary clauses to be NP-complete. However, if $\mathcal{F}$ contains at most binary clauses, then the SAT problem can be solved in polynomial time.

2.1.2 Resolution

The resolution rule in Boolean logic is an inference rule that allows for the creation of a new valid clause [118]. It requires two clauses, $c_1, c_2 \in \mathcal{F}$, that contain for the variable v the complementary literals $\lambda, \neg\lambda$. Let $c_1 = (\neg\lambda \vee \alpha)$ and $c_2 = (\lambda \vee \beta)$, where α and β are both a disjunction of some literals $\lambda' \in \mathcal{L}(\mathcal{F})$. The derived clause $(\alpha \vee \beta)$ is called the resolvent of c_1 and c_2 on the variable v. If α and β contain a pair of complementary literals, then $(\alpha \vee \beta)$ is a tautology. It is satisfied for any assignment of variables and can be removed from the formula $\mathcal{F}$.

2.2 Basic SAT Solving Techniques

SAT solvers can generally be categorized into two distinct types, complete and incomplete solvers. Given a satisfiable formula $\mathcal{F}$ in CNF, both kinds of solvers are

able compute a satisfying assignment for $\mathcal{F}$. However, complete solvers can also prove unsatisfiability for formulae that cannot be satisfied by any assignment.

Incomplete solvers are mostly local search approaches [54], that use stochastic models to decide how the current assignment τ should be changed. They have been shown to be especially successful for satisfiable random SAT instances. Solving SAT with local search is beyond the scope of this thesis and we refer the reader to [74].

The central technique of modern complete SAT solvers is the DPLL algorithm [37, 36] extended by conflict-driven clause learning (CDCL). However, we will only introduce the DPLL algorithm here, since it covers already the basic techniques (like *unit propagation* and *variable decision*) we will be referring to throughout this work.

2.2.1 DPLL algorithm

The original variant of the DPLL algorithm that was presented by Davis and Putnam in 1960 [37] is based on propositional resolution. Two years later the first approach was reworked by Davis, Logemann and Loveland [36] to cope with memory restrictions [20]. The algorithm can be described by the following rules:

R1 *unit propagation*: If there is a unit clause $c = (\lambda_v)$, add the corresponding variable v with the polarity of λ_v to the assignment τ. Furthermore, remove all clauses that contain the literal λ_v and remove the literal $\neg\lambda_v$ from any clause it is contained in.

R2 *pure literal assignment*: If there is a variable v such that λ_v occurs only in positive polarity or only in negative polarity, add the variable v with the polarity of λ_v to the assignment τ. Furthermore, remove all clauses that contain the literal λ_v and remove the literal $\neg\lambda_v$ from any clause it is contained in.

R3 *decision and branching*: Choose one (unassigned) variable u of the formula $\mathcal{F}$ and examine both subproblems $(\mathcal{F} \wedge (\lambda_u))$ and $(\mathcal{F} \wedge (\neg\lambda_u))$.

The main idea of this algorithm is a branch-and-bound technique over all possible assignments of the variables of $\mathcal{F}$. The search space can be visualized as a tree, with the nodes being partial assignments, where the decision and branching rule was executed. Whenever the algorithm finds a contradiction (an empty clause), it prunes the search tree at a partial assignment, where the last decision had to be done. However, there is no information about the structure of this contradiction. This means that the DPLL algorithm probably runs into the same situation for another partial assignment, finding the same contradiction again. This can be avoided by the so-called *conflict driven clause learning*, an efficient extension of the DPLL algorithm [125].

Adding redundant constraints, that enables the SAT solver to find conflicts earlier in the search tree, can have a significant impact on the performance of the solver. We will observe this behavior in Chapter 5 of this thesis.

2.2.2 SAT solving with selector variables

Clause-selector variables (or short selector variables) can be used to augment a CNF formula $\mathcal{F}$ in such a way that standard Boolean satisfiability (SAT) solvers can manipulate and, in effect, reason about the formula's clauses without any modification to the solver itself. This augmentation has been used in many algorithms [96, 112, 91, 106].

Every clause c_i in a CNF formula $\mathcal{F}$ is augmented with a negated selector variable s_i to create $c'_i = (\neg s_i \vee c_i)$ in a new (augmented) formula $\mathcal{F}'$. Notice that each augmented clause c'_i is an implication: $c'_i = (s_i \rightarrow c_i)$. Assigning a particular s_i the value `true` implies the original clause, essentially enabling it. Conversely, assigning s_i to `false` has the effect of disabling or removing c_i from the set of constraints, as the augmented clause c'_i is satisfied by the assignment to s_i. This augmentation gives an unmodified, standard SAT solver the ability to enable and disable constraints as part of its normal search, checking the satisfiability of the enabled subsets of the formula $\mathcal{F}$ within a single backtracking search tree.

2.3 Minimal Unsatisfiable Subsets

The majority of this thesis will handle unsatisfiable formulae and the problems that arise when dealing with these. In the past few years, there has been an upswing of interest and research in a mechanism for providing information beyond the basic response that a given formula $\mathcal{F}$ is unsatisfiable: extraction of Minimal Unsatisfiable Subsets of constraints (MUSes), also called unsatisfiable cores. Given an unsatisfiable formula $\mathcal{F}$, an MUS of $\mathcal{F}$ is a subset of $\mathcal{F}$ that is (i) unsatisfiable and (ii) minimal in the sense that removing any one of its elements renders the remaining set of constraints satisfiable. It is defined as follows:

Definition 2.3 (Minimal Unsatisfiable Subset)**.** *A subset $M \subseteq \mathcal{F}$ is an MUS $\Leftrightarrow$ M is unsatisfiable and $\forall c \in M : M \setminus \{c\}$ is satisfiable*

An MUS minimizes an unsatisfiable constraint set to a "core" proof of its inconsistency. They are called "unsatisfiable cores" in some works, but we use the term MUS instead. Unsatisfiable formulae often contain many MUSes, and the presence of any one makes the formula unsatisfiable. Finding a single MUS is like pointing to a single hole to explain why a sieve does not hold the water, and "repairing" a single MUS (by relaxing constraints in the MUS to make it satisfiable) will not necessarily affect other MUSes, leaving the formula $\mathcal{F}$ infeasible.

This observation leads us to the closely related concept of Minimal Correction Subsets (MCS):

Definition 2.4 (Minimal Correction Subset)**.** *A subset $M \subset \mathcal{F}$ is an MCS $\Leftrightarrow$ $\mathcal{F} \setminus M$ is satisfiable and $\forall c \in M : (\mathcal{F} \setminus M) \cup \{c\}$ is unsatisfiable*

The removal of an MCS from the formula $\mathcal{F}$ restores its satisfiability ("corrects" it). The minimality is again not in cardinality, but in the fact that no proper subset of an MCS is a correction set itself. An MCS can also be defined as the complement of a Maximal Satisfiable Subset (MSS):

Definition 2.5 (Maximal Satisfiable Subset)**.** *A subset $M \subseteq \mathcal{F}$ is an MSS $\Leftrightarrow$ M is satisfiable and $\forall c \in (\mathcal{F} \setminus M) : M \cup \{c\}$ is unsatisfiable*

A very common problem regarding MSSes is finding the MSS with the largest cardinality. It is also well-known as the MaxSAT problem. Any MaxSAT solution is an MSS, but the converse does not necessarily hold.

2.3.1 Deletion-based MUS extraction

Many of the high-performance algorithms for MUS extraction from CNF instances can be classified as deletion-based. In such algorithms, starting from some unsatisfiable subset, we gradually try to remove clauses while making sure that the candidate set stays unsatisfiable. When the candidate set cannot be further reduced, we have arrived at an MUS.

Algorithm 2.1 A deletion-based MUS extraction algorithm using selector variables.

Require: an unsatisfiable SAT instance $\mathcal{F} = \{c_1, \ldots, c_m\}$ in CNF
Ensure: some MUS $\subseteq \mathcal{F}$
1: $\mathcal{F}' \leftarrow \emptyset$
2: **for all** clauses $c_i \in \mathcal{F}$ **do**
3: $\quad \mathcal{F}' \leftarrow \mathcal{F}' \cup \{c_i \vee \neg s_i\}$ ▷ add selector variables
4: **end for**
5: $\texttt{UC} \leftarrow \mathcal{F}$ ▷ clauses of unknown status
6: $\texttt{MUS} \leftarrow \emptyset$ ▷ this set will be the MUS in the end
7: **while** UC is not empty **do**
8: $\quad c_i \leftarrow$ select one clause $\in$ UC
9: $\quad \texttt{UC} \leftarrow \texttt{UC} \setminus \{c_i\}$
10: $\quad res = SAT(\mathcal{F}', \{s_i : c_i \in \texttt{UC} \cup \texttt{MUS}\})$ ▷ reduction attempt
11: $\quad$ **if** $res =$ `true` **then** ▷ unsuccessful reduction, c_i is critical
12: $\quad\quad \texttt{MUS} \leftarrow \texttt{MUS} \cup \{c_i\}$
13: $\quad$ **end if**
14: **end while**
15: **return** MUS

In the pseudocode, the set MUS collects the indices of clauses known to be critical. A clause c_i is said to be *critical* in an unsatisfiable subset F' if its deletion from F' will cause that $F' \setminus \{c_i\}$ becomes satisfiable. Thus, the critical clauses form an MUS, since the deletion of any of them results in a satisfiable subformula. The set UC contains clauses that have to be tested for criticality.

In line 10 of Algorithm 2.1 a SAT solver is called with two parameters: the formula $\mathcal{F}$ and a set of selector variables in positive polarity. This set of selector variables can be seen as forced variables assignments, that are given as an input to a SAT solver. As described in Section 2.2.2, the clauses corresponding to these selector variables will be enabled. All other clauses of the formula will be disabled by the *pure literal rule* of the underlying DPLL algorithm (see Section 2.2.1).

The complexity of MUS extraction algorithms is commonly measured in the number of necessary SAT calls. Obviously, the number of SAT calls for this algorithm is $\in O(m)$, i.e. linear in the size of the input formula $\mathcal{F} = \{c_1, \ldots, c_m\}$.

Clause set refinement

The advantage of deletion-based approaches lies in the possibility to execute larger reduction steps by analyzing the refutation proofs produced by the SAT solver, rather than only eliminating one constraint c from the US F'. When the SAT solver determines that a current subformula is unsatisfiable, the refutation proof returned by the solver is guaranteed to contain at least one MUS. If we generate such a proof and only select those clauses which were used in it, we are guaranteed to receive an unsatisfiable subset F'' smaller than or equal to the reduced clause set, $F'' \subseteq F' \setminus \{c\}$. This technique is often referred to as clause set refinement in the literature. It was made popular by Alexander Nadel [106], who uses it in the following simple, but very efficient algorithm for deletion-based MUS extraction:

Algorithm 2.2 A deletion-based MUS extraction algorithm using selector variables and clause set refinement.

Require: an unsatisfiable SAT instance $\mathcal{F} = \{c_1, \ldots, c_m\}$ in CNF
Ensure: some MUS $\subseteq \mathcal{F}$
1: $\mathcal{F}' \leftarrow \emptyset$
2: **for all** clauses $c_i \in \mathcal{F}$ **do**
3: $\quad \mathcal{F}' \leftarrow \mathcal{F}' \cup \{c_i \vee \neg s_i\}$ ▷ add selector variables
4: **end for**
5: $\langle res, proof \rangle = SAT(\mathcal{F}', \{s_1, \ldots, s_m\})$
6: **for all** clause $c_j \notin proof$ **do** ▷ clause set refinement
7: $\quad \mathcal{F}' \leftarrow \mathcal{F}' \cup \{\neg s_j\}$ ▷ disable the clauses that are not used for the proof
8: **end for**
9: `US` $\leftarrow \{c_i : c_i \in proof\}$ ▷ clauses of unknown status
10: `MUS` $\leftarrow \emptyset$ ▷ this set will be the MUS in the end
11: **while** `US` is not empty **do**
12: $\quad c_i \leftarrow$ select one clause $\in$ `US` $\setminus$ `MUS`
13: $\quad$ `US` $\leftarrow$ `US` $\setminus \{c_i\}$
14: $\quad \langle res, proof \rangle = SAT(\mathcal{F}', \{s_i : c_i \in$ `US`$\})$ ▷ reduction attempt
15: $\quad$ **if** $res =$ `true` **then** ▷ unsuccessful reduction, c_i is critical
16: $\quad\quad$ `MUS` $\leftarrow$ `MUS` $\cup \{c_i\}$
17: $\quad$ **else**
18: $\quad\quad$ `US` $\leftarrow \{c_j : c_j \in proof\}$
19: $\quad\quad$ **for all** clause $c_j \notin proof$ **do** ▷ clause set refinement
20: $\quad\quad\quad \mathcal{F}' \leftarrow \mathcal{F}' \cup \{\neg s_j\}$ ▷ disable the clauses that are not used for the proof
21: $\quad\quad$ **end for**
22: $\quad$ **end if**
23: **end while**
24: **return** `MUS`

In comparison to Algorithm 2.1 a few things have changed: In addition to the basic result whether the given formula is satisfiable or not, we are using the capability of a SAT solver to return a refutation proof for the unsatisfiability for the given formula $\mathcal{F}'$ (see line 5 and line 14 in Algorithm 2.2). For simplicity reasons we interpret the proof as a set of clauses that are used to deduce the unsatisfiability. In addition to the set `MUS`, that continues to collect the clauses known to be *critical*, we use the set `US` to store the clauses of the current unsatisfiable subset, which is always a superset of `MUS`.

The asymptotic complexity of the Algorithm 2.2 is $\in O(m)$ as well, but the actual number of SAT solver calls is in many cases much less than without the clause set refinement.

2.3.2 Model rotation

Another very useful technique for speeding up deletion-based MUS extraction is called model rotation and was introduced by Marques-Silva and Lynce [127]. This technique uses the model returned by the SAT solver to get a lot more information out of unsuccessful reduction attempts. Recall that an unsuccessful reduction attempt results in the detection of one critical clause c_i that has to be present in the MUS.

A critical clause c_i in an unsatisfiable clause set $\mathcal{F}$ is characterized by having an *associated assignment*, i.e. an assignment τ which satisfies $\mathcal{F} \setminus \{c_i\}$, but not c_i. Model rotation exploits this property by cheaply deriving from an associated assignment other assignments which can quickly be tested whether that model is associated to another clause, which then has to be critical and hence is a part of the MUS that is extracted. A formal proof of the fact that a clause is critical if and only if it has an associated assignment can be found in [10].

Algorithm 2.3 The recursive model rotation routine.

Require: an unsatisfiable SAT instance $\mathcal{F} = \{c_1, \ldots, c_m\}$ in CNF, critical clause $c_i \in \mathcal{F}$, model τ of $\mathcal{F} \setminus \{c_i\}$
Ensure: a set of further critical clauses $\texttt{crit} \subseteq \mathcal{F}$

```
1:  crit ← ∅
2:  for each variable v_j ∈ c_i do
3:      τ(v_j) ← 1 − τ(v_j)                          ▷ flip the variable v_j in the model τ
4:      unsat = {c_k : τ(c_k) = 0}   ▷ collect clauses not satisfied by rotated model τ
5:      for each clause c_l ∈ F do
6:          if unsat = {c_l} then                    ▷ τ is an associated assignment for c_l
7:              crit ← crit ∪ {c_l}
8:              crit ← crit ∪ modelRotation(F, c_l, τ)          ▷ recursive call
9:          end if
10:     end for
11:     τ(v_j) ← 1 − τ(v_j)                          ▷ flip the variable v_j back
12: end for
13: return MUS
```

In Algorithm 2.3 a variant called recursive model rotation [10] is presented.

Since this algorithm identifies other critical clauses without having to perform a costly SAT solver call, it should be executed after each unsuccessful reduction attempt in deletion-based MUS extraction to significantly decrease the number of SAT solver calls necessary to arrive at an MUS.

Siert Wieringa [146] gives an alternative description of recursive model rotation based on traversals of the flip graph, achieves further improvements to the algorithm based on these insights, and provides some analysis of benchmark instances which (partially) explain its high usefulness in practice. Marques-Silva and Lynce [127] determine in benchmarks that recursive model rotation is the single most effective technique for speeding up MUS extraction.

2.3.3 Insertion-based MUS extraction

A second class of MUS extraction algorithms works in a dual fashion to the deletion-based approach. In insertion-based approaches, we start with a satisfiable subset of the formula, gradually expanding it by additional clauses until our candidate set becomes unsatisfiable.

While insertion-based algorithms tend to require a higher number of SAT solver calls, an advantage of them is that they profit immensely from the use of incremental SAT solving. Incremental SAT solving describes a process of adding clauses to a SAT solver iteratively. That way, the SAT solver will store the data it derived (e.g. variable assignments, learnt clauses, heuristic measures) while processing a clause set, to reuse the data to solve later inputs more efficiently, since a large subset of the new formula was already known to the SAT solver. In an insertion-based approach which gradually adds more clauses to a candidate set, the individual calls to a SAT solver can therefore be performed incrementally at a much lower cost.

Algorithm 2.4 An insertion-based MUS extraction algorithm.

Require: an unsatisfiable SAT instance $\mathcal{F} = \{c_1, \ldots, c_m\}$ in CNF
Ensure: some MUS $\subseteq \mathcal{F}$

```
1:  MUS ← ∅                                   ▷ this set will be the MUS in the end
2:  F' ← F                                    ▷ candidates for critical clauses
3:  while |MUS| < |F'| do
4:      F'' ← MUS
5:      lastAppended ← ∅
6:      for all c_i ∈ F' \ MUS do
7:          res = SAT(F'' ∪ ¬c_i, {})         ▷ insertion attempt
8:          if res = true then
9:              F'' ← F'' ∪ {c_i}
10:             lastAppended ← {c_i}          ▷ detect the last non-redundant element
11:         end if
12:     end for
13:     MUS ← MUS ∪ lastAppended
14:     F' ← F''
15: end while
16: return MUS
```

The first algorithm for insertion-based MUS extraction was presented by Hans van Maaren and Siert Wieringa [141]. Their algorithm operates in rounds that repeatedly enlarge a satisfiable under-approximation until a new critical clause is found. During the enlargement, redundant clauses are detected and pruned away for the next iteration. A clause c is redundant in a CNF formula $\mathcal{F}$ if $\mathcal{F} \setminus \{c\} \cup \neg c$ is unsatisfiable. Here "$\neg c$" denotes the set of unit clauses $\{\{\neg\lambda_1\}, \ldots, \{\neg\lambda_k\}\}$ for the clause $c = (\lambda_1 \vee \ldots \vee \lambda_k)$. This result can be exploited to create a more constrained SAT instance without losing any satisfying assignments, often considerably reducing the time needed for each SAT solver run.

The last non-redundant element (line 10 in Algorithm 2.4) that could be added during inflation before $\mathcal{F}$ becomes unsatisfiable, is identified as a critical clause. The outer *while* loop (line 3 to 15) terminates when `lastAppended` is empty, which is the case when every element remaining in $\mathcal{F} \setminus$ `MUS` is redundant. The number of SAT solver calls is $\in O(k * m)$ with k being the size of the MUS that is extracted.

Marques-Silva and Lynce developed a variant [127] that improves the number of SAT solver calls to $O(m)$ for this paradigm as well, showing that insertion-based approaches are not asymptotically slower than deletion-based algorithms.

2.3.4 Hybrid MUS extraction

Incorporating the ideas from both, the deletion-based and insertion-based MUS extraction, Marques-Silva and Lynce [127] proposed the Algorithm 2.5. The hybrid algorithm uses the efficient redundancy check by van Maaren and Wieringa [141] and the clause set refinement from Nadel [106]. However, clause set refinement cannot be used every time the deletion of one candidate resulted in an unsatisfiable subset S of $\mathcal{F}'$, since the redundancy check in line 6 adds new unit clauses to the formula. These unit clauses can mask other clauses of $\mathcal{F}'$ in a sense that without the redundant unit clauses, other elements of $\mathcal{F}'$ would have been used to prove the unsatisfiability of S.

Algorithm 2.5 A hybrid MUS extraction approach that incorporates the ideas from deletion and insertion-based approaches.

Require: an unsatisfiable SAT instance $\mathcal{F} = \{c_1, \ldots, c_m\}$ in CNF
Ensure: some MUS $\subseteq \mathcal{F}$

```
1:  MUS ← ∅                                   ▷ this set will be the MUS in the end
2:  F' ← F                                    ▷ candidates for critical clauses
3:  while F' is not empty do
4:      c_i ← select one clause ∈ F'
5:      F' ← F' \ {c_i}
6:      ⟨res, proof⟩ = SAT(MUS ∪ F' ∪ ¬c_i, {})        ▷ redundancy check
7:      if res = true then
8:          MUS ← MUS ∪ {c_i}
9:      else if proof ∩ ¬c_i = ∅ then
10:         F' ← proof \ MUS                  ▷ clause set refinement
11:     end if
12: end while
13: return MUS
```

3 Visualizing MUS extraction

3.1 Introduction

Finding small reasons for unsatisfiability of a SAT formula by extracting a Minimal Unsatisfiable Subset (MUS) is used in many different settings. Examples of applications include inconsistency measurement [76, 151], type error debugging [40, 4], debugging of relational specifications [138, 139], analysis of over-constrained temporal problems [90], axiom pinpointing in description logics [123], software and hardware model checking [2], among many others [132, 112].

Aided by the broad range of applications a remarkable amount of work in recent years caused a significant progress in efficient extraction of a MUS. The development of *model rotation* (see Section 2.3.2), an improved use of resolution [107] and refutation proofs [8] or completely new algorithms [100] are examples for the fact, that the field of MUS extraction has become an emerging research field in the SAT community, leading to the introduction of a special MUS track in the SAT competition 2011[1]. However, most of the formulas that can be tackled relatively easily by SAT solvers are very hard for MUS extraction algorithms, since the number of SAT solver calls to extract one MUS is $\in O(m)$ (see Section 2.3) with m being the number of clauses in a CNF formula $\mathcal{F}$.

In general the algorithms for MUS extraction can be characterized as *constructive* (insertion-based), *destructive* (deletion-based) or a *hybrid* combination of both (see Section 2.3), but they all focus exclusively on the number of clauses or clause sets (called *groups*) [11] present in the minimal explanation.

In this chapter we introduce a completely new approach to guiding the basic destructive MUS extraction algorithm, which is due to the effectivity of the *clause set refinement* (see Section 2.3.1) and *model rotation* still among the top performing algorithms for industrially relevant instances [126]. The central idea of destructive MUS extraction, which was first proposed more than 20 years ago [29, 5], is to perform a series of *reduction steps*, moving into smaller unsatisfiable subsets $\mathcal{F}'$

[1] http://satcompetition.org/2011/#tracks

until all subsets $\mathcal{F}'' \subset \mathcal{F}'$ are satisfiable.

The idea of *interactive MUS extraction* is to give a user full control over the individual reduction steps, providing an interface for interactively focusing on or eliminating parts of the search space. This is done in our tool MUStICCa, an abbreviation for "MUS extraction with interactive choice of candidates", by reverting to intermediate results, the non-minimal unsatisfiable subsets (US), and exploring new parts of the search space by choosing alternative deletion candidates. This feature will not only help domain experts (who usually have good intuitions about the clauses relevant for good explanations) to find meaningful explanations of unsatisfiability, but also researchers and students in analyzing unsatisfiable instances and evaluating the effects of different heuristics for selecting deletion candidates.

During the extraction of different MUSes, the user (or any other MUS extraction algorithm) has to perform "unnecessary" deletion tests. These tests are unnecessary, since the same information that is obtained with this step is already known due to other reduction steps within another part of the search space. We will show how these unnecessary steps can be avoided by an efficient representation of all the clause criticality information that was gained anywhere in the search space. By that we are able to reuse this information to not only save expensive SAT solver calls for the user, but also to pave the way for the ideas that were used to boost state-of-the-art MUS enumeration approaches (see Chapter 4).

This chapter is based on the work published in [45] and is organized as follows. In the next section we describe a useful classification scheme of clauses in unsatisfiable clause sets, that will be used throughout this work. The scheme divides the set of clauses regarding the information, whether they are present in all MUSes, only in some MUSes, or in no MUSes at all. Section 3.3 introduces the tool's user interface and the possibilities to guide the interactive MUS extraction procedure. We focus on the graphical representation of the search space, and of the criticality information of clauses, and introduce the possibilities to start automatic procedures that reduce the current unsatisfiable subset to reach an MUS finally. The main contribution of this chapter is the development of an additional SAT formula on selector variables, the so-called *meta instance*. It is introduced in Section 3.4. We will show in Section 3.5 how the meta instance can be extended to represent not only the criticality information of clauses, but also to enforce properties on the MUS that is extracted. A short summary concludes this chapter.

3.2 Classification of Clauses

Typical MUS extraction problems contain not only a single MUS, but rather a large set of different ones. These MUSes of an unsatisfiable SAT instance tend to overlap, containing a common core set of clauses, which is satisfiable on its own. To arrive at some MUS, this common core must be made unsatisfiable by adding some combination of other clauses. Kullmann, Lynce and Marques-Silva [27] build on this observation to formally distinguish different degrees of necessity for clauses with respect to MUSes. These definitions introduce some very useful vocabulary

for talking about the role of different clauses in interactive MUS extraction. We use $\mathcal{F}$ for some unsatisfiable SAT formula, and $MUSes(\mathcal{F})$ for the set of all MUSes of the formula $\mathcal{F}$.

Definition 3.1 (Necessary Clause)**.** *A constraint $c \in \mathcal{F}$ is called* necessary *if and only if $\mathcal{F} \setminus \{c\}$ is satisfiable. Thus, the constraint c is* critical *in every MUS $M \in MUSes(\mathcal{F})$.*

The set of necessary clauses forms the mentioned core that is contained in every MUS, and can therefore be written as $\bigcap MUSes(\mathcal{F})$. Given m clauses in $\mathcal{F}$, $\bigcap MUSes(\mathcal{F})$ can be computed by m calls to a SAT solver by checking if $\mathcal{F} \setminus \{c\}$ is satisfiable for each constraint c. This process can be speeded up by a large factor by using the aforementioned techniques of *model rotation* (see Section 2.3.2) and *clause set refinement* (see Section 2.3.1).

The dual to this strongest notion of necessity is a very weak notion of redundancy of a clause. It describes the fact that unsatisfiability is maintained when the single clause c is removed from the original formula $\mathcal{F}$. Thus, there exists at least one MUS where c is not present. However, it does not ensure that two unnecessary clauses can be removed at the same time.

Definition 3.2 (Unnecessary Clause)**.** *A constraint $c \in \mathcal{F}$ is called* unnecessary *if and only if $\mathcal{F} \setminus \{c\}$ is unsatisfiable. Thus, $\exists M \in MUSes(\mathcal{F})$ with $c \notin M$.*

The strongest redundancy of a clause is defined by the following.

Definition 3.3 (Never Necessary Clause)**.** *A constraint $c \in \mathcal{F}$ is called* never necessary *if and only if c is unnecessary in all unsatisfiable subsets $\mathcal{F}' \subseteq \mathcal{F}$. Thus, $c \notin \bigcup MUSes(\mathcal{F})$.*

The definition implies that we can safely remove any clause known to be *never necessary*, although this might make the unsatisfiability a lot harder to prove because they can contribute to much shorter proofs of unsatisfiability, which might be a lot easier to understand than the proof of a MUS. We will not explore this issue further here, as it is a general problem with the approach of taking minimal subsets as formal approximations to small explanations of infeasibility.

3.3 The Application

By default, the graphical user interface (see Figure 3.1) consists of three main view components. The central component is a representation of the current knowledge about the search space. Explored USes are inspected in a separate *US view* component, which also provides the interface for starting reduction steps. The third component is responsible for administering automated reduction procedures, the *agents*.

The interface is based on the open-source Kahina framework [44] for graphical debugging, which was chosen because it already provided the needed view components and native support for managing a database of computation steps as nodes in a graph structure.

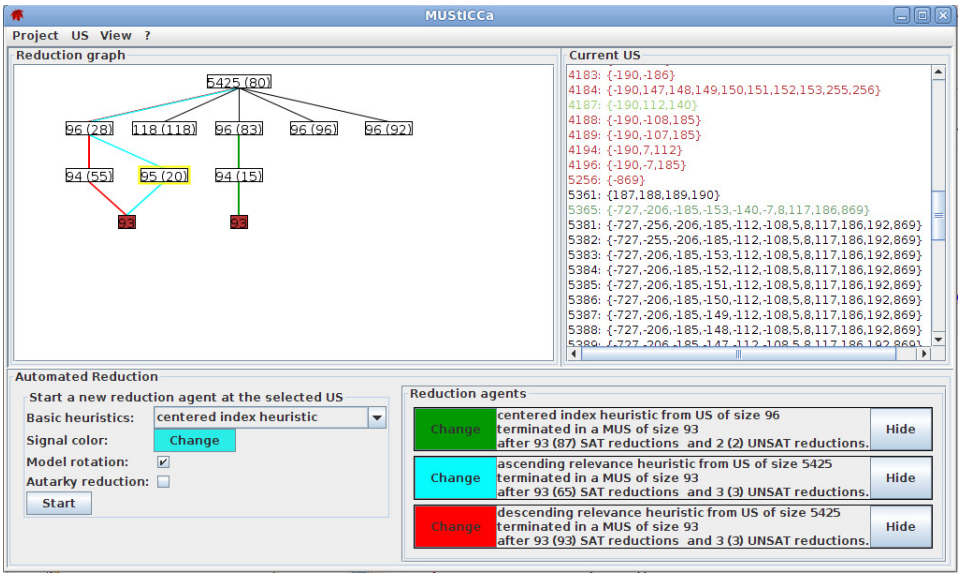


Figure 3.1. Screenshot of MUStICCa's default user interface.

Since deletion-based MUS extraction can be viewed as a downward traversal of the powerset lattice for an unsatisfiable clause set $\mathcal{F}$, the explored part of the search space is modeled by means of a *reduction graph*, which is a subsemilattice of the powerset lattice whose edges represent successful reduction attempts. The powerset lattice will be introduced in a more detailed fashion in Section 4.3.

A deterministic deletion-based MUS extraction algorithm will only create a linear structure of US states connected by the subset relation. But as soon as we allow the exploration of multiple deletion alternatives, the structure of the encountered USes will probably branch out. Since there can be different sequences of deletions leading to the same US, the structure is a directed acyclic graph (dag) which can be defined as:

Definition 3.4 (Reduction graph)**.** *Let $\mathcal{F}$ be an unsatisfiable clause set. A reduction graph $R = (V, E)$ for $\mathcal{F}$ is defined as a tuple of vertices V and edges E. V is a set of unsatisfiable subsets $V \subset 2^{\mathcal{F}}$, and an edge $(V_1, V_2) \in E$ exists if and only if V_2 was the result of removing some non-critical clause from V_1, possibly followed by clause set refinement.*

The USes encountered on reduction paths are the obvious choice for defining the meaningful steps of an MUS extraction process, and were therefore chosen as the nodes in the graph. Since every US is a unique subset of the original clause set, it can be represented and uniquely identified by a set of clause indices representing the clauses it contains.

In the visualization of USes, the number on each node gives the size of the corresponding US, followed in brackets by the number of clauses of unknown criticality. MUSes are marked in red, dark green color marks non-MUSes where all reduction options have been explored, and light green color marks USes where all clauses are

known to be either critical or unnecessary. The unnecessary clauses are unexplored reduction options. Note that not every unnecessary clause c is unusable; we only know that there is at least one MUS which does not contain c.

Whenever a node in the reduction graph view is selected, the US corresponding to the node is displayed in the *US view*. The default format for clauses in the *US view* consists of the clause ID (numbered according to the order in the input DIMACS file) and the set of integers representing its literals. The clauses in the *US view* are color-coded to reflect their criticality status: critical clauses are displayed in red, explicitly reduced clauses in a dark green, other unnecessary clauses in a lighter green, and clauses of unknown status in black. The color codes make it easy to spot interesting deletion candidates for reduction steps, which are started by double-clicking on clauses in the *US view*.

For advanced interactions, one or more clauses in the *US view* can be selected, and are then highlighted by a yellow background color. The selection of interesting clauses is supported by a selection refinement interface in the form of a hierarchy of sub-menus in the *US view*'s context menu. This menu provides the options of manually executing reduction attempts followed by *model rotation*, and two different options for executing reduction operations on sets of clauses: (i) *Semi-automation* initiates a batch processing of deletion attempts for all the clauses in the current selection. This helps to quickly open up several new branches in the reduction graph at once, or to speed up series of criticality checks. (ii) *Simultaneous reduction* is an attempt to delete all the currently selected clauses at once. If the attempt was successful, the new node (or link) will appear in the reduction graph, just like in the case of deleting a single clause. If a simultaneous reduction attempt fails, this only yields very weak criticality information, since it might have been possible to delete some of the clauses under the condition that others stay in.

3.3.1 Automation through reduction agents

During a process of interactive extraction, a user will often want to quickly explore parts of the search space without having to manually execute hundreds of reduction attempts, especially in contexts where domain knowledge has not yet become relevant. For this purpose, our tool MUStICCa includes an automated reduction mechanism in the form of *reduction agents* which in essence act like autonomous additional users who were given sets of simple instructions. In addition to predefined reduction agents which emulate standard deletion-based algorithms, it is possible to implement own heuristics to select deletion candidates as well.

The most important option in the dialog for creating and starting new reduction agents serves to select one of the predefined heuristics from a drop-down menu. Each new agent is initialized with a random signal color that can freely be redefined. The new reduction agent starts at the US that is currently selected in the reduction graph, and runs until it has determined an MUS. The downward path of an agent through the powerset lattice (see Section 4.3 for more details) is highlighted in its signal color.

3.4 Saving unnecessary SAT Solver Calls

Whenever at least two different branches in the reduction graph exist, the possibility of unnecessary SAT solver calls arises. We start our analysis of this problem by considering the powerset lattice of another small example instance $\{c_1, c_2, c_3, c_4\}$. To simplify the example, we will assume that clause set refinement is of no relevance here, so that each single reduction step will at most lead one step down in the powerset lattice. Assume a state (see Figure 3.2) of interactive reduction where the reduction graph spans the edges colored in black (fat), and the grey edges and subsets have not yet been explored. In the subset $\{c_2, c_3, c_4\}$, the clause c_2 has just been determined to be critical, which we again symbolize by coloring critical clauses and the corresponding transition edges in red, fat and dashed.

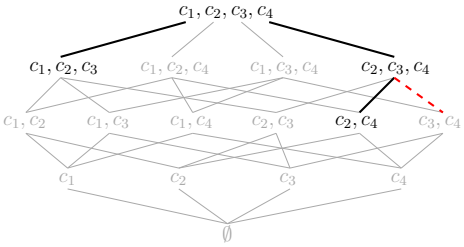


Figure 3.2. c_2 is critical in $\{c_2, c_3, c_4\}$

The criticality of c_2 in $\{c_2, c_3, c_4\}$ is propagated downwards in the reduction graph via the black edge to the subset $\{c_2, c_4\}$, but not to the other subsets. Now assume that in the next reduction step, we successfully delete c_1 from the set $\{c_1, c_2, c_3\}$. The propagation and the reduction attempt together lead us to the following new state of the reduction graph shown in Figure 3.3. We do not derive the criticality information of c_2 in the new US $\{c_2, c_3\}$, causing us to waste a call to a SAT solver when we attempt to delete c_2 later. This type of problem is quite virulent in larger instances with a lot of different MUSes, because it is likely to occur whenever one branch of the reduction graph arrives at a US which would also be reachable by a reduction starting in some other node where some unsuccessful reduction attempts have already been performed.

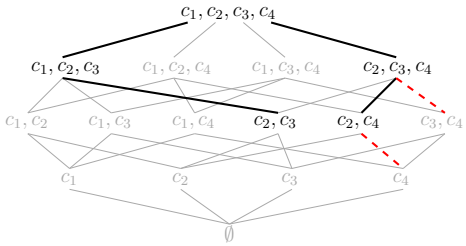


Figure 3.3. propagated criticality

The approach developed in this chapter aims to achieve the storage of the criticality information by explicitly representing and processing this information in the form of Boolean constraints over selector variables. Before, selector variables s_i had the only purpose of extending the clauses c_i of the original instance in such a way that the clause c_i can effectively be switched off by assigning the corresponding selector variable s_i to `false`. In this chapter, we will go beyond that by reusing the selector variables as direct handles for talking about the presence or absence of clauses in subsets of the formula $\mathcal{F}$. As we have seen in our motivating example, whenever we find during our reduction attempts a set of clauses $\{c_1, c_2\}$ that cannot be reduced at the same time (otherwise c_2 would not be critical for $\{c_2, c_3, c_4\}$), we can save this information by an additional clause in the form of $(s_1 \vee s_2)$. It

contains positive literals for the two selector variables corresponding to c_1 and c_2. This clause enforces the presence of either c_1 or c_2 in any subset we consider in the future. We will be using a collection of such clauses to store all the derived criticality information, and use it to efficiently determine all the clauses already known to be critical in new unsatisfiable subsets.

All these additional clauses will be saved in the so-called *meta instance*. In the following we will introduce shortly what kind of information we can store, and how to use this information during the execution of MUSes. We will use the meta instance to express conditions for clause set unsatisfiability. Thus, if the meta instance is satisfiable (under given assumptions), the subset that corresponds to the given assumptions is unsatisfiable. Analogously, if the meta instance is unsatisfiable under given assumptions, the subset that corresponds to the assumptions is satisfiable.

We will see, that the *meta instance* will contain only clauses with *selector variables* in positive polarity. Thus, without further assumptions the additional SAT formula is always satisfiable by the *pure literal rule* (see Section 2.2.1). Assuming the meta instance contains the clause $c = (s_i \vee s_j \vee s_k)$. The instance is unsatisfiable if and only if at least one clause (w.l.o.g. c) is unsatisfiable, which is the case if all variables of c are assumed to be negative.

3.4.1 Criticality information via unsuccessful reductions

Assume that we attempt to delete a clause c_i from an unsatisfiable clause set $U \subseteq \mathcal{F}$, and that this results in a satisfiable instance $U \setminus \{c_i\}$. Thus, c_i is now known to be critical in the US U. This criticality information depends on all the clauses $\overline{U} = \mathcal{F} \setminus U$ which had already been reduced or fallen away in the state where we attempted the deletion. This is often described as the complement of U. The new information can be expressed as $(\bigwedge \neg s_j) \rightarrow s_i : c_j \in \overline{U}$. This can directly be expressed as a single clause in CNF $(\bigvee s_j \vee s_i)$ for $c_j \in \overline{U}$, which is added to the meta instance.

The added constraint $(\bigvee s_j \vee s_i)$ can be read as preventing all the elements of the set $\{\overline{U} \cup s_i\}$ from being removed at the same time, by requiring that one element of the set must be present in any unsatisfiable subset. In our motivating example, we would learn the meta clause $(s_1 \vee s_2)$, expressing that either c_1 or c_2 must be present in any unsatisfiable subset of $\{c_1, c_2, c_3, c_4\}$.

We use that criticality information during the MUS extraction in the following way. Assume again that in the next reduction step, we successfully reduce $\{c_1, c_2, c_3\}$ to $\{c_2, c_3\}$. If we now want to determine whether c_2 is critical in $\{c_2, c_3\}$, we can perform a first check for the satisfiability of $\{c_3\}$ by testing the satisfiability of the meta instance under the assumptions $\{\neg s_1 \wedge \neg s_2 \wedge \neg s_4\}$. Given that the meta instance contains the constraint $(s_1 \vee s_2)$, the meta instance will be unsatisfiable, indicating that the clause set $\{c_3\}$ is satisfiable. Thus, c_2 must be critical in $\{c_2, c_3\}$. As intended, we do not need to waste a much more costly SAT solver call on the original instance any longer.

Using *model rotation*, we potentially find an entire set $\{c_{i_1}, \ldots, c_{i_k}\}$ of k new critical clauses in the current US U. This leads to the new constraint $(\bigwedge \neg s_j) \rightarrow (s_{i_1} \wedge \ldots \wedge s_{i_k}) : c_j \in \overline{U}$, which could be converted into k constraints in CNF: $(\bigvee s_j \vee s_{i_1})$, ... , $(\bigvee s_j \vee s_{i_k})$ for $c_j \in \overline{U}$. From the perspective of our meta instance the result of a *model rotation* is equivalent to k sequential unsuccessful deletion attempts of the clauses $\{c_{i_1}, \ldots, c_{i_k}\}$.

Unsuccessful reduction of a clause set

Assume the simultaneous deletion of a set of k clauses $\{c_{i_1}, \ldots, c_{i_k}\}$ from a US $U = \{c_1, \ldots, c_m\}$ leads to a satisfiable problem. We gain the knowledge that the k clauses may not be deleted together. In other words, at least one of the k clauses is critical for a subset $U' \subseteq U$. This information can be expressed with the constraint $(\bigwedge \neg s_j) \rightarrow (s_{i_1} \vee \ldots \vee s_{i_k}) : c_j \in \overline{U}$. Converting this constraint into CNF leads to the addition of a single clause $(\bigvee s_j \vee s_{i_1} \vee \ldots \vee s_{i_k})$ for $c_j \in \overline{U}$.

3.4.2 Successful reductions

Assume that our attempt to delete a clause c_i from an unsatisfiable clause set $U = \{c_1, \ldots, c_m\}$ is successful. We arrive at a new unsatisfiable subset $U' \subseteq U \backslash \{c_i\}$. Equality of the sets U' and $U \setminus \{c_i\}$ hold if and only if *clause set refinement* has not caused any further clauses to be deleted from U.

With this result we know that we can safely delete all the clauses in $U \setminus U'$ from any US that contains the clauses in U'. If we express this connection in terms of selector variables, we can simply say that it suffices for unsatisfiability that all clauses in U' are present, i.e. if $(\bigwedge s_j) : c_j \in U'$ holds.

Note that this formula is of a very different nature from what we have derived for the case of unsuccessful reductions. The meta instance shall encode conditions for unsatisfiability, but we are now dealing with a constraint that is a *sufficient* condition, not a *necessary* condition as before.

In fact, $(\bigwedge s_j) : c_j \in U'$ is a minterm which would need to be added as a disjunct to the meta instance to represent one way to fulfill the condition that a subset is unsatisfiable. But this disjunctivity means that we cannot simply integrate this information by adding a set of clausal constraints to the meta instance.

As for unsuccessful reductions, the case of successful simultaneous reduction has in principal no difference to the case of a sequence of successful deletions of a single clause. Assume that a set of constraints $\{c_{i_1}, \ldots, c_{i_k}\}$ is simultaneously deleted from an US U. If the result set $U' := U \setminus \{c_{i_1}, \ldots, c_{i_k}\}$ is unsatisfiable, we can again store the minterm $(\bigwedge s_j) : c_j \in U'$, which would be the result of k sequential reductions of the constraints $\{c_{i_1}, \ldots, c_{i_k}\}$. This observation also allows us to not consider the consequences of *clause set refinement* in this section, since it does not change anything except for the size of the derived minterm.

3.4.3 Blocks of selector variables

The clauses that are added to the meta instance tend to become very large, since each of them enumerates all the individual selector variables which correspond to the elements of the complement of some satisfiable subset. Even the most extreme case of two large meta clauses only differing in a single literal is very common, since this is what results from unsuccessful attempts to delete different clauses from the same unsatisfiable subset, and *model rotation*.

Using an automated scheme which groups the selector variables that often occur together in meta clauses into *blocks*, we introduce *block variables* b_i as shorthands for referring to large disjunctions of selector variables in our meta instance. This approach can be viewed as using the block representation as a compression scheme. The space savings achieved in this way turn out to be so significant that this measure alone makes usage of the meta instance on benchmark instances of any interesting size much more efficient. In Table 3.1 the size of the original and the compressed *meta-instance* are shown for 10 different instances. The first set of instances is taken from a hardware verification domain, specifically automated test pattern generation [128]. The second set of instances is taken from the Daimler testset for automotive product configuration [132]. For both sets a significant difference in the number of literals, that is in fact the sum of all clause sizes, can be observed. The presented values are calculated by running three reduction agents on every configuration until each of them detected an MUS. The difference in the number of literals is expected to grow even further for every new criticality information that is obtained by executing further reduction steps.

			uncompressed		compressed	
test set	instance	$\lvert\mathcal{F}\rvert$	clauses	literals	clauses	literals
atpg	`bf1355-127.cnf`	7,306	439	3,143,824	441	8,186
	`bf1355-462.cnf`	7,305	469	3,353,061	472	8,493
	`bf1355-530.cnf`	7,305	253	1,827,249	255	7,813
	`bf1355-666.cnf`	7,305	343	2,466,969	345	7,882
	`bf1355-741.cnf`	7,307	247	1,784,903	249	7,803
Daimler	`C168_FW_SZ_66.cnf`	5,425	280	1,493,047	284	6,355
	`C202_FW_SZ_103.cnf`	10,283	428	4,252,915	437	12,193
	`C208_FA_SZ_121.cnf`	5,278	97	508,990	99	5,474
	`C210_FW_RZ_57.cnf`	7,405	76	560,980	78	7,559
	`C220_FV_SZ_55.cnf`	5,753	916	4,989,549	923	10,133

Table 3.1. The number of clauses and literals of the original and compressed *meta instance*. For every input 3 reduction agents executed a series of reduction steps until detecting an MUS.

Apart from the compression quality, the clause sets represented by the *blocks* can be interpreted in a more semantically motivated way as well. As the block structure progressively becomes more granular, the block structure tends to group together clauses which often occur together as subtrees of refutation proofs. The reason for this connection between inferred *blocks* and refutation proofs lies in the fact that the operation of clause set refinement prunes unsatisfiable subsets to only

those clauses which occur in some unsatisfiability proof.

Assume that the clauses $(s_1 \vee \ldots \vee s_i \vee s_k)$ and $(s_1 \vee \ldots \vee s_i \vee s_l)$ are the only clauses added to the meta instance. These clauses are added due to the fact that the constraints c_k and c_l are both critical in the US $F \setminus \{c_1, \ldots, c_i\}$ (see Section 3.4.1). By inserting a fresh variable b_i representing a *block* of selector variables we obtain the following clauses: $(s_1 \vee \ldots \vee s_i \vee b_i)$, $(\overline{b_i} \vee s_k)$ and $(\overline{b_i} \vee s_l)$.

We will not go into more details here about the algorithm that is used to determine the blocks of selector variables, nor will we prove any properties for the equisatisfiability of the original meta instance and the new meta instance, which contains the *blocks* of selector variables. We refer the reader to the work of Dellert [43], a co-supervised student research project, that culminated in the tool presented in this chapter.

3.5 Additional Meta Constraints

By representing the criticality information of clauses in unsatisfiable clause sets in the meta instance we have a powerful tool at our disposal. However, we can extend the current meta instance by additional constraints to specify additional properties of the MUSes we extract. This general idea leads to a number of possible applications, some of which are presented in the following.

The applications proposed in this section demonstrate the potential benefits of adding constraints over selector variables. However, none of them has actually been implemented as part of our tool. The main reason for this is the unclear connection to interactive MUS extraction caused by the lack of monotonicity in some interesting properties. For interactive reduction, the purpose of the meta instance is to axiomatize unsatisfiability of clause sets, whereas now, we are additionally using it to axiomatize further criteria for the sets we want to find. If these criteria do not hold for each superset of a set where they hold, the satisfiability check against the meta instance rejects unsatisfiable subsets that would have reached the criterion by future reductions. Therefore, some of the ideas presented in this section only make sense in the mode of full SAT solving against meta constraints (without any given assumptions), where they can be used to encode constraints on the combinations of selector variables. The models of the meta instance then correspond to USes that incorporate the additional encoded properties.

3.5.1 Inclusion or exclusion of specific clauses in USes

One big problem we want to solve with interactive MUS extraction is the loss of clauses in USes which were supposed to be part of the desired explanation of infeasibility. By allowing to revert reduction decisions and to explore alternative paths that do not result in the deletion of the desired clauses the problem can potentially be solved. However, because this approach does not change anything about the fact that we are making reduction attempts which can in principle cause arbitrary other clauses to fall away by *clause set refinement*, a lot of trial and error

might still be involved until the user arrives at an MUS containing a specific set of desired clauses.

It becomes possible to explicitly specify such requirements of (un)desired clauses using the meta instance. If we have a subset $\{c_{i_1}, \ldots, c_{i_k}\}$ which we desire to be part of the extracted MUS, we can simply add the unit constraints $(s_{i_1}), \ldots, (s_{i_k})$ to the meta instance in order to ensure that the desired clause set must be part of each unsatisfiable subset we encounter. We do not necessarily arrive at an MUS by using this method, but we are guaranteed that the USes we arrive at are minimal under the condition that all the clauses in $\{c_{i_1}, \ldots, c_{i_k}\}$ are present.

An analogous scenario arises when we are sure that some clauses, that are part of the original instance, cannot be part of any interesting MUS. Just as we enforce the presence of a subset $\{c_{i_1}, \ldots, c_{i_k}\}$, we can also enforce its absence by adding the unit constraints $(\neg s_{i_1}), \ldots, (\neg s_{i_k})$ to the meta instance. However, this is a non-monotonic property, as with these constraints added, the unsatisfiability of the meta instance will not tell us anymore that all subsets of the tested clause set violate the desired property. It might well be that we end up in a proper US after removing some more clauses.

3.5.2 Expressing GMUS extraction

The standard approach to solve the task of GMUS (short for grouped MUS) extraction is to assign the same selector variable to all clauses of a group. Doing so ensures that certain groups of clauses are always removed together. The possibility to define additional constraints over the selector variables allows us to emulate this behavior despite the fact that we identify every clause by a different selector variable.

Assume that the clauses $\{c_1, \ldots, c_4\}$ belong to the same group. We can enforce these clauses to be either all absent or present by adding the clauses $(\neg s_1 \vee s_2)$, $(\neg s_2 \vee s_3)$,$(\neg s_3 \vee s_4)$, and $(\neg s_4 \vee s_1)$ to the meta instance. The circular dependency between these binary clauses enforces that as soon as one of the $s_i : 1 \leq i \leq 4$ is assigned to `false`, repeated unit propagation forces all the three other s_i to also be assigned to `false`, and analogously for the assignment to `true`.

The advantages of encoding the groups implicitly rather than using a single selector variable become visible when we generalize the notion of a GMUS. A standard GMUS problem effectively defines a partition of the instance, as every clause is assigned to exactly one group. Assume that we have an application where these groups overlap, e.g. two groups $\{c_1, c_2, c_3\}$ and $\{c_1, c_4, c_5\}$ where the presence of c_1 only forces the clauses from either one of the groups to be present, but the absence of c_1 forces all clauses from both groups to be absent. Such connections between groups might be applied in identifying incorrect hardware or software components that share parts.

Given our example the necessary conditions can easily be expressed by the meta constraints $(\neg s_1 \vee s_2 \vee s_4)$, $(\neg s_2 \vee s_3)$, $(\neg s_3 \vee s_1)$, $(\neg s_3 \vee s_2)$, $(\neg s_4 \vee s_5)$, $(\neg s_5 \vee s_1)$, and $(\neg s_5 \vee s_4)$. The same disjunctive dependency of the clause c_1 could not be expressed within the standard GMUS paradigm. Depending on the application, even more complex connections between overlapping groups might be desirable. For such purposes, being able to freely define dependencies between clauses via

additional meta constraints is very helpful.

3.5.3 Enforcing limits on MUS sizes

Much more complex constraints than the previously presented are possible as well. One of the potentially most attractive options is to express an upper bound k on the size of the desired MUS, allowing us to explicitly check whether any MUS of a given size exists. The basic procedure for adding such a parametrization is simple: we use an efficient encoding for *at-most-k constraints* (such as the ones described by Sinz [131] or Ben-Haim et al. [13]) to express the requirement $\sum_i s_i \leq k$, and add the resulting clauses to the meta instance.

Running the SAT solver on the meta instance would then only generate US candidates of a size at most k. Once the meta instance is unsatisfiable by the clauses that were added from unsuccessful reduction attempts and the at-most-k constraints, this means that no further USes of size $\leq k$ exist.

3.6 Summary

Finding small reasons for unsatisfiability of a SAT formula by extracting minimal unsatisfiable subformulae is an emerging research field in the SAT community. However, the common MUS extraction algorithms focus exclusively on the number of clauses. All clauses are treated equal, because the algorithms do not have any information about the meaning of any of the clauses in the underlying problem. In this chapter we presented our tool MUStICCa that was designed to provide the user with the possibility to guide the well-known deletion-based MUS extraction algorithm through the powerset lattice towards different MUSes using domain-specific information to reach meaningful explanations of unsatisfiability.

MUStICCa is the first application that reuses criticality information from other parts of the search space to avoid unnecessary execution steps. We are confident that our tool is helpful to domain experts in analyzing inconsistencies in SAT formulae. Moreover, MUStICCa can be used to create and evaluate new heuristics to find "good" deletion candidates.

However, the main takeaway from this chapter is rather the methodical background that was introduced here and will be heavily used throughout this thesis. Starting from the basic idea of explicitly modeling the search space of deletion-based MUS extraction as a graph of USes, this reduction graph was conceived as a subset lattice. We have analyzed the information that can be gained from successful and unsuccessful reduction attempts in the deletion-based MUS extraction paradigm, and developed a scheme for learning and retrieving this information for a maximum of information reuse during interactive search space exploration. We have shown that a clausal meta instance over the selector variables can store information about all the encountered satisfiable subsets, which enables us to retrieve the clauses which are implied to be critical in some US.

The general idea of expressing connections between clauses as meta constraints over selector variables was shown to have other potential applications such as an axiomatization of the desired MUS size or a generalized variant of group-based

MUS extraction. Throughout the remaining chapters of this work we will be using several approaches that define additional constraints for selector variables to enforce desired properties (mainly regarding the size) of satisfiable or unsatisfiable subsets of a given formula $\mathcal{F}$.

Block-based representations were developed mainly in order to derive more compact representations of a clausal meta instance, but they are interesting from a theoretical perspective as well. The clauses that are represented via one block of selector variables are clauses that either occur in the same refutation proofs for unsatisfiable subsets or are not present at all. The idea to group clauses together is the basis for the ideas that are able to develop a technique to boost the enumeration of MUSes, which will be presented in the next chapter. In contrast to the *block* notion represented here, we will generalize it to incorporate sets of clauses, that show an equal behavior regarding their presence and absence in the $MUSes(\mathcal{F})$. In the new sense, the clauses of one *block* are either all together in any MUS $M \in MUSes(\mathcal{F})$ or none of the clauses is present in the MUS M.

4 Improving MUS enumeration

4.1 Introduction

Many algorithms in common applications of constraint systems cover problems of finding a satisfying assignment, commonly known as model. Applications require either a single model, a set of these or even all models for a given problem [122, 145, 20].

On the other hand, constraint sets without any model are target for the "infeasibility analysis" algorithms, which can be partitioned into two groups. Their tasks are a) finding a - preferably very large - part of the constraint set that is still satisfiable and b) locating the area of the constraint set where the reason for unsatisfiability lies. These two categories are known by different names: Maximal Satisfiable Subsets (MSS) and Maximum Feasibly Subset (MaxFS) for the former and Minimal(ly) Unsatisfiable Subset or Core (MUS/MUC) and Irreducible Infeasible Subsystem (IIS) for the latter. Although "Max / Min" and "SAT / UNSAT" seem to be completely opposite, they are strongly connected via a hitting set relationship [39, 117].

Minimal reasons of infeasibility in linear programming [140, 64, 29] and in artificial intelligence [41] have been studied since the 1980s. Finding MUSes in SAT covers a lot of applications, including debugging of relational specifications [138] or type errors [4] and model checking [142, 31].

The relationship of Maximal Satisfiability and Minimal Unsatisfiability is based on the following: any satisfiable subset of an infeasible constraint set cannot completely contain any unsatisfiable subset (US) of the formula, and thus must at least exclude one constraint from every US. Searches for results of these can be guided by the results of the other. Algorithms exist that compute MUSes with the help of MSSes [4], vice versa [53] and even ones that use non-minimal USes to support MSS solution finding to finally produce MUSes [92]. The latest improvements for (partially) enumerating MUSes are based on this as well [89, 116].

We propose a novel approach to improve the partial enumeration of MUSes by

using the information which clauses are very similar according to their presence or absence in MUSes. We first define basic terms and concepts (Section 4.2), before describing the related work (Section 4.3) and especially the MARCO algorithm [89] (Section 4.4). We present the new techniques in Sections 4.5, 4.6 and an extensive practical analysis (Section 4.7) before concluding the work and discussing some possibilities for future research (Section 4.8).

4.2 Hitting Set Duality of MUSes and MCSes

Recall that an MCS is a subset of an infeasible constraint set (SAT formula) whose removal from that formula results in a satisfiable set of constraints. Thus, the removal "corrects" the infeasibility. MCSes are minimal in the sense that any proper subset does not have that correcting property.

The following unsatisfiable formula $\mathcal{F}$ in CNF is used as a running example throughout this chapter. We refer to the 4 clauses of the formula $\mathcal{F}$ as $c_1, \ldots, c_4$.

Example 1:

$$\mathcal{F} = \bigwedge c_i : 1 \leq i \leq 4$$

$$c_1 = (\overline{x_1}) \quad c_2 = (x_1 \vee x_2) \quad c_3 = (x_2) \quad c_4 = (\overline{x_2})$$

MUSes($\mathcal{F}$)
$\{c_1, c_2, c_4\}$
$\{c_3, c_4\}$

MCSes($\mathcal{F}$)	MSSes($\mathcal{F}$)
$\{c_4\}$	$\{c_1, c_2, c_3\}$
$\{c_1, c_3\}$	$\{c_2, c_4\}$
$\{c_2, c_3\}$	$\{c_1, c_4\}$

The formula $\mathcal{F}$ has 2 MUSes and 3 MCS/MSS pairs. For simplicity we denote any MUS, MCS and MSS as a set of clauses throughout this work. Note that any MCS is a complement of an MSS and vice versa.

Furthermore, the set of MUSes of a formula $\mathcal{F}$ and the set of MCSes of $\mathcal{F}$ are "hitting set duals" of one another. All MUSes of $\mathcal{F}$ form a set that is equivalent to the set of all irreducible hitting sets of the MCSes and analogously the set of MCSes is equivalent to all irreducible hitting sets of the MUSes. The following Theorem 4.1 is proven formally in [22].

Theorem 4.1. *Let $\mathcal{F}$ be an unsatisfiable formula, $MUSes(\mathcal{F})$ the set of all minimal unsatisfiable subsets of $\mathcal{F}$ and $MCSes(\mathcal{F})$ the set of all minimal correction sets.*

1. *$U \subset \mathcal{F}$ is an MUS $\Leftrightarrow$ U is an irreducible hitting set of $MCSes(\mathcal{F})$*
2. *$C \subset \mathcal{F}$ is an MCS $\Leftrightarrow$ C is an irreducible hitting set of $MUSes(\mathcal{F})$*

We recall an intuitive explanation for this from Liffiton et al. [91] here. An unsatisfiable formula $\mathcal{F}$ has at least one MUS M. Due to the minimality of an

MUS it can be made satisfiable by simply deleting a single clause of it. Therefore, in order to make the whole formula $\mathcal{F}$ feasible, one has to "dispose" its MUSes by removing at least one clause from every MUS. An MCS corresponds to a set of clauses that accomplishes this: its removal restores the satisfiability of $\mathcal{F}$. Thus, any MCS has to contain at least one element of every MUS of $\mathcal{F}$ and due to its minimality the irreducibility of the hitting set is obtained. A similar argument can be found for the fact that MUSes are irreducible hitting sets of MCSes.

Example 2: We explain the property using the example from above.

		MCSes		
		$\{c_4\}$	$\{c_1, c_3\}$	$\{c_2, c_3\}$
MUSes	$\{c_1, c_2, c_4\}$	x	x	x
	$\{c_3, c_4\}$	x	x	x

Whenever an MUS and MCS have a clause in common they intersect each other, which is denoted by an "x". Each clause in the intersection of all MUSes infers an MCS of size one, in this example $\{c_4\}$. These clauses are also called *necessary* clauses (see Section 3.2). All other MCSes are of larger size.

4.3 Related Work

The existing work on MUS enumeration can be viewed as methodical explorations of power sets, since each subset of the whole constraint set, in our case the SAT formula F, is a possible MUS or MSS/MCS. The goal of an MUS enumeration algorithm is the exploration of the complete power set of the input formula. The power set can be visualized as a lattice in a Hasse diagram (see Figure 4.1). Each level of the diagram contains subsets of a fixed size and edges connect sets with their immediate supersets and subsets in the consecutive levels.

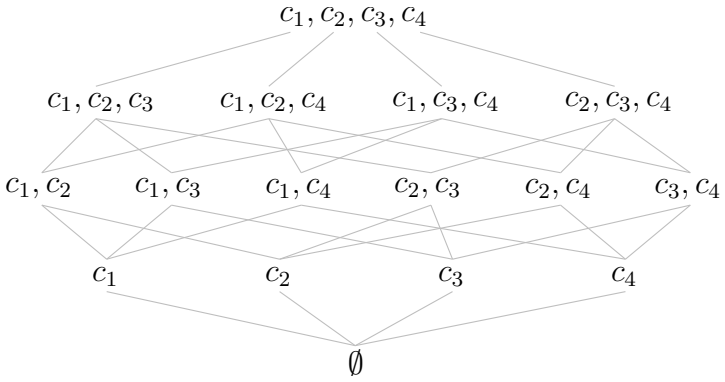


Figure 4.1. Hasse diagram of the power set lattice for a formula of four constraints

Exploring this power set lattice, an enumeration algorithm will determine the feasibility of various subsets of the formula $\mathcal{F}$. A function that assigns each subset

its feasibility

$$feas : S \subseteq \mathcal{F} \rightarrow \{SAT, UNSAT\}$$

can be represented by coloring each node in the power set lattice with a color representing its satisfiability. A "map" will be created with exactly two regions: satisfiable and unsatisfiable subsets. For our running example from Section 4.2, the fully colored map is shown in Fig. 4.2.

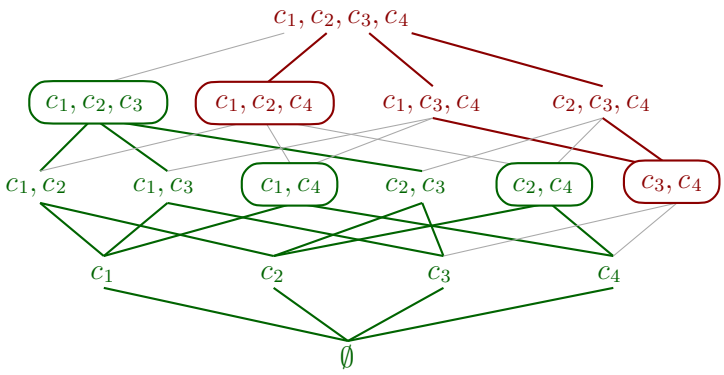


Figure 4.2. The colored Hasse diagram of our running example, the unsatisfiable (satisfiable, respectively) region is marked red (green, respectively). Furthermore the MUSes and MSSes are marked with a box in their respective color.

For a given infeasible constraint set (in our case the SAT formula $\mathcal{F}$), the following simple facts and their connection to the corresponding Hasse diagram can be stated:

- Every subset of the constraint set is either SAT or UNSAT. Trivially, $\mathcal{F}$ is unsatisfiable and the empty set of constraints is satisfiable. In the diagram we will mark nodes that correspond to satisfiable subsets green, and nodes that correspond to unsatisfiable subset red.
- If a subset X is satisfiable (unsatisfiable, respectively), then all of it subsets (supersets, respectively) are satisfiable (unsatisfiable, respectively). For every green (red, respectively) node in the diagram all of its neighbors in the next lower (higher, respectively) level have the same color.
- Given the minimality (maximality, respectively) of MUSes (MSSes, respectively), all subsets (supersets, respectively) have to be satisfiable (unsatisfiable, respectively). The MUSes and MSSes are marked in the diagram with a box in the corresponding color.

The problems that analyze the infeasibility of constraint sets, can be viewed as problems of coloring the whole lattice or only a single point and therefore, depending on the satisfiability of the corresponding point, all of its subsets/supersets. Whereas algorithms that enumerate all MUSes/MSSes color the whole lattice, an MUS extraction algorithm will result in coloring a node of the diagram and all of its supersets red, and an MSS extraction algorithm (or Max-SAT calculation) will

result in coloring a node of the diagram and all of its subsets green.

In comparison to the amount of research done on extracting single MUSes, the existing work on enumerating MUSes of infeasible constraint sets is relatively limited.

Several MUS enumeration approaches that are specialized on a particular type of constraints are known. In the constraint programming domain, methods for computing the MUSes of over-constrained NCSPs were developed by Gasca, et al. [57]. NCSPs are numerical constraint satisfaction problems that consist of numeric variables $\in \mathbb{R}$ and constraints in the form of (in-)equalities between linear or polynomial combinations of elements of the variable set. This technique enumerates all possible subsets of the constraint problem and prunes unnecessary collections of subsets in between. The pruning is done based on rules, that are structure specific to NCSPs and therefore are not easily generalizable.

In the field of operations research, in particular for linear programs (LP) and integer linear programs (ILP), MUSes are known as *Irreducible Inconsistent Subsystems* (IISes). The methods [140, 64] that compute all IISes of an LP rely on techniques like constructing polytopes and the simplex method. Since these are specific to linear programming, these methods cannot be generalized easily to be applicable for other constraint systems.

In the following we will introduce more generalizable approaches more specifically, since they are more relevant as predecessors for the MARCO algorithm.

4.3.1 Enumerating subsets

Early approaches to enumerate MUSes are based on an exhaustive search on the power set lattice. The explicit enumeration uses an HS-tree data structure [75] with pruning rules to avoid multiple satisfiability tests for a single subset. Every node in the tree corresponds to a subset S of the formula $\mathcal{F}$, and every child node is labeled with a subset $S' \subset S$. In a depth-first fashion the subsets are tested for unsatisfiability. Each unsatisfiable node whose children are all found to be satisfiable is marked as an MUS. This can be visualized in the power set lattice (see Figure 4.1) as traversing top-down in a depth-first fashion, calling a SAT-solver for each node (subset S of $\mathcal{F}$) on the path, backtracking when a satisfiable S is found, and marking unsatisfiable nodes, that are the "low-points" (thus, do not have unsatisfiable subsets) as MUSes. Several improvements could be made for this technique [69], but the iterative SAT-solver calls and the explicit enumeration of all possible subsets of $\mathcal{F}$ are a performance bottleneck [4].

Junker [80] briefly described a method that focuses on extracting "preferred explanations", e.g. MUSes with regard to preferences on constraints. The method operates on branching on each constraint $c_i \in \mathcal{F}$. The first direction of the branch eliminates c_i and enumerates recursively MUSes without c_i if the remaining subset of $\mathcal{F}$ is unsatisfiable. The second direction is an unspecified mechanism, which removes other constraints to find MUSes that contain c_i in the end. Effectively, this is an exhaustive search as well. The major drawback, like before, is the large number of SAT-solver calls which is required to determine the infeasible region of the power set lattice explicitly.

4.3.2 The algorithm CAMUS

The hitting set MUS/MCS duality discussed in Section 4.2 can be used to enumerate MUSes. The CAMUS algorithm by Liffiton and Sakallah [91] works in two phases: the first phase computes all MSSes/MCSes in a "top-down" search through the power set lattice of a constraint set by searching level-by-level (a level contains all subsets of a particular size) for satisfiable subsets. These subsets must not be subsumed by some larger satisfiable subset found at a higher level, since that invalidates the MSS property. The second phase is a hitting set approach that is completely independent of any constraint solver. It starts when all MSSes/MCSes are found.

To enumerate all MCSes in the first phase a *selector variable* approach (see Section 2.2.2) is used. By adding cardinality constraints on the selector variables which are assigned to `false`, the search is restricted to a particular size of the MCSes. Using a linear progression on the bound of the size of computed MCSes, CAMUS computes MCSes of increasing size (MSSes of decreasing size) in the first phase. For every found MCS M a new clause

$$\bigvee_{i:c_i \notin M} s_i$$

is added that ensures that all proper subsets of the corresponding MSS are *blocked* in the future.

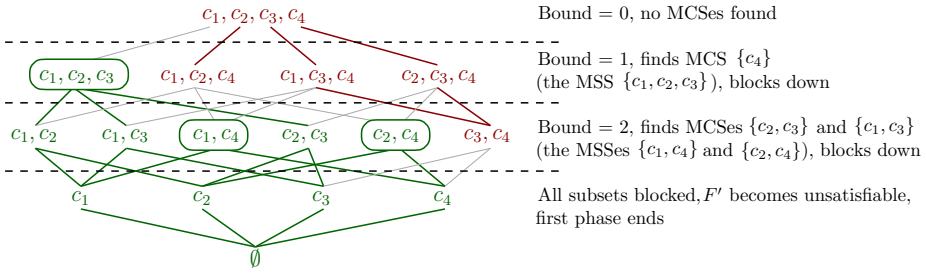


Figure 4.3. Illustration of the first phase of the CAMUS algorithm on our running example F. The levels are explored one-by-one from top to bottom.

The execution of the first phase of the CAMUS algorithm is shown in Figure 4.3. The SAT and UNSAT regions are known implicitly due to the augmented formulation of $\mathcal{F}'$, since the models of $\mathcal{F}'$ correspond to satisfiable subsets within the power set lattice. For every bound k any reported model corresponds to an MCS of size k. Every found MCS causes the *blocking* of its proper subsets. When no further models can be found the remaining subsets in that level can be implicitly marked as unsatisfiable. The combination of these implicitly marked unsatisfiable regions together with the blocked subsets of found MSSes/MCSes eventually covers the whole power set lattice and the first phase of CAMUS terminates.

The search for MCSes was boosted by Grégoire et al. [65] using an incomplete local search oracle to identify possible MCSes. This method is more efficient than complete MCS enumeration but still relies on it for completeness and correctness.

A significant shortcoming of CAMUS is the possible intractability of the first phase. A constraint system may have an exponential number of MCSes. Enumerating all MCSes in the first phase before computing the MUSes in the hitting set phase is not possible within limited time in these cases. To cope with this intractability, the concept of a *partial correction set* (PCS), a subset of some MCS, was introduced by Liffiton and Sakallah [91]. The technique can be incorporated in the first phase of CAMUS in the following way: every found MCS is reduced to a specified size $k' \leq k$ by eliminating arbitrary constraints from the MCS to reach that size. By eliminating elements from an MCS to build a PCS, we construct an over-approximation of an MSS. By *blocking* proper subsets of this over-approximation subsequently we mark a (potentially large) region of the power set lattice as already inspected, and cut the search space, in which many more MCSes could have been located. The hitting set approach of the second phase is unaltered and still computes valid MUSes. Thus, using PCSes instead of MCSes sacrifices completeness for the sake of reporting a subset of MUSes faster.

4.3.3 The algorithm DAA

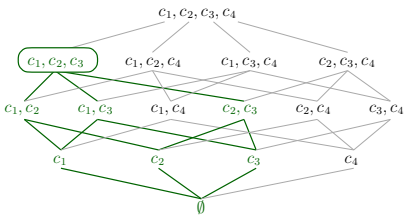
Another algorithm that explores the hitting set duality between MCSes and MUSes was developed by Bailey and Stuckey [4]. The so-called Dualize and Advance (DAA) method is an incremental hitting set approach that computes both, MCSes and MUSes, during its execution. In every iteration a satisfiable subset (in the beginning the empty set) is grown into an MSS. Its complement MCS is added to the set of already found MCSes. On this set of MCSes all minimal hitting sets (possible MUSes) are computed. Each MUS candidate is then checked for unsatisfiability. Each candidate is either a known MUS and filtered out, a new unexplored MUS that will be reported immediately, or a new unexplored satisfiable subset. In the latter case, it is used as a starting point for the computation of a new MSS in the next iteration.

This technique of testing candidate MUSes while "jumping" from one solution in the power set lattice to another possible solution is very similar to the technique we developed to speed up the MUS enumeration. Therefore, we will show in the following an example execution of DAA on our running Example 1.

Executing DAA on our example

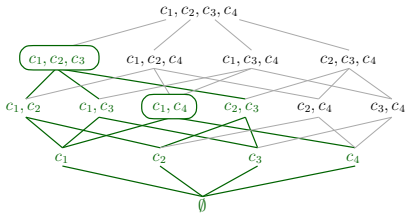
Initially, no MCSes have been found. Thus, the first known satisfiable subset is the empty set. It will be used to obtain an MSS via a so-called **grow** method (see Section 4.4 for details). The complementary MCS is added to the set of known MCSes, from which all minimal hitting sets are computed.

- **grow**$(\emptyset) \rightarrow \{c_1, c_2, c_3\} \leftrightarrow$ MCS:$\{c_4\}$
- MCSes $= \{\{c_4\}\}$
- **HittingSets**(MCSes)$\rightarrow \{\{c_4\}\}$



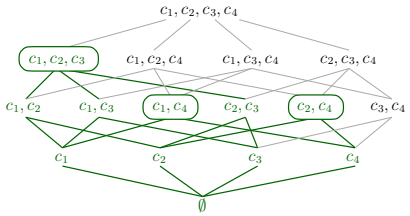
The hitting set, which is a minimal point in the region above the known satisfiable region, turns out to be a satisfiable subset. It is taken as the next starting point for the **grow** method.

- **grow**$(\{c_4\}) \rightarrow \{c_1, c_4\} \leftrightarrow$ MCS:$\{c_2, c_3\}$
- MCSes $= \{\{c_4\}, \{c_2, c_3\}\}$
- **HittingSets**(MCSes)$\rightarrow \{\{c_2, c_4\}, \{c_3, c_4\}\}$



One of these hitting sets ($\{c_3, c_4\}$) is unsatisfiable and is reported as an MUS. The other is taken as the next starting point for the **grow** method.

- **grow**$(\{c_2, c_4\}) \rightarrow \{c_2, c_4\} \leftrightarrow$ MCS:$\{c_1, c_3\}$
- MCSes $= \{\{c_4\}, \{c_2, c_3\}, \{c_1, c_3\}\}$
- **HittingSets**(MCSes)$\rightarrow \{\{c_1, c_2, c_4\}, \{c_3, c_4\}\}$



One of these hitting sets ($\{c_3, c_4\}$) has already been found and is skipped, the other ($\{c_1, c_2, c_4\}$) is unsatisfiable and is reported as an MUS. Since no other hitting sets are found to be satisfiable, the algorithm terminates. All MUSes have been enumerated.

In contrast to CAMUS, the DAA algorithm reports MUSes and MCSes interleaved. It produces MUSes before all MCSes are computed and avoids the possible intractability of the first phase from CAMUS. However, the number of hitting sets computed in each iteration can be exponential as well, resulting in another intractability.

4.4 The MARCO Algorithm

Independently of each other Previti and Marques-Silva [116] and Liffiton and Malik [89] developed two very similar algorithms called eMUS and MARCO. Both can be seen as *partial* (or incremental) MUS enumerators, not replacing any state-of-the-art complete MUS enumerators like CAMUS [91] but providing a viable alternative for satisfiability instances where the full enumeration is computationally infeasible in limited time. Their major advantage is in reporting the first MUS as quickly as state-of-the-art MUS extractors and reporting further MUSes with a similar delay.

We will describe first how the power set lattice which represents the whole search space of an MUS enumeration algorithm can be implemented, such that we create a data structure which gives us the possibility to obtain a not yet visited node of the Hasse diagram.

Let *expl* be a function that assigns any subset $X \subseteq \mathcal{F}$ the value `true`, if the subset is **un**explored and its feasibility is **un**determined and the valie `false` oth-

erwise:

$$expl : X \subseteq \mathcal{F} \rightarrow \{\texttt{true}\ (\text{unexplored}), \texttt{false}\ (\text{explored})\}$$

We can then define the *Unexplored Subset Problem* for a constraint set, in our case a SAT-formula $\mathcal{F}$, a set of known-unsatisfiable subsets U, and a set of known-satisfiable subsets S:

Definition 4.2. *Let $\mathcal{F}$ be a constraint set, U the set of its known-satisfiable subsets, and S the set of known-unsatisfiable subsets of $\mathcal{F}$. A subset $X \subseteq \mathcal{F}$ is said to be dominated by a subset $Y \in U \Leftrightarrow X \supseteq Y$, and a subset $X \subseteq \mathcal{F}$ is said to be dominated by a subset $Z \in S \Leftrightarrow X \subseteq Y$. In other words, the subset X is dominated by an element of U or S if and only if the feasibility of the subset is known and thus is "explored".*
The solution to the Unexplored Subset Problem *is a subset $X \subseteq \mathcal{F}$ that is not dominated by any element of U or S if X exists or* `NULL` *otherwise.*

Note that the elements of U and S do not have to be minimal or maximal, respectively. Furthermore, U and S can be empty.

Any function $f : X \subseteq \mathcal{F} \rightarrow \{\texttt{true}, \texttt{false}\}$ can be implemented by a propositional formula over $|\mathcal{F}|$ variables. The *Unexplored Subset Problem* can be solved as a Boolean CNF formula, called `map`. The formula `map` is defined by its set of variables and a set of clauses in the following way:

- The formula contains $|\mathcal{F}|$ variables, for every constraint $c_i \in \mathcal{F}$ there exists a variable s_i (similar to the concept of *selector variables*).
- Any complete assignment of the variables correspond to a subset $X \subseteq \mathcal{F}$ by the variables assigned to `true`:

  $$s_i = \texttt{true} \Leftrightarrow c_i \in X$$

- For every known-unsatisfiable subset $Y \in U$ the formula contains the clause

  $$\bigvee_{i:c_i \in Y} \neg s_i$$

  This clause ensures that every superset of Y is marked as explored, since all of them have to be unsatisfiable.

- For every known-satisfiable subset $Z \in S$ the formula contains the clause

  $$\bigvee_{i:c_i \notin Z} s_i$$

  This clause ensures that every subset of Z is marked as explored, since all of them have to be satisfiable.

Lemma 4.3 (Correctness)**.** *Given the above formulation of the* Unexplored Subset Problem *for the constraint system $\mathcal{F}$ as a formula called* `map`*. Any model of* `map` *represents a subset $X \subseteq \mathcal{F}$, which is not dominated by any subset in the set of known-unsatisfiable subsets U, nor by any subset in the known-satisfiable subsets S. In other words: the subset X is unexplored.*

Proof. We have to consider both cases: (i) X is not dominated by any subset in U, (ii) nor by any subset in S.

Case (i): Let m be a model of `map` and its corresponding subset of $\mathcal{F}$ is dominated by a known-unsatisfiable subset $Y \in U$. Thus, $\forall c_i \in Y, s_i$ is assigned to `true` in the model m. However, the formula `map` contains a clause $\bigvee_{i:c_i \in Y} \neg s_i$, which is not satisfied by the model. Thus, the assumption cannot be true.

Case (ii): Let m be a model of `map` and its corresponding subset of $\mathcal{F}$ is dominated by a known-satisfiable subset $Z \in S$. Thus, $\forall c_i \notin Z, s_i$ is assigned to `false` in the model m. However, the formula `map` contains a clause $\bigvee_{i:c_i \notin Z} s_i$, which is not satisfied by the model. Thus, the assumption cannot be true.

By the combination of both cases we prove that any model m of `map` corresponds to a subset of $\mathcal{F}$, which is not dominated by any subset in the set of known-unsatisfiable subsets U, nor by any subset in the known-satisfiable subsets S. □

Lemma 4.4 (Completeness)**.** *Given the above formulation of the* Unexplored Subset Problem *for the constraint system $\mathcal{F}$ as a formula called* `map` *that is unsatisfiable if and only if the set of known-unsatisfiable subsets U contains all MUSes of $\mathcal{F}$ and the set of known-satisfiable subsets S contains all MSSes of $\mathcal{F}$. In other words: all subsets of $\mathcal{F}$ are explored.*

Proof. The formula `map` is unsatisfiable if and only if every complete variable assignment does not satisfy at least one clause in `map`. A specific clause cls in `map` is not satisfied by complete variable assignments that correspond to subsets of $\mathcal{F}$, which are dominated by the subset that caused the addition of the clause cls. Thus, every complete assignment does not satisfy at least one clause in `map` if and only if every subset of $\mathcal{F}$ is dominated by one known-unsatisfiable subset in U or one known-satisfiable subset in S. Every possible subset of $\mathcal{F}$ is dominated by one subset in U or one subset in S if and only if U contains all MUSes of $\mathcal{F}$ and S contains als MSSes of $\mathcal{F}$. □

Theorem 4.5. *The above defined formulation of the Boolean CNF formula* `map` *presents a correct and complete solution for the* Unexplored Subset Problem.

Proof. Follows from Lemma 4.3 and Lemma 4.4. □

4.4.1 Implementation

We have shown before that the *Unexplored Subset Problem* can be solved by formulating it as a SAT problem. For any algorithm that enumerates MUSes that power set is initially unexplored, hence `map` $= \top$. Every time the algorithm explores a subset $X \subseteq \mathcal{F}$ and its feasibility, it should be marked as explored in the `map` by adding at least one clause that "blocks" the the corresponding model.

Assume that an algorithm explores $\mathcal{F}$ itself and determines it to be unsatisfiable. To block that particular subset (and possible supersets) for the future, the corresponding model $\{s_1 = s_2 = s_3 = s_4 = \texttt{true}\}$ has to be forbidden by adding a single clause to `map`: $(\neg s_1, \neg s_2, \neg s_3, \neg s_4)$. The `map` represents then, that the subset equal to the whole formula $\mathcal{F}$ is explored, and all proper subsets of $\mathcal{F}$ remain unexplored. Note that $\mathcal{F}$ would be added to the known-unsatisfiable subsets U. Since

U does not necessarily contain only Minimal Unsatisfiable Subsets that is valid. In fact, during the execution of `MARCO` only MUSes will be added to U and only MSSes will be added to S.

In the following we will present possible tasks that can be executed in the context of exploring the power set lattice. They will be presented as subroutines which will be later used in the pseudocode of the algorithms throughout this chapter.

- **getModel**(`map`) →subset $X \subseteq \mathcal{F}$
  Assuming `map` is satisfiable, which indicates that at least one subset of $\mathcal{F}$ is unexplored, this method will obtain an unexplored subset of $\mathcal{F}$ by returning a model of `map`. Any constraint solver can be used for that.

- **SAT**(subset X) $\rightarrow \{\text{SAT}, \text{UNSAT}\}$
  By sending a subset $X \subseteq \mathcal{F}$ to a simple constraint solver, we can check whether X is satisfiable or not.

- **shrink**(unsatisfiable subset X) $\rightarrow$ MUS M
  We extract a MUS M from a given known-unsatisfiable subset X ($M \subseteq X$) by executing a single MUS extraction algorithm. The simplest possible variant is given in Algorithm 4.1.

- **grow**(satisfiable subset X) $\rightarrow$ MSS M
  Analogously to before we extract an MSS M from a given known-satisfiable subset X ($M \supseteq X$) by executing a single MSS extraction algorithm. The simplest possible variant is given in Algorithm 4.2. Note that this is *not* equivalent to solving the Max-SAT problem, since an arbitrary MSS does not need to have the largest possible cardinality.

Algorithm 4.1 The shrink method

Input: $X \subseteq \mathcal{F}$
Output: one MUS of $\mathcal{F}$
1: **for** $c \in X$ **do**
2: **if** $X \setminus \{c\}$ is unsatisfiable **then**
3: $X \leftarrow X \setminus \{c\}$
4: **end if**
5: **end for**
6: **return** X

Algorithm 4.2 The grow method

Input: $X \subseteq \mathcal{F}$
Output: one MSS of $\mathcal{F}$
1: **for** $c \in \mathcal{F} \setminus X$ **do**
2: **if** $X \cup \{c\}$ is satisfiable **then**
3: $X \leftarrow X \cup \{c\}$
4: **end if**
5: **end for**
6: **return** X

- **complement**(MSS/MCS M) $\rightarrow$ MCS/MSS M'
  Any subset can be seen as a complete variable assignment of `map`. By flipping all variable assignments, we get the complement of an MSS, which is an MCS and vice versa.

- **blockMUS**(MUS M) $\rightarrow$ clause cls
  By adding a new clause to the `map` we can mark one region of the power set lattice as explored. The clause cls will be

$$\bigvee_{i:C_i \in M} \neg s_i$$

Algorithm 4.3 The basic MARCO algorithm

Input: unsatisfiable formula $\mathcal{F} = \{c_1, \ldots, c_n\}$
Output: MCSes and MUSes of $\mathcal{F}$ as they are discovered

```
1:  map ← BoolFormula(s_1, ..., s_n)              ▷ s_i are "selector variables"
2:  while map is satisfiable do
3:      seed ← getModel(map)                      ▷ get an unexplored subset of F
4:      if SAT(seed) then
5:          MSS ← grow(seed)
6:          MCS ← complement(MSS)
7:          print MCS                             ▷ print the MCS without ending the algorithm
8:          map ← map ∧ blockMSS(MSS)
9:      else
10:         MUS ← shrink(seed)
11:         print MUS                             ▷ print the MUS without ending the algorithm
12:         map ← map ∧ blockMUS(MUS)
13:     end if
14: end while
```

- **blockMSS**(MSS M)→ clause cls
  By adding a new clause to the map we can mark one region of the power set lattice as explored. The clause cls will be

$$\bigvee_{i:C_i \notin M} s_i$$

The basic version of the MARCO algorithm is presented as pseudocode in Algorithm 4.3 and uses the subroutines introduced before. The fundamental execution process are the following four steps, which are repeated until all MUSes and MSSes/MCSes are found eventually:

1. Get an unexplored subset of the power set lattice. This is the starting point for further calculations and is called *seed*.

2. Check the satisfiability of the *seed* to decide, whether it is a superset of an MUS, or the subset of an MSS.

3. According to the satisfiability, **grow** the *seed* to an MSS or **shrink** it to an MUS.

4. Report the result and mark the corresponding region of the power set lattice as explored.

Each iteration identifies either a new MUS or a new MSS/MCS. The process terminates when all possible subsets of the formula $\mathcal{F}$ are explored and all MUSes and MSSes/MCSes are reported.

The following example illustrates the execution of MARCO on our running example.

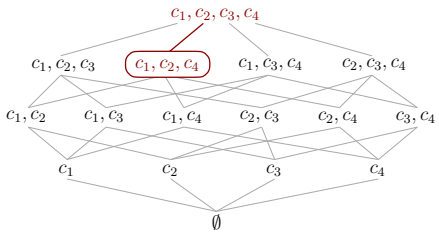
Executing MARCO on our example

$\mathcal{F} = \{c_1 = (\overline{x_1}), c_2 = (x_1 \vee x_2), c_3 = (x_2), c_4 = (\overline{x_2})\}$

In the beginning, the map is a tautology (true in every model), meaning that no subset has been explored. The **getModel**(map)-method could return any subset of $\mathcal{F}$ as a *seed*.

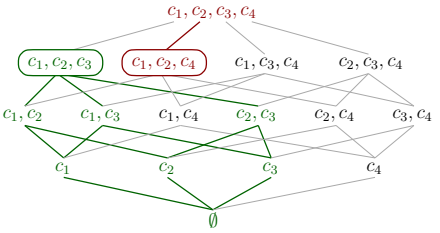
In our example it returns the complete formula $\mathcal{F}$ as *seed*. Since it is unsatisfiable, a MUS will be extracted:

- **getModel**(map) $\rightarrow \{c_1, c_2, c_3, c_4\}$
- **SAT**($\{c_1, c_2, c_3, c_4\}$) $\rightarrow$ false (UNSAT)
- **shrink**($\{c_1, c_2, c_3, c_4\}$) $\rightarrow \{c_1, c_2, c_4\}$
- map $\leftarrow$ map $\cup$ **blockMUS**($\{c_1, c_2, c_4\}$)



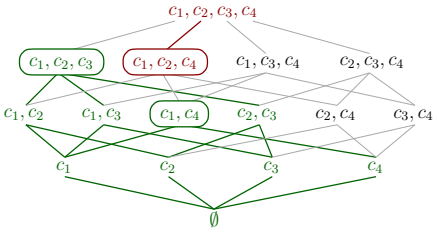
The next *seed* ($\{c_3\}$) is SAT, resulting in an MSS:

- **getModel**(map) $\rightarrow \{c_3\}$
- **SAT**($\{c_3\}$) $\rightarrow$ true (SAT)
- **grow**($\{c_3\}$) $\rightarrow \{c_1, c_2, c_3\}$
- map $\leftarrow$ map $\cup$ **blockMSS**($\{c_1, c_2, c_3\}$)

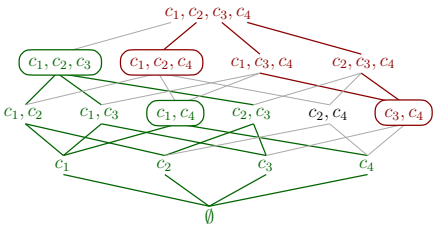


And so on...

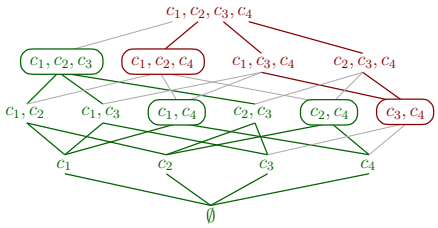
- **getModel**(map) $\rightarrow \{c_4\}$
- **SAT**($\{c_4\}$) $\rightarrow$ true (SAT)
- **grow**($\{c_4\}$) $\rightarrow \{c_1, c_4\}$
- map $\leftarrow$ map $\cup$ **blockMSS**($\{c_1, c_4\}$)



- **getModel**(map) $\rightarrow \{c_2, c_3, c_4\}$
- **SAT**($\{c_2, c_3, c_4\}$) $\rightarrow$ false (UNSAT)
- **shrink**($\{c_2, c_3, c_4\}$) $\rightarrow \{c_3, c_4\}$
- map $\leftarrow$ map $\cup$ **blockMUS**($\{c_3, c_4\}$)



- **getModel**(map) $\rightarrow \{c_2, c_4\}$
- **SAT**($\{c_2, c_4\}$) $\rightarrow$ true (SAT)
- **grow**($\{c_2, c_4\}$) $\rightarrow \{c_2, c_4\}$
- map $\leftarrow$ map $\cup$ **blockMSS**($\{c_2, c_4\}$)



At this point every subset of $\mathcal{F}$ is explored. The map is unsatisfiable and the algorithm terminates.

Lemma 4.6 (Correctness)**.** *Every reported subset of $\mathcal{F}$ is either an MUS or an MCS.*

Proof. All MUSes reported by MARCO were calculated by the **shrink** method, similarly all MSSes are calculated by the **grow** method. With the correctness of both methods and the correctness of the **complement** method, it follows that only correct MUSes and MCSes are reported. □

The completeness of the results follows from the observation that no single result (MUS or MCS) can be reported repeatedly, since blocking the first occurrence of it in the map forbids any subset that is dominated by earlier results as new *seeds*.

Lemma 4.7 (No repeated results)**.** *If* MARCO *(Algorithm 4.3) reports an MUS or MCS, the same subset will not be reported again.*

Proof. Let M be the MUS (MCS/MSS) reported by Algorithm 4.3. Via the **blockMUS** (**blockMSS**) all models that correspond to a superset of the MUS (subset of the MSS) are blocked in map. With Lemma 4.3 of Section 4.4 it follows, that **getModel**(map) will never return a model corresponding to a superset of the MUS (subset of the MSS). Due to the facts that only **shrink** and **grow** are calculating the reported MUSes and MSSes/MCSes and **shrink** needing a superset of M and **grow** needing a subset of M to extract M again, it follows that once M is found, no further iteration of the **while** loop (Algorithm 4.3 lines 2 to 14) will report M again. □

Theorem 4.8 (Completeness)**.** MARCO *(Algorithm 4.3) reports all MUSes and all MCSes of an infeasible constraint set, in our case a SAT formula $\mathcal{F}$.*

Proof. Each iteration of the **while** loop (Algorithm 4.3 lines 2 to 14) will report an MUS or MCS that has not been reported before (Lemma 4.7). As long as there is at least one MUS or one MCS left to be found map is satisfiable, since at least one model corresponding to a superset of that MUS or at least one model corresponding to a subset of the MSS is not yet blocked (Lemma 4.4 of Section 4.4).

Therefore, as long as there are unreported MUSes or MCSes, each iteration of the **while** loop will report a new MUS or MCS, ultimately reporting all MUSes and all MCSes of $\mathcal{F}$. □

Theorem 4.9 (Termination)**.** MARCO *(Algorithm 4.3) will terminate.*

Proof. By Theorem 4.8 all MUSes and all MCSes will be found eventually. If every MUS and every MCS has been found, then every superset of an MUS and every subset of an MSS has been blocked in `map` by the lines 12 and 8. By Lemma 4.4 of Section 4.4 it follows that `map` is unsatisfiable at this point in time, causing the termination of the **while** loop and the termination of the whole `MARCO` algorithm. □

4.4.2 Variants of `MARCO`

Analyzing the performance from the basic `MARCO` version results in possible optimizations which we will describe in the following.

`MARCO` reports the first MUS very fast. In fact, it reports it as fast as any state-of-the-art single MUS extractor. Assume `MARCO` starts with the whole formula $\mathcal{F}$ as the first seed. Since $\mathcal{F}$ has to be unsatisfiable, we can directly call **shrink** to extract one MUS out of it. Since **shrink** can be any state-of-the-art MUS extraction algorithm, especially the best known algorithm, `MARCO` will report the first MUS as fast as any MUS extraction algorithm [89]. Directly following from that is the fact that no other MUS enumeration algorithm can report a first MUS faster.

To report successive MUSes with a similar delay all successive *seeds* would have to be unsatisfiable. This cannot be guaranteed, which causes that MCSes and MUSes will be reported interleaving. However, it is possible to bias the algorithm to unsatisfiable *seeds* early in the execution. Furthermore, successive calls to the **shrink** method can be even faster than the first call of the same method. Since the first extracted MUS M will cause a call of the **blockMUS** method, a region of the power set lattice will be marked as explored. This region has to include the complete formula $\mathcal{F}$. Thus, we know that the successive calls of the **shrink**(*seed*) method will be executed on a *seed* that is smaller than $\mathcal{F}$. The search space is reduced, which should result in a shorter run time. Further optimizations to boost a single call of the **shrink** method will be introduced in this section.

The overall runtime of the `MARCO` algorithm is heavily dependent on the **shrink** and **grow** methods. Both of these are working on a subset of the formula $\mathcal{F}$. All the other subroutines operate on the simple clause set `map` and are insignificant.

Due to this analysis three possible optimizations can be stated:

1. Favor unsatisfiable *seeds* rather than satisfiable *seeds* at the beginning
2. Reduce some or even all calls to **shrink** or **grow**
3. Boost individual **shrink** calls

We will introduce all three optimizations shortly.

Maximal models

The first two goals can be reached by the same technique: using maximal models as *seeds*. A maximal model is a model m where none of the literals that are assigned to `false` can be flipped.

Therefore, m_{max} will correspond to a subset of $\mathcal{F}$, whose supersets are all explored. Of course, it is not guaranteed to produce an unsatisfiable subset, but the larger the subset is, the higher the probability to produce an unsatisfiable subset of $\mathcal{F}$. Assume the intermediate state of `map` shown in Figure 4.4. The maximal models of `map` correspond to the subsets $\{c_1, c_2, c_3\}$ and $\{c_1, c_2, c_4\}$. The first subset is unsatisfiable and is the second MUS of the formula $\mathcal{F}$, the second subset is satisfiable and is an MSS.

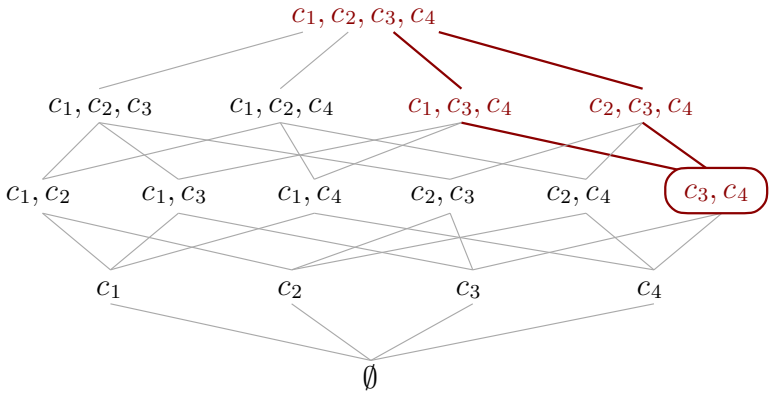


Figure 4.4. A possible intermediate state of `map` after finding the MUS $\{c_3, c_4\}$ as the first MUS. The maximal models are corresponding to the subsets $\{c_1, c_2, c_3\}$ and $\{c_1, c_2, c_4\}$.

In fact, whenever a subset corresponding to a maximal model of `map` is satisfiable, this subset builds an MSS. This is stated in the following theorem:

Theorem 4.10 (Maximal models)**.** *If the subset $X \subset \mathcal{F}$ corresponding to a maximal model m_{max} of* `map` *is satisfiable it is an MSS of $\mathcal{F}$.*

Proof. We have to show the following properties: (i) all immediate supersets of X have to be explored, (ii) all supersets of X have to be unsatisfiable

(i): Assume that there exists a superset $Y \supset X$ corresponding to the complete variable assignment $m' = m_{max} \cup s_i : c_i \notin X$ that is unexplored. Thus, it has to be a model for `map`. Since Y is a superset of X and since the clause c_i that corresponds to the variable s_i in `map` is not contained in X, s_i has to be assigned to `false` for m_{max}. Thus, m_{max} could not be a maximal model, since s_i violates the "maximality" property; s_i could be assigned to `true` and would still satisfy `map`. The resulting complete variable assignment would be m'.

(ii): Assume that there exists a superset $Y \supset X$ that is satisfiable. We know from the first part of the proof that it has to be marked as already explored. By Lemma 4.3 from Section 4.4 it follows that X would be dominated by Y and thus, also marked as explored. □

With Theorem 4.10 all calls to the **grow** method are obsolete when the *seeds* return. Thus, using maximal models to compute seeds will not only bias the algorithm towards finding and reporting MUSes early, but also removes all calls to the **grow** method, potentially resulting in an even bigger performance boost of the algorithm. The resulting algorithm is shown as pseudocode in Algorithm 4.4.

Algorithm 4.4 The `MARCO` algorithm using maximal models

Input: unsatisfiable formula $\mathcal{F} = \{c_1, \ldots, c_n\}$
Output: MCSes and MUSes of $\mathcal{F}$ as they are discovered

```
1: map ← BoolFormula(s_1, ..., s_n)              ▷ s_i are "selector variables"
2: while map is satisfiable do
3:     seed ← getMaximalModel(map)               ▷ get an unexplored subset of F
4:     if SAT(seed) then
5:         MCS ← complement(seed)
6:         print MCS                             ▷ print the MCS without ending the algorithm
7:         map ← map ∧ blockMSS(seed)
8:     else
9:         MUS ← shrink(seed)
10:        print MUS                             ▷ print the MUS without ending the algorithm
11:        map ← map ∧ blockMUS(MUS)
12:    end if
13: end while
```

Note that this optimization has a dual: if any *seed* obtained from the `map` is a minimal model, then all unsatisfiable subsets that correspond to these minimal models are guaranteed to be MUSes. Thus, the calls to the **shrink** method could be saved. At first glance that may be attractive, since the **shrink** subroutine can be very expensive, but we will see in the practical analysis that this does not lead to a better performance overall.

Boost individual MUS extractions

The development of the recursive model rotation [10] led to major improvements for (single) MUS extraction algorithms in recent years [9] and led to the fact that dropping the **shrink** calls completely from `MARCO` is not beneficial for the overall performance. Another route of optimization is to provide the MUS extraction algorithm with additional information to boost its execution.
During its execution, `MARCO` collects viable information on some constraints of $\mathcal{F}$. For example, certain constraints can be found to be *necessary* for every MUS. A constraint c is necessary in a formula $\mathcal{F}$, if $\mathcal{F} \setminus \{c\}$ is satisfiable. Algorithm 4.1 - the basic MUS enumeration algorithm - executes for all constraints $c_i \in X : X \subseteq \mathcal{F}$ the test, whether c_i is a critical clause or not. Thus, knowing in advance which constraints are necessary could save a lot of time during **shrink**.

Specifically, every found MCS that contains only one constraint is a necessary constraint for every MUS. Note that such an MCS $M = \{c_i\}$ will cause the addition of the unit clause (s_i) to `map` (since it is the only constraint not in the complementary MSS). More generally, any literal that is implied to be assigned to `true` by the `map` formula corresponds to a necessary constraint that is included in every MUS of $\mathcal{F}$. These implications can be easily retrieved from the `map` formula and given to the MUS extraction algorithm as so-called hard clauses.

Note that this idea has a dual as well: any by the `map` implied variable assignment to `false` would correspond to a constraint that has to be present in any MCS

(absent in any MSS). However, for Boolean formulae this is not possible at all, since a single constraint cannot induce a conflict on its own.

4.4.3 Practical analysis of MARCO and its optimizations

Before introducing the new techniques which boost the MUS enumeration, we want to compare the different variants of the original MARCO algorithm with each other and with the best known existing MUS enumeration algorithms, CAMUS and DAA. CAMUS typically outperforms DAA for complete MUS enumeration [91], and DAA outperforms the subset enumeration algorithms (see Section 4.3 for details) [4]. Since DAA has an incremental MUS enumeration behavior (it reports MUSes and MCSes interleaving and "early") its addition to this analysis is vital. CAMUS on the other hand, is in its basic version a complete MUS enumerator. When complete MUS enumeration is intractable, CAMUS will not report any MUSes within a given time limit. To cope with that problem a variant of CAMUS was added to the analysis setup that truncates the MCSes calculated in the first phase of the algorithm to partial correction sets of size 2 ("2PCSes").

The following three variants of the original MARCO algorithm were tested:

- MARCO (all opt) - the optimized version of the MARCO algorithm using maximal models to remove the calls to the **grow** method (see Algorithm 4.4) and using the map to identify necessary constraints
- MARCO (basic) - the basic variant of the algorithm (see Algorithm 4.3)
- MARCO (MCS bias) - a variant of MARCO biased towards finding MCSes early using minimal models instead of maximal models

Every algorithm was run on a collection of 300 unsatisfiable Boolean CNF benchmarks, the special MUS track benchmarks of the 2011 SAT competition[1]. These instances were drawn from a large variety of applications, with the most prominent being hardware and software verification, product configuration and bounded model checking. The size of the instances range from 26 to 4.4 million variables and from 70 to 16.0 million constraints. All experiments were run by the authors of the original MARCO algorithm [89] on Amazon Elastic Compute Cloud (EC2) "cc2.8xlarge" cores (Intel Xeon E5-2670 processors). Every single execution ran with a time limit of one hour (3600 seconds) and a memory limit of 3000 MB RAM. The detailed results can be found on the homepage from Mark Liffiton[2].

Due to the potentially exponential number of MUSes, complete MUS enumeration is in general an intractable problem. Due to this, no algorithm was able to complete the enumeration within the timeout of one hour for more than 29 out of 300 instances. The runtimes for the tested algorithms are shown in a logarithmic *cactus plot* in Figure 4.5. A *cactus plot* is created by sorting and plotting values in order within each series. They show the value distribution of the series, but do not allow a pairwise comparison. Each point (x, y) indicates that x instances have a value (e.g. runtime, MUSes, etc.) of y or less. CAMUS is the fastest approach for

[1] http://www.satcompetition.org/2011/

[2] http://sun.iwu.edu/~mliffito/marco/enumeration_results_201312.ods

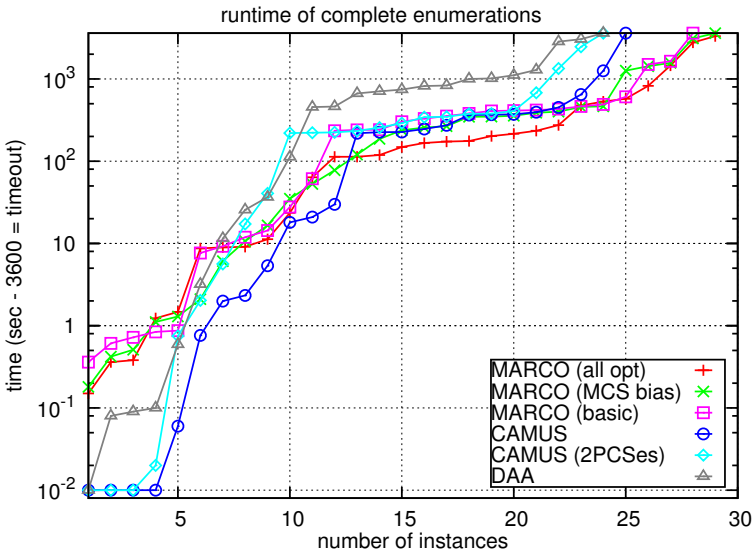


Figure 4.5. Cactus plot of the total runtime to complete the MUS enumeration for each algorithm

13 instances, but MARCO (all opt) is better for the next 16 instances. Comparing all three variants of MARCO no substantial difference can be seen, though the MARCO (all opt) seems to have a slight advantage.

Nevertheless, the algorithms tested here are better analyzed in terms of how many results they produce in comparison to the runtime of completion, which will in most cases be impossibly long to determine. For this purpose we created the reverse cactus plots in Figure 4.6. A point (x, y) can be interpreted in these plots as "x instances have at least y MUSes / MCSes". We can see that MARCO (all opt) vastly outperforms the other approaches for (partial) MUS enumeration. For 235 out of 300 instances it reports at least two MUSes, while the next best old approach, CAMUS (2PCSes), reports more than one MUS for only 58 instances. The behavior of MARCO (MCS bias) in comparison to the basic MARCO shows only a minor gain, supporting the observation that using maximal models to get rid of the calls to the **grow** method is much more effective than using minimal models to get rid of the calls to the **shrink** method.

There are approximately 20 instances in which either version of CAMUS reports at least two orders of magnitude more MUSes than the best variant of the MARCO algorithm. In these cases, CAMUS finds the complete set of MCSes or PCSes very quickly. With its efficient hitting set algorithm the MUSes are then enumerated much faster than any **shrink** call of MARCO can handle this. In some of these cases, CAMUS outputs millions of MUSes even before any version of MARCO completes its first call to **shrink**.

The results are somehow flipped when analyzing the number of MCSes found during the execution of the algorithms. Unfortunately, the versions of CAMUS do not report the number of MCSes, so we were not able to add these values into the

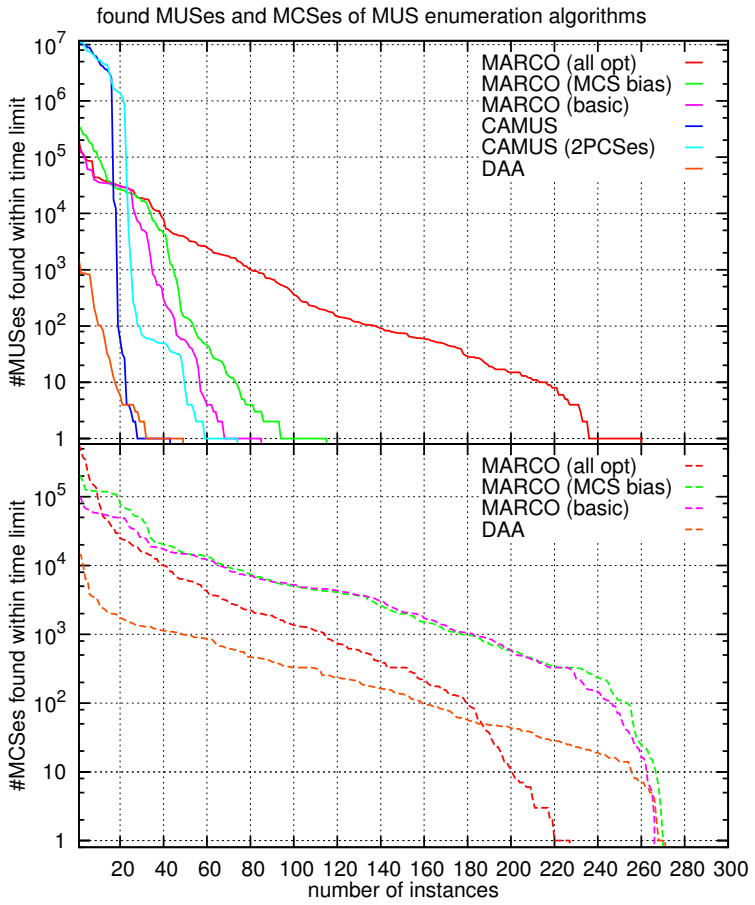


Figure 4.6. Reverse cactus plots of the number of computed MUSes and MCSes within the time and memory limits for each algorithm

plot. The MARCO variant which has a bias towards reporting MCSes early has a slight advantage against the basic MARCO version, but all in all their performance is almost the same. It is remarkable that for all but the first 9 instances, MARCO (all opt) reports less MCSes than the other two variants of MARCO. But since, the gap on the number of MUSes is much larger than the gap on the number of MCSes between MARCO (all opt) and its other two variants, MARCO (all opt) is still the best overall approach.

All of these results show the power of the MARCO approach in general, and the "all opt" variant in particular. Because of this, we decided to incorporate the new techniques to boost MUS enumeration, which will be described in the next sections, into the "MARCO (all opt)" variant. Since the **shrink** method is the most time consuming subroutine of MARCO, it is the obvious target for further improvements to the algorithm.

We will show in the next section, how the `map` can be used to detect not only necessary clauses, but also some critical clauses for the current *seed* to decrease the search space for the MUS extraction algorithms even further. Motivated by this extension, we use a generalization to *clausal blocks* (see Section 3.4.3) to investigate the possibility to speed-up the computation of MCSes as well as the MUS extraction in detail afterwords.

4.5 Determine more MUS Members via `map`

In the previous section we introduced one extension which uses the `map` to identify necessary MUS members, which could then be given as hard clauses to an MUS extraction algorithm, saving potentially for every such clause one call to a SAT solver during the extraction of the MUS.

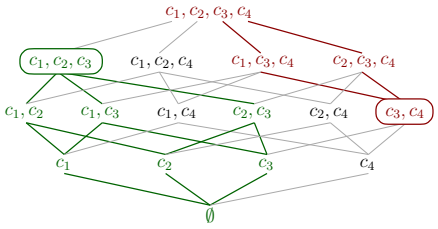
The approach uses so-called top-level assignments within the `map` to identify the hard clauses. Top-level assignments are implications that were caused by the propositional logic of the `map` formula without any further assumptions of variables. These top-level assignments may be caused for example by unit clauses that were added by finding MCSes of size one. Assume that in our running example, the MCS $\{c_3\}$ was found before all MUSes are detected. The corresponding MSS $\{c_1, c_2, c_4\}$ would cause the addition of the unit clause (s_3) (by calling **blockMSS**$(\{c_1, c_2, c_4\})$) indicating that the clause c_3 has to be present in every MUS $M \subset \mathcal{F}$.

The reason for this is the following: Assume that there exists an MUS M that does not contain c_3. Then M would be a subset of the MSS $S = \{c_1, c_2, c_4\}$ that caused the addition of the unit clause (s_3). Since the MSS is satisfiable and $M \subseteq S$, M has to be satisfiable as well, which violates the assumption.

However, the `map` is even capable of determining the hitting set property of MUSes and MCSes (see Section 4.3). Thus, we decided to extend the detection of necessary MUS members in the following way. Given a formula $\mathcal{F}$ and a $seed \subset \mathcal{F}$. Whenever *seed* is unsatisfiable we determine the set of positive implied assignments of the `map` that are forced by adding the corresponding negative literals $\overline{s_i}$ for each clause $c_i \in (\mathcal{F} \setminus seed)$ as assumptions to the `map`. This is done via the method **getImplies**(*seed*).

Assume the current state of the `MARCO` algorithm is the following:

- found MUSes = $\{\{c_3, c_4\}\}$
- found MSSes = $\{\{c_1, c_2, c_3\}\}$
- next *seed* = $(\{c_1, c_2, c_4\})$
- **SAT**(*seed*) → `false` (UNSAT)



The **shrink** method will be called to extract one new MUS. By the hitting set property between MUSes and MCSes and the knowledge of the MCS $\{c_3, c_4\}$, we know that c_4 has to be present in the MUS, since $c_3 \notin seed$. The **shrink** method

would provide this result as well, but has to use one SAT solver call to detect this.

We will show in the practical analysis in Section 4.7 that this extension alone does not lead to an improved performance. The effort to compute the set of forced positive assignments that correspond to necessary members of an MUS is larger than the savings during the subsequent **shrink** method.

This is the reason we will introduce another extension to the approach (see Section 4.6.4) which aims to find a lot more MCSes faster to boost the performance of the `MARCO` algorithm with the help of the extended **getImplies**(*seed*) method.

4.6 Using Blocks to boost MUS Enumeration

All our extensions to the state-of-the-art partial MUS enumeration approach are based on the following *block* property of clauses, which is very similar to the idea of *generalized nodes* to speed-up hitting set computations in hypergraphs by Kavvadias et al. [83].

Definition 4.11 (Block property)**.** *Given an unsatisfiable formula $\mathcal{F}$. A* block *b is a set of clauses $b = \{c_x, c_y, \ldots, c_z\}$ that are always either exactly altogether present in an MUS or not:*

$$\forall M \in MUSes(\mathcal{F}) : b \cap M = \emptyset \vee b \cap M = b$$

The blocks are clause maximal, meaning that the block b cannot be extended by any clause $c_i \in \mathcal{F} \setminus b$ without losing the block property.

Some trivial observations derived from this definition are that every clause belongs to exactly one block and the set of blocks $\mathcal{B}$ is a partition of the unsatisfiable formula $\mathcal{F}$. We denote b_0 as the block of clauses that do not belong to any MUS of $\mathcal{F}$. Then $\mathcal{F} \setminus b_0$ is the union of all MUSes.

4.6.1 Determine the blocks

To obtain the set of blocks $\mathcal{B}(\mathcal{F})$ for an unsatisfiable formula $\mathcal{F}$ the straight-forward approach is to enumerate all MUSes of $\mathcal{F}$ and use the following split routine. Initially $\mathcal{B}_0 = b_0 = \mathcal{F}$. With no found MUS all clauses of the formula belong to the default block of clauses. Note that M_i denotes the i-th found MUS and therefore $\mathcal{B}_i$ denotes the set of (interim) blocks that are formed by the MUSes $M_1, \ldots, M_i$. Please note that interim blocks only permit the block property for the MUSes $M_1, \ldots, M_i$, and not necessarily for the later ones. Nevertheless we drop the word "interim" from it in the remaining part of this chapter.

Definition 4.12 (Splitting blocks)**.** *Let $\mathcal{B}_i = \{b_0, \ldots, b_x\}$ be the set of blocks for an unsatisfiable formula $\mathcal{F}$ which were obtained by splitting the blocks via the MUSes $\{M_1, \ldots, M_i\}$ and $M_{i+1} \subset \mathcal{F}$ be the next MUS that was discovered during the MUS enumeration algorithm. Then $\mathcal{B}_i$ is updated to $\mathcal{B}_{i+1}$ via Algorithm 4.5.*

Lemma 4.13. *(Correctness) The* **splitblocks** *subroutine (Algorithm 4.5) computes blocks $\mathcal{B}_{i+1}$ that do not violate the* block property*.*

Algorithm 4.5 The splitblocks routine

Input: blocks $\mathcal{B}_i = \{b_0, \ldots, b_x\}$ and MUS $M_{i+1} \subset \mathcal{F} = \{c_1, \ldots, c_n\}$
Output: blocks $\mathcal{B}_{i+1}$

1: $\mathcal{B}_{i+1} \leftarrow \emptyset$
2: $m \leftarrow x + 1$ ▷ new block index (b_x is last element in $\mathcal{B}_i$)
3: **for** $b_i \in \mathcal{B}_i$ **do**
4: **if** $i == 0$ **and** $0 < |b_i \cap M_{i+1}|$ **then** ▷ clauses that were in no MUS until now
5: $b_m \leftarrow b_i \cap M_{i+1}$ ▷ build new block b_m
6: $b_i \leftarrow b_i \setminus b_m$ ▷ update the old block b_i
7: $\mathcal{B}_{i+1} \leftarrow \mathcal{B}_{i+1} \cup b_i \cup b_m$ ▷ add both blocks b_i, b_m
8: $m \leftarrow m + 1$
9: **else if** $0 < |b_i \cap M_{i+1}| < |b_i|$ **then** ▷ proper subset
10: $b_m \leftarrow b_i \cap M_{i+1}$ ▷ build new block b_m
11: $b_i \leftarrow b_i \setminus b_m$ ▷ update the old block b_i
12: $\mathcal{B}_{i+1} \leftarrow \mathcal{B}_{i+1} \cup b_i \cup b_m$ ▷ add both blocks b_i, b_m
13: $m \leftarrow m + 1$
14: **else** ▷ block unchanged
15: $\mathcal{B}_{i+1} \leftarrow \mathcal{B}_{i+1} \cup b_i$
16: **end if**
17: **end for**

Proof. Assume that the block b_k is violating the *block property* (Definition 4.11) for at least one of the MUSes $\{M_1, \ldots, M_{i+1}\}$. We have to consider the following two main cases: (i) b_k violates the property for one of the MUSes $\{M_1, \ldots, M_i\}$, (ii) b_k violates the property for the MUS M_{i+1}.

Case (i): Since $\mathcal{B}_i$ was a set of blocks that did not violate the *block property* for any of the old MUSes, b_k has to be created in the last call to the **splitblocks** method. This can happen either in lines 5-8 or in lines 10-13. Since in both parts of the algorithm an old block b_i will be partitioned into the blocks b_m and $b_{i'}$ with $b_i = b_m \cup b_{i'}$ (lines 6 and 11), b_k has to be a proper subset of b_i and cannot violate one of the MUSes $\{M_1, \ldots, M_i\}$, due to the fact, that if b_k violates the MUS $M_j : 1 \leq j \leq i$, any superset of b_k (in particular b_i) has to violate M_j as well.

Case (ii): Two subcases have to be considered here: either b_k is unchanged from $\mathcal{B}_i$, or b_k is created within the current call of **splitblocks**. The block b_k cannot be unchanged from $\mathcal{B}_i$ (line 15), since both sets $b_k \cap M_{i+1}$ and $M_{i+1} \setminus b_k$ have to be nonempty to violate the *block property* and thus, the **splitblocks** would never reach line 15. With a similar observation as in case (i) we know that b_k cannot be created within the current call either, since it would have been created in lines 5-8 or in lines 10-13. By that, b_k is either the new block (lines 5 or 10), or the updated old block (lines 6 or 11. Both cannot violate the *block property* for the MUS M_{i+1} by definition. □

Lemma 4.14. *(Termination) The* ***splitblocks*** *subroutine (Algorithm 4.5) terminates.*

Proof. Trivial, since there are at worst m different blocks in the set $\mathcal{B}_i$ with m

being the number of clauses of the formula $\mathcal{F}$. □

Since the blocks can only get smaller, each block in the set $\mathcal{B}_k$ is an ancestor for at least one block in $\mathcal{B}(\mathcal{F})$. Each ancestor block can be seen as an over-approximation of a block that gets tighter with more MUSes found until ultimately reaching equality. Tightness is reached at latest when all MUSes were enumerated, but could be obtained earlier as well. For example, whenever $\mathcal{B}_k \neq \mathcal{B}(\mathcal{F})$ contains a block of size 1 that block cannot be split any further.

Example 3: The **splitblocks** method results in the following blocks for our running example $\mathcal{F}$ and the given sequence of enumerated MUSes.

Initialization:		$\mathcal{B}_0 = \{(c_1, c_2, c_3, c_4\}$
1st MUS:	$M_1 = \{c_1, c_2, c_4\}$	$\mathcal{B}_1 = \{(c_3), (c_1, c_2, c_4)\}$
2nd MUS:	$M_2 = \{c_3, c_4\}$	$\mathcal{B}_2 = \{(c_1, c_2), (c_4), (c_3)\} = \mathcal{B}(\mathcal{F})$

Real-world instances show some important properties: The block of clauses that are present in no MUS at all (b_0) is normally the largest block and although there are a lot of blocks containing only one clause, several blocks of larger sizes are present as well.

4.6.2 Proving the block property

With the block property proven, we could save many SAT solver calls due to the fact that whenever one clause of the block is determined to be present or absent in the current MUS within the **shrink** subroutine, all other members of the block are determined as well, without using any additional SAT solver calls. To prove the block property we could use the available `map` instance. Recall that the `map` is used to determine already covered areas of the search space. It provides the main method with a *seed* from an area of the search space, that was not yet covered by the algorithm and therefore offers a new result, either an MCS or an MUS.

After the addition of (an over-approximation of) a block $b_i \in \mathcal{B}_k$ to the `map` via **blockMUS**(b_i) the `map` provides *seeds* where not the whole block b_i is present. Let cls_i be the clause that was added to the `map` via **blockMUS**(b_i) and the *seed* returned from the `map` to be unsatisfiable. During the subsequent **shrink** method the SAT solver either deletes all members of b_i from the *seed* to find a new MUS, or at least one member of b_i is still present in the new MUS M_{k+1}. In the first case, the block b_i is not touched and thus is still valid. The algorithm could go on with the proposed block b_i added to the `map`. In the second case, the block b_i is divided into two new blocks $b_{i'}$ and $b_{i''}$ with $b_{i'}$ consisting of the elements of b_i that are present in the MUS M_{k+1}, and $b_{i''} = b_i \setminus b_{i'}$. Suppose we continue trying to prove the block property for $b_{i'}$, since b_i was proven to be an over-approximation. Adding $b_{i'}$ to the `map` would make b_i obsolete, since $b_{i'}$ is a subset of b_i and thus the clause $cls_{i'}$ added via **blockMUS**(b'_i) subsumes the clause cls_i. From now on the `map` provides *seeds* where not the whole block $b_{i'}$ is present until it reaches unsatisfiability. In the end, the block property of the current block is proven, since

every unsatisfiable *seed* that is provided by the `map` without the current block has to contain the whole block.

The problem of this approach to prove the block property for a block b_i is, that it finds all MUSes $\mathcal{M}$ that do not contain b_i. Thus, the proven property is only available when enumerating the remaining MUSes $\mathcal{M}' = MUSes(\mathcal{F}) \setminus \mathcal{M}$, a part of them already enumerated and used to redefine b_i by the **splitblocks** method (Algorithm 4.5). Therefore, we show in the next sections how unproven blocks (that are over-approximations of blocks) are used to support the MUS and MCS detection.

4.6.3 Using block information during shrink

The **shrink** method can be any state-of-the-art MUS extraction algorithm. `MARCO` uses `muser2` [11]. One major advantage and prerequisite for our extension is that the solver is able to cope with so-called *group*-MUS instances [91].

Definition 4.15. *Given an explicitly partitioned unsatisfiable CNF formula* $\mathcal{F} = D \cup \bigcup_{G \in \mathcal{G}} G$ *with* $\mathcal{G} = \{G_1, \ldots, G_k\}$*,* D *and* G_i *being disjoint sets of clauses, a* ***group oriented MUS*** *of* $\mathcal{F}$ *is a subset* $\mathcal{G}'$ *of* $\mathcal{G}$*, such that* $D \cup \bigcup_{G \in \mathcal{G}'}$ *is unsatisfiable, and* $\forall \mathcal{G}'' \subset \mathcal{G}' : D \cup \bigcup_{G \in \mathcal{G}''}$ *is satisfiable.*

D is the default group (often denoted as being group G_0) that has to be present in every MUS. It consists of the clauses that correspond to the implied variable assignments given by the `map` (via the **getImplied(***seed***)** method) as described in Section 4.5.

The possibility to define partitioned groups allows us to use the block information of clauses rather straight-forward. All blocks b_i that are present in the *seed* get their own group $G_{n+i} = \{seed \cap b_i\}$ with n being the number of clauses in the unsatisfiable formula $\mathcal{F}$. The only exception from this rule is the block b_0 of clauses which were not present in any MUS until now. Each of these clauses $c_i \in \{seed \cap b_0\}$ form their own group $G_i = \{c_i\}$. Due to the blocks being over-approximations, the block property is not proven for any of its members. Thus, executing the MUS extractor on this *grouped* instance does not return an MUS, but rather an over-approximation of an MUS $gM \supseteq M$ as well. We have to run the MUS extractor a second time if and only if $\exists G_i \in gM : |G_i| > 1$. For every $G_i \in gM$ with $|G_i| = 1$ we know that the clause representing this group has to be in M. We add that clause to G_0 for the second MUS extractor call. Since it was found to be critical for the over-approximation gM, it has to be *critical* for each subset of gM, especially M, as well. Recall that a clause c is *critical* when the deletion of it from an unsatisfiable set of clauses U causes $U \setminus \{c\}$ being satisfiable.

For all other groups $G_j \in gM$ with $|G_j| > 1$ every clause $c_i \in G_j$ forms its own group G_i for the second call. Together with the increased G_0 which can be possibly (when $gM \subset$ seed) further increased by new forced implications recognized via **getImplies(gM)** they form a new instance where every non-default group is of size one. Running the MUS extractor on this finally returns a valid MUS $M \subset \mathcal{F}$.

We have seen that using the block property within the **shrink** subroutine may cause that two calls to a *group*-MUS extractor have to be used to determine a single

MUS. Nevertheless the sum of SAT solver calls in those two MUS extractor runs is potentially much smaller in comparison to the normal **shrink** call, when a large group G_i could be deleted from the seed within the first run. During the practical analysis of our extensions we will show the effect of using this extended **shrink** method.

4.6.4 Using block information to find more MCSes

To gain additional boost of the **getImplies(***seed***)** method we present an approach that uses the block information to determine likely candidates for other MCSes. As we have seen in Section 4.5, a clause cls is added to the `map` with every **blockMSS()**. The clause cls can be used to infer clauses $c_i \in \mathcal{F}$ which have to be part of an MUS via the hitting set property of MUSes and MCSes.

Recall that when two clauses c_i and c_j are present in the same block b_k, the clauses do not appear separately in any MUS. This leads to the following Lemma.

Lemma 4.16. *Let block b_k have at least two clauses c_i and c_j. For every MCS M with $c_i \in M$, there has to be another MCS $M' = c_j \cup (M \setminus \{c_i\})$.*

Proof. By the hitting set property of MCSes and MUSes and the minimality of MCSes we know that there has to be at least one MUS U with $U \cap M = c_i$. If there is no such U, then M would not be minimal. It would be possible to eliminate c_i from M and not lose the hitting set property of the set of MUSes. But since c_i and c_j are in the same block b_k all MUSes that were hit by c_i are hit by c_j as well. Therefore $M' = c_j \cup (M \setminus \{c_i\})$ is a valid MCS by the hitting set property. □

Based on this Lemma we present the following algorithm which is called whenever a new MCS is found. This either happens via the **grow** method of the basic `MARCO` algorithm (line 5 in Algorithm 4.3) or whenever we detect a satisfiable *seed* (line 5) using the maximal model approach from Algorithm 4.4.

Algorithm 4.6 The moreMCS routine

Input: blocks $\mathcal{B}_i$ and MCS $C = \{c_1, \ldots, c_n\}, C \subset \mathcal{F}$
Output: MCSes and MUSes of $\mathcal{F}$ as they are discovered

1: find block $blk(c_j) \in \mathcal{B}_i$ for every $c_j \in C$
2: **for** every possible combination $MCSc$ in $\{blk(c_1)\} \times \ldots \times \{blk(c_n)\}$ **do**
3: $MSSc \leftarrow$ **complement(**$MCSc$**)** ▷ get the MSS candidate
4: **if** $MSSc$ is satisfiable **then** ▷ new MCS found
5: **yield** $MCSc$ ▷ print the MCS without ending the algorithm
6: **else** ▷ unsatisfiable seed for MUS extraction
7: $MUS \leftarrow$ **shrink**$(MSSc)$ ▷ extract new MUS
8: **yield** MUS ▷ print the MUS without ending the algorithm
9: $\mathcal{B}_{i+1} \leftarrow$ **splitblocks**(MUS) ▷ use MUS to split blocks
10: find more MCSes/MUSes in split blocks ▷ see Example 4
11: **return**
12: **end if**
13: **end for**

Theorem 4.17. *(Correctness) The **moreMCS** subroutine (Algorithm 4.6) reports only correct MCSes (line 5) and correct MUSes (line 8) of $\mathcal{F}$.*

Proof. By the correctness of the **shrink** method we know that every reported MUS is correct. Furthermore we know that the reported $MCSc$ cannot be a superset of an MUS, since it is satisfiable (line 4). It is left to show that $MSSc$ cannot be a real subset of an MSS in $\mathcal{F}$. This follows directly from Lemma 4.16. □

The presented algorithm tests all possible combinations as long as the resulting candidate MSSes $MSSc$ are satisfiable. Whenever the algorithm detects an unsatisfiable $MSSc \subset \mathcal{F}$, it is used as the *seed* for the **shrink** method to extract a new MUS. The extracted MUS is used to split the blocks, causing at least one of the blocks $\{blk(c_1), \ldots, blk(c_n)\}$ for the original MCS $C = \{c_1, \ldots, c_n\}$ to be split. We prove this formally via the following lemma.

Lemma 4.18. *(MUSes split original blocks) Whenever a new MUS is found, the **splitblocks** method (line 9) will split at least one of the blocks $\{blk(c_1), \ldots, blk(c_n)\}$ for the MCS $C = \{c_1, \ldots, c_n\}$ that caused the execution of the **moreMCS** subroutine (Algorithm 4.6).*

Proof. Let $b_i = \{c_k, c_l, \ldots\}$ be the block that contains at least the two elements c_k and c_l. Suppose that the exchange of these two elements violates the *block property*. This means that there exists an MCS C with $c_k \in C$, but no MCS $C' = c_l \cup (C \setminus \{c_k\})$ (see Lemma 4.16). Thus, **complement**(C') is unsatisfiable and will be used as the *seed* for the **shrink** method to extract a new MUS M in line 7 of the Algorithm 4.6. We know that c_l will not be present in the extracted MUS M, since it is not part of the $seed =$ **complement**(C'). But c_k is present in the *seed* (since $c_k \notin C'$) and c_k will be even critical for the MUS M, since $seed \setminus \{c_k\} \subset$ **complement**(C), which is satisfiable. Thus, $c_k \in M$ and $c_l \notin M$, causing at least a split of the block b_i into $b_m = \{c_k, \ldots\}$ and $b_i = b_i \setminus b_m = \{c_l, \ldots\}$ (Algorithm 4.5 line 11). □

This approach used in the **moreMCS** subroutine (Algorithm 4.6) of testing a candidate for a special property (here: being an MSS) and using that candidate as a *seed* for extracting a new MUS is very similar to the DAA approach (see Section 4.3), which tests candidate MUSes for unsatisfiability and grows them into an MSS if the candidate is satisfiable.

Example 4: Suppose $C = \{c_1, c_4\}$ is the MCS that triggered the call of **moreMCS**, $blk(c_1) = \{c_1, c_2, c_3\}$, $blk(c_4) = \{c_4, c_5, c_6\}$. That leads to $|blk(c_1)| * |blk(c_4)| - 1 = 3 * 3 - 1 = 8$ possible new MCSes since $\{c_1, c_4\}$ has not to be tested. Suppose that $\{c_1, c_5\}, \{c_1, c_6\}$ are tested successfully as MCSes, but $\{c_2, c_4\}$ is not an MCS. We know that (at least) c_2 has to leave the block $blk(c_1)$ due to Lemma 4.18.

Suppose the new blocks after the split operation are $b_i = \{c_1, c_3\}, b'_i = \{c_2\}, b_j = \{c_4, c_5\}, b'_j = \{c_6\}$. The new possible combinations are $b_i \times b_j, b_i \times b'_j, b'_i \times b_j, b'_i \times b'_j$. Please note, that in line 10 of Algorithm 4.6 the combinations $b'_i \times b_j, b'_i \times b'_j$ would not be tested, since no MCS was found that hits these combinations of blocks.

Furthermore, the implementation ensures that no subsets are tested twice during the recursion to prevent doubled results. For example, the MCS candidate $\{c_1, c_6\}$ from the combination $b_i \times b'_j$ is not tested again, but the candidate $\{c_3, c_6\}$ from the same combination is tested.

4.7 Practical Results

To evaluate the extensions to the MARCO algorithm and to compare it to the previous approaches for (partial) MUS enumeration, MARCO and eMUS, we ran all algorithms on a set of 207 instances from the Boolean satisfiability domain. These instances were drawn from a large variety of applications, with the most prominent being hardware and software verification, product configuration and bounded model checking. The benchmark set is a subset of the MUS track of the 2011 SAT competition[3] containing only instances where at least two MUSes and one MCS are found within the time limit of one hour. This decision is based on the fact that our techniques for boosting the computation of MUSes and MCSes use the block information, which is inferred from the already enumerated MUSes. The presented techniques (**shrink** in Section 4.6.3, **moreMCS** in Section 4.6.4) are triggered for the first time when the original MARCO algorithm found the first MCS, respectively starts to extract the second MUS from a part of the formula. Thus, we focus our analysis of the effects on the performance on these instances.

We used the latest MARCO release[4] v1.0.1 that implements the Algorithm 4.4 as the framework for our extension. It is written as a python script that uses the MiniSAT [47] solver for the formula $\mathcal{F}$ as well as the map. The **shrink** method uses muser2 [11] as a MUS and *group*-MUS extraction algorithm. All experiments were run on 2.83GHz Intel Xeon CPUs with a 3600 second timeout and a 16 GB memory limit.

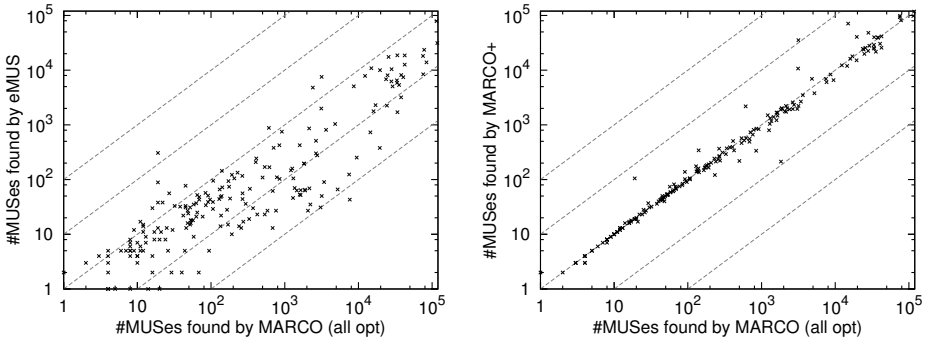
We use the following terminology to describe the different versions of the algorithm and its possible combinations:

- MARCO the variant "(all opt)" by Liffiton et al. [89] (see Section 4.4)
- MARCO+ more critical clauses obtained by **getImplies(***seed***)** method (see Section 4.5)
- MARCOs block information used during **shrink** (see Section 4.6.3)
- MARCOm block information used to find more MCSes faster (see Section 4.6.4)

Thus, when mentioning for example MARCO+m the second and fourth option are used in parallel.

The first results (Figure 4.7) show that MARCO finds more MUSes than eMUS for 182 instances, 23 times eMUS reports more MUSes and both provide the same amount only twice. 44 times the number of MUSes found by MARCO is one order of magnitude higher than the number found by eMUS. In comparison to MARCO+ the results are not so clear. In that case MARCO reports more MUSes for 86 instances, in 70 out of 207 instances MARCO+ finds more MUSes and for the remaining 51 instances both versions find the same amount of MUSes (see numbers on the right hand side in Figure 4.9). The additional effort to compute forced MUS members shown in Section 4.5 is not worth it when using the original MARCO algorithm without any further extensions.

[3] http://www.satcompetition.org/2011/
[4] http://sun.iwu.edu/~mliffito/marco/



(a) (b)

Figure 4.7. Comparing `MARCO` (all opt) to `eMUS` (a) and `MARCO+` (b): number of MUSes found within time limits of 3600 seconds. Each point declares one out of the 207 instances

The additional use of `MARCOm` changes that. `MARCO+` benefits from the MCSes that have been produced by `MARCOm` earlier. The resulting version `MARCO+m` reports more MUSes for 105 instances, less MUSes in 87 and for the remaining 15 instances the same amount as the original `MARCO` within the time limits. Nearly 94% of the found MCSes for all 207 instances ($6,944,690$ out of $7,390,727$) are reported by the Algorithm 4.6 presented in Section 4.6.4.

We present in Figure 4.8 the reverse cactus plots for each possible combination of our extensions and "`MARCO` (all opt)", which was the best variant in the experiments presented in Section 4.4.3. It is obvious that the extensions are all very similar in their performance. Looking at the results for the number of MCSes found by the different combinations it can be seen that every single combination that used the **moreMCS** subroutine (`MARCOm`, `MARCO+m`, `MARCOms`, `MARCO+ms`), performs better than any combination without this subroutine. Furthermore, all pairs for the combinations with and without the extension that computes more necessary members of MUSes via the **getImplied**(*seed*) method are "grouped" together in these reverse cactus plots. This indicates that it makes nearly no difference whether this option (`+`) is activated or not. This behavior seems strange, considering the values from the pairwise comparisons for each of the 207 instances. To understand the problem of evaluating the approaches only on raw numbers, we shift the focus towards an analysis of the relative number of found MUSes and MCSes combined.

Figure 4.9 shows the relative number of MUSes and MCSes found by two extensions, `MARCO+m` and `MARCO+`, in comparison to `MARCO` (all opt). The values on the x-axis are computed by the logarithm (to the base 2) of the fraction of MUSes found by the extensions and by `MARCO`. The y-values show the ratio of found MCSes. The black points correspond to `MARCO+`, the red points to `MARCO+m`.

Points in the positive region of the x-axis denote instances where `MARCO+` (or `MARCO+m`) found more MUSes in the same time limit than `MARCO` (all opt). The same applies correspondingly for the y-axis and the number of MCSes. On the right-hand side of Figure 4.9 it is shown, how many instances are located within the four quadrants in the plane, as well as how many instances are located on the

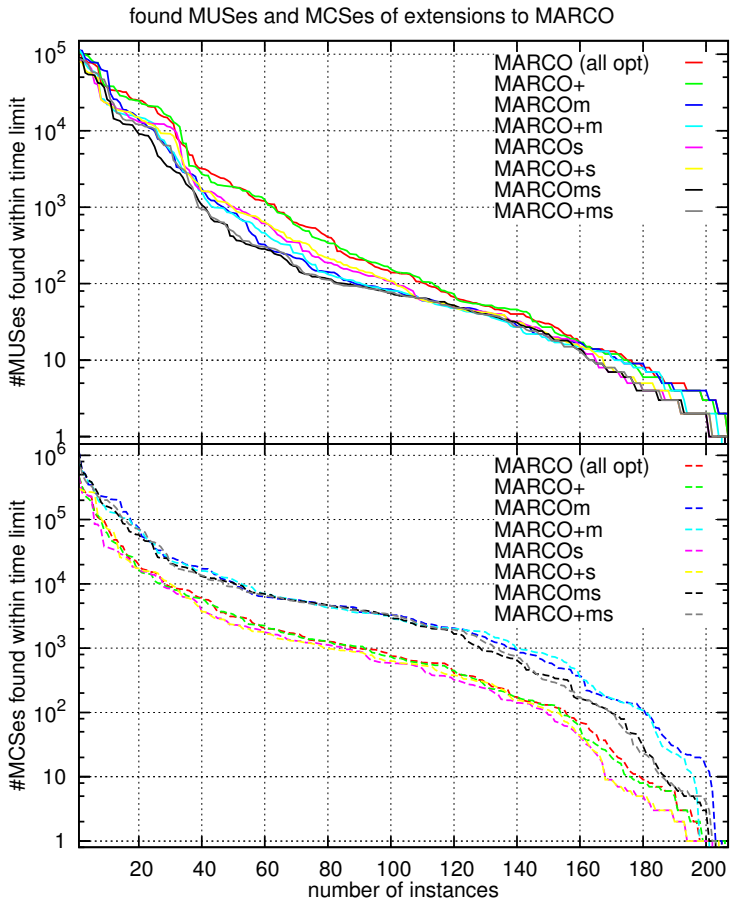


Figure 4.8. Reverse cactus plots of the number of computed MUSes and MCSes within the time limits for each possible combination of the extensions and the variant of `MARCO` that was the best overall ("`MARCO` (all opt)")

(positive and negative) parts of the axes. Black numbers correspond to `MARCO+`, red numbers to `MARCO+m`. The following things can be observed:

The relative numbers of `MARCO+` in comparison to `MARCO` (all opt) are very small: 184 points are located within the interval $[-1 \leq x \leq 1], [-1 \leq y \leq 1]$. In other words `MARCO+` reported for 184 instances more/less MUSes and MCSes with a factor of at most 2. 144 of these points are even within the interval $[-0.322 \leq x \leq 0.322], [-0.322 \leq y \leq 0.322]$, indicating that for these instances the factor of more/less MUSes(MCSes) is at most approximately 1.25.

The relative numbers of `MARCO+m` are much more spread out: 35 points are located within the interval $[-1 \leq x \leq 1], [-1 \leq y \leq 1]$ and only 15 are located within the interval $[-0.322 \leq x \leq 0.322], [-0.322 \leq y \leq 0.322]$.

For the vast majority of 87.4% (181 out of 207) of the instances `MARCO+m` reports more MCSes than `MARCO` (all opt). The same holds for just 34.8% (72 out of 207)

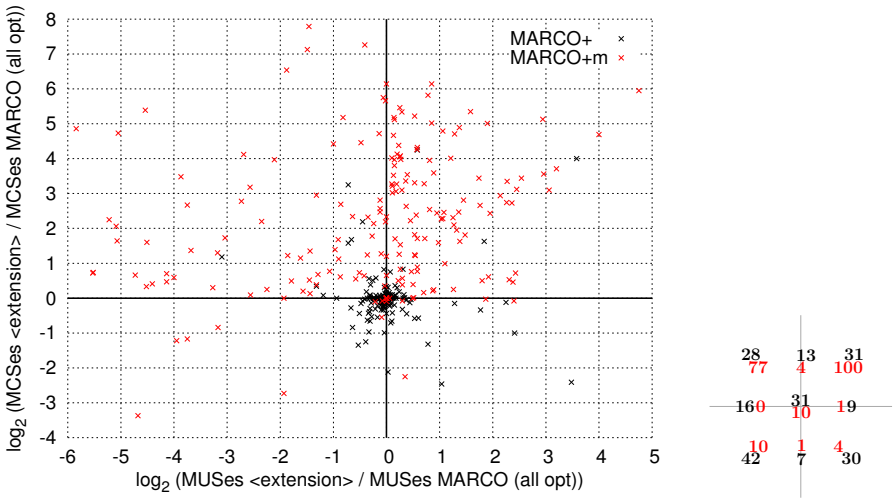


Figure 4.9. $\log_2$ of the relative number of MUSes on x-axis and of MCSes on y-axis; together with the amount of points (instances) for every quadrant in the plane and on the axes for `MARCO+` (black) and `MARCO+m` (red) in comparison to `MARCO` (all opt)

for `MARCO+`.

For 50.7% of the instances (105 out of 207) `MARCO+m` outperforms `MARCO` (all opt) on both, MUSes and MCSes (for 4 instances it computes more MCSes and the same number of MUSes and for 1 instance it `MARCO+m` computes more MUSes and the same amount of MCSes). The opposite holds for just 5.3% of the instances (11 out of 207 - one of them it detects the same amount of MUSes, but less MCSes). For 10 instances the number of found MUSes and MCSes are identical.

The remaining 81 instances where `MARCO+m` either found less MUSes, but more MCSes or vice versa, cannot simply be evaluated by the raw numbers of found MUSes and MCSes since the effort to compute one MUS, or one MCS respectively, varies. To overcome this problem we introduce a new way to measure the results of (partial) MUS enumerators.

4.7.1 Workload computation

Since the partial MUS enumerators introduced in this chapter produce two different results, MUSes and MCSes, our goal is to shift the evaluation and comparison of different approaches to schemes and measures that cover both results as well. Until now, the number of computed MCSes is seen as a side-effect of the computation, which is sometimes even not reported (like in `CAMUS` [91]). The MCSes are needed to terminate the algorithm by marking the satisfiable regions of the `map` as explored, but we have shown in Section 4.5 how the MCSes can be used to speed up every MUS extraction. Before we present detailed results on the effect of known MCSes for the MUS extraction later, we introduce a new scoring scheme of partial MUS enumeration approaches which is based on the number of detected MUSes and MCSes, as well as the runtime the algorithm spends to report a new result.

Thus, we introduce the following scoring function, called the *additional expected workload* to compare the results of two algorithms A_1 and A_2.

Definition 4.19 (The additional expected workload)**.** *Let Algorithm A_1 and Algorithm A_2 be two partial MUS enumerators and the respective number of found MUSes $nU(A_i)$ and MCSes $nC(A_i)$. With the time used to compute all found MUSes $tU(A_i)$ and MCSes $tC(A_i)$ for a fixed instance we define the additional expected workload of A_1 in comparison to A_2 as:*

$$wl(A_1, A_2) = nU(A_1)\frac{tU(A_1) + tU(A_2)}{nU(A_1) + nU(A_2)} + nC(A_1)\frac{tC(A_1) + tC(A_2)}{nC(A_1) + nC(A_2)} - (tU(A_1) + tC(A_1))$$

The first term of the formula describes the expected time that is needed to find the number of MUSes by algorithm A_1. It is computed via the average time both algorithms need to compute a single MUS. The second term describes the same for the MCSes found by algorithm A_1. Subtracting the real times the algorithm A_1 spends computing MUSes and MCSes from this sum we get a positive value if and only if A_1 performed better than A_2 because the expected runtime is higher than the actual runtime. Note that $wl(A_1, A_2) = -wl(A_2, A_1)$.

For the aforementioned 81 instances represented by the red points in the second and forth quadrant of the Cartesian plane shown in Figure 4.9 we get a sum of the *additional expected workload* of 3412.86 seconds for `MARCO+m` in comparison to `MARCO` (all opt). When expanding the sum to all 207 instances in the benchmark set we get the *additional expected workload* of 99911, 44 seconds with a median of 376.26 seconds and an average of 482.66 seconds. The extension `MARCO+m` clearly outperforms the state-of-the-art `MARCO` (all opt) algorithm.

To put these results in perspective: the overall runtime for `MARCO+m` on the 207 instances is approximately 730000 seconds, with the *additional expected workload* of 99911.44 seconds `MARCO+m` performs 13.7% better than `MARCO` (all opt).

Table 4.1. The sum of the *additional expected workload* for all possible extensions in comparison to `MARCO` (all opt), the best value is marked bolt.

extension	add. exp. wl.
`MARCO+`	-9,735.52
`MARCOm`	79,786.84
`MARCO+m`	**99,911.44**
`MARCOs`	-152,012.84
`MARCO+s`	-142,216.26
`MARCOms`	-18,123.78
`MARCO+ms`	-43,770.99

The sum of the *additional expected workload* for all possible extensions in comparison to `MARCO` (all opt) is shown in Table 4.1. Only two out of 7 possible extensions outperform `MARCO` (all opt). Whenever the *block property* is used within the **shrink** method (Section 4.6.3), the results are much worse than without this option activated.

4.7.2 Influence of enumerated MCSes on shrink

In Section 4.5 we introduced a technique to increase the number of critical clauses via an extensive use of the `map` formula. We expect that a higher number of already known MUS members leads to an improved performance of the **shrink** method, since the MUS extractor saves for every already known critical clause potentially one SAT solver call which would have been used otherwise to detect the necessity of them.

In general it is hard to find a fair way to evaluate the influence of the number of MCSes on each **shrink** call, since for the majority of instances the difference of found MUSes (see Figure 4.9) is very large. The MUS extractions at a later point in the execution time of `MARCO` are expected to have a shorter runtime, since the *seed* that is used as a starting point for the **shrink** method is expected to be smaller than earlier. This effect is caused by using maximal models and the observation that the region of unexplored subsets of the `map` shrinks with every found MUS and MCS.

Due to this, we present in the following some results on 9 instances, whose complete MUS/MCS enumeration finished within the time limit of an hour. The numbers of MUSes and MCSes and the minimum and maximum runtime (in seconds) for any of the extensions is reported in Table 4.2.

Table 4.2. complete enumerated instances: the numbers of MUSes and MCSes together with the minimum and maximum runtime (in seconds) for any of the extensions

instance	MUSes	MCSes	min time	max time
1) atpg_ssa2670-140.cnf	58	696	777.545	2319.576
2) atpg_ssa2670-140.cnf	12296	1997	537.362	1429.512
3) bmc-default_barrel2.cnf	27	137	0.155	0.491
4) bmc-default_longmult0.cnf	4	90	0.088	0.169
5) design-debugging_c1_DD_s3_f1_e2_v1-[...].cnf	4	1296	981.740	1706.239
6) equivalence-checking_c2670.cnf	4	6185	459.273	556.23
7) hardware-verification_dlx2_aa.cnf	32	1124	7.671	10.617
8) product-configuration_C168_FW_UT_851.cnf	852	30	8.899	82.631
9) product-configuration_C208_FA_UT_3254.cnf	25600	155	307.488	1243.728

Since we want to study the effect of the number of known MCSes on the performance of **shrink** the majority of these instances are not usable for different reasons:

- instances 3 and 4, since both are enumerated within a few tenths of a second
- instances 5 and 6, since these only have a very small amount of MUSes
- instances 2, 8 and 9, since the ratio of MUSes and MCSes is heavily skewed towards MUSes

We are left with the instances 1 and 7. For the instance 1 the best performance was achieved by `MARCOm` 777.545 seconds, `MARCO` (all opt) needed 1024.013 seconds.

The behavior of the instance `hardware-verification_dlx2_aa.cnf` is shown in a more detailed fashion in Figures 4.10 and 4.11. The Figure 4.10 compares the number of detected MUSes and MCSes during each point in the execution for every extension and `MARCO` (all opt). It can be seen that all the extensions with the activated **moreMCS** option (`m`) detect the MCSes much earlier. `MARCOm` and `MARCO+m` are the fastest and detected all 1124 MCSes after 4.75 seconds. The fastest version without the `m` option is `MARCO` (all opt) approximately 4 seconds later. Furthermore the Figure 4.10 shows that the extensions with activated `m` option enumerate nearly all the MCSes, before extracting the fourth and all later MUSes. `MARCO+m` for example extracts the fourth MUS when it already has detected

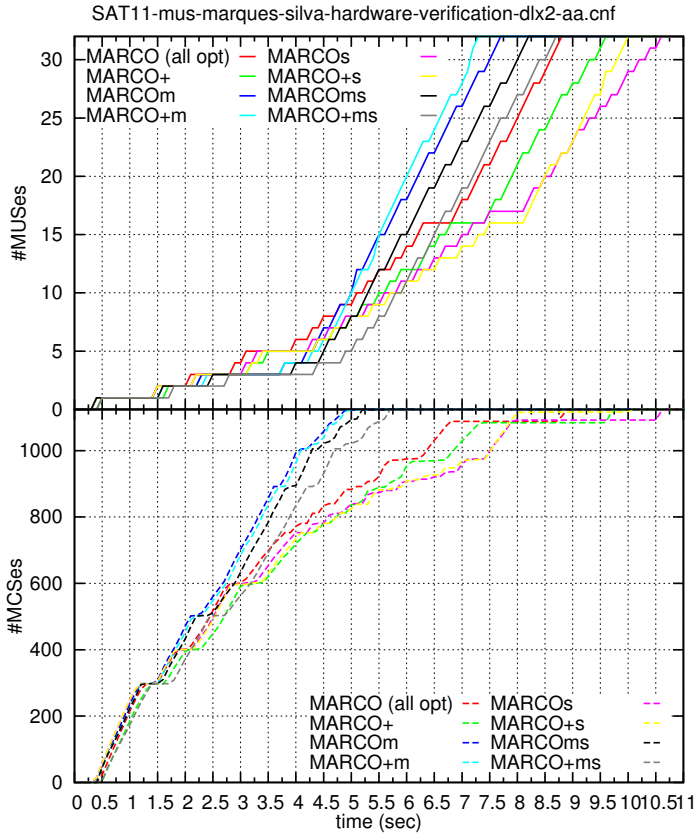


Figure 4.10. The number of found MUSes (top) and MCSes (bottom) for the instance `dlx2_aa.cnf` during the execution of all extensions and `MARCO` (all opt)

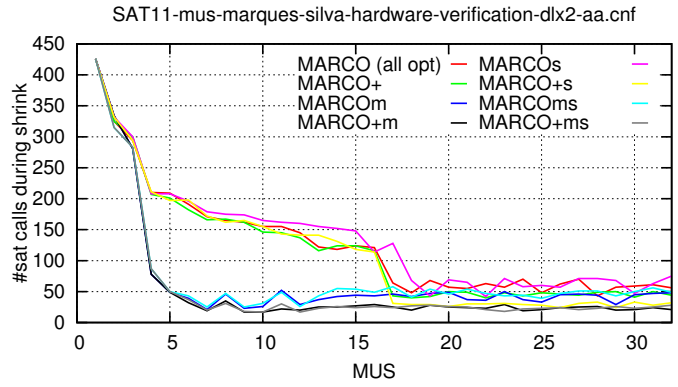


Figure 4.11. The number of SAT solver calls needed within every **shrink** method for all extensions. The numbers for the MUSes number 4 - 17 are significantly smaller for all extensions with activated **moreMCS** option (`m`).

circa 900 MCSes. `MARCO` (all opt) extracts the fourth MUS earlier in the execution (circa at the 2.55 seconds mark), when it has only detected about 600 MCSes.

Figure 4.11 compares the number of SAT solver calls within each **shrink** method for all extensions for the same instance. We see a significant difference of SAT solver calls for the extraction of MUSes 4 to 17 between all extensions with and without the activated **moreMCS** option (`m`). The higher number of MCSes present at the start of the fourth (and any following) MUS extraction enables the detection of more *critical* clauses by the **moremcs** method (Section 4.6.4), which leads to a significantly smaller number of SAT solver calls during the **shrink** method. After the seventeenth detected MUS, all versions have found the vast majority of MCSes. Thus, no significant difference in number of SAT solver calls can be observed anymore.

4.7.3 Using block property within shrink

As already mentioned, the extensions of `MARCO` (all opt) that use the block information within **shrink** calls (`MARCOs`, `MARCO+s`, `MARCOms` and `MARCO+ms`, Section 4.6.3) do not result in any improvements of the performance over all instances (Table 4.1). However, there is a particular set of benchmarks, the `rand_net` set from Eugene Goldberg, where the extended **shrink** method helps to improve the performance of `MARCO` (all opt). The `rand_net` benchmarks are "miter" CNFs (all unsatisfiable) produced from randomly generated circuits that contain AND and OR gates but no inverters. Each circuit implements a monotone function and is rectangular, i.e. the number of primary inputs, the number of gates of m-th level, and the number of primary outputs are all equal to a given parameter n. To check if a circuit is equivalent to itself, a so-called "miter" is formed. The specification and implementation are equivalent if this "miter" is unsatisfiable.

For this particular benchmark set, which contains 9 different instances, the sum of the *additional expected workload* from `MARCOms` in comparison to `MARCO` (all opt) is 3434.89. This is particular remarkable, since for the same variant without using the block property within **shrink** (`MARCOm`) the sum of the *additional expected workload* in comparison to `MARCO` (all opt) is -2227.12. With the additional usage of the extended shrink an approach which was performing worse than `MARCO` (all opt) improves significantly, such that it outperforms `MARCO` (all opt) now. A more detailed analysis the number of SAT calls for each **shrink** for a particular instance is shown in Figure 4.12. Since the extensions with activated **getImplies**(*seed*) (`+`) have the exact same behavior as the extensions without this option, we decided to present only the 4 versions without this option activated.

It can be seen that the extensions with activated `s`-option need a smaller amount of SAT calls to determine every single MUS. Up to 10% of the SAT solver calls could be saved this way. Please note that towards the end of the enumeration `MARCOm` has closed the gap and needs approximately the same amount of SAT solver calls within each **shrink**. It appears that the extended **shrink** method can be beneficial especially at the beginning of an MUS enumeration procedure. This can be explained by the fact that the blocks are rather big when only a small amount of MUSes is detected. The bigger the amount of detected MUSes the smaller the blocks are expected to be, since the over-approximations get tighter whenever an

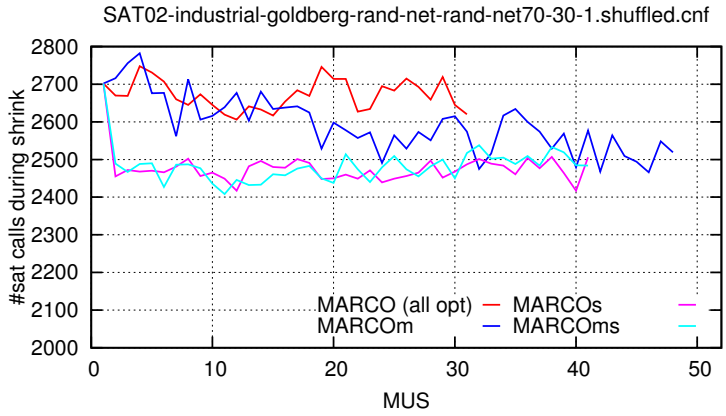


Figure 4.12. The number of SAT solver calls needed within every **shrink** method for all extensions.

MUS is used to split blocks via Algorithm 4.5.

4.8 Summary

In this chapter we presented a technique to boost partial MUS enumeration. In fact, these partial MUS enumerators not only enumerate MUSes, but MCSes as well. We introduced the state-of-the-art approach, that was developed by Liffiton and Malik [89], and Previti and Marques-Silva [116] in detail, since they use an additional meta instance over selector variables to determine the already processed parts of the search space. Indeed, this is very similar to the idea we proposed in the last chapter.

Based on a generalization of the *block* idea from Chapter 3 we introduced the *block property* of clauses, which can be used to identify clauses that are always present together or not present at all in an MUS. We showed a way to prove this property for a given block, but due to its limited effectivity we opted to use unproven intermediate blocks, which are over-approximations of real blocks.

We incorporated the block information in the detection of MUSes and MCSes, and showed how both detections benefit possibly. Furthermore, we presented an extension to the already available clause necessity detection of the `map` instance, that enabled us to infer even more clauses that are known to be *critical* in an MUS by the hitting set property of MUSes and MCSes.

In combination with our technique to detect MCSes faster and earlier in the search, it is possible that any MUS extraction algorithm can save potentially a large amount of SAT solver calls, since the criticality for these clauses does not have to be tested during the MUS extractor call, but can be given in advance.

With the help of an extensive empirical analysis we could show that our exten-

sions lead to a better performance of our `MARCO+m` variant regarding the number of found MUSes and MCSes, as well as the *expected additional workload* within a time limit of one hour for every instance. We did not analyze the performance of `MARCO+m` in comparison to the state-of-the-art MUS enumerator `CAMUS`, since the results obtained in the original work about partial MUS enumeration algorithms [89, 116] do not change: partial MUS enumeration does not replace state-of-the-art MUS enumerators, but offers a viable option for instances where the full enumeration is computationally infeasible in limited time. `MARCO+m` offers better results than `MARCO` / `eMUS`, but does not close the gap completely.

Since the MUS extraction is the most time consuming step within MUS enumeration algorithms, we further analyzed the effect of our extensions to it. On the one hand we saw that it is beneficial to have a larger set of detected MCSes before starting a MUS extraction step, since it allows us to infer more criticality information. On the other hand we analyzed that the usage of the *block property* during the MUS extraction is not beneficial for the majority of instances.

The ability to prove blocks in an efficient manner could be the key to overcome this problem. Other possible directions of future work will be presented in Chapter 7, where the work is concluded.

5 Using SAT to reconstruct phylogenies

5.1 Introduction

Since Charles Darwin postulated his thesis of *genetic evolution* being the basis for the diversity of the organisms on earth in 1859, scientists are searching for the *tree of life*, a phylogenetic tree that puts all organisms into relationship with each other. The problem of finding the *tree of life* can be described in general as finding the optimal phylogenetic tree for a set of taxa, e.g. organisms. This problem is known to be NP-hard under the most popular optimization criteria (maximum parsimony and maximum likelihood) [52, 119]. Nevertheless there are several approaches to solve this problem, one of them a divide-and-conquer-approach that splits the problem into subproblems, solves them and recombines the solutions to form a complete tree that represents all taxa under consideration [77, 16].

In this chapter we will focus on *Quartet Compatibility* (QC), an approach that tries to find a tree that preserves all the information of a set of binary unrooted trees with 4 taxa : the quartets $\mathcal{Q}$. If $\mathcal{Q}$ denotes the set of all $\binom{n}{4}$ quartets of a phylogenetic tree T, then T is uniquely defined by $\mathcal{Q}$ and can be computed in polynomial time [49]. Once $\mathcal{Q}$ is incomplete or contains elements that contradict each other QC becomes NP-complete [133]. Closely related to this is the *Maximum Quartet Consistency* (MQC) problem, where a tree that agrees with the maximum number of quartet topologies is computed. Due to the NP-completeness of both, QC and MQC, only a few algorithmic approaches are known, mostly using graph-theory-based methods like *edge colorings* [66] or special techniques on limited graph classes, like *chordal graphs* [124]. The most efficient solutions for MQC use *Answer Set Programming* [150], *Pseudo Boolean Optimization* or *SAT Modulo Theories* [104].

In this chapter we introduce a new way to tackle QC as well as MQC by encoding it as a satisfiability (SAT) problem. Due to the emergence of SAT solving in practical applications [132, 38] and highly-optimized SAT solvers the idea is to

develop new practicable approaches based on the SAT-methodology.

This chapter is organized as follows. Section 5.2 defines the problem and introduces the background of this work. The most efficient solutions for MQC are presented in Section 5.3. Section 5.4 describes the developed SAT encoding for QC. We show in Section 5.5 how the SAT formulation is used to solve the MQC problem. The practical results of the comparison of the novel SAT formulation to the state-of-the-art approaches is shown in Section 5.6. The chapter is finished with a small summary in Section 5.7.

5.2 Preliminaries

We take the basic graph-related definitions for granted and refer to textbook literature [14]. A *phylogeny* on a given set of taxa $\mathcal{S} = \{s_1, \ldots, s_n\}$ is a tree whose n leaves are mapped one-to-one to all elements from S. It can be *rooted* or *unrooted*. We call a *phylogeny binary* or *resolved* when all inner nodes have a degree of three (in a *rooted phylogeny* the only exception is the root which has degree two). From now on whenever we say *phylogeny* we implicitly refer to *unrooted binary phylogenies*. Each inner node in a phylogeny represents a hypothesized (or an extinct) ancestor of the taxa which are contained in the subtrees connected to the inner node.

We will not go into details about *phylogenetics*, a field of interaction between mathematics, statistics, computer science and biology, but rather refer the reader to the basic literature [124] for further information.
A subset of $\mathcal{S}$ containing four different taxa is called *quartet*. A phylogeny for a quartet is called a *quartet topology* (or topology short). It is the smallest phylogeny containing any information regarding evolutionary relations between its associated taxa. With $\{a, b, c, d\} \subseteq \mathcal{S}$ being the quartet, disregarding symmetric cases, each quartet can have three different topologies $ab|cd, ac|bd$ and $ad|bc$ shown in Figure 5.1. The phylogenetic information from $ab|cd$ is that the pair of a and b is closer related to each other than to c or d, i.e. the path from a to b in the phylogeny for S does not intersect the path from c to d.

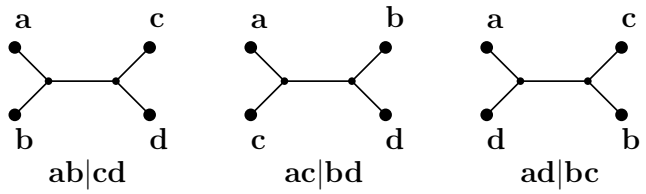


Figure 5.1. the three different topologies $ab|cd, ac|bd$ and $ad|bc$ for a quartet $\{a, b, c, d\}$

Let T be a phylogeny on S and $q \subseteq \mathcal{S}$ a quartet. The *restriction* of T to q, denoted $T|q$, is the topology T' that is constructed by deleting all leaves labeled with an element from $\mathcal{S} \backslash q$ at first and subsequently removing all vertices with *degree* two, merging their *adjacent* edges. If T' and the topology of q are equivalent (i.e. isomorphic) we say T *displays* q. See Figure 5.2 for an example.

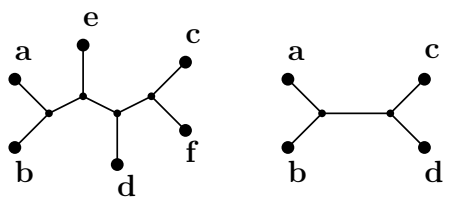


Figure 5.2. An example for a phylogeny and a displayed topology.

The *Quartet Compatibility Problem* can be defined as

Definition 5.1 (Quartet Compatibility (QC))**.** *Let* $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$ *be a set of topologies. The* Quartet Compatibility Problem *is the decision problem whether there exists a phylogeny* T *that displays all topologies* $q_i \in \mathcal{Q}$*.*

The set of topologies $\mathcal{Q}$ on the set of taxa $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ is said to be *complete*, if $\mathcal{Q}$ contains a topology for each of the $\binom{n}{4}$ quartets of $\mathcal{S}$, otherwise $\mathcal{Q}$ is *incomplete*.

It is known that QC can be answered in polynomial time if $\mathcal{Q}$ is *complete*. Given a compatible complete topology set $\mathcal{Q}$, the associated phylogeny is unique and can be constructed within the same time complexity [49]. However, if $\mathcal{Q}$ is incomplete, it has been shown by Steel [133] that QC is NP-complete. The more computationally interesting *Maximum Quartet Consistency Problem* arises when $\mathcal{Q}$ is not compatible is defined as follows:

Definition 5.2 (Maximum Quartet Consistency (MQC))**.** *Let* $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$ *be a set of topologies. The* Maximum Quartet Consistency Problem *determines a phylogeny* T *which displays the maximum number of topologies* $q_i \in \mathcal{Q}$*.*

MQC is shown to be a NP-hard problem if $\mathcal{Q}$ is complete [17], although it admits a polynomial-time approximation scheme [79]. The existing approaches to solve MQC can be categorized as either heuristic or exact.

The broad range of existing heuristics contains for example the short quartet methods of Erdös et al. [50], the approach of semi-definite programming by Ben-Dor et al. [12], the quartet cleaning algorithm of Berry at al. [17] and an algorithm by Wu et al. [148] that computes a phylogeny with a high success probability.

Due to the fact that this chapter focuses on exact solutions for the MQC problem, we will introduce the most important ones in the next section.

5.3 Related Work

This section will introduce the most efficient techniques to solve the MQC problem exactly: In 2004 Wu et al. [150] proposed the use of Answer Set Programming (ASP) for MQC, and in 2008 Morgado and Marques-Silva [103] translated the problem of MQC to Pseudo Boolean Optimization (PBO).

However, there are some other approaches as well. Ben-Dor et al. [12] proposed the use of Dynamic Programming to solve the problem. In their setting the topologies have a weight assigned additionally. The objective is to compute a phylogeny

with a maximal score, which is a phylogeny whose sum of weights of displayed topologies is maximal.

Gramm and Niedermeier [63] developed a fixed-parameter algorithm whose objective was to compute a phylogeny that displays less or equal to k topologies. This algorithm runs in time $O(4^k n + n^4)$ with the number of errors k and $n = |\mathcal{S}|$. A look-ahead branch-and-bound algorithm by Wu et al. [149] can be seen as an improvement for that. It runs in the same running time complexity, but does not require the number of errors k to be known.

5.3.1 Answer set programming

The central point in the approach proposed by Wu et al. [150] is the *ultrametric* phylogeny T. Wu et al. use a *rooted binary* tree as T. This does not violate our definition from before, since every unrooted binary tree can be rooted by inserting a single node on one of its *inner edges*. An *ultrametric* phylogeny needs a special labeling scheme for the set of inner nodes of T to the set of integers $\{1, 2, \ldots, n-1\}$. Since T is binary and rooted, there are exactly $n-1$ inner nodes. However, the labeling scheme does not have to be bijective; two different inner nodes can be assigned to the same integer. A labeling scheme is said to be *ultrametric*, if along each path from the root to any leaf the labels of the inner nodes is strictly decreasing [68]. One phylogeny together with an ultrametric labeling scheme is called an *ultrametric phylogeny*.

With the help of the lowest common ancestors it is tested whether a topology q_k is satisfied by an *ultrametric phylogeny*. The lowest common ancestor of two leaves in a rooted tree is the inner node on the path between the two leaves that is the closest to the root. See Figure 5.3 for an example: the lowest common ancestor of s_1 and s_5 is labeled with the 3.

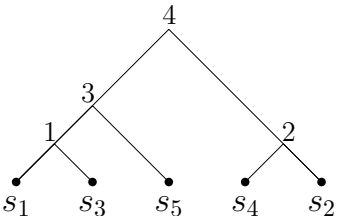


Figure 5.3. An ultrametric phylogeny.

The authors showed that MQC can be interpreted as finding an *optimal ultrametric phylogeny* which satisfies the maximum number of topologies from the input $\mathcal{Q}$. Let $LCAl(s_i, s_j)$ be the label of the lowest common ancestor of the leaves s_i and s_j. Then the topology $q = s_a s_b | s_c s_d, q \in \mathcal{Q}$ is satisfied by an ultrametric phylogeny T, if and only if the following inequality holds:

$$min\{LCAl(s_a, s_c), LCAl(s_b, s_d)\} > min\{LCAl(s_a, s_b), LCAl(s_c, s_d)\}$$

They formulated the problem of finding an *optimal ultrametric phylogeny* as an answer set program (ASP). ASP is a form of logic programming that has its strengths in solving constraint problems in a declarative way. In ASP a given problem is expressed as a logic program. Each answer of this program corresponds to a solution of the given problem. The declarative knowledge representation allows the ASP solver to use different techniques to efficiently compute an answer. In general an ASP can be translated into SAT to be solved by any SAT solver. However, the authors stated that the use of a specialized ASP solver is much more effective. They use the solver `smodels` [130] which is over 6 years old for the experiments. We will

add the more recent ASP solver `clasp` [58, 59] to the experimental setup (see Section 5.6) to ensure that the ASP approach is solved by a solver which incorporates the latest algorithmic improvements of the ASP community.

5.3.2 Pseudo boolean optimization

The PBO approach that was first introduced in 2008 by Morgado and Marques -Silva [103] and extended in 2010 by the same group [104] uses the same central concept of an *ultrametric phylogeny*. In fact they use a matrix representation of the $LCAl(s_i, s_j)$ (see Table 5.1). For the matrix M whose values are $M[s_i, s_j] = LCAl(s_i, s_j)$ to be a *ultrametric matrix* the following properties must be satisfied: (i) all values in M have to be between 1 and n. In fact Wu et al. [150] have shown that the entries can be restricted to $M[s_i, s_j] \leq \lceil \frac{n}{2} \rceil$. (ii) M is symmetric, thus $M[s_i, s_j] = M[s_j, s_i]$, and (iii) for each triple $(s_i, s_j, s_l) : 1 \leq i, j, l \leq n$

Table 5.1. The ultrametric matrix M for the example shown in Figure 5.3.

	s_1	s_2	s_3	s_4	s_5
s_1	0	4	1	4	3
s_2		0	4	2	4
s_3			0	4	3
s_4				0	4
s_5					0

$$
\begin{aligned}
(M[s_i, s_j] = M[s_i, s_l] \quad &\wedge \quad M[s_i, s_l] > M[s_j, s_l]) \vee \\
(M[s_i, s_j] = M[s_j, s_l] \quad &\wedge \quad M[s_j, s_l] > M[s_i, s_l]) \vee \\
(M[s_j, s_l] = M[s_i, s_l] \quad &\wedge \quad M[s_i, s_l] > M[s_i, s_j])
\end{aligned}
$$

The authors developed over the years different encodings, that are combinations of its three main parts: 1. the encoding of the ultrametric matrix, 2. the encoding of the consistency of the topologies, and 3. the encoding of the cost function, that is used in the target function. The authors use the most well-known PBO solver `minisat+` [48] for their approach. `minisat+` translates the pseudo-boolean constraint into *clauses* that can be handled by any SAT solver.

However, it is also possible to convert PBO problems into MaxSAT problems. In fact the benchmark set which is used to evaluate the MaxSAT solvers every year in the MaxSAT-evaluation[1] contains several instances that solve the MQC problem.

Our experiments in Section 5.6 will cover the most effective PBO encodings which will be solved with the help of `minisat+`, as well as the MaxSAT benchmarks. Since we will use MaxSAT for our approach to solve MQC (see Section 5.5), the comparison with existing MaxSAT instances is a good indicator of the performance of our approach.

5.4 SAT Formulation

Let $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$ be a set of topologies on the set of taxa $\mathcal{S} = \{s_1, \ldots, s_n\}$ for which we seek to decide, whether there exists a phylogeny T_n, that displays all topologies $q_i \in \mathcal{Q}$. In the following, we describe a logic formula $\mathcal{F}(\mathcal{Q})$ that will solve this problem by encoding it as a SAT instance. Recall that any SAT problem can

[1] `http://maxsat.ia.udl.cat/introduction/`

be described in *conjunctive normal form* (CNF), which is a conjunction of clauses; each clause being a disjunction of (possibly negated) literals. We will define $\mathcal{F}(\mathcal{Q})$ by its set of variables and a corresponding set of rules. The rules will ensure the proper assignment of the variables and will be given in propositional logic, which can be converted into CNF clauses straightforwardly [115].

Before introducing the SAT formulation we want to outline the idea behind it. The encoding will be composed of five different parts. The first part will ensure a construction of a phylogeny T_n for the complete set of taxa $\mathcal{S}_{input}$ from the input ($|\mathcal{S}_{input}| = n$) by using a sequence of split operations on "intermediate" trees $T_3, \ldots, T_{n-1}$. The second part will determine for each inner edge e in the phylogeny T_n which taxa are existent in the two remaining subtrees if the edge e would be deleted from T_n. We call this the *covering* property of edges. This information will be used to determine, whether T_n displays all input topologies $q_i \in \mathcal{Q}$ in the third part of the encoding. The forth and fifth part will use two different approaches to increase the performance of the SAT solver by pruning a part of the search space that cannot lead to satisfying variable assignments.

5.4.1 The split encoding

This SAT formulation is based on the *split operation* representing the fact that a phylogeny T_n for $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ can be constructed via splitting a single edge e from the phylogeny T_{n-1} for $\mathcal{S} = \{s_1, s_2, \ldots, s_{n-1}\}$ by inserting a new inner node v_n. This inner node v_n will be connected to both endpoints of e, as well as to a new leaf l_n which will be associated with the new taxa s_n.

A phylogeny for a set of taxa $\mathcal{S}$ with $|\mathcal{S}| = i, i \geq 3$ has $2i - 3$ edges, thus there are $2n - 3$ possibilities to construct T_n, given a phylogeny T_{n-1}. Figure 5.4 illustrates the possibilities for the first two split operations: there are three different operations to create T_4 for $\mathcal{S} = \{a, b, c, d\}$ by splitting one of the three *leaf edges* of T_3 for $\mathcal{S} = \{a, b, c\}$, and five possibilities to construct T_5 for $\mathcal{S} = \{a, b, c, d, e\}$ given one of the three versions of T_4 for $\mathcal{S} = \{a, b, c, d\}$.

Building a phylogeny T_n

The variables of $\mathcal{F}(\mathcal{Q})$ should model a phylogeny T_n which displays all topologies $q_i \in \mathcal{Q}$, if it exists. We will handle splits of edges that are between inner nodes (called *inner edges*) and splits of edges that are between one inner node and one leaf (called *leaf edges*) separately. Thus, we will use different variables to determine what kind of edge will be split. The variables $\iota(s_i, e_j)$ determine whether the *inner edge* e_j is split to insert s_i into the phylogeny T_{i-1}. Similarly, the variables $\lambda(s_i, e_k)$ determine whether the *leaf edge* e_k is split to insert s_i into the phylogeny T_{i-1}. Since a phylogeny T_{i-1} contains $i - 4$ *inner* and $i - 1$ *leaf edges*, $\iota(s_i, e_j)$ are defined for all $5 \leq i \leq n, 1 \leq j \leq i - 4$, and $\lambda(s_i, e_k)$ is defined for all $4 \leq i \leq n, 1 \leq k \leq i - 1$.

For every taxa $s_i \in \mathcal{S}$ with $4 \leq i \leq n$ at least one of these variables has to be set to `true` which is ensured by the following rule:

$$((\bigvee \iota(s_i, e_j)) \vee (\bigvee \lambda(s_i, e_k))$$
$$\forall i : 4 \leq i \leq n$$

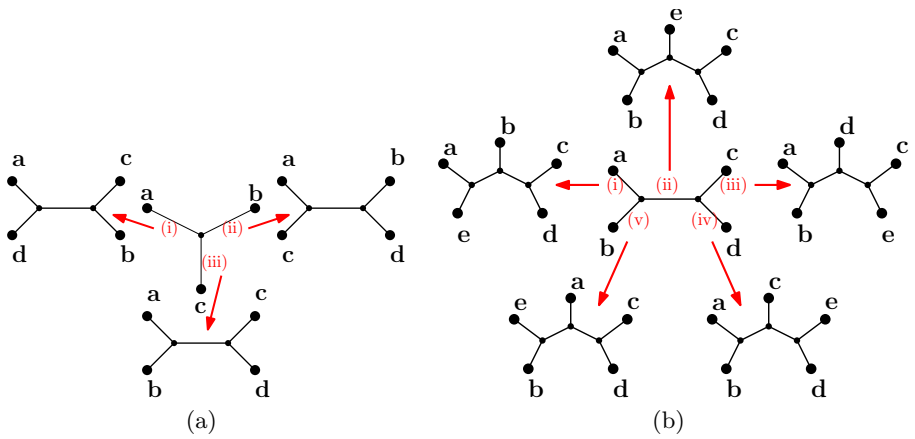


Figure 5.4. (a) The three possibilities to construct T_4 by splitting an one of the three *leaf edges* of T_3. (b) The five possibilities to construct T_5 by splitting either the *inner edge* or one of the four *leaf edge* of T_4.

Together with the following rule which ensures that at most one of these variables is set to `true` we have the *splitting rule*. This forces that every taxa $s_i \in \mathcal{S}$ with $4 \leq i \leq n$ splits exactly one of the edges $\in T_{i-1}$.

$$\neg\iota(s_i, e_j) \vee \neg\iota(s_i, e_k)$$
$$\forall i, j, k : 4 \leq i \leq n, 1 \leq j \leq i-4, 1 \leq k \leq i-4, j \neq k$$
$$\neg\iota(s_i, e_j) \vee \neg\lambda(s_i, e_k)$$
$$\forall i, j, k : 4 \leq i \leq n, 1 \leq j \leq i-4, 1 \leq k \leq i-1$$
$$\neg\lambda(s_i, e_j) \vee \neg\lambda(s_i, e_k)$$
$$\forall i, j, k : 4 \leq i \leq n, 1 \leq j \leq i-1, 1 \leq k \leq i-1, j \neq k$$

Please note that the phylogeny T_4 has exactly one *inner edge* which will be called *inner edge* e_1. Every split introduces a new inner edge which has to be identified uniquely for further split operations and resulting variable assignments the following way.

- splitting a *leaf edge* is trivial, no special considerations needed (see Figure 5.5(a)). The new inner edge gets the index $i-3$.
- splitting an *inner edge* e_j will cause the new *inner edge* e_{i-3} to be inserted on the path from *inner edge* e_j to *inner edge* e_1 right next to e_j (see Figure 5.5(b))

Determine the covered labels

So far we have managed to encode the splitting operations. The next part of the SAT formulation determines for each inner edge e_i in the phylogeny T_n which taxa

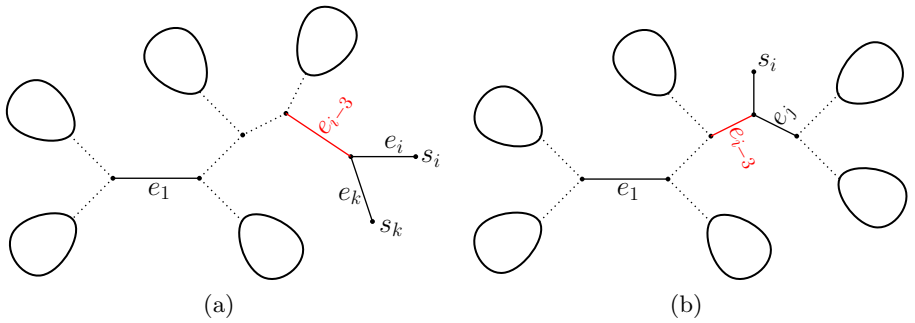


Figure 5.5. (a) The phylogeny T_i after splitting *leaf edge* e_k. (b) The phylogeny T_i after splitting *inner edge* e_j.

s_j are existent in which of the two remaining subtrees if the edge e_i would be deleted from T_n. We call this the *covering* property of edges.

To encode this information we introduce the variables $\rho(e_i, s_j)$ for each pair of *inner edge* $e_i \in T_n$ and taxa $s_j \in \mathcal{S}$. These variables represent for every *inner edge* e_i whether the path from the *leaf edge* e_j (the edge where the leaf associated to the taxa s_j is connected to) to *inner edge* e_1 contains *inner edge* e_i. In other words, these variables describe whether taxa s_j would be in a different connected component than *inner edge* e_1 after deleting *inner edge* e_i from the phylogeny T_n. For *inner edge* e_1 these variables represent whether the taxa s_i would be in the same connected component as s_1 after deleting the *inner edge* e_1. Exemplary variable assignments for the example in 5.5(b) are $\rho(e_{i-3}, s_i) = \texttt{true}$ and $\rho(e_j, s_i) = \texttt{false}$.

To ensure the correct assignment of the $\rho(e_i, s_j)$ variables which will be used to test whether the phylogeny T_n displays all the topologies from the input set, we have to introduce variables $\varphi(e_k, e_l)$ for every distinct pair of *inner edges* $e_k, e_l \in T_n$ which represent the same *cover* relation for *inner edges*.

Any time a *leaf edge* e_k is split (see Figure 5.5(a)) to insert s_i into the phylogeny T_{i-1} the following assignments have to be made via the *split leaf edge rule*:
(i) the new *inner edge* e_{i-3} will cover the taxa s_i and s_k

$$\lambda(s_i, e_k) \rightarrow (\rho(e_{i-3}, s_i) \wedge \rho(e_{i-3}, s_k))$$
$$\forall i, k : 4 \leq i \leq n, 1 \leq k \leq i-1$$

(ii) the new *inner edge* e_{i-3} will not cover the taxa s_j that were already present before s_i is added

$$\lambda(s_i, e_k) \rightarrow \neg\rho(e_{i-3}, s_j)$$
$$\forall i, j, k : 4 \leq i \leq n, 1 \leq j \leq i-1, 1 \leq k \leq i-1, j \neq k$$

(iii) the already present *inner edges* e_j will *cover* the new *inner edge* e_{i-3} as well as the taxa s_i if and only if they *cover* taxa s_k

$$(\lambda(s_i, e_k) \wedge \rho(e_j, s_k)) \rightarrow (\varphi(e_j, e_{i-3}) \wedge \rho(e_j, s_i))$$
$$(\lambda(s_i, e_k) \wedge \neg\rho(e_j, s_k)) \rightarrow (\neg\varphi(e_j, e_{i-3}) \wedge \neg\rho(e_j, s_i))$$
$$\forall i, j, k : 5 \leq i \leq n, 1 \leq j \leq i-4, 1 \leq k \leq i-1$$

Any time an *inner edge* e_k is split (see Figure 5.5(b)) to insert s_i into the phylogeny we have to force the following via the *split inner edge rule*:
(i) the new *inner edge* e_{i-3} will cover the *inner edge* e_k as well as the taxa s_i, whereas the *inner edge* e_k will not cover *inner edge* e_{i-3} nor the taxa s_i

$$\iota(s_i, e_k) \rightarrow (\varphi(e_{i-3}, e_k) \wedge \rho(e_{i-3}, s_i) \wedge \neg\varphi(e_k, e_{i-3}) \wedge \neg\rho(e_k, s_i))$$
$$\forall i, k : 5 \leq i \leq n, 1 \leq k \leq i-4$$

(ii) the new *inner edge* e_{i-3} covers the already present taxa $s_j \neq s_k$ if and only if the *inner edge* e_k covers it, same with already present *inner edges* e_j

$$(\iota(s_i, e_k) \wedge \rho(e_k, s_j)) \rightarrow (\rho(e_{i-3}, s_j))$$
$$(\iota(s_i, e_k) \wedge \neg\rho(e_k, s_j)) \rightarrow (\neg\rho(e_{i-3}, s_j))$$
$$(\iota(s_i, e_k) \wedge \varphi(e_k, e_j)) \rightarrow (\varphi(s_{i-3}, e_j))$$
$$(\iota(s_i, e_k) \wedge \neg\varphi(e_k, e_j)) \rightarrow (\neg\varphi(s_{i-3}, e_j))$$
$$\forall i, j, k : 5 \leq i \leq n, 1 \leq j \leq i-4, 1 \leq k \leq i-4, j \neq k$$

(iii) the already present *inner edges* e_j will *cover* the new *inner edge* e_{i-3} as well as the taxa s_i if and only if they *cover* the *inner edge* e_k

$$(\iota(s_i, e_k) \wedge \varphi(e_j, e_k)) \rightarrow (\varphi(e_j, e_{i-3}) \wedge \rho(e_j, s_i))$$
$$(\iota(s_i, e_k) \wedge \neg\varphi(e_j, e_k)) \rightarrow (\neg\varphi(e_j, e_{i-3}) \wedge \neg\rho(e_j, s_i)$$
$$\forall i, j, k : 5 \leq i \leq n, 1 \leq j \leq i-4, 1 \leq k \leq i-4, j \neq k$$

Topology display test

So far, we have managed to encode the splitting operations to create T_n, and the *covering* information of all *inner edges* $\in T_n$. It remains to ensure that every topology $q_i \in \mathcal{Q}$ is displayed by the phylogeny T_n. Thus, we introduce variables $\delta(e_i, q_k)$ that represent whether *inner edge* e_i separates the phylogeny T_n into two components, with one of the components containing s_a and s_b, but not s_c nor s_d for the topology $q_k = s_a s_b | s_c s_d$. They are properly assigned by the following *edge separation rule*:

$$(\rho(e_i, s_a) \wedge \rho(e_i, s_b) \wedge \neg\rho(e_i, s_c) \wedge \neg\rho(e_i, s_d)) \vee$$
$$(\neg\rho(e_i, s_a) \wedge \neg\rho(e_i, s_b) \wedge \rho(e_i, s_c) \wedge \rho(e_i, s_d))$$
$$\rightarrow \delta(e_i, q_k)$$
$$\forall i, k : 1 \leq i \leq n-3, q_k = s_a s_b | s_c s_d \in \mathcal{Q}$$

Of course, for every input topology q_k on of the $\delta(e_i, q_k)$ has to be set to `true`. Otherwise the topology q_k would not be displayed by the phylogeny T_n. This is ensured by the *displayed quartet rule*:

$$(\bigvee \delta(e_i, q_k))$$
$$\forall i, k : 1 \leq i \leq n-3, q_k \in \mathcal{Q}$$

Theorem 5.3. *(Correctness) Let $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$ be a set of topologies on the set of taxa $\mathcal{S} = \{s_1, \ldots, s_n\}$. Then, there exists a phylogeny T_n which displays all topologies $q_i \in \mathcal{Q}$ if and only if $\mathcal{F}(\mathcal{Q})$ is satisfiable.*

Proof. To prove this theorem we have to show that: (i) a phylogeny T_n which displays all topologies $q_i \in \mathcal{Q}$ yields a satisfying assignment of $\mathcal{F}(\mathcal{Q})$ and (ii) a satisfying assignment of $\mathcal{F}(\mathcal{Q})$ yields a phylogeny T_n which displays all topologies $q_i \in \mathcal{Q}$.

(i) From a phylogeny to an assignment: Assume, that T_n is the phylogeny for the set of taxa $\mathcal{S} = \{s_1, s_2, \ldots, s_n\}$ which displays all topologies $\mathcal{Q}$. We define an assignment $(\hat{\iota}, \hat{\lambda}, \hat{\rho}, \hat{\varphi}, \hat{\delta})$ to the ι-, λ-, ρ, φ and δ-variables consistent with the intended meaning of the variables:

- $\hat{\iota}(s_i, e_j) =$ `true`, if and only if the restriction of T_n to $\mathcal{S}_i = \{s_1, \ldots, s_i\}$ $(T_n|\mathcal{S}_i)$ is created by splitting the *inner edge* e_j from the $T_n|\mathcal{S}_{i-i}$ with $\mathcal{S}_{i-1} = \{s_1, \ldots, s_{i-1}\}$
- $\hat{\lambda}(s_i, e_k) =$ `true`, if and only if the restriction of T_n to $\mathcal{S}_i = \{s_1, \ldots, s_i\}$ $(T_n|\mathcal{S}_i)$ is created by splitting the *leaf edge* e_k from the $T_n|\mathcal{S}_{i-i}$ with $\mathcal{S}_{i-1} = \{s_1, \ldots, s_{i-1}\}$
- $\hat{\rho}(e_1, s_j) =$ `true`, if and only if the leaf that is associated with the taxa s_j is in the same connected component as s_1 when the *inner edge* e_1 is removed from the phylogeny T_n
- $\hat{\rho}(e_i, s_j) =$ `true` with $e_i \neq e_1$, if and only if the leaf that is associated with the taxa s_j is in a different connected component as the *inner edge* e_1 when the *inner edge* e_i is removed from the phylogeny T_n
- $\hat{\varphi}(e_1, e_j) =$ `true`, if and only if the *inner edge* e_j is in the same connected component as s_1 when the *inner edge* e_1 is removed from the phylogeny T_n
- $\hat{\varphi}(e_i, e_j) =$ `true` with $e_i \neq e_1$, if and only if the *inner edge* e_j is in a different connected component as the *inner edge* e_1 when the *inner edge* e_i is removed from the phylogeny T_n
- $\hat{\delta}(e_i, q_k)) =$ `true`, if and only if T_n is partitioned in the connected components C_1 and C_2, for which w.l.o.g. C_1 contains $\{s_a, s_b\}$ and C_2 contains $\{s_c, s_d\}$ for the quartet $q_k = s_a s_b | s_c s_d$

To prove that the assignment $(\hat{\iota}, \hat{\lambda}, \hat{\rho}, \hat{\varphi}, \hat{\delta})$ satisfies $\mathcal{F}(\mathcal{Q})$, we consider all rules of $\mathcal{F}(\mathcal{Q})$:

- The *splitting rule* is satisfied by $(\hat{\iota}, \hat{\lambda}, \hat{\rho}, \hat{\varphi}, \hat{\delta})$, since there is exactly one split of an *inner edge* or *leaf edge* which creates the phylogeny $T_n|\mathcal{S}_i$ given the next smaller phylogeny $T_n|\mathcal{S}_{i-1}$.
- To prove that the *split leaf edge rule* is satisfied we have to consider all three subcases of it:

Case (i) and (ii) are trivially satisfied, since when deleting the newly inserted *inner edge* e_{i-3} from $T_n|\mathcal{S}_i$, only the taxa $s_k, k < i$ associated to the split *leaf edge* e_k and the newly added s_i are in a different connected component than s_1 (case (i)). All other taxa $s_j, j \neq \{1, i, k\}$ are in the same component as s_1 (case (ii)).

Case (iii): This case ensures the transitivity of the cover relationship. With the property that T_n and every intermediate phylogeny T_i with $3 \leq i \leq n-1$ are trees this case is satisfied trivially as well.

- We consider each of the three subcase of the *split inner edge rule*:

  Case (i): This subcase ensures the antisymmetry of the cover relationship for *inner edges* which is trivially satisfied, since the newly inserted *inner edge* e_{i-3} is directly adjacent to the split *inner edge* e_k (see Figure 5.5(b)).

  Case (ii): Since the newly inserted *inner edge* e_{i-3} is directly adjacent to the split *inner edge* e_k, it has to cover the same taxa as e_k. This transitivity is ensured here and by the property that we are working on phylogenies (that are trees) this subcase is satisfied trivially.

  Case (iii): This case is done analogously to the case (iii) from the *split leaf edge rule*.

- The *edge separation rule* is satisfied due to the definition of the assignment $(\hat{\iota}, \hat{\lambda}, \hat{\rho}, \hat{\varphi}, \hat{\delta})$.

- It remains to show that the *displayed quartet rule* is satisfied which is trivially true, since T_n was given as a phylogeny that displays all quartets from the input $\mathcal{Q}$.

(ii) From an assignment to a phylogeny: Let $(\hat{\iota}, \hat{\lambda}, \hat{\rho}, \hat{\varphi}, \hat{\delta})$ be a satisfying assignment to $\mathcal{F}(\mathcal{Q})$. Let T_n be the corresponding phylogeny that is created by a sequence of split operations for the taxa $s_4, \ldots, s_n \in \mathcal{S}$. Every split that introduces a new taxa s_i into the phylogeny T_{i-1} can be uniquely identified, since exactly one of the split variables for *inner edges* or *leaf edges* has to be `true` due to the *splitting rule*. We start from a 3-star (which equals the complete bipartite graph $K_{1,3}$) that is acyclic and connected (see Figure 5.4(a)). Since the split operations conserve the connectivity and acyclicity, we know that T_n is a tree. Furthermore we know that the degree requirements of a unrooted binary tree are not violated, thus T_n is a phylogeny. For the sake of contradiction, assume that T_n is not displaying a particular $q_k = s_a s_b | s_c s_d, q_k \in \mathcal{Q}$ Then T_n has to display either $q' = s_a s_c | s_b s_d$ or $q'' = s_a s_d | s_b s_c$, which are the other two topologies for the quartet $\{s_s, s_b, s_c, s_d\}$ (see Figure 5.1). W.l.o.g. we assume that T_n displays q'. In this case one of the following to cases must hold for at least one *inner edge* e_i:

C.1 $\rho(e_i, s_a) = \texttt{true}, \rho(e_i, s_c) = \texttt{true}, \rho(e_i, s_c) = \texttt{false}, \rho(e_i, s_d) = \texttt{false}$

C.2 $\rho(e_i, s_a) = \texttt{false}, \rho(e_i, s_c) = \texttt{false}, \rho(e_i, s_c) = \texttt{true}, \rho(e_i, s_d) = \texttt{true}$

However, neither C.1 nor C.2 is possible, since both confidurations do not comply with the *edge separation rule* of $\mathcal{F}(\mathcal{Q})$. Therefore, T_n does not display q' nor q'' which means directly that T_n displays q_k, as desired. □

The split encoding is able solve the Quartet Consistency problem. However, with the help of an extensive practical analysis in Section 5.6 we will see that the current SAT formulation has only limited efficiency.

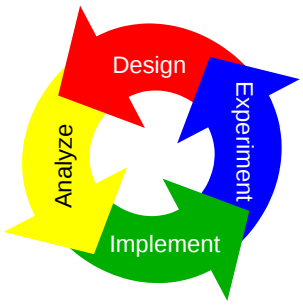


Figure 5.6. The main algorithm engineering cycle.

The first iteration of the main algorithm engineering cycle (see Figure 5.6) is finished with the experiments of the basic SAT formulation $\mathcal{F}(\mathcal{Q})$. We analyzed the experimental results and recognized that the SAT solvers spend a lot of their time in areas of the search space where no model can be found. Assuming that the decisions in the DPLL algorithm are made exclusively on the ι and λ variables which represent the splitting options, all the other variables will be assigned based on either the *pure literal rule* or the *unit propagation rule*. Thus, the conflicts can be detected only via the clauses that were added by the *displayed quartet rule*. Note that the proper assignment of the $\delta(e_i, s_x)$ variables is forced by the $\rho(e_i, s_y)$ variables and the proper assignment of the $\rho(e_i, s_y)$ variables is forced by the ι and λ variables for a taxon s_{y+3}. Thus, a conflict can occur only when all n taxa are inserted into the phylogeny. To overcome that problem we introduce two extensions to the formulation that enable us to

(i) detect conflicts much earlier; specifically at the time when the last taxa of a quartet is inserted into the phylogeny by the split operation (Section 5.4.2),

(ii) forbid splittings that do not lead to a phylogeny that will display a quartet $q_i \in \mathcal{Q}$ (Section 5.4.3).

5.4.2 Using non-displayed quartets to prune search space

This section introduces an extension to the SAT formulation $\mathcal{F}(\mathcal{Q})$ which is based on the following:

Definition 5.4. *(Violated topology) Let $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$ be a set of topologies on the set of taxa $\mathcal{S} = \{s_1, \ldots, s_n\}$ and T_n a phylogeny on $\mathcal{S}$. A topology $q_i = s_a s_b | s_c s_d \in \mathcal{Q}$ is violated by an* inner edge *e_j, if and only if e_j separates the phylogeny T_n into two (connected) components, where both components contain exactly one member of $\{s_a, s_b\}$ and one member of $\{s_c, s_d\}$.*

Theorem 5.5. *Assume T_n is a phylogeny. If T_n contains an* inner edge e_i *which violates the topology $q_k = s_a s_b | s_c s_d$, then $\nexists e_j$, such that $\delta(e_j, q_k) =$ `true`. In other words, q_k cannot be displayed by any* inner edge $\in T_n$.

Proof. It follows directly with the observation that a phylogeny T_n displays exactly one of the three topologies for the quartet $\{s_a, s_b, s_c, s_d\}$: (i) $s_a s_b | s_c s_d$ (ii) $s_a s_c | s_b s_d$ (iii) $s_a s_d | s_b s_c$. If the *inner edge* e_i violates the topology $q_k = s_a s_b | s_c s_d$, then it displays one of the other topologies (denoted q') and thus, the whole T_n cannot display q_k, since it displays q'. □

We will extend the formula $\mathcal{F}(\mathcal{Q})$ with new variables and rules to encode the Theorem 5.5. The resulting formula will be denoted $\mathcal{F}_v(\mathcal{Q})$. We introduce a new variable $\delta(q_k)$ for every $q_k \in \mathcal{Q}$ which describes whether the topology q_k is displayed by at least one *inner edge* of the phylogeny T_n. The old *displayed quartet rule* from $\mathcal{F}(\mathcal{Q})$ will be dropped in favor of the *new displayed quartet rule*:

$$
\begin{gathered}
(\bigvee \delta(e_i, q_k)) \leftrightarrow \delta(q_k) \\
\delta(q_k) \\
\forall i, k : 1 \leq i \leq n-3, q_k \in \mathcal{Q}
\end{gathered}
$$

Note that the second part of the *new displayed quartet rule* is represented by unit clauses which fix $\delta(q_k)$ to be `true`. Via *unit propagation* (which is a basic operation performed by all SAT solvers [47]) other constraints may become simpler or even already satisfied. We are using this unit clause to force a conflict via the *violating topology rule*:

$$
\begin{gathered}
((\rho(e_i, s_a) \wedge \neg\rho(e_i, s_b) \wedge \rho(e_i, s_c) \wedge \neg\rho(e_i, s_d)) \vee \\
(\rho(e_i, s_a) \wedge \neg\rho(e_i, s_b) \wedge \neg\rho(e_i, s_c) \wedge \rho(e_i, s_d)) \vee \\
(\neg\rho(e_i, s_a) \wedge \rho(e_i, s_b) \wedge \rho(e_i, s_c) \wedge \neg\rho(e_i, s_d)) \vee \\
(\neg\rho(e_i, s_a) \wedge \rho(e_i, s_b) \wedge \neg\rho(e_i, s_c) \wedge \rho(e_i, s_d))) \\
\rightarrow \neg\delta(q_k) \\
\forall i, k : 1 \leq i \leq n-3, q_k = s_a s_b | s_c s_d \in \mathcal{Q}
\end{gathered}
$$

With the help of this conflict the SAT solver "recognizes" much earlier that the current T_n is not displaying $\mathcal{Q}$. The old formulation $\mathcal{F}(\mathcal{Q})$ does not allow this early recognition. Once a topology q_k is not displayed by an *inner edge* e_i the SAT solver would still have the chance to satisfy the clauses added by the *displayed quartet rule* in $\mathcal{F}(\mathcal{Q})$ via the other $\delta(e_j, q_k), (e_j \neq e_i)$ variables. Thus, the SAT solver is allowed spend a lot of computation time in areas of the search space where no satisfying assignment can be found. The extended formulation $\mathcal{F}_v(\mathcal{Q})$ removes these areas of the search space by introducing the conflict which should result in a much better performance.

Theorem 5.6. *(Correctness) Let $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$ be a set of topologies on the set of taxa $\mathcal{S} = \{s_1, \ldots, s_n\}$. Then, there exists a phylogeny T_n which displays all topologies $q_i \in \mathcal{Q}$ if and only if $\mathcal{F}_v(\mathcal{Q})$ is satisfiable.*

Proof. Follows directly from the correctness of $\mathcal{F}(\mathcal{Q})$. □

5.4.3 Using the input to prune the search space

This section introduces another approach to reduce the available search space for the SAT solver. We will use the information given by the set of input topologies $\mathcal{Q}$ to forbid particular splits of *leaf edges* as well as *inner edges* to prevent that the SAT solver spends time in a subtree of the search space that cannot lead to a satisfying variable assignment for the particular SAT formula.

This approach is based on the following two theorems. We will extend the formula $\mathcal{F}(\mathcal{Q})$ with new variables and rules to encode both theorems. The resulting formula will be denoted $\mathcal{F}^s(\mathcal{Q})$. In fact, the formula $\mathcal{F}_v(\mathcal{Q})$ which incorporates the extension introduced in Section 5.4.2 can be extended as well. The resulting formula will be denoted $\mathcal{F}_v^s(\mathcal{Q})$

Theorem 5.7 (Pruned leaf edge splits)**.** *W.l.o.g. we assume that s_d is the taxa with the highest index in the topology $q_k = s_a s_b | s_c s_d, q_k \in \mathcal{Q}$. The following splits will never lead to a phylogeny T_n that displays q_k and should be forbidden:*

1. *s_d splits* leaf edge e_a *or* leaf edge e_b

2. *s_d splits* leaf edge e_x *if* $\exists$ inner edge e_i*, such that e_i* covers *exactly one of $\{s_x, s_c\}$ and exactly one of $\{s_a, s_b\}$ with s_x being the taxa that is connected via the leaf edge e_x*

Proof. We have to consider both forbidden cases:
Case 1: Assume that s_d splits the *leaf edge* $e_a \in T_{d-1}$. With the *split leaf edge rule* (see Figure 5.5(a), Section 5.4.1) a new *inner edge* e_{d-3} is inserted, which *covers* exactly s_d and s_a. Neither s_c, nor s_b is covered by the new *inner edge* $e_{d-3} \in T_d$. Thus, the *inner edge* violates the topology q_k. With Theorem 5.5 we know that T_d does not display q_k, and thus T_n will not display q_k either, falsifying our assumption. The case for splitting the *leaf edge* e_b is done analogously.

Case 2: Assume that s_d splits the *leaf edge* e_x. Furthermore assume $\exists$ *inner edge* e_i, such that e_i *covers* s_x and exactly s_a, but does not *cover* s_b, nor s_c. With the *split leaf edge rule* (see Figure 5.5(a), Section 5.4.1) we know that the *inner edge* e_i *covers* s_a and s_d now (since it covers s_x as well), but neither s_b, nor s_c. Thus, e_i is violating the topology q_k for the phylogeny T_d. With Theorem 5.5 we know that T_d does not display q_k, and thus T_n will not display q_k either, falsifying our assumption. The other three possible combinations of *covered* taxa are done analogously. □

The corresponding *pruned leaf edge split rule* which incorporates Theorem 5.7 is

$$(\neg\lambda(s_d, e_a) \wedge \neg\lambda(s_d, e_b))$$
$$\forall k : q_k = s_a s_b | s_c s_d \in \mathcal{Q} \text{ with } a, b, c < d$$

$$\begin{gathered}((\rho(e_i, s_x) \wedge \rho(e_i, s_a) \wedge \neg\rho(e_i, s_b) \wedge \neg\rho(e_i, s_c)) \vee \\ (\rho(e_i, s_x) \wedge \neg\rho(e_i, s_a) \wedge \rho(e_i, s_b) \wedge \neg\rho(e_i, s_c)) \vee \\ (\neg\rho(e_i, s_x) \wedge \rho(e_i, s_a) \wedge \neg\rho(e_i, s_b) \wedge \rho(e_i, s_c)) \vee \\ (\neg\rho(e_i, s_x) \wedge \neg\rho(e_i, s_a) \wedge \rho(e_i, s_b) \wedge \rho(e_i, s_c))) \\ \rightarrow \neg\lambda(s_d, e_x) \\ \forall x, k : q_k = s_a s_b | s_c s_d \in Q \text{ with } a, b, c < d, 1 \leq x \leq d-1\end{gathered}$$

The first part of the *pruned leaf edge split rule* is again represented by a set of unit clauses that fix some λ variables to be assigned to `false`.

Theorem 5.8 (Pruned inner edge splits)**.** *W.l.o.g. we assume that s_d is the taxa with the highest index in the topology $q_k = s_a s_b | s_c s_d, q_k \in \mathcal{Q}$. The following splits will never lead to a phylogeny T_n that displays q_k and should be forbidden:*

1. *s_d splits* inner edge *e_x if e_x covers s_c and exactly one of $\{s_a, s_b\}$*
2. *s_d splits* inner edge *e_x if* $\exists$ inner edge *e_y, such that e_y covers exactly one out of $\{e_x, s_c\}$ and exactly one out of $\{s_a, s_b\}$*
3. *s_d splits* inner edge *e_x if e_x covers either exactly all or none of $\{s_a, s_b, s_c\}$ and* $\exists$ inner edge *e_y, such that e_y covers s_c and exactly one of $\{s_a, s_b\}$*

Proof. We have to consider all four cases:
The cases 1 and 2 are done analogously to the cases 1 and 2 from Theorem 5.7.

Case 3: Assume s_d splits the *inner edge* e_x which covers $\{s_a, s_b, s_c\}$. Furthermore assume $\nexists$ *inner edge* e_y, such that e_y covers s_c and exactly one out of $\{s_a, s_b\}$. With the *split inner edge rule* (see Figure 5.5(b)) we know that e_x will not cover s_d. Thus, an *inner edge* which displays q_k has to be in the set of *inner edges* that are covered by e_x. Since e_y cannot cover s_d, it violates the topology q_k. By Theorem 5.5 we know that q_k cannot be displayed, falsifying the assumption. The other subcases are done analogously. □

The corresponding *pruned inner edge split rule* which incorporates Theorem 5.8 is

$$\begin{gathered}((\rho(e_x, s_c) \wedge \rho(e_x, s_a) \wedge \neg\rho(e_x, s_b)) \vee \\ (\rho(e_x, s_c) \wedge \neg\rho(e_x, s_a) \wedge \rho(e_x, s_b)) \\ \rightarrow \neg\iota(s_d, e_x) \\ \forall x, k : q_k = s_a s_b | s_c s_d \in \mathcal{Q} \text{ with } a, b, c < d, 1 \leq x \leq d-4\end{gathered}$$

$$\begin{gathered}((\varphi(e_y, e_x) \wedge \rho(e_y, s_a) \wedge \neg\rho(e_y, s_b) \wedge \neg\rho(e_y, s_c)) \vee \\ (\varphi(e_y, e_x) \wedge \neg\rho(e_y, s_a) \wedge \rho(e_y, s_b) \wedge \neg\rho(e_y, s_c)) \vee \\ (\neg\varphi(e_y, e_x) \wedge \rho(e_y, s_a) \wedge \neg\rho(e_y, s_b) \wedge \rho(e_y, s_c)) \vee \\ (\neg\varphi(e_y, e_x) \wedge \neg\rho(e_y, s_a) \wedge \rho(e_y, s_b) \wedge \rho(e_y, s_c))) \\ \rightarrow \neg\iota(s_d, e_x) \\ \forall y, x, k : q_k = s_a s_b | s_c s_d \in \mathcal{Q} \text{ with } a, b, c < d, 1 \leq x, y \leq d-4\end{gathered}$$

$$
\begin{gathered}
((\rho(e_x, s_a) \wedge \rho(e_x, s_b) \wedge \rho(e_x, s_c) \wedge \rho(e_y, s_a) \wedge \neg\rho(e_y, s_b) \wedge \rho(e_y, s_c)) \vee \\
(\rho(e_x, s_a) \wedge \rho(e_x, s_b) \wedge \rho(e_x, s_c) \wedge \neg\rho(e_y, s_a) \wedge \rho(e_y, s_b) \wedge \rho(e_y, s_c)) \vee \\
(\neg\rho(e_x, s_a) \wedge \neg\rho(e_x, s_b) \wedge \neg\rho(e_x, s_c) \wedge \rho(e_y, s_a) \wedge \neg\rho(e_y, s_b) \wedge \rho(e_y, s_c)) \vee \\
(\neg\rho(e_x, s_a) \wedge \neg\rho(e_x, s_b) \wedge \neg\rho(e_x, s_c) \wedge \neg\rho(e_y, s_a) \wedge \rho(e_y, s_b) \wedge \rho(e_y, s_c))) \\
\rightarrow \neg\iota(s_d, e_x) \\
\forall y, x, k : q_k = s_a s_b | s_c s_d \in \mathcal{Q} \text{ with } a, b, c < d, 1 \leq x, y \leq d - 4
\end{gathered}
$$

Theorem 5.9. *(Correctness) Let $\mathcal{Q} = \{q_1, q_2, \ldots, q_k\}$ be a set of topologies on the set of taxa $\mathcal{S} = \{s_1, \ldots, s_n\}$. Then, there exists a phylogeny T_n which displays all topologies $q_i \in \mathcal{Q}$ if and only if $\mathcal{F}^s(\mathcal{Q})$ is satisfiable.*

Proof. Follows directly from the correctness of $\mathcal{F}(\mathcal{Q})$. □

5.5 Solving MQC

To solve the Maximum Quartet Consistency (MQC) problem we use the concept of a *Minimal Correction Set* (MCS). Recall that an MCS is a subset of an infeasible constraint set (SAT formula) whose removal from that formula results in a satisfiable set of constraints. Thus, the removal "corrects" the infeasibility. MCSes are minimal in the sense that any proper subset does not have that correcting property.

Definition 5.10 (Minimal Correction Set (MCS))**.** *A subset $M \subset C$ is an MCS if $C \setminus M$ is satisfiable and $\forall c_i \in M : C \setminus (M \setminus \{c_i\})$ is unsatisfiable.*

There are several algorithms available that enumerate all MCSes of a SAT formula, e.g. `MCSls` [99], `picoMCS` [111] or `CAMUS` [91]. We are only interested in the smallest MCS of all MCSes. For this purpose we will use `CAMUS` and an upgraded version of `picoMCS`.

We want to recall here the basic idea of `CAMUS` (see Section 4.3.2). `CAMUS` is known as a complete MUS enumerator, that works in two phases. The first phase computes all MCSes in a "top-down" search through the power set lattice of a constraint set by searching level-by-level (a level contains all subsets of a particular size). In the second phase, the MUSes will be computed by a hitting set approach. We will neglect the whole second part of the computation and the majority of the first phase, since we are only interested in the smallest possible MCS. Since `CAMUS` computes the MCSes in its first phase in sorted fashion, starting with the smallest MCSes, we can abort the whole computation after the first MCS is reported by the algorithm.

To be able to report the MCSes in a sorted fashion, `CAMUS` introduces a second layer of variables and clauses to the original formula. The variables in this second layer are so-called selector variables. Every constraint C_i in the formula F will be extended by a new selector variable s_i, thus $C_i' = C_i \cup \{s_i\}$. Since these selector variables are only present in its corresponding extended clause, they can be used to implicitly delete the clause C_i' from F by assigning s_i to `true`. Assigning s_i to `false` would restore the "old" clause C_i and its satisfiability. Adding new

cardinality constraints to the second layer, that only work on selector variables enables `CAMUS` to search for MCSes in ascending order.

This approach of `CAMUS` was implemented by us into `picoMCS` to force the solver into finding the MCSes in ascending order. By doing so, we can abort the computation of further MCSes after the first MCS is found again, since we know, that all further MCSes have to have at least the same size as the first reported MCS.

Recall that an MCS can also be defined as the complement of a **Maximal Satisfiable Subset** (MSS):

Definition 5.11 (Maximal Satisfiable Subset (MSS))**.** *A subset $M \subseteq F$ is an MSS $\Leftrightarrow$ M is satisfiable and $\forall c \in (F \setminus M) : M \cup c$ is unsatisfiable.*

The problem of finding the MSS with the highest cardinality is also known as the maximum satisfiability problem (MaxSAT). Clearly, any MaxSAT solution is an MSS, but the converse does not necessarily hold.
MaxSAT, as well as MCS, can be lifted to so-called group SAT instances, where constraints are modeled as a *set* or *group* of clauses, leading to *group MaxSAT* (gMaxSAT) [73] and *group MCS* (gMCS).

Definition 5.12 (group SAT instance)**.** *A group SAT instance is an explicitly partitioned CNF formula $F = D \cup \bigcup_{G \in \mathcal{G}} G$ with $\mathcal{G} = \{G_1, \ldots, G_k\}$, D and each G_i disjoint sets of clauses.*

D is the default group (often denoted as being group G_0). In gMCS and gMaxSAT this group has to be considered specifically. Whereas G_0 is not allowed to be part of any correction set, the clauses of this group have to be satisfied in gMaxSAT, leading to a partial gMaxSAT instance.

Definition 5.13 (group Minimal Correction Set (gMCS))**.** *A subset $gM \subseteq \mathcal{G}$ of a group SAT instance F is a gMCS if $D \cup (\mathcal{G} \setminus gM)$ is satisfiable and $\forall G_i \in gM$: $D \cup (\mathcal{G} \setminus (gM \setminus \{G_i\})$ is unsatisfiable.*

Definition 5.14 ((partial) group MaxSAT)**.** *Given a group SAT instance F,* group MaxSAT (gMaxSAT) *is the problem of finding a variable assignment that satisfies D (hard constraints) and minimizes the amount of unsatisfied groups $G_i \in \mathcal{G}$. If D is non-empty the problem is called* partial gMaxSAT.

Solving MQC requires the usage of a group SAT instance. The corresponding (grouped) SAT formula will be created as follows. Any input topology $q_i \in \mathcal{Q}$ causes the addition of a set of clauses to the formula due to the *displayed quartet rule*. For every topology q_i this set will form a distinct group G_i. All other clauses of the formula, that were added due to the other rules to $\mathcal{F}(\mathcal{Q})$, will be the so-called **phylogeny-construction** group $D = G_0$.

Note, that the groups $G_i : i > 0$ contain only a single clause. In that case a partial gMaxSAT instance can be also interpreted as a "normal" partial MaxSAT instance, where the soft clauses are not grouped together. The same holds for the extended formula $\mathcal{F}_v(\mathcal{Q})$. In that case, only the unit clauses, that are added via the *new displayed quartet rule* are soft clauses, all other clauses of $\mathcal{F}_v(\mathcal{Q})$ are hard.

Any formula, that incorporates the second extension (*pruned leaf edge split rule* and *pruned inner edge split rule*) for a topology q_i has to be considered to be a grouped instance, since the clauses that are added by these two rules have to added to the groups G_i.

5.6 Practical Results

This section analyzes the results obtained from running the existing ASP solution [150], the most efficient PBO models [104] as well as gMaxSAT and gMCS approaches based on the SAT encoding for QC described in this work. First, the experimental setup is described, followed by the analysis of the experimental results for the sets of instances.

The instances were obtained from [150] and correspond to a set of artificially generated instances. For a set of n taxa the authors generate a phylogeny by recursively joining randomly selected subtrees. The subtrees are selected from a set that initially contained the one-node subtrees corresponding to the given taxa. When two subtrees are joined they are replaced in the set by the generated subtree. This procedure continues until the set contains only one tree: a phylogeny on n taxa. From that phylogeny the set of $\binom{n}{4}$ quartet topologies are derived. To introduce potential errors p% of the topologies are altered. Note that this process guarantees an upper bound on the number of quartet errors in the dataset of $\frac{p}{100}\binom{n}{4}$. Since some combinations of topology alterations might result in a new compatible set of quartet topologies the number of quartet errors can be less than the number of alterations. The possible error percentages p are 0%, 1%, 2%, 5%, 10%, 15%, 20%, 25%, and 30%. 10 different datasets for each pair (n, p) build one test set. We report the average runtime to solve the 10 instances for every pair (n, p).

The encoders/solvers used in the experiments can be divided in three categories, (i) **phy+ASP solver**, (ii) **PBO encoder+PBO solver**, and (iii) **SAT**-based solver.

(i) **phy+ASP solver**. `phy` is an encoder of the MQC problem into an answer set program (ASP) from [150]. The approach of using `phy` with an ASP solver is known to be the best for error percentages of 5% to 15%. `phy` receives the number of taxa n, the maximum number of allowed quartet errors err_{max} and the topologies $\mathcal{Q}$ as an input.

The ASP solver `smodels`[130] that is known to be particular efficient for this encoding is executed to check the feasibility to build an topology T_n given that at most err_{max} topologies $q_i \in \mathcal{Q}$ are not displayed in T_n.

Since the latest `smodels` version (`v2.36`) is from May 2009 we tested the very common ASP solver `clasp` [58] (`v3.1.0`) as well to ensure that the ASP approach is solved by a solver that incorporates the latest algorithmic improvements of the ASP community.

(ii) **PBO encodings+PBO solver**. This combination is the best approach in the literature for error percentages above 20%. It is characterized by first

using one of the encoders described in [104] to obtain a pseudo boolean optimization (PBO) file which is given to a PBO solver. For these experiments we use the most efficient PBO encodings `bin` and `bin-sc` [104] and the `minisat+` solver (`v1.0`) [48] that proved to be most efficient for these encodings.

(iii) **SAT+gMCS/gMaxSAT solver**. This approach uses the SAT encoding for the QC problem presented in section 5.4 to create a group SAT instance as described in Section 5.5.

The gMCS solvers of our choice are `CAMUS` and `picoMCS`. `CAMUS` is started with the parameter `-l 2` to ensure that the computation aborts after the first MCS which does not contain the phylogeny-construction group $D = G_0$ is found. `PicoMCS` uses selector-variables to determine which groups are present in the current MCS. We upgraded `picoMCS`, such that it uses additional constraints over these selector variables to find the smallest MCS. The gMaxSAT approach uses the $\top$-encoding [73] for encoding groups since it proved to be the most effective for our purpose. As MaxSAT solvers we chose the solvers that were particular effective for a set of application instances in the partial MaxSAT track of the *MaxSAT evaluation 2014*: `eva500a` [108], `mscg` [78] and `qMaxSAT` [87].

The results were obtained on quad-core CPUs with 2.83 GHz and 8 GB RAM [1].

5.6.1 Comparison of SAT approaches and formulations

The average runtime of the gMCS and the gMaxSAT approaches for all possible solvers on the test set with 10 taxa is shown in Table 5.2. The best formulation for nearly all combinations is $\mathcal{F}_v(\mathcal{Q})$. Only for the solver `picoMCS` the formulation $\mathcal{F}_v^s(\mathcal{Q})$ was better for 7 out of 9 error percentages.

It is obvious that the basic SAT formulation $\mathcal{F}(\mathcal{Q})$ without any additional clauses to prune the search space is the worst for every solver and all gMaxSAT approaches are far better than the gMCS approaches. Even with using the basic encoding $\mathcal{F}(\mathcal{Q})$ all instances could be solved by every MaxSAT solver. However, it is nearly two orders of magnitude slower than the best encoding.

The redundant clauses which were added to $\mathcal{F}_v(\mathcal{Q})$ improve the performance of all solvers significantly by helping to detect conflicts earlier in the search space. The positive effect of the redundant clauses that were added to $\mathcal{F}^s(\mathcal{Q})$ is much less.

The best overall solver is `qmaxsat` which needs approximately only one third of the runtime of the other two MaxSAT solvers for error percentages of more than 20%.

5.6.2 Comparison to PBO and ASP

Table 5.3 shows the average runtime of state-of-the-art approaches and the best formulation of the gMCS and gMaxSAT approaches for the set of artificial instances on 10 taxa. The behavior of the state-of-the-art approaches is similar to the published results [104]. The PBO approaches beat the ASP solvers for an error percentage of 30%. However, PBO does not outperform ASP for 20% and 25% as previously stated.

		0%	1%	2%	5%	10%	15%	20%	25%	30%
CAMUS	$\mathcal{F}(\mathcal{Q})$	1.505	2.422	5.330	47.644	341.771	1084.689	>1800	>1800	>1800
	$\mathcal{F}_v(\mathcal{Q})$	**1.011**	**1.119**	**1.297**	**1.398**	**2.424**	**6.309**	**38.265**	**210.870**	**(2) 1290.957**
	$\mathcal{F}^s(\mathcal{Q})$	2.088	2.577	3.792	(1) 7.276	(6) 303.893	(9) 1625.087	>1800	>1800	>1800
	$\mathcal{F}^s_v(\mathcal{Q})$	1.947	2.390	2.896	14.679	>1800	>1800	>1800	>1800	>1800
picomcs	$\mathcal{F}(\mathcal{Q})$	0.768	18.605	9.657	74.133	940.392	(9) 1594.836	>1800	>1800	>1800
	$\mathcal{F}_v(\mathcal{Q})$	**0.609**	**1.563**	4.628	12.829	39.021	159.880	352.360	(1) 882.679	>1800
	$\mathcal{F}^s(\mathcal{Q})$	1.185	3.668	4.518	15.330	280.552	(3) 1044.056	>1800	>1800	>1800
	$\mathcal{F}^s_v(\mathcal{Q})$	0.929	2.651	**2.419**	**4.397**	**31.079**	**54.940**	**223.765**	**(1) 792.249**	**(8) 829.974**
eva500a	$\mathcal{F}(\mathcal{Q})$	0.656	3.916	6.086	16.516	39.832	81.496	145.284	304.831	922.730
	$\mathcal{F}_v(\mathcal{Q})$	**0.586**	**0.610**	**0.647**	**0.897**	**1.453**	**1.989**	**3.726**	**7.059**	**17.822**
	$\mathcal{F}^s(\mathcal{Q})$	1.022	1.395	1.740	3.682	8.631	16.482	36.781	78.830	220.369
	$\mathcal{F}^s_v(\mathcal{Q})$	1.170	1.417	1.609	2.169	3.460	4.749	7.166	10.375	20.114
mscg	$\mathcal{F}(\mathcal{Q})$	0.565	1.258	2.396	14.832	62.439	157.526	376.058	687.401	1178.281
	$\mathcal{F}_v(\mathcal{Q})$	**0.500**	**0.538**	**0.574**	**0.646**	**0.979**	**1.610**	**3.625**	**7.078**	**16.373**
	$\mathcal{F}^s(\mathcal{Q})$	1.196	1.210	1.232	1.718	7.221	15.408	49.971	130.635	321.007
	$\mathcal{F}^s_v(\mathcal{Q})$	1.242	1.249	1.391	1.549	2.235	3.789	7.189	12.071	24.427
qmaxsat	$\mathcal{F}(\mathcal{Q})$	4.981	5.647	4.114	13.477	27.836	59.476	141.031	186.902	381.100
	$\mathcal{F}_v(\mathcal{Q})$	**0.531**	**0.576**	**0.576**	**0.644**	**0.739**	**0.912**	**1.267**	**2.608**	**5.151**
	$\mathcal{F}^s(\mathcal{Q})$	2.119	2.139	1.933	3.142	4.759	8.316	21.407	53.330	129.453
	$\mathcal{F}^s_v(\mathcal{Q})$	1.349	1.504	1.374	1.645	1.960	2.429	2.805	5.646	8.773

Table 5.2. Average runtime for all SAT-based approaches on all possible SAT formulations in seconds for the artificial instances with 10 taxa. Figures in parentheses denote the number (out of 10) of instances which ran into the timeout of 1800 seconds. Bolt values mark the best approach for this particular combination of error percentage and solver.

The performance of both ASP solvers is very similar for error percentages of 15% and less, but it is remarkable that the `smodels` solver outperforms `clasp` for larger error percentages considerably. The higher the error rate, the larger the difference of the performance of both solvers, culminating in a difference of more than one order of magnitude. In contrast to that, both PBO encodings show no significant difference from each other.

The gMCS approach with `CAMUS` as a solver does provide a competitive approach for error percentages of 20% and 25% in comparison to `clasp`, and for error percentages between 5% and 15% in comparison to the PBO encodings. The extended `picoMCS` solver is considerably slower than `CAMUS` by at least one order of magnitude for error percentages of 5% and more.

It is remarkable that all solvers used in the SAT+gMaxSAT approach were able to beat the PBO and ASP approaches for error percentages of 15% and more. For error percentages of 20% and more the difference between the best MaxSAT solver `qMaxSAT` and the best ASP and PBO approaches is approximately one order of magnitude.

For the benchmark set covering 15 taxa the results are even more convincing (see Table 5.4). The gMaxSAT approach was the best approach for any error percentage. For error percentages of less than 2% the solver `mscg` is the best. For higher percentages the solver `qMaxSAT` performed better than all others. However, not a single instance for an error percentage of 20% or more could be solved by any approach.

Another significant difference can be observed when analyzing the single approaches in more detail. In contrast to the instances with 10 taxa the behavior of the two ASP solvers, as well as the performance of both PBO encodings for instances with 15 taxa show a significant difference. The ASP solver `clasp` is able to outperform `smodels` considerably for all error percentages of 1% and more for the instances with more taxa. A similar behavior can be seen by the PBO encodings as the `pbo-bin-sc` is able to solve three more instances for an error percentage of 2%.

A strange behavior is shown by the gMCS approach. The extended `picoMCS` solver is significantly better than `CAMUS` for an error percentage of 0% and 5%, but is much worse for the error percentages of 1% and 2%. The results for the MaxSAT solvers `eva500a` and `mscg` show a similar behavior. Whereas `mscg` clearly outperforms `eva500a` for error percentages up to 10%, the results flip for 15%. In that case `eva500a` is able to solve 5 more instances than `mscg`.

5.6.3 Comparison to available MaxSAT approaches

Table 5.6 shows the runtime for the available MaxSAT instances from the benchmark set of the MaxSAT-evaluation[2] for all three used solvers and the runtime of the best SAT formulation based on the split encoding which was introduced in this chapter. The MaxSAT instances from the benchmark set of the MaxSAT-evaluation were created by translating the pseudo boolean optimization problems into SAT [104].

[2] `http://maxsat.ia.udl.cat/introduction/`

	0%	1%	2%	5%	10%	15%	20%	25%	30%
phy+smodels	**0.220**	**0.266**	**0.290**	0.548	1.725	2.951	10.490	34.675	73.107
phy+clasp	0.408	0.437	0.438	**0.491**	1.169	2.652	52.534	311.707	(1) 1136.024
pbo-bin+minisat+	0.917	0.998	0.917	2.460	7.809	13.407	19.629	38.329	60.086
pbo-bin-sc+minisat+	0.821	0.919	0.946	2.328	7.134	12.465	20.754	35.265	58.549
sat+CAMUS ($\mathcal{F}_v(\mathcal{Q})$)	1.011	1.119	1.297	1.398	2.424	6.309	38.265	210.870	(2) 1290.957
sat+picoMCS ($\mathcal{F}_v^s(\mathcal{Q})$)	0.929	2.651	2.419	4.397	31.079	54.940	223.765	(1) 792.249	(8) 829.974
sat+eva500a ($\mathcal{F}_v(\mathcal{Q})$)	0.586	0.610	0.647	0.897	1.453	1.989	3.726	7.059	17.822
sat+mscg ($\mathcal{F}_v(\mathcal{Q})$)	0.500	0.538	0.574	0.646	0.979	1.610	3.625	7.078	16.373
sat+qMaxSAT ($\mathcal{F}_v(\mathcal{Q})$)	0.531	0.576	0.576	0.644	**0.739**	**0.912**	**1.267**	**2.608**	**5.151**

Table 5.3. Average runtime in seconds for the artificial instances with 10 taxa. Figures in parentheses denote the number (out of 10) of instances which ran into the timeout of 1800 seconds. Bolt values mark the best approach for this particular error percentage.

	0%	1%	2%	5%	10%	15%	20%
phy+smodels	2.578	14.691	77.563	(3) 726.182	(3) 950.303	(6) 1027.792	>1800
phy+clasp	3.682	7.995	31.909	376.980	(2) 522.090	(6) 865.765	>1800
pbo-bin+minisat+	59.650	43.931	(4) 938.297	>1800	>1800	>1800	>1800
pbo-bin-sc+minisat+	17.990	50.477	(1) 530.269	>1800	>1800	>1800	>1800
sat+CAMUS ($\mathcal{F}_v(\mathcal{Q})$)	7.055	18.168	34.707	>1800	>1800	>1800	>1800
sat+picoMCS($\mathcal{F}_v^s(\mathcal{Q})$)	19.685	(1) 888.566	(2) 988.009	>1800	>1800	>1800	>1800
sat+eva500a ($\mathcal{F}_v(\mathcal{Q})$)	2.598	16.766	44.195	124.368	500.425	(5) 1353.779	>1800
sat+mscg ($\mathcal{F}_v(\mathcal{Q})$)	**2.561**	**3.222**	**4.844**	33.981	340.113	>1800	>1800
sat+qMaxSAT ($\mathcal{F}_v(\mathcal{Q})$)	4.276	5.278	6.935	**23.276**	**324.628**	**(1) 966.242**	>1800

Table 5.4. Average runtime in seconds for the artificial instances with 15 taxa. Figures in parentheses denote the number (out of 10) of instances which ran into the timeout of 1800 seconds. Bolt values mark the best approach for this particular error percentage.

For the `eva500a` solver the results are very balanced. Whereas the novel *split encoding* leads to a smaller runtime for 18 out of 28 instances covering 10 taxa, the pbo-maxsat translation had a smaller runtime for all 10 instances covering 15 taxa.

For the other two MaxSAT solvers `mscg` and `qMaxSAT` the split encoding offers a better runtime in all but 2 instances (for `qMaxSAT`.) The difference of the runtime between the two approaches is often more than one order of magnitude. However, the gap between the PBO translation and the split encoding is significantly less for 15 taxa, but that is to be expected, since the split encoding is particular effective for higher error percentages (see Section 5.6.2) and the instances available for 15 taxa have a very low error percentage of 1%.

Table 5.5. The instances of inaccurate optimal values of the linear pbo-maxsat encoding.

instance	opt	res
n10-tree1-10p	21	20
n10-tree1-20p	42	41
n10-tree3-05p	10	9
n10-tree3-15p	31	30
n10-tree3-25p	52	48
n10-tree4-05p	10	9
n10-tree4-20p	42	41
n10-tree4-30p	63	62
n10-tree5-15p	31	29

Please note that we detected a possible error in the linear encoding ("n") for the pbo-maxsat approach. For 9 different instances (see Table 5.5) the reported optimal value ("res") differs from the optimal value ("opt") that were reported by all other approaches. We mark the runtime that resulted in a report of inaccurate values in Table 5.6 with a gray color.

5.7 Summary

Encoding problems from Bioinformatics as satisfiability problems was successfully established by Lynce and Marques-Silva with their work on *haplotype inference* [94] and by Bonet and St. John [25].

In this chapter we present a novel approach to solve *Maximum Quartet Consistency* by encoding *Quartet Compatibility* as a satisfiability problem. The Quartet Compatibility problem describes, whether it is possible to construct a big evolutionary tree (phylogeny) that maintains the evolutionary relations which were given in smaller input trees. Since many of these problems are unsatisfiable the *Maximum Quartet Consistency* is more interesting in practice. It searches for a phylogeny that maintains the maximum number of evolutionary relations from the input data.

We describe the possibilities of using group *Minimal Correction Set* (MCS) and group *MaxSAT* techniques to tackle this problem. Due to the fact that finding a single MCS or a MaxSAT solution itself requires several calls of a SAT solver on one particular instance the SAT encoding has to be especially efficient to be able to cope with the state-of-the-art approaches *answer set programming* (ASP) or *pseudo boolean optimization* (PBO).

By adding redundant information to the basic SAT formulation we were able to boost the performance of state-of-the-art MaxSAT solvers on our formulation, such that the MaxSAT approach was considerably faster than any of the ASP and PBO approaches for complete input sets for 10 taxa with 10% error percentage and more, and for complete input sets for 15 taxa with any error percentage.

Moreover, we compared the existing MaxSAT instances that were created by

	eva500a				mscg				qmaxsat			
instance	pbo-maxsat		split		pbo-maxsat		split		pbo-maxsat		split	
n10-tree1-10p	1.477	n	**1.430**	$\mathcal{F}_v(\mathcal{Q})$	15.481	n	**0.589**	$\mathcal{F}_v(\mathcal{Q})$	13.314	n	**0.612**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree1-15p	**1.608**	nlog	1.768	$\mathcal{F}_v(\mathcal{Q})$	20.482	nlog	**1.013**	$\mathcal{F}_v(\mathcal{Q})$	18.744	nlog	**0.414**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree1-20p	12.163	n	**2.737**	$\mathcal{F}_v(\mathcal{Q})$	91.953	n	**2.967**	$\mathcal{F}_v(\mathcal{Q})$	53.779	n	**2.492**	$\mathcal{F}_v^s(\mathcal{Q})$
n10-tree1-25p	**4.766**	nlog	6.664	$\mathcal{F}_v(\mathcal{Q})$	68.977	nlog	**5.404**	$\mathcal{F}_v(\mathcal{Q})$	48.175	nlog	**2.365**	$\mathcal{F}_v^s(\mathcal{Q})$
n10-tree1-30p	55.359	n	**15.709**	$\mathcal{F}_v(\mathcal{Q})$	297.881	n	**17.870**	$\mathcal{F}_v^s(\mathcal{Q})$	212.689	n	**8.978**	$\mathcal{F}_v^s(\mathcal{Q})$
n10-tree2-10p	**0.821**	nlog	0.918	$\mathcal{F}_v(\mathcal{Q})$	6.434	nlog	**0.351**	$\mathcal{F}_v(\mathcal{Q})$	4.306	nlog	**0.138**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree2-15p	6.992	n	**1.494**	$\mathcal{F}_v(\mathcal{Q})$	38.402	n	**2.302**	$\mathcal{F}_v(\mathcal{Q})$	28.305	n	**0.618**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree2-20p	**3.170**	nlog	5.504	$\mathcal{F}_v(\mathcal{Q})$	52.948	nlog	**3.450**	$\mathcal{F}_v(\mathcal{Q})$	28.808	nlog	**0.775**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree2-25p	24.333	n	**9.498**	$\mathcal{F}_v(\mathcal{Q})$	63.281	n	**12.878**	$\mathcal{F}_v^s(\mathcal{Q})$	58.744	n	**2.823**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree2-30p	**14.586**	nlog	22.678	$\mathcal{F}_v(\mathcal{Q})$	124.666	nlog	**18.452**	$\mathcal{F}_v(\mathcal{Q})$	56.780	nlog	**5.939**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree3-05p	1.612	n	**0.674**	$\mathcal{F}_v(\mathcal{Q})$	1.935	n	**0.135**	$\mathcal{F}_v^n(\mathcal{Q})$	2.176	n	**0.224**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree3-10p	**0.799**	nlog	1.021	$\mathcal{F}_v(\mathcal{Q})$	4.197	nlog	**0.305**	$\mathcal{F}_v(\mathcal{Q})$	2.074	nlog	**0.168**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree3-15p	4.943	n	**2.900**	$\mathcal{F}_v(\mathcal{Q})$	36.258	n	**1.919**	$\mathcal{F}_v(\mathcal{Q})$	25.202	n	**0.369**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree3-20p	**2.917**	nlog	3.821	$\mathcal{F}_v(\mathcal{Q})$	47.052	nlog	**3.573**	$\mathcal{F}_v(\mathcal{Q})$	19.599	nlog	**0.979**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree3-25p	37.362	n	**12.674**	$\mathcal{F}_v(\mathcal{Q})$	64.364	n	**11.878**	$\mathcal{F}_v^s(\mathcal{Q})$	80.452	n	**5.981**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree3-30p	**16.816**	nlog	20.920	$\mathcal{F}_v^s(\mathcal{Q})$	96.531	nlog	**19.970**	$\mathcal{F}_v(\mathcal{Q})$	122.641	nlog	**3.294**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree4-05p	**0.589**	nlog	0.743	$\mathcal{F}_v(\mathcal{Q})$	0.722	nlog	**0.152**	$\mathcal{F}_v(\mathcal{Q})$	0.408	nlog	**0.139**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree4-10p	4.113	n	**1.832**	$\mathcal{F}_v(\mathcal{Q})$	9.201	n	**0.582**	$\mathcal{F}_v(\mathcal{Q})$	12.697	n	**0.451**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree4-15p	**2.112**	nlog	2.355	$\mathcal{F}_v(\mathcal{Q})$	14.285	nlog	**0.815**	$\mathcal{F}_v(\mathcal{Q})$	6.930	nlog	**0.810**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree4-20p	21.583	n	**5.359**	$\mathcal{F}_v(\mathcal{Q})$	76.197	n	**4.333**	$\mathcal{F}_v^s(\mathcal{Q})$	62.608	n	**0.804**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree4-25p	6.347	nlog	**5.814**	$\mathcal{F}_v(\mathcal{Q})$	65.546	nlog	**6.603**	$\mathcal{F}_v(\mathcal{Q})$	27.371	nlog	**1.186**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree4-30p	76.561	n	**19.638**	$\mathcal{F}_v(\mathcal{Q})$	322.684	n	**29.449**	$\mathcal{F}_v^s(\mathcal{Q})$	152.715	n	**7.310**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree5-05p	0.641	nlog	**0.430**	$\mathcal{F}_v(\mathcal{Q})$	0.690	nlog	**0.142**	$\mathcal{F}_v(\mathcal{Q})$	0.680	nlog	**0.301**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree5-10p	2.128	nlog	**0.667**	$\mathcal{F}_v(\mathcal{Q})$	4.622	nlog	**0.255**	$\mathcal{F}_v(\mathcal{Q})$	1.882	nlog	**0.194**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree5-15p	9.336	n	**2.465**	$\mathcal{F}_v(\mathcal{Q})$	27.958	n	**1.633**	$\mathcal{F}_v(\mathcal{Q})$	29.135	n	**0.548**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree5-20p	5.532	nlog	**2.658**	$\mathcal{F}_v(\mathcal{Q})$	16.432	nlog	**2.049**	$\mathcal{F}_v(\mathcal{Q})$	16.300	nlog	**0.756**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree5-25p	43.777	n	**12.193**	$\mathcal{F}_v(\mathcal{Q})$	192.391	n	**9.064**	$\mathcal{F}_v(\mathcal{Q})$	100.273	n	**4.343**	$\mathcal{F}_v(\mathcal{Q})$
n10-tree5-30p	25.751	nlog	**18.384**	$\mathcal{F}_v^s(\mathcal{Q})$	102.494	nlog	**12.798**	$\mathcal{F}_v(\mathcal{Q})$	59.594	nlog	**3.720**	$\mathcal{F}_v(\mathcal{Q})$
n15-tree1-01p	**5.442**	nlog	29.776	$\mathcal{F}_v(\mathcal{Q})$	16.687	nlog	**1.574**	$\mathcal{F}_v(\mathcal{Q})$	7.393	nlog	**1.816**	$\mathcal{F}_v(\mathcal{Q})$
n15-tree2-01p	**13.948**	nlog	27.995	$\mathcal{F}_v(\mathcal{Q})$	23.339	nlog	**1.565**	$\mathcal{F}_v(\mathcal{Q})$	14.627	nlog	**4.462**	$\mathcal{F}_v(\mathcal{Q})$
n15-tree3-01p	**5.837**	nlog	23.011	$\mathcal{F}_v(\mathcal{Q})$	44.685	nlog	**1.690**	$\mathcal{F}_v(\mathcal{Q})$	17.839	nlog	**2.752**	$\mathcal{F}_v(\mathcal{Q})$
n15-tree4-01p	**6.285**	nlog	10.701	$\mathcal{F}_v(\mathcal{Q})$	32.242	nlog	**1.653**	$\mathcal{F}_v(\mathcal{Q})$	15.629	nlog	**10.912**	$\mathcal{F}_v(\mathcal{Q})$
n15-tree5-01p	**6.509**	nlog	10.229	$\mathcal{F}_v(\mathcal{Q})$	17.185	nlog	**1.287**	$\mathcal{F}_v(\mathcal{Q})$	**7.368**	nlog	8.668	$\mathcal{F}_v(\mathcal{Q})$
n15-tree6-01p	**6.559**	nlog	23.373	$\mathcal{F}_v(\mathcal{Q})$	22.638	nlog	**2.230**	$\mathcal{F}_v(\mathcal{Q})$	11.884	nlog	**8.999**	$\mathcal{F}_v(\mathcal{Q})$
n15-tree7-01p	**5.251**	nlog	23.285	$\mathcal{F}_v(\mathcal{Q})$	20.844	nlog	**1.988**	$\mathcal{F}_v(\mathcal{Q})$	13.437	nlog	**6.174**	$\mathcal{F}_v(\mathcal{Q})$
n15-tree8-01p	**4.250**	nlog	18.245	$\mathcal{F}_v(\mathcal{Q})$	7.291	nlog	**1.608**	$\mathcal{F}_v(\mathcal{Q})$	10.517	nlog	**4.103**	$\mathcal{F}_v(\mathcal{Q})$
n15-tree9-01p	**3.732**	nlog	18.330	$\mathcal{F}_v(\mathcal{Q})$	9.733	nlog	**2.170**	$\mathcal{F}_v(\mathcal{Q})$	13.181	nlog	**5.778**	$\mathcal{F}_v(\mathcal{Q})$
n15-tree10-01p	**2.983**	nlog	13.432	$\mathcal{F}_v(\mathcal{Q})$	2.188	nlog	**1.413**	$\mathcal{F}_v(\mathcal{Q})$	**5.143**	nlog	6.658	$\mathcal{F}_v(\mathcal{Q})$

Table 5.6. Comparison of MaxSAT-approaches for each of the three used solvers. We report the best runtime together with the used encoding for every solver. Bolt values denote the best value for a particular solver. Underlined values denote the best approach for that particular instance for all solvers.

translating the PBO encodings into SAT encodings with the novel SAT formulation. The results have shown that the fastest solution was always obtained by using our SAT formulation for every problem.

However, all results show that the available approaches lose their usability when the number of taxa (in combination with the error percentage) rises. The possibilties to overcome this problem will be discussed in Chapter 7, where the work of this thesis is concluded.

6 Using SAT to embed graphs in books

6.1 Introduction

Embedding graphs in books is a fundamental issue in graph theory that has received considerable attention (see, e.g., [21, 114] for an overview). In a book embedding [113] the vertices of a graph are restricted to a line, referred to as the *spine* of the book, and the edges are drawn at different half-planes delimited by the spine, called *pages* of the book. The task is to find a so-called *linear order* of the vertices along the spine and an assignment of the edges of the graph to the pages of the book, so that no two edges of the same page cross (see Figure 6.1b). The *book thickness* or *page number* of a graph is the smallest number of pages that are required by any book embedding of the graph.

Problems on book embeddings are mainly classified into two categories based on whether the graph to be embedded is planar or not. For non-planar graphs it is known that there exist graphs on n vertices that have book thickness $\Theta(n)$, e.g., the book thickness of the complete graph K_n is $\lceil n/2 \rceil$ [15]. Sublinear book thickness have, e.g., graphs with subquadratic number of edges [98], subquadratic genus [97] or sublinear treewidth [46]. Constant book thickness have, e.g., all minor-closed graphs [23] or the k-trees for fixed k [56]. Another class of non-planar graphs that was recently proved to have constant book thickness is the class of 1-planar graphs [6]. Note that 1-planar graphs are not necessarily closed under minors [109].

For planar graphs a remarkable result is due to Yannakakis who back in 1986 proved that any planar graph can be embedded in a book with four pages [152]. However, more restricted subclasses of planar graphs allow embeddings in books with fewer pages. Bernhart and Kainen [15] showed that the graphs which can be embedded in single-page books are the outerplanar graphs while the graphs which can be embedded in books with two pages are the subhamiltonian ones.

It is known that not all planar graphs are subhamiltonian and the corresponding decision problem whether a maximal planar graph is Hamiltonian (and therefore two-page book embeddable) is NP-complete [147]. However, several subclasses

of planar graphs are known to be either Hamiltonian or subhamiltonian, e.g., 4-connected planar graphs [110], planar graphs without separating triangles [81], Halin graphs [34], planar 2-trees [30], or planar graphs of maximum degree 3 or 4 [71, 7].

A well-known non-Hamiltonian graph is the Goldner-Harary one [62]. This particular graph, however is a planar 3-tree and hence 3-page book embeddable [70]. To the best of our knowledge there is no planar graph given in the literature whose page number is four. In other words it is not yet known whether the upper bound of four pages of Yannakakis [152] is tight for planar graphs or not.

Our contribution

We suggest an alternative approach to the problem of determining whether a graph can be embedded in a book of a certain number of pages without imposing any further restrictions on the graph's structure like planarity or simplicity. We propose a formulation of the problem as a SAT formula which can be useful in practice. Apart from their independent theoretical interest book embeddings find applications in several contexts, such as VLSI design, fault-tolerant processing, sorting networks and parallel matrix multiplication, see e.g., [30, 72, 120, 137]. It turns out that our formulation is of a simple nature, quite intuitive and easy-to-implement, but simultaneously robust enough to solve non-trivial instances of the problem in reasonable amount of time (e.g., within 20 minutes we can test whether a maximal planar graphs with up to 400 vertices is 3-page book embeddable), as we will see in our experimental study.

Note that SAT formulations are not so common in graph drawing. A few notable exceptions are [18, 28, 55]. In our context of particular interest is the work of Biedl et al. [18] who proposed ILP and SAT formulations for several grid-based graph problems (including pathwidth, bandwidth, optimum st-orientation and visibility representation). Of course, their general formulation can be easily extended to solve our problem as well. However, from the authors' experimental evaluation (and we could also confirm it) it follows that their approach is limited to solve relatively small instances within reasonable time, e.g., within 20 minutes one can cope with graphs whose size in vertices and edges does not exceed 100.

A list of hypotheses

When we started working on this project we placed several hypotheses that we wanted to prove or disprove. So, before we proceed with the description of our formulation, we first list and then discuss the most important ones:

H1: There is a (maximal) planar graph whose book thickness is four.

H2: There is a 1-planar graph whose book thickness is (at least) four.

H3: There is a (maximal) planar graph which cannot be embedded in a book of three pages, if the subgraphs embedded at each page must be acyclic.

H4: There is a maximal planar graph, say G_a, which in any of its book embeddings on three pages has at least one face whose edges are on the same page.

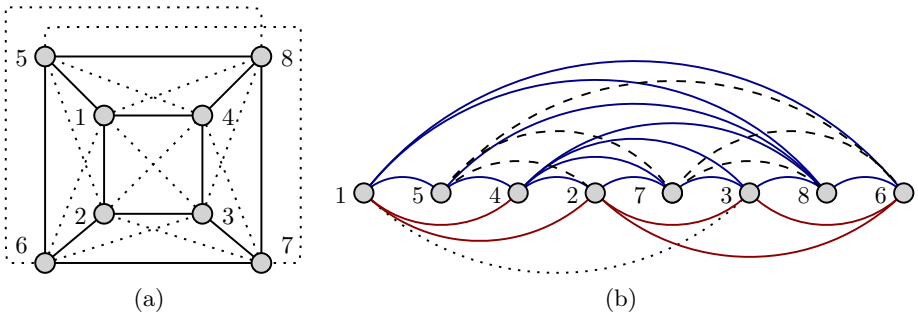


Figure 6.1. (a) An optimal 1-planar graph whose underlying planar structure (solid drawn) is the cube graph. (b) A corresponding embedding in a book with 4 pages. Observe that the fourth page contains just a single edge (dotted drawn).

H5: There is a maximal planar graph, say G_c, which in any of its book embeddings on three pages has a face, say f_c^*, whose edges cannot be on the same page.

Summary and discussion

Clearly, our ultimate goal was to find a planar graph supporting Hypothesis 1. During our extensive practical analysis we tested several hundreds maximal planar graphs (both randomly created and crafted), but we did not manage to find one supporting Hypothesis 1. We also tested a specific graph with roughly 600 vertices out of the family of planar graphs that Yannakakis proposed to require page number four, but it turned out to be 3-page book embeddable for this particular size.

On the positive side, we proved that the weakest version of Hypothesis 2 holds. In particular, we managed to find a relatively small 1-planar graph whose book thickness is exactly four (see Figure 6.1). To the best of our knowledge this is the first (non-trivial) lower bound on the book thickness of 1-planar graphs. Recall that a 1-planar graph on n vertices is said to be *optimal*, if it has exactly $4n - 8$ edges which is the maximum possible [24]. By a work of Suzuki [136], who described how one can generate all optimal 1-planar graphs, it follows that the graph of Figure 6.1 is the smallest optimal 1-planar graph.

We were surprised that we did not succeed in proving that Hypothesis 3 holds. Note that it is very natural to try to embed a tree-structured subgraph at each of the available pages of the book, if one seeks to prove that indeed all planar graphs can be embedded in books of three pages. For example, Heath [70] who constructively proved that all planar 3-trees fit into books with three pages used exactly this approach: the subgraphs embedded at each of the three pages of the book are acyclic, one of them is a tree and the other two are forests, each consisting of two trees.

Note that we managed to prove a weaker version of Hypothesis 3 according to which the input maximal planar graph has n vertices and cannot be embedded in a book with three pages so that: (i) the subgraph assigned to each of the three pages is a tree on $n - 1$ vertices and, additionally, (ii) the three vertices that are not spanned by the three trees are pairwise adjacent forming a face f_o of the graph,

say w.l.o.g. its outerface. The graph supporting this weaker hypothesis is given in Figure 6.2 and shows that it is not always possible to construct a 3-page book embedding based on a Schnyder decomposition into three trees regardless of the linear order underneath. We refer the reader to Section 6.2.1 for further information regarding the SAT formulation we used to identify the graph. Note that we tried to modify this special graph in various ways, but we could not confirm Hypothesis 3 in full generality.

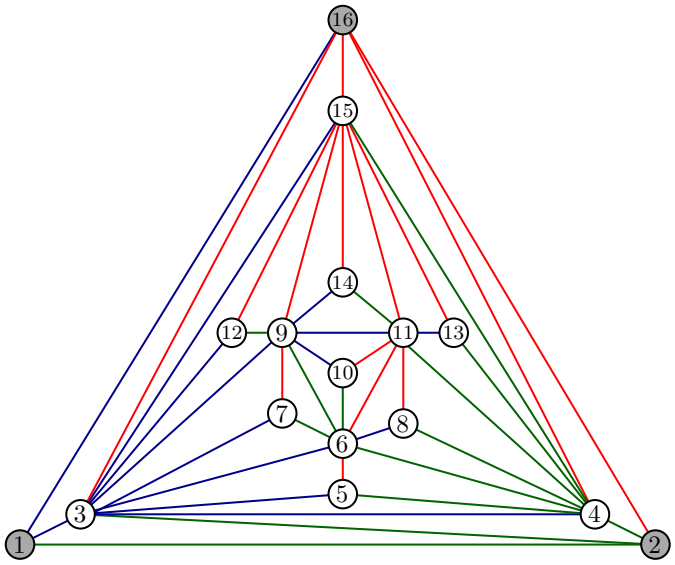


Figure 6.2. A maximal planar graph on 16 vertices supporting a weaker version of Hypothesis 3.

From our experimental evaluation (see Section 6.3) we quickly observed that the practical limitation of testing the book-embedability of maximal planar graphs on 3 pages with our SAT formulation (that we will shortly present in Section 6.2) lies at around 600 to 700 vertices. Larger graphs lead to instance sizes which excess several gigabytes of random access memory. Hypotheses 4 and 5 in conjunction describe an approach that could potentially overcome this bottleneck. To see this assume that the two planar graphs, denoted by G_a and G_c in Hypotheses 4 and 5 respectively, exist (note however, that we did not succeed in finding them). If for each face f_a of G_a we create a copy of graph G_c and identify each of the vertices of face f_c^* of G_c with one of the vertices of face f_a of G_a, then we will obtain a (drastically larger) planar graph which is not 3-page book-embeddable. This is because G_a must contain at least one face whose edges are on the same page, while in the same time face f_c^* would require at least one of them not to be at the same page.

In the next section we introduce the details of our SAT formulation for the Book Embedding problem, and show how this basic formulation can be extended straightforwardly to investigate the mentioned hypotheses. In Section 6.3 the experimental setting is described. Due to the fact that the Book Embedding problem was not

tackled by SAT before, we are required to create benchmark sets that model the possible inputs properly. We show how to generate different benchmark sets based on the Hypotheses 1 to 5 we want to prove or disprove for particular graph classes.

6.2 SAT Formulation

Let $G = (V, E)$, with $V = \{v_1, v_2, \ldots, v_n\}$ and $E = \{e_1, e_2, \ldots, e_m\}$, be a graph for which we seek to decide whether it can be embedded in a book with $p \geq 2$ pages. In the following we describe a logic formula $\mathcal{F}(G, p)$ that will solve this problem by encoding it as a SAT instance. Recall that any SAT problem can be described in *conjunctive normal form* (CNF) which is a conjunction of clauses; each clause being a disjunction of (possibly negated) literals. We will define $\mathcal{F}(G, p)$ by its set of variables and a corresponding set of rules. The rules will ensure the proper assignment of the variables and will be given in propositional logic which can be converted into CNF clauses straightforwardly [115].

The variables of $\mathcal{F}(G, p)$ should model a book embedding of G in a book with p pages, if it exists. Thus, we use variables $\sigma(v_i, v_j)$ for each pair of vertices $v_i, v_j \in V$ to determine whether vertex v_i precedes vertex v_j along the spine, meaning that vertex v_i is to the left of vertex v_j. If that is the case the assignment is $\sigma(v_i, v_j) = \texttt{true}$. In this so-called *relative encoding* the variables encode a relative order between the vertices. Clearly, asymmetry has to hold for these variables which is ensured by the following rule:

$$\sigma(v_i, v_j) \leftrightarrow \neg\sigma(v_j, v_i)$$

With this *asymmetry rule* variables $\sigma(v_i, v_j)$ can be defined only for $i < j$. The other literals $\sigma(v_i, v_j)$ with $i > j$ can be replaced by $\neg\sigma(v_j, v_i)$. This results in a significantly smaller formula which is easier to be solved by a SAT solver [144]. The following *transitivity rule* for the σ-relation ensures a proper order of the vertices along the spine:

$$\sigma(v_i, v_j) \wedge \sigma(v_j, v_k) \rightarrow \sigma(v_i, v_k) \qquad \forall \text{ pairwise distinct } v_i, v_j, v_k \in V$$

The search space of possible satisfying assignments can be reduced by choosing a particular vertex as the first vertex along the spine and even further by choosing exactly one other pairwise order, e.g., that v_2 is to the left of v_3. These choices can be easily encoded by the *direction rules*:

$$\begin{aligned} &\sigma(v_1, v_i) \quad \forall v_i \in V \text{ with } i > 1 \\ &\sigma(v_2, v_3) \end{aligned}$$

Note that the direction rules are represented by unit clauses that fix $\sigma(v_1, v_i)$ for $i > 1$ and $\sigma(v_2, v_3)$ to be $\texttt{true}$. Via *unit propagation* (that is a basic operation performed by all SAT solvers [47]) other constraints may become simpler or even already satisfied.

So far we have managed to encode the linear order of the vertices along the spine. To encode the assignment of the edges to the pages of the book we introduce

a variable $\phi_q(e_i)$ for every edge $e_i \in E$ and every possible page $1 \leq q \leq p$. Thereby $\phi_q(e_i) = \texttt{true}$ means that the edge e_i is assigned to the q-th page of the book. Of course, every edge has to be assigned to (at least) one page which is ensured by the *page assignment rule*:

$$\phi_1(e_i) \vee \phi_2(e_i) \vee \ldots \vee \phi_p(e_i) \qquad \forall e_i \in E$$

We can again reduce the search space by the *fixed page assignment rule* that fixes a single edge on a particular page, e.g., edge $e_1 \in E$ to page 1:

$$\phi_1(e_1)$$

Next we describe how to forbid crossings among edges of the same page. We first introduce a variable $\chi(e_i, e_j)$ for each pair of edges $e_i, e_j \in E$ which describes whether e_i and e_j are assigned to the same page. $e_i, e_j \in E$ are assigned to the same page, if they are both assigned to one of the available pages which is ensured by the *same page rule*:

$$((\phi_1(e_i) \wedge \phi_1(e_j)) \vee \ldots \vee (\phi_k(e_i) \wedge \phi_k(e_j))) \rightarrow \chi(e_i, e_j) \qquad \forall e_i, e_j \in E$$

To ensure planarity it is enough to ensure that two edges which are assigned to the same page do not cross. So, if $(v_i, v_j), (v_k, v_\ell) \in E$ are two edges of G, such that vertices v_i, v_j, v_k and v_ℓ are pairwise different. This can be ensured by the following *planarity rule*:

$$\begin{array}{ll}
\chi((v_i, v_j), (v_k, v_\ell)) \rightarrow & \\
\quad \neg(\sigma(v_i, v_k) \wedge \sigma(v_k, v_j) \wedge \sigma(v_j, v_\ell)) & \wedge\neg(\sigma(v_i, v_\ell) \wedge \sigma(v_\ell, v_j) \wedge \sigma(v_j, v_k)) \\
\wedge\neg(\sigma(v_j, v_k) \wedge \sigma(v_k, v_i) \wedge \sigma(v_i, v_\ell)) & \wedge\neg(\sigma(v_j, v_\ell) \wedge \sigma(v_\ell, v_i) \wedge \sigma(v_i, v_k)) \\
\wedge\neg(\sigma(v_k, v_i) \wedge \sigma(v_i, v_\ell) \wedge \sigma(v_\ell, v_j)) & \wedge\neg(\sigma(v_k, v_j) \wedge \sigma(v_j, v_\ell) \wedge \sigma(v_\ell, v_i)) \\
\wedge\neg(\sigma(v_\ell, v_i) \wedge \sigma(v_i, v_k) \wedge \sigma(v_k, v_j)) & \wedge\neg(\sigma(v_\ell, v_j) \wedge \sigma(v_j, v_k) \wedge \sigma(v_k, v_i))
\end{array}$$

Theorem 6.1. *Let $G = (V, E)$ be a graph and $p \in \mathbb{N}$. Then G admits a book embedding on p pages, if and only if, $\mathcal{F}(G, p)$ is satisfiable. In addition $\mathcal{F}(G, p)$ has $O(n^2 + m^2 + pm)$ variables and $O(n^3 + m^2)$ clauses.*

Proof. The number of σ-, ϕ- and χ-variables are $O(n^2)$, $O(m^2)$ and $O(pm)$ respectively, which implies that $\mathcal{F}(G, p)$ has $O(n^2 + m^2 + pm)$ variables. The number of clauses of $\mathcal{F}(G, p)$ is dominated by the number of transitivity, same-page and planarity rules, which yield in total $O(n^3 + m^2)$ clauses. So, to prove this theorem it remains to show that: (i) a book embedding on p pages yields a satisfying assignment of $\mathcal{F}(G, p)$ and (ii) a satisfying assignment of $\mathcal{F}(G, p)$ yields a book embedding on p pages.

***(i)* From an embedding to an assignment:** Assume that G has a book embedding $\mathcal{E}(G, p)$ on p pages. From $\mathcal{E}(G, p)$ we obtain an order of the vertices along the spine and an assignment of the edges to the pages. We define an assignment $(\hat{\sigma}, \hat{\phi}, \hat{\chi})$ to the σ-, ϕ- and χ-variables of $\mathcal{F}(G, p)$ consistent with the intended meaning of the variables:

- $\hat{\sigma}(v_i, v_j) = \texttt{true}$, if and only if v_i is before v_j along the spine

- $\hat{\phi}_q(e_i) = \texttt{true}$, if and only if e_i is assigned to the q-th page
- $\hat{\chi}(e_i, e_j) = \texttt{true}$, if and only if e_i and e_j are assigned to the same page

To prove that assignment $(\hat{\sigma}, \hat{\phi}, \hat{\chi})$ satisfies $\mathcal{F}(G, p)$ we consider all rules of $\mathcal{F}(G, p)$:

- The *transitivity* and *asymmetry rules* are satisfied by $(\hat{\sigma}, \hat{\phi}, \hat{\chi})$, since $\hat{\sigma}$ is a complete order over the vertices of G (by definition of the assignment).
- The *direction rules* are also satisfied, since we can assume w.l.o.g. that in $\mathcal{E}(G, p)$ vertex $v_1 \in V$ is the first vertex along the spine and v_2 is to the left of v_3. Note that if this is not the case then we can circularly-shift the vertices of G along the spine and potentially mirror $\mathcal{E}(G, p)$ and obtain an equivalent embedding which has the aforementioned properties; see e.g., [152].
- The *page assignment rule* is trivially satisfied by the definition of the assignment and the fact that $\mathcal{E}(G, p)$ was given.
- The *fixed page assignment rule* can be satisfied as well, since we can assume w.l.o.g. that the first page of $\mathcal{E}(G, p)$ is the page where edge $e_1 \in E$ is assigned to.
- The *same page rule* is trivially satisfied due to the definition of the assignment.
- It remains to show that all *planarity rules* are satisfied. For the sake of contradiction assume that the assignment $(\hat{\sigma}, \hat{\phi}, \hat{\chi})$ violates a planarity rule for some pair of edges (v_i, v_j) and (v_k, v_ℓ). We know that $\hat{\chi}((v_i, v_j), (v_k, v_\ell)) = \texttt{true}$ and w.l.o.g. assume further $\hat{\sigma}(v_i, v_k) = \hat{\sigma}(v_k, v_j) = \hat{\sigma}(v_j, v_\ell) = \texttt{true}$. Hence, in $\mathcal{E}(G, p)$ we have that v_k is between v_i and v_j, while v_ℓ is not between v_i and v_j. Thus, (v_i, v_j) and (v_k, v_ℓ) are on the same page and cross in $\mathcal{E}(G, p)$ which is a clear contradiction.

***(ii)* From an assignment to an embedding:** Let $(\hat{\sigma}, \hat{\phi}, \hat{\chi})$ be a satisfying assignment to $\mathcal{F}(G, p)$. Let $\xi : V \mapsto \{1, \ldots, n\}$ be a function which maps each vertex $v \in V$ to a position along the spine. Based on the assignment $(\hat{\sigma}, \hat{\phi}, \hat{\chi})$, ξ can be defined as follows:

$$\xi(v_i) = 1 + |\{v_j : \sigma(v_j, v_i) = \texttt{true}, 1 \leq j \leq n,\ j \neq i\}|$$

Since $(\hat{\sigma}, \hat{\phi}, \hat{\chi})$ satisfies the *asymmetry* and *transitivity rules*, it follows that all positions assigned to the vertices of G are pairwise different. Therefore, a proper global ordering is obtained. Since by the *page assignment rule*, every edge of G is assigned to at least one page we only have to show that each page is crossing-free. Assume for the sake of contradiction that (v_i, v_j) and (v_k, v_ℓ) are two edges of G that are assigned to the same page and cross. In this case one of the following relationships must hold:

R.1: $\min\{\xi(v_i), \xi(v_j)\} < \min\{\xi(v_k), \xi(v_\ell)\} < \max\{\xi(v_i), \xi(v_j)\} < \max\{\xi(v_k), \xi(v_\ell)\}$

R.2: $\min\{\xi(v_k), \xi(v_\ell)\} < \min\{\xi(v_i), \xi(v_j)\} < \max\{\xi(v_k), \xi(v_\ell)\} < \max\{\xi(v_i), \xi(v_j)\}$

However, neither R.1 nor R.2 is possible, since both configurations do not comply with the *planarity rule* of $(\hat{\sigma}, \hat{\phi}, \hat{\chi})$. Therefore, each page is crossing-free, as desired. □

So far, we have described a SAT formulation that tests, whether a given graph $G = (V, E)$ admits an embedding in a book with $p \geq 2$ pages. Of course, this formulation can be extended with additional variables and rules. In the following we will introduce three different extensions, which encode Hypotheses 3, 4 and 5. Finally we discuss in Section 6.2.4 a fourth extension which seeks for embeddings in a fixed number of pages with minimum number of edges on the last page. This variant is useful, e.g., in order to determine how many edges must be removed from a p-page book embeddable graph in order to become $(p-1)$-page book embeddable which is clearly an NP-hard problem.

6.2.1 A variant to check Hypothesis 3

In this subsection, we present a SAT formulation to check Hypothesis 3. Recall that we seek to check whether a maximal planar graph G can be embedded in $p = 3$ pages, so that the subgraph assigned to each of the three pages is an acyclic graph. In the following we will extend formula $\mathcal{F}(G, 3)$ with new variables and rules to encode the additional requirement. We denote by $\mathcal{F}_f(G, 3)$ the resulting formula.

Let $\mathcal{N}(v)$ be the set of vertices adjacent to $v \in V$. For every edge $(v_i, v_j) \in E$ the variable $\pi_q(v_i, v_j)$ describes whether vertex v_i is the *parent* of vertex v_j in the forest of page $q \in \{1, 2, 3\}$. Variable $\pi_q(v_j, v_i)$ is defined symmetrically. We ensure that exactly one of the two variables is `true` when the edge (v_i, v_j) is assigned to page q, and both of the variables are `false` when the edge (v_i, v_j) is not assigned to page q by the *parent rules*:

$$\begin{aligned} \phi_q((v_i, v_j)) &\rightarrow (\pi_q(v_i, v_j) \wedge \neg\pi_q(v_j, v_i)) \vee (\neg\pi_q(v_i, v_j) \wedge \pi_q(v_j, v_i)) \\ \neg\phi_q((v_i, v_j)) &\rightarrow (\neg\pi_q(v_i, v_j) \wedge \neg\pi_q(v_j, v_i)) \end{aligned}$$

We also have to ensure that every vertex $v_i \in V$ has at most one parent vertex in the forest of page q which can be done via the *single parent rule*:

$$(\neg\pi_q(v_k, v_i) \vee \neg\pi_q(v_\ell, v_i)),\ \forall v_k, v_\ell \in \mathcal{N}(v_i) : v_k \neq v_\ell$$

To ensure acyclicity we use variables $\alpha_q(v_i, v_j)$ that describe whether vertex v_i is an *ancestor* of v_j in the forest of page q. We know that whenever for an edge $(v_i, v_j) \in E$ vertex v_i is the parent of vertex v_j on page q that v_i is the ancestor for v_j on that page as well which results in the *parent ancestor rule*:

$$\pi_q(v_i, v_j) \rightarrow \alpha_q(v_i, v_j)$$

Clearly, *transitivity* as well as *antisymmetry* has to hold for the ancestor relationship:

$$\begin{aligned} (\alpha_q(v_i, v_j) \wedge \alpha_q(v_j, v_k)) \rightarrow &\quad \alpha_q(v_i, v_k) \quad \forall \text{ pairwise distinct } v_i, v_j, v_k \in V \\ \alpha_q(v_i, v_j) \rightarrow &\quad \neg\alpha_q(v_j, v_i) \quad \forall \text{ pairwise distinct } v_i, v_j \in V \end{aligned}$$

Theorem 6.2. *Let $G = (V, E)$ be a (maximal) planar graph. Then G admits a book embedding on three pages, so that the subgraph assigned to each of the three pages is a forest, if and only if $\mathcal{F}_f(G, 3)$ is satisfiable.*

Proof. We use the same technique as in the proof of Theorem 6.1. Consider an embedding $\mathcal{E}(G,3)$ in three pages yield by our formulation. We claim that the subgraphs embedded at each page are acyclic. For contradiction assume that there is a cycle $\mathcal{C}_q$ on page q. If we direct each edge of $\mathcal{C}_q$ from the child to the parent vertex then all edges of $\mathcal{C}_q$ must have the same orientation, that is either clockwise or counterclockwise along $\mathcal{C}_q$ (otherwise there is a vertex of $\mathcal{C}_q$ that has two parents, deviating the single parent rule). The transitivity of the ancestor relationship implies that the antisymmetry property is deviated along $\mathcal{C}_q$ which is a contradiction. Hence, the subgraphs embedded at each page of $\mathcal{E}(G,3)$ are indeed acyclic. Following similar arguments as in the second part of the proof of Theorem 6.1 we can easily prove that a satisfying assignment of $\mathcal{F}_f(G,3)$ yields a book embedding on 3 pages in which the subgraph assigned to each page is a forest which completes the proof. □

Note that $\mathcal{F}_f(G,3)$ has asymptotically the same number of variables and clauses as $\mathcal{F}(G,3)$. Also note that our formulation can be easily adjusted to check whether the subgraph assigned to each page is a tree (which will be denoted by $\mathcal{F}_t(G,3)$). In this scenario we employ an additional variable $\rho_q(v_i)$ for each vertex $v_i \in V$ that describes whether v_i is the *root* of the tree of page $q \in \{1,2,3\}$. Vertex v_i is the root of the tree of page q if and only if it has no parent and (at least) one child at page q, which can be ensured via the following *root rule*:

$$\Big(\bigwedge_{v_j \in \mathcal{N}(v_i)} \neg\pi_q(v_j,v_i)\Big) \wedge \Big(\bigvee_{v_k \in \mathcal{N}(v_i)} \pi_q(v_i,v_k)\Big) \leftrightarrow \rho_q(v_i)$$

We ensure that there are not two or more roots on the same page q via the *single root rule*:

$$(\neg\rho_q(v_i) \vee \neg\rho_q(v_j)),\ \forall v_i, v_j \in V; i \neq j$$

Theorem 6.3. *Let $G = (V,E)$ be a (maximal) planar graph. Then G admits a book embedding on three pages, so that the subgraph assigned to each of the three pages is a tree, if and only if $\mathcal{F}_t(G,3)$ is satisfiable.*

Proof. Consider an embedding $\mathcal{E}(G,3)$ in three pages yield by our formulation. We claim that the subgraphs embedded at each page are three trees. For contradiction assume that there is a forest F_q on page q. By acyclicity we know that at least two vertices on page q have no father, but at least one child, forcing them to be a root on page q via the *root rule* which contradicts the *single root rule*. Hence, the subgraphs embedded at each page of $\mathcal{E}(G,3)$ are indeed trees. Following similar arguments as in the second part of the proof of Theorem 6.1 we can easily prove that a satisfying assignment of $\mathcal{F}_t(G,3)$ yields a book embedding on 3 pages in which the subgraph assigned to each page is a tree which completes the proof. □

Further extensions to test a weaker version of Hypothesis 3

The weaker version of Hypothesis 3 shall test whether it is not always possible to construct a 3-page book embedding based on a Schnyder decomposition into three trees (regardless of the linear order underneath). For this purpose it is required that (i) the subgraph assigned to each of the three pages is not only a tree, but

a tree on $n-1$ vertices and additionally, (ii) that the three vertices which are not spanned by the three trees are pairwise adjacent forming a face f^* of the graph.

Since we search for book embeddings in which every subgraph assigned to each of the three pages is a tree we will use $\mathcal{F}_t(G,3)$ as a base and extend it by the following two rules to obtain $\mathcal{F}_s(G,3)$. Via the *fixed root rule* we set each of the three vertices from the face f^* as the root of one of the three pages. Assume that the three vertices of f^* are v_i, v_j, v_k:

$$(\rho_1(v_i) \wedge \rho_2(v_j) \wedge \rho_3(v_k))$$

Additionally we have to ensure that each of the three vertices (v_i, v_j, v_k) of f^* is not spanned by one of the three trees. For this purpose we define $\mathcal{I}(v)$ to be the set of edges incident to $v \in V$. The *page assignment rule* from $\mathcal{F}(G,3)$ which is therefore also still present in $\mathcal{F}_f(G,3)$ and $\mathcal{F}_t(G,3))$ will be replaced by the following new *forbid page assignment rule.*

$$\bigwedge_{e \in \mathcal{I}(v_i)} \neg\phi_2(e) \wedge \bigwedge_{e \in \mathcal{I}(v_j)} \neg\phi_3(e) \wedge \bigwedge_{e \in \mathcal{I}(v_k)} \neg\phi_1(e)$$

This rule unsures that all incident edges of v_i are either on the first page where v_i is the root, or on the third page, but not on the second page. Analogously none of the incident edges of v_j are on the third page, and none of the incident edges of v_k are on the first page.

Theorem 6.4. *Let $G = (V, E)$ be a (maximal) planar graph. Then G admits a book embedding on three pages, so that (i) the subgraph assigned to each of the three pages is not only a tree, but a tree on $n-1$ vertices and additionally, (ii) that the three vertices, that are not spanned by the three trees are pairwise adjacent forming a face f^* of the graph, if and only if $\mathcal{F}_s(G,3)$ is satisfiable.*

Proof. Directly follows from the validity of $\mathcal{F}_t(G,3)$. □

6.2.2 A variant to check Hypothesis 4

Assume that $G_a = (V_a, E_a)$ is a maximal planar graph that is embeddable in a book with 3 pages. Let $\Delta(G_a) = \{f_1, f_2, \ldots, f_{2|V_a|-4}\}$ be the set of (triangular) faces of G_a. In the following we describe an extension to the formula $\mathcal{F}(G_a,3)$ that forbids the so-called *unicolored* faces. These are faces whose edges are assigned to the same page of the book. We denote the resulting formula by $\mathcal{F}_a(G_a,3)$.

In comparison to our previous approaches we are not searching for a single book embedding for which an additional property must hold, but rather we have to test whether all possible book embeddings have this property. We will use the *same page variables* already present in $\mathcal{F}(G_a,3)$ to ensure this property via the *forbid unicolored face rule*:

$$(\neg\chi(e_i,e_j) \vee \neg\chi(e_i,e_k)) \quad \forall f = \{e_i,e_j,e_k\} \in \Delta(G_a)$$

Theorem 6.5. *$\mathcal{F}_a(G_a,3)$ is unsatisfiable, if and only if for every possible book embedding $\mathcal{E}(G_a,3)$ there exists a unicolored face $f_i \in \Delta(G_a)$, $i = 1, \ldots, 2|V_a| - 4$.*

Proof. Directly follows from the validity of $\mathcal{F}(G_a,3)$. □

6.2.3 A variant to check Hypothesis 5

Assume that $G_c = (V_c, E_c)$ is a maximal planar graph that is embeddable in a book with 3 pages and let $\Delta(G_c)$ be the set of (triangular) faces of G_c. To check whether a particular face $f^* = \{e_i, e_j, e_k\} \in \Delta(G_c)$ cannot be unicolored in any possible book embedding of G_c we use the already present *same page variables* of $\mathcal{F}(G_c, 3)$ again:

$$(\chi(e_i, e_j) \wedge \chi(e_i, e_k)), f^* = \{e_i, e_j, e_k\} \in \Delta(G_c)$$

The addition of the *force unicolored face rule* to the formula $\mathcal{F}(G_c, 3)$ yields a new formula which we denote by $\mathcal{F}_c(G_c; f^*, 3)$. By the following theorem it follows that in order to check Hypothesis 5 one has to check $2|V_c| - 4$ different formulas; one for each face of G_c.

Theorem 6.6. *$\mathcal{F}_c(G_c; f^*, 3)$ is unsatisfiable, if and only if there exists no book embedding of G_c with f^* being unicolored.*

Proof. Directly follows from the validity of $\mathcal{F}(G_c, 3)$. □

6.2.4 Finding "difficult" graphs

This variant aims in computing graph embeddings in books with $p \geq 2$ pages which contain the minimum number of edges on the p-th page. In other words, one can use this variant to determine how many edges must be removed from a p-page book embeddable graph, in order to become $(p-1)$-page book embeddable[1]. The motivation of this approach comes from the search for unsolvable graph instances, say for planar graphs that cannot be embedded in three pages. To construct such a graph we want to take a "difficult" graph and modify it appropriately to make it impossible for 3-page book embedding. A way to define the difficulty of a graph is by its number of edges at the last page. We minimize the number of edges on the last page and if a graph has many edges on the last page then it is considered to be a difficult one.

To tackle this variant of the problem we will use *partial MaxSAT* [102]. Partial MaxSAT sits between the classic SAT problem and the MaxSAT problem. Recall that MaxSAT seeks to maximize the number of satisfied clauses of a given formula. Partial MaxSAT combines SAT and MaxSAT by having certain clauses marked as *soft* or *relaxable* and the others as *hard* or *non-relaxable*. Given a certain number of soft and hard clauses the objective is to find a variable assignment that satisfies all hard clauses together with the maximum number of soft clauses.

We employ partial MaxSAT to minimize the number of edges of the p-th page of a p-page book embeddable graph G as follows. Our base is formula $\mathcal{F}(G, p-1)$. The clauses added due to the *page assignment rule* are the only *soft* clauses of $\mathcal{F}(G, p-1)$. All other clauses are *hard* and must be satisfied. Partial MaxSAT will maximize the number of edges assigned to the first $p-1$ pages, such that these edges do not cross. All we have to ensure is that the remaining edges (that are

[1]Clearly, this is an NP-hard problem, since it can be easily reduced to the problem of determining whether a maximal planar graph is Hamiltonian.

inevitably at the p-th page) do not cross. To achieve this we have to ensure that two edges which are not assigned to the first $p-1$ pages must be identified as *on the same page* (recall that for this purpose we have already defined the χ-variables) which can be done by the following *on last page rule*:

$$(\neg\phi_1(e_i) \wedge \ldots \wedge \neg\phi_{p-1}(e_i) \wedge \neg\phi_1(e_j) \wedge \ldots \wedge \neg\phi_{p-1}(e_j)) \rightarrow \chi(e_i, e_j)\ \forall e_i, e_j \in E$$

All clauses implied by this rule are *hard*. These clauses together with the already present *forbid crossing rule* of $\mathcal{F}(G, p-1)$ ensure that all edges assigned to the p-th page of the book will not cross as well.

Note that even if there exist several algorithms and solvers to tackle partial MaxSAT (see e.g., [53, 87]) in practice we realized that this particular variant is of limited impact, since it is time demanding even for relatively small graphs, as we will see in the following experimental section.

6.3 Practical Results

In this section we present an experimental evaluation of our SAT formulation. We ran our experiments on a Linux machine with four cores at 2.5 GHz and 8 GB of RAM. The implementation that creates the SAT instances was done in Java. For solving the SAT instances we used the `SApperloT` solver [88]. This solver is as fast as the well-known `minisat` [47] solver for smaller graphs, but it considerably outperforms `minisat` for increasing instance sizes. The runtime we report consists of both the time to create the instance and the time to solve it. Note that the time to create the SAT instance for small graphs is neglectable. For large graphs, however, that step can take a few minutes.

6.3.1 Established benchmark sets

Since the Rome and the North graphs are popular test sets for planar and nearly planar graphs we also used them as test sets for our experiment (cf. `http://www.graphdrawing.org`). The *Rome graphs* are 11534 graphs; 3281 of them are planar and 8253 are non-planar. Their average density is 0.069, where the *density* of a graph $G = (V, E)$ is $2|E|/(|V|(|V|-1))$. The number of vertices of the Rome graphs range from 10 to 110. The corresponding number of edges range from 9 to 158.

It is eye-catching that all planar Rome graphs are 2-page book embeddable (see Table 6.1). The non-planar ones on the other hand are 3-page embeddable. But since the Rome graphs are very sparse this result was more or less expected. Note that 99% of the planar Rome graphs (that is, 3248 out of 3282) are solved within 2 seconds. For the non-planar Rome graphs, the same ratio (that is, 8169 out of 8253) is achieved after 6.25 seconds.

As a second benchmark set we used the *North graphs* which are 1277 directed acyclic graphs (obtained from `http://www.graphdrawing.org`); 854 of them are planar and 423 are non-planar. The number of vertices of the North graphs range from 10 to 100. The corresponding number of edges range from 9 to 241. On average these graphs are nearly twice as dense as the Rome graphs; their average density is 0.13. Again all planar graphs were 2-page book embeddable (see also

Table 6.1. Overview of the results for the established benchmark sets.

	planar		nonplanar				
Graph class	#	$p = 2$	#	$p = 3$	$p = 4$	$p = 5$	see below
Rome	3281	3281	8253	8253	0	0	
North	854	854	423	329	25	8	61

Table 6.1). The runtime to compute the corresponding embedding for the vast majority of the planar North graphs was rather small. In particular 97.5% of them (that is, 833 out of 854 graphs) were solved within 3 seconds, with the maximum runtime being 9.4 seconds.

For the non-planar North graphs which are more dense than the non-planar Rome graphs we could determine the page number of 344 out of 423 graphs within the time limit of 1200 seconds. We also observed that for these graphs finding a 3-page book embedding is much faster than proving that such an embedding does not exist (see Figure 6.3b). For the remaining 79 graphs we increased the time limit to 3 hours and managed to get at least some partial results: (i) for 18 graphs we computed their exact page number (see also Table 6.1), (ii) 27 graphs fit in four pages, but we were not able to determine whether they could fit in three pages, (iii) 32 graphs did not fit in three pages (and 6 out of them did not even fit in four pages), but we did not manage to determine their page number. Nevertheless, all non-planar North graphs could fit into 8 pages and since the focus of this chapter is on planar and 1-planar graphs, we did not further investigated the book embeddability of these graphs.

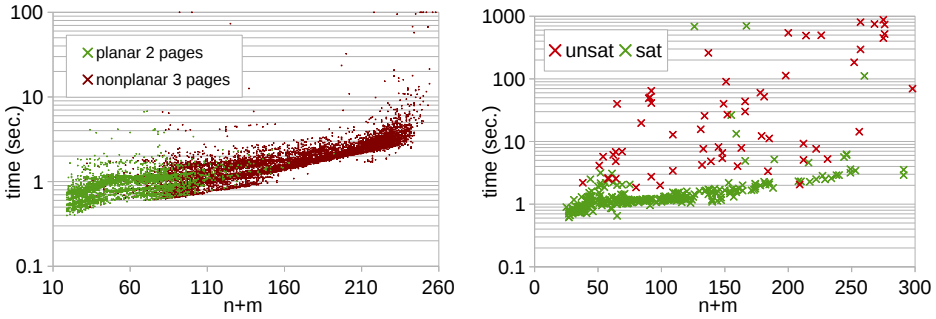


Figure 6.3. (a) Rome graphs: Runtime to compute either 2-page embeddings for planar Rome graphs (green) or 3-page embeddings for non-planar Rome graphs (red). (b) Non-planar North graphs: The time needed either to compute 3-page embeddings (green) or to prove that no 3-page embedding exist (red).

6.3.2 Crafted graphs

To prove Hypothesis 1 we also crafted several maximal planar graphs with at least 500 vertices each which we tested for 3-page book embeddability. To avoid testing Hamiltonian graphs we adopted a two-step approach that was inspired by the graph class that Yannakakis proposed as candidate to require four pages. In the first step, we randomly chose a triangulated planar (not necessarily non-Hamiltonian) graph as the base for the second step. In the second step we augmented the base graph by specific operations to make it non-Hamiltonian (and therefore at least not 2-page embeddable). Examples of these operations are:

(i) *stellate* a face f, that is, introduce a new vertex and connect it to all vertices of f,

(ii) replace a triangular face by an octahedron,

(iii) embed a non-Hamiltonian (maximal) planar graph G_f to a face f by identifying the vertices of f with the vertices of a particular face of G_f,

(iv) add a non-Hamiltonian (maximal) planar graph G_f into a face f, connect the vertices of f with the vertices of a particular face of G_f (w.l.o.g. the outer face f_o) in an octahedronic fashion (see Figure 6.4).

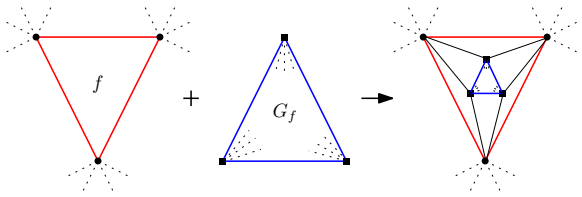


Figure 6.4. Add a non-Hamiltonian (maximal) planar graph G_f into a face f in a octahedronic fashion.

In practice we observed that these simple operations most of the times yield non-Hamiltonian planar graphs (mainly because of the presence of several separating triangles). The graphs that we crafted and tested with this approach were all maximal planar with at least 500 and at most 700 vertices. The runtime to check each instance was ranging from a couple of hours to at most a couple of days. Of course, we also tested a specific graph with roughly 600 vertices out of Yannakakis' graph class, but it turned out to be 3-page book embeddable (for this particular size).

6.3.3 Finding "difficult" graphs

The operations (iii) and (iv) from above yield non-Hamiltonian graphs. Since our goal was to find a planar graph that requires 4 pages we used the partial MaxSAT approach from Section 6.2.4 to find "difficult" graphs. These are graphs for which the minimum number of edges on the third page is as high as possible. By combining these graphs via the operations (iii) and (iv) we would be able to create

larger graphs, that are required to have a large number of edges on the third page, increasing the possibility that this third page is non-planar.

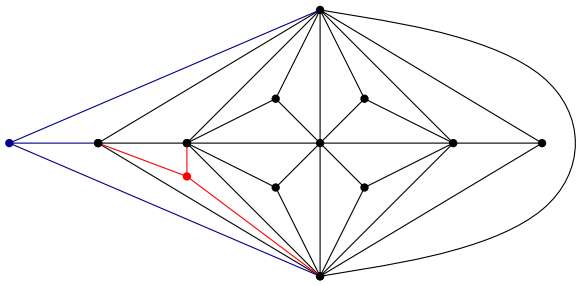


Figure 6.5. The black substructure is the well-known Goldner-Harary graph, which is the only maximal planar graph with 11 vertices (out of 1249 [26]), that requires 3 pages. Adding the blue (red, respectively) vertex and edges leads to the only two maximal planar graphs with 12 vertices (out of 7595 [26]), that require 3 pages.

In our search for small graphs that have a high minimum number of edges on the last page we tested all isomorphic maximal planar graphs for $n \in \{11, \ldots, 16\}$. The Goldner-Harary graph is the only maximal planar graph with 11 vertices which requires 3 pages (and therefore at least one edge on the third page). For 12 vertices there exist only 2 graphs that require 3 pages (both have at least one edge on the third page). Both of these graphs can be created by stellating one out of two particular faces of the Goldner-Harary graph (see Figure 6.5). With increasing graph size the number of non-Hamiltonian graphs rises rapidly (see Table 6.2). The smallest graph that requires at least two edges on the third page has 14 vertices, and among all tested maximal planar graphs with at most 16 vertices, not a single one needed to have 3 edges on the third page.

	overview		min. edges on page 3		
n	# [26]	non-Ham.	1	2	3
11	1,249	1	1	0	0
12	7,595	2	2	0	0
13	49,566	30	30	0	0
14	339,722	239	237	2	0
15	2,406,841	2,369	2,361	8	0
16	17,490,241	37,348	37,119	229	0

Table 6.2. The minimum number of edges on the third page for maximal planar graphs with at most 16 vertices.

Due to the size of the search space and the fact that the runtime of the partial MaxSAT approach increases rapidly with increasing graph size we were not able to test any further complete set of maximal planar graphs for larger n.

For $n = 5$ ($n = 6$, respectively), the graphs G' created by stellating every face are the only maximal planar graphs with $n' = 11$ ($n' = 14$, respectively) vertices that require one (two, respectively) edges to be embedded on the third page.

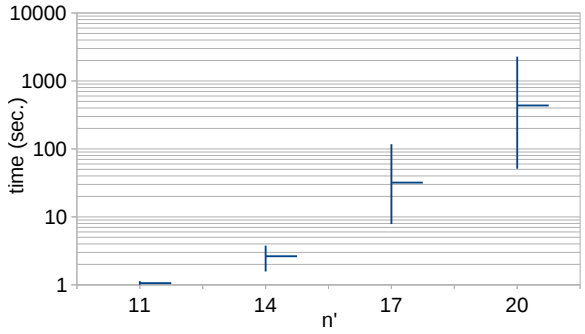


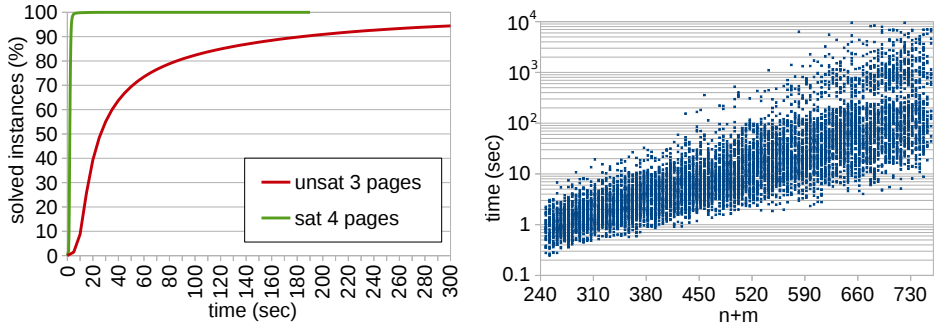
Figure 6.6. The runtime to compute the minimum number of edges on the third page for stellated maximal planar graphs. The horizontal line marks the average runtime for a given size of the graph, whereas the two end points of the vertical lines denote the minimum and maximum runtime. Every graph was tested three times.

As already mentioned, the partial MaxSAT approach reaches its practical limit already for graphs with more than 20 vertices. We created, based on all maximal planar graphs for $n \in \{5, \ldots, 8\}$, the stellated graphs G' of size $n' \in \{11, 14, 17, 20\}$ and computed the minimum amount of edges on the third page three times for every G'. For graphs larger than 14 vertices the average runtime for a single instance increased by one order of magnitude in comparison to the next smaller group of graphs (see Figure 6.6). While the average runtime for graphs of size $n' = 14$ was 2.64 seconds the average runtime increased to 31.93 seconds for $n' = 17$ and to 435.54 seconds for the stellated maximal planar graphs with $n' = 20$ vertices.

6.3.4 1-planar graphs

To check Hypothesis 2 for more than four pages we initially generated all 2,098,675 planar triconnected quadrangulations with 25 vertices and minimum degree three using the tool `plantri` [26]. By augmenting every face with two crossing edges, the generated quadrangulations yield optimal 1-planar graphs. Our experiments showed that all tested optimal 1-planar graphs required four pages. The runtime distribution is shown in Figure 6.7a. Computing a 4-page embedding was always fast: For 99.06% of the graphs the solver found a solution within 4.7 seconds. The maximum runtime for a single instance was 186 seconds. Proving that no 3-page embedding existed was harder. In less than 5 minutes 94.4% of the instances could be solved. However, for very few instances this could take up to two hours.

To obtain a better understanding of the connection between the runtime of our approach and the size of the graphs we randomly created 8312 optimal 1-planar graphs of different sizes varying from 50 to 155 vertices. Starting from the cube graph (see Figure 6.1) we iteratively applied one of the two operations described by Suzuki [136] in order to generate all optimal 1-planar graphs, until we reached the desired size of the graph. The runtime to compute 4-page embeddings for these graphs is shown in Figure 6.7b. For nearly all graphs up to 100 vertices a 4-page embedding could be computed within two minutes. However, with increasing vertex-count the amount of graphs that could take up to several hours of CPU time



(a) (b)

Figure 6.7. (a) The runtime for maximal 1-planar graphs with 25 vertices. The red curve shows the runtime to prove that no 3-page embedding exist; the green curve shows the runtime to compute 4-page embeddings. (b) The runtime to compute 4-page embeddings for randomized optimal 1-planar graphs.

to compute an embedding on four pages rises rapidly.

6.3.5 Phase transition

To further investigate the runtime behavior of our SAT approach we created random optimal 1-planar graphs of different sizes with the method described before. Let G be an optimal 1-planar graph that is created at random with the method we described earlier in this chapter. We denote G_i the subgraph of G that is obtained by removing i randomly selected edges of G. For every (large enough) graph G there exists a z, such that G_z is 3-page embeddable and G_{z-1} is not. These are the instances considered to be hard.

We are interested in this phase transition (see Figure 6.8). We created more than 200 graphs of varying sizes ($n = 16, \ldots, 21$), grouped the instances according to their z-value and computed the average runtime of each group. As expected, the runtime increases from G_1 towards G_{z-1} and decreases afterwards. At some point the runtime is mainly dominated by the time used to parse the graph and create the SAT instance. Hence, sparse graphs that are far away from the phase transition can be embedded very fast. It is significantly harder to prove that a graph G_{z-1} is not embeddable on 3 pages than computing a 3-page book embedding for G_z.

Another interesting observation can be done when comparing the runtime of graphs with the same amount of vertices, but different z-value. The dashed curves are always above the continuous curves in the unsatisfiable area of the diagram (that is located left of the gray line). This indicates, that it is harder to find a proof for nonembeddability for sparser graphs. On the first look that may seem counter-intuitive, since the search space for sparse graphs is smaller than for their dense counterparts, but could be explained by the idea that the denser the graph is the more possibilities to find a counter example exist.

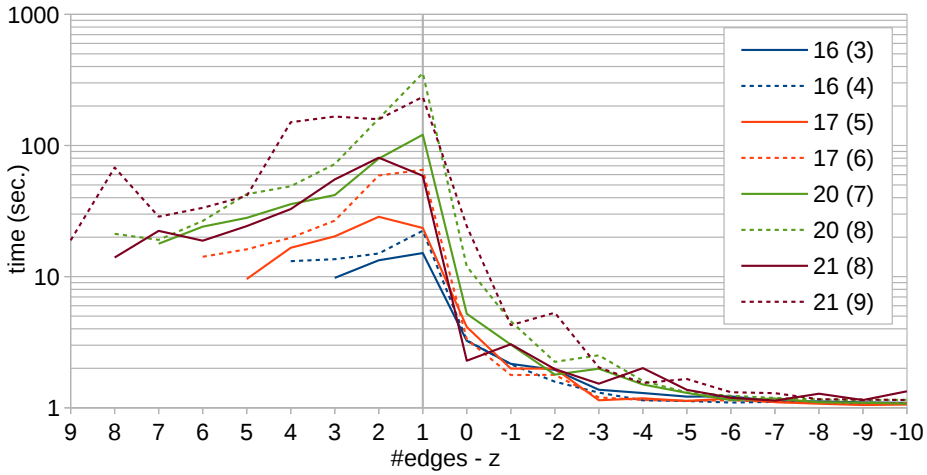


Figure 6.8. Runtime of the SAT approach for generated subgraphs of optimal 1-planar graphs to test whether the graph is 3-page book embeddable.

6.3.6 Randomized planar graphs

To check Hypotheses 3-5 we generated a large set of random maximal planar graphs as follows. We applied Delaunay triangulation [42] on a set of randomly created points within a triangular region. To avoid Hamiltonian graphs we simply stellated every face of the produced Delaunay triangulations. Our results are summarized in the following. Note that none of the tested graphs corroborate Hypotheses 3-5.

- For Hypothesis 3 we tested 15040 maximal planar graphs of varying number of vertices between 25 and 80. The SAT approach was able to solve 70.78% of the instances (that is, 10646 out of 15040) within 3 minutes and 76.37% (that is, 11487 out of 15040) of the instances within 20 minutes, which was the time limit of the computation.

- For Hypothesis 4 we tested 7174 maximal planar graphs of varying number of vertices between 59 and 125. The SAT approach was able to solve 92.75% of the instances (that is, 6621 out of 7174) within 10 minutes and 99% of the instances within an hour. The maximum time that was needed to solve a single instance was 5 hours and 6 minutes.

- For Hypothesis 5 we managed to test 277284 maximal planar graphs of varying number n of vertices between 59 and 95. Every single instance, each containing $2n - 4$ different SAT formulas, required only few seconds to be tested.

6.4 Summary

In this chapter we approached the problem of determining whether a graph can be embedded in a book of a certain number of pages from a SAT solving perspective.

We presented a novel SAT formulation for the Book Embedding problem which is of a simple nature, quite intuitive and easy-to-implement, but simultaneously robust enough to solve non-trivial instances of the problem in a reasonable amount of time. It is possible to solve large planar graphs of up to 700 nodes within a few hours of computation time. However, around the optimal solution, where the problem switches from unsatisfiable to satisfiable, we observe the well-known *phase transitional behavior* for SAT problems [27].

The SAT formulation is designed in a very modular way. It can be separated in three main parts: (i) the creation of a complete order of the vertices of the input graph which represents the position of each vertex on the spine of the book, (ii) the assignment of all edges to one of the available possible pages, (iii) ensuring the property of non-crossing edges on each of the p pages.

Because of this modularity it is very easy to formulate SAT encodings that differ in one or more aspects from the original Book Embedding formulation. For example, it is quite easy to modify the original formulation so to allow up to a maximum number of crossing on any page, or require that a single edge is embedded on more than one page. Motivated by the Hypotheses 3, 4 and 5 we have shown how to extend the basic SAT formulation by inserting new constraints to incorporate the additional requirements.

Unfortunately, we were able neither to find a planar graph that requires 4 pages, nor to find a 1-planar graph that requires more than 4 pages. However, we were able to find a relatively small 1-planar graph whose book thickness is exactly four and thus, prove the first (non-trivial) lower bound on the book thickness of 1-planar graphs.

We will get back to the possibility of applying SAT techniques to problems from the field of graph theory and graph drawing in Chapter 7, where the work of this thesis is concluded. We will also give some ideas and directions for future research regarding the usage of SAT to solve other interesting problems in graph theory in general, and for the Book Embedding problem specifically.

7 Conclusion

This thesis can be divided into main parts. The first part focuses on minimal unsatisfiable subsets, and specifically on the enumeration of them. We elaborated an interactive approach to extract different MUSes, and used the developed methodologies for the interactive extraction to improve the automatic enumeration of MUSes. The second part of the thesis introduces two novel applications for SAT. The first problem stems from Bioinformatics and searches for an evolutionary tree. The second applications is based in the field of graph theory and graph drawing.

Results of the thesis

The motivating idea for the work presented in Chapter 3 was to turn deletion-based MUS extraction into an interactive process, with the goal of allowing experts to use their domain knowledge while looking for good explanations of infeasibilities. Different MUSes for one instance often overlap in a *core region*, which is satisfiable itself but has to be "enlarged" by only a few other constraints from the formula to construct an MUS. Due to this, the user could have to detect the same core region for different MUSes in an interactive process multiple times. To overcome this problem we developed a scheme for learning and retrieving the information that can be gained from successful and unsuccessful reduction attempts for a maximum of information reuse during interactive search space exploration. The so-called *meta instance* over selector variables can store the information about all encountered *critical clauses*, and can be used to retrieve the clauses which are implied to be critical in some US by solving this meta instance with assumptions. By grouping selector variables into *blocks* it was possible not only to compress the meta instance in a significant way, but also to introduce the methodical foundations for the next chapter.

The ideas of an extended meta instance and a generalization of the blocks were incorporated into state-of-the-art MUS enumeration algorithms in Chapter 4. The development of an identification procedure for a set of clauses that exhibit the *block property* aided new techniques to improve the MUS extraction. In addition, we put the focus on the computation of an often undesired by-product of MUS enumeration algorithms, the minimal correction sets (MCSes). By using a meta instance extensively we showed how to benefit from these additional MCSes and were able to speed up a single MUS extraction by reducing the search space. Combining all these

techniques enabled us to outperform state-of-the-art MUS enumeration algorithms.

In Chapter 5 we moved away from the enumeration of MUSes and developed a novel approach to solve *Maximum Quartet Consistency* by encoding *Quartet Compatibility* as a satisfiability problem. The encoding is based on the split operation, that creates iteratively a phylogeny covering all the taxa from the input. However, the first version of the encoding was not efficient enough, since the conflicts were detected at a very low level in the DPLL search tree. We elaborated two different extensions, that tackle this problem and introduce redundant clauses to allow the SAT solver the deduction of conflicts at a higher level of the search tree. The practical analysis showed that we outperform all available ASP, PBO and MaxSAT approaches.

The second new SAT formulation we developed during this work solves the Book Embedding problem. The formulation is of a simple nature, intuitive and easy-to-implement, but simultaneously robust enough to solve non-trivial instances of the problem in a reasonable amount of time. We proved a first (non-trivial) lower bound on the book thickness of 1-planar graphs by finding a relatively small 1-planar graph whose book thickness is exactly four. Given the modular structure of our SAT formulation it can easily be extended to incorporate additional requirements of the embedding. We have shown different extensions to the basic formulation and were able to find a proof, that it is not always possible to construct a 3-page book embedding for planar graphs that are based on a Schnyder decomposition into three trees, regardless of the linear order of the vertices on the spine.

Directions for future work

There are several possibilities for future research on the different topics studied in this thesis, some of which we present here:

The interactive MUS extraction paradigm introduced in Chapter 3 is far away from being fully explored. Some interesting ideas are:

- Incorporate the *insertion-based* MUS extraction to the interactive MUS extraction paradigm. This extension would require a completely different approach to the interactive extraction, since the Algorithm 2.4 detects only one critical clause in each iteration. A successful insertion attempt does not lead necessarily to the addition of the clause to the MUS, since only the last successful insertion attempt of one iteration adds one clause to the MUS.

- Extend the interactive paradigm to the *hybrid* MUS extraction algorithm. The hybrid approach selects a single clause in each iteration and tests, whether that clause is critical or not. Since this is analogous to the *deletion-based* algorithm, no further issues have to be considered.

- The already mentioned possibilities in using the *meta instance* to impose additional properties on the encountered USes (see Section 3.5) should be implemented.

Future research directions for the usage of the *block property*, which was introduced in Chapter 4, are for example the following.

- Explore the possibilities to use the block information directly within any MSS/MCS oracle.
- Incorporate sophisticated data structures into the algorithms that use the block property to detect more MCSes.
- Prove the block property without the current problems to be able to extract MUSes by a single call of an MUS enumerator.
- Develop novel quality measures of MUSes. At the moment, the number of MUSes seems to be the natural measure when assessing partial MUS enumeration techniques, but finding one MUS of superb quality with respect to the application may be better than finding many MUSes of poor quality with respect to the application.

Since the split encoding, that we introduced in Chapter 5 to solve the Maximum Quartet Consistency problem, reaches its practical limit for approximately 20 taxa some interesting directions are:

- In its current version, the split encoding is particular efficient for small binary phylogenies. Can the split encoding be generalized to be similarly efficient for a general tree?
- Develop a SAT formulation that solves the Quartet Compatibility problem on an incomplete set of quartet topologies for a higher number of taxa ($n \geq 100$) efficiently.
- Evolutionary trees are more and more replaced by evolutionary networks. Can the split encoding be modified, such that it incorporates network structures?

Since we did not prove (or disprove) any of the Hypotheses we stated for the Book Embedding problem in Chapter 6, they remain open problems. Additional possible research directions are:

- Develop a SAT formulation for other "one-dimensional" and grid-based problems (like pathwidth, bandwidth, optimal st-orientation, visibility representation, and feedback arc set) based on the *relative encoding* used also for the Book Embedding problem.
- Reduce the number of clauses that are used within the Book Embedding formulation to be able to solve denser graphs and a higher number of pages.
- Explore the possibility to use an iterative SAT solving approach, that adds the edges of a graph in a particular sequence. By adding the edges not all at once, the SAT solver solves embeds subgraphs, and is able to reuse the information he gained for subgraphs for the next iterations.

References

[1] bwGRiD (http://www.bw-grid.de), member of the German D-Grid initiative, funded by the Ministry for Education and Research (Bundesministerium fuer Bildung und Forschung) and the Ministry for Science, Research and Arts Baden-Wuerttemberg (Ministerium fuer Wissenschaft, Forschung und Kunst Baden-Wuerttemberg).

[2] Zaher S. Andraus, Mark H. Liffiton, and Karem A. Sakallah. Reveal: A formal verification tool for verilog designs. In *Logic for Programming, Artificial Intelligence, and Reasoning - LPAR 2008 - 15th International Conference*, volume 5330 of *LNCS*, pages 343–352. Springer, 2008.

[3] Alessandro Armando and Luca Compagna. Automatic SAT-Compilation of Protocol Insecurity Problems via Reduction to Planning. In *Formal Techniques for Networked and Distributed Systems - FORTE 2002*, pages 210–225, 2002.

[4] James Bailey and Peter J. Stuckey. Discovery of Minimal Unsatisfiable Subsets of Constraints Using Hitting Set Dualization. In *Practical Aspects of Declarative Languages - PADL 2005 - 7th International Symposium*, volume 3350 of *LNCS*, pages 174–186. Springer, 2005.

[5] R. R. Bakker, F. Dikker, F. Tempelman, and P. M. Wognum. Diagnosing and Solving Over-Determined Constraint Satisfaction Problems. In *13th International Joint Conference on Artificial Intelligence - IJCAI 1993*, pages 276–281. Morgan Kaufmann, 1993.

[6] Michael A. Bekos, Till Bruckdorfer, Michael Kaufmann, and Chrysanthi N. Raftopoulou. 1-Planar Graphs have Constant Book Thickness. In *Algorithms - ESA 2015 - 23rd Annual European Symposium*, volume 9294 of *LNCS*, pages 130–141. Springer, 2015.

[7] Michael A. Bekos, Martin Gronemann, and Chrysanthi N. Raftopoulou. Two-Page Book Embeddings of 4-Planar Graphs. In *Theoretical Aspects of Computer Science - STACS 2014 - 31st International Symposium*, volume 25 of *LIPIcs*, pages 137–148. Schloss Dagstuhl - Leibniz-Zentrum fuer Informatik, 2014.

[8] Anton Belov, Marijn Heule, and João Marques-Silva. MUS Extraction Using Clausal Proofs. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, volume 8561 of *LNCS*, pages 48–57. Springer, 2014.

[9] Anton Belov, Inês Lynce, and João Marques-Silva. Towards efficient MUS extraction. *AI Communications*, 25(2):97–116, 2012.

[10] Anton Belov and João Marques-Silva. Accelerating MUS extraction with recursive model rotation. In *Formal Methods in Computer-Aided Design - FMCAD 2011 - 11th International Conference*, pages 37–40. FMCAD Inc., 2011.

[11] Anton Belov and João Marques-Silva. MUSer2: An Efficient MUS Extractor. *Journal on Satisfiability, Boolean Modeling and Computation*, 8(3/4):123–128, 2012.

[12] Amir Ben-Dor, Benny Chor, Dan Graur, Ron Ophir, and Dan Pelleg. From four-taxon trees to phylogenies (preliminary report): the case of mammalian evolution. In *Research in Computational Molecular Biology - RECOMB'98 - Second Annual International Conference*, pages 9–19. ACM, 1998.

[13] Yael Ben-Haim, Alexander Ivrii, Oded Margalit, and Arie Matsliah. Perfect hashing and CNF encodings of cardinality constraints. In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference*, volume 7317 of *LNCS*, pages 397–409. Springer, 2012.

[14] Claude Berge. *The theory of graphs and its applications*. Methuen, 1962.

[15] Frank Bernhart and Paul C. Kainen. The book thickness of a graph. *Journal of Combinatorial Theory, Series B*, 27(3):320–331, 1979.

[16] Vincent Berry and Olivier Gascuel. Inferring evolutionary trees with strong combinatorial evidence. *Theoretical Computer Science*, 240(2):271–298, 2000.

[17] Vincent Berry, Tao Jiang, Paul E. Kearney, Ming Li, and Todd Wareham. Quartet Cleaning: Improved Algorithms and Simulations. In *7th Annual European Symposium on Algorithms - ESA '99*, volume 1643 of *LNCS*, pages 313–324. Springer, 1999.

[18] Therese C. Biedl, Thomas Bläsius, Benjamin Niedermann, Martin Nöllenburg, Roman Prutkin, and Ignaz Rutter. Using ILP/SAT to Determine Pathwidth, Visibility Representations, and other Grid-Based Graph Drawings. In *Graph Drawing - GD 2013 - 21st International Symposium*, volume 8242 of *LNCS*, pages 460–471. Springer, 2013.

[19] Armin Biere, Edmund M. Clarke, and Yunshan Zhu. Combining Local and Global Model Checking. *Electronic Notes in Theoretical Computer Science*, 23(2):34–45, 1999.

[20] Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[21] Tomasz Bilski. Optimum embedding of complete graphs in books. *Discrete Mathematics*, 182(1-3):21–28, 1998.

[22] Elazar Birnbaum and Eliezer L. Lozinskii. Consistent subsets of inconsistent systems: structure and behaviour. *Journal of Experimental and Theoretical Artificial Intelligence*, 15(1):25–46, 2003.

[23] Robin L. Blankenship. *Book Embeddings of Graphs*. Phd thesis, Louisiana State University, 2003.

[24] R. Bodendiek, H. Schumacher, and K. Wagner. Über 1-optimale Graphen. *Mathematische Nachrichten*, 117(1):323–339, 1984.

[25] Maria Luisa Bonet and Katherine St. John. Efficiently calculating evolutionary tree measures using SAT. In *Theory and Applications of Satisfiability Testing - SAT 2009 - 12th International Conference*, volume 5584 of *LNCS*, pages 4–17. Springer, 2009.

[26] Gunnar Brinkmann, Sam Greenberg, Catherine S. Greenhill, Brendan D. McKay, Robin Thomas, and Paul Wollan. Generation of simple quadrangulations of the sphere. *Discrete Mathematics*, 305(1-3):33–54, 2005.

[27] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *AI*, pages 331–340. Morgan Kaufmann, 1991.

[28] Markus Chimani, Petra Mutzel, and Immanuel M. Bomze. A New Approach to Exact Crossing Minimization. In *Algorithms - ESA 2008 - 16th Annual European Symposium*, volume 5193 of *LNCS*, pages 284–296. Springer, 2008.

[29] John W. Chinneck and Erik W. Dravnieks. Locating Minimal Infeasible Constraint Sets in Linear Programs. *INFORMS Journal on Computing*, 3(2):157–168, 1991.

[30] Fan R. K. Chung, Frank Thomson Leighton, and Arnold L. Rosenberg. Embedding Graphs in Books: A Layout Problem with Applications to VLSI Design. *SIAM Journal on Algebraic and Discrete Methods*, 8(1):33–58, 1987.

[31] Edmund M. Clarke, Daniel Kroening, and Flavio Lerda. A Tool for Checking ANSI-C Programs. In *Tools and Algorithms for the Construction and Analysis of Systems - TACAS 2004 - 10th International Conference*, volume 2988 of *LNCS*, pages 168–176. Springer, 2004.

[32] Stephen A. Cook. The Complexity of Theorem-proving Procedures. In *Third Annual ACM Symposium on Theory of Computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.

[33] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to Algorithms, Third Edition*. The MIT Press, 3rd edition, 2009.

[34] Gérard Cornuéjols, Denis Naddef, and William R. Pulleyblank. Halin graphs and the Travelling Salesman Problem. *Mathematical Programming*, 26(3):287–294, 1983.

[35] James M. Crawford and Andrew B. Baker. Experimental Results on the Application of Satisfiability Algorithms to Scheduling Problems. In *Twelfth National Conference on Artificial Intelligence (Vol. 2)*, AAAI'94, pages 1092–1097. American Association for Artificial Intelligence, 1994.

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

[37] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7(3):201–215, 1960.

[38] Hidde de Jong and Michel Page. Search for Steady States of Piecewise-Linear Differential Equation Models of Genetic Regulatory Networks. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 5(2):208–222, 2008.

[39] Johan de Kleer and Brian C. Williams. Diagnosing Multiple Faults. *Artificial Intelligence*, 32(1):97–130, 1987.

[40] Maria J. García de la Banda, Peter J. Stuckey, and Jeremy Wazny. Finding all minimal unsatisfiable subsets. In *5th International ACM SIGPLAN Conference on Principles and Practice of Declarative Programming*, pages 32–43. ACM, 2003.

[41] J. L. de Siqueira N. and Jean-Francois Puget. Explanation-Based Generalisation of Failures. In *8th European Conference on Artificial Intelligence - ECAI 88*, pages 339–344. Pitmann Publishing, 1988.

[42] Boris N. Delaunay. Sur la sphère vide. *Bulletin of Academy of Sciences of the USSR*, (6):793–800, 1934.

[43] Johannes Dellert. Interactive Extraction of Minimal Unsatisfiable Cores Enhanced by Metalearning. Diplomarbeit, Universität Tübingen, 2013.

[44] Johannes Dellert, Kilian Evang, and Frank Richter. Kahina: A Hybrid Trace-Based and Chart-Based Debugging System for Grammar Engineering. In *ESSLLI 2013 Workshop on High-level Methodologies for Grammar Engineering - HMGE 2013*, 2013.

[45] Johannes Dellert, Christian Zielke, and Michael Kaufmann. MUStICCa: MUS Extraction with Interactive Choice of Candidates. In *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference*, volume 7962 of *LNCS*, pages 408–414. Springer, 2013.

[46] Vida Dujmovic and David R. Wood. Graph Treewidth and Geometric Thickness Parameters. *Discrete & Computational Geometry*, 37(4):641–670, 2007.

[47] Niklas Eén and Niklas Sörensson. An Extensible SAT-solver. In *Theory and Applications of Satisfiability Testing - SAT 2003 - 6th International Conference*, volume 2919 of *LNCS*, pages 502–518. Springer, 2003.

[48] Niklas Eén and Niklas Sörensson. Translating Pseudo-Boolean Constraints into SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 2(1-4):1–26, 2006.

[49] Péter L. Erdös, Michael Anthony Steel, László A. Székely, and Tandy Warnow. A Few Logs Suffice to Build (almost) All Trees: Part II. *Theoretical Computer Science*, 221(1-2):77–118, 1999.

[50] Péter L. Erdös, Mike A. Steel, László A. Székely, and Tandy Warnow. Constructing Big Trees from Short Sequences. In *24th International Colloquium on Automata, Languages and Programming - ICALP'97*, volume 1256 of *LNCS*, pages 827–837. Springer, 1997.

[51] Paolo Ferraris and Enrico Giunchiglia. Planning as Satisfiability in Nondeterministic Domains. In *Seventeenth National Conference on Artificial Intelligence and Twelfth Conference on Innovative Applications of Artificial Intelligence, AAAI 2000, IAAI 2000*, pages 748–753, 2000.

[52] L.R. Foulds and R.L. Graham. The steiner problem in phylogeny is np-complete. *Advances in Applied Mathematics*, 3(1):43 – 49, 1982.

[53] Zhaohui Fu and Sharad Malik. On Solving the Partial MAX-SAT Problem. In *Theory and Applications of Satisfiability Testing - SAT 2006 - 9th International Conference*, volume 4121 of *LNCS*, pages 252–265. Springer, 2006.

[54] Oliver Gableske. An Ising Model Inspired Extension of the Product-Based MP Framework for SAT. In *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference*, volume 8561 of *LNCS*, pages 367–383. Springer, 2014.

[55] Graeme Gange, Peter J. Stuckey, and Kim Marriott. Optimal k-Level Planarization and Crossing Minimization. In *Graph Drawing - GD 2010 - 18th International Symposium*, volume 6502 of *LNCS*, pages 238–249. Springer, 2010.

[56] Joseph L. Ganley and Lenwood S. Heath. The pagenumber of k-trees is o(k). *Discrete Applied Mathematics*, 109(3):215–221, 2001.

[57] Rafael M. Gasca, Carmelo Del Valle, María Teresa Gómez López, and Rafael Ceballos. NMUS: Structural Analysis for Improving the Derivation of All MUSes in Overconstrained Numeric CSPs. In *12th Conference of the Spanish Association for Artificial Intelligence - CAEPIA 2007*, volume 4788 of *LNCS*, pages 160–169. Springer, 2007.

[58] Martin Gebser, Benjamin Kaufmann, André Neumann, and Torsten Schaub. *clasp* : A Conflict-Driven Answer Set Solver. In *Logic Programming and Nonmonotonic Reasoning - LPNMR 2007 - 9th International Conference*, volume 4483 of *LNCS*, pages 260–265. Springer, 2007.

[59] Martin Gebser, Benjamin Kaufmann, and Torsten Schaub. Conflict-driven answer set solving: From theory to practice. *Artificial Intelligence*, 187:52–89, 2012.

[60] Enrico Giunchiglia, Alessandro Massarotto, and Roberto Sebastiani. Act, and the Rest Will Follow: Exploiting Determinism in Planning as Satisfiability. In *Fifteenth National Conference on Artificial Intelligence and Tenth Innovative Applications of Artificial Intelligence Conference, AAAI 98, IAAI 98*, pages 948–953, 1998.

[61] Evguenii I. Goldberg, Mukul R. Prasad, and Robert K. Brayton. Using SAT for combinational equivalence checking. In *Design, Automation and Test in Europe - DATE 2001*, pages 114–121. ACM, 2001.

[62] A. Goldner and Frank Harary. Note on a smallest nonhamiltonian maximal planar graph. *Bulletin of the Malaysian Mathematical Sciences Society*, 6(1):41–42, 1975.

[63] Jens Gramm and Rolf Niedermeier. A fixed-parameter algorithm for minimum quartet inconsistency. *Journal of Computer and System Sciences*, 67(4):723–741, 2003.

[64] Harvey J. Greenberg and Frederic H. Murphy. Approaches to Diagnosing Infeasible Linear Programs. *INFORMS Journal on Computing*, 3(3):253–261, 1991.

[65] Éric Grégoire, Bertrand Mazure, and Cédric Piette. Using local search to find MSSes and MUSes. *European Journal of Operational Research*, 199(3):640–646, 2009.

[66] Stefan Grünewald, Peter J. Humphries, and Charles Semple. Quartet Compatibility and the Quartet Graph. *Electronic Journal of Combinatorics*, 15(1), 2008.

[67] Jun Gu, Paul W. Purdom, John Franco, and Benjamin W. Wah. Algorithms for the Satisfiability (SAT) Problem: A Survey. In *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, volume 35, pages 19–152. American Mathematical Society, 1996.

[68] Dan Gusfield. *Algorithms on Strings, Trees, and Sequences - Computer Science and Computational Biology*. Cambridge University Press, 1997.

[69] Benjamin Han and Shie-Jue Lee. Deriving minimal conflict sets by CS-trees with mark set in diagnosis from first principles. *IEEE Transactions on Systems, Man, and Cybernetics, Part B*, 29(2):281–286, 1999.

[70] Lenwood S. Heath. Embedding Planar Graphs in Seven Pages. In *Foundations of Computer Science - FOCS 84 - 25th Annual Symposium*, pages 74–83. IEEE Computer Society, 1984.

[71] Lenwood S. Heath. *Algorithms for Embedding Graphs in Books*. Phd thesis, University of North Carolina, 1985.

[72] Lenwood S. Heath, Frank Thomson Leighton, and Arnold L. Rosenberg. Comparing Queues and Stacks as Mechanisms for Laying out Graphs. *SIAM Journal on Discrete Mathematics*, 5(3):398–412, 1992.

[73] Federico Heras, António Morgado, and João Marques-Silva. An Empirical Study of Encodings for Group MaxSAT. In *Advances in Artificial Intelligence - AI 2012 - 25th Canadian Conference*, volume 7310 of *LNCS*, pages 85–96. Springer, 2012.

[74] Holger Hoos and Thomas Stützle. *Stochastic Local Search: Foundations & Applications*. Morgan Kaufmann Publishers Inc., San Francisco, CA, USA, 2004.

[75] Aimin Hou. A Theory of Measurement in Diagnosis from First Principles. *Artificial Intelligence*, 65(2):281–328, 1994.

[76] Anthony Hunter and Sébastien Konieczny. On the measure of conflicts: Shapley inconsistency values. *Artificial Intelligence*, 174(14):1007–1026, 2010.

[77] Daniel H. Huson, Scott Nettles, and Tandy Warnow. Disk-Covering, a Fast-Converging Method for Phylogenetic Tree Reconstruction. *Journal of Computational Biology*, 6(3/4):369–386, 1999.

[78] Alexey Ignatiev, António Morgado, Vasco M. Manquinho, Inês Lynce, and João Marques-Silva. Progression in Maximum Satisfiability. In *21st European Conference on Artificial Intelligence - ECAI 2014*, volume 263 of *Frontiers in Artificial Intelligence and Applications*, pages 453–458. IOS Press, 2014.

[79] Tao Jiang, Paul E. Kearney, and Ming Li. Orchestrating Quartets: Approximation and Data Correction. In *Foundations of Computer Science - FOCS '98 - 39th Annual Symposium*, pages 416–425. IEEE Computer Society, 1998.

[80] Ulrich Junker. QUICKXPLAIN: Preferred Explanations and Relaxations for Over-Constrained Problems. In *Nineteenth National Conference on Artificial Intelligence, Sixteenth Conference on Innovative Applications of Artificial Intelligence*, pages 167–172. AAAI Press / The MIT Press, 2004.

[81] Paul C. Kainen and Shannon Overbay. Extension of a theorem of Whitney. *Applied Mathematics Letters*, 20(7):835–837, 2007.

[82] Henry Kautz and Bart Selman. Planning As Satisfiability. In *10th European Conference on Artificial Intelligence*, ECAI '92, pages 359–363. John Wiley & Sons, Inc., 1992.

[83] Dimitris J. Kavvadias and Elias C. Stavropoulos. An Efficient Algorithm for the Transversal Hypergraph Generation. *Journal of Graph Algorithms and Applications*, 9(2):239–264, 2005.

[84] Vlado Keselj and Nick Cercone. A formal approach to subgrammar extraction for NLP. *Mathematical and Computer Modelling*, 45(3-4):394–403, 2007.

[85] L. G. Khachiyan. A polynomial algorithm in linear programming. *Doklady Akademii Nauk SSSR*, 244:1093–1096, 1979.

[86] V. Klee and G. J. Minty. How Good is the Simplex Algorithm? In *Inequalities III*, pages 159–175. Academic Press Inc., 1972.

[87] Miyuki Koshimura, Tong Zhang, Hiroshi Fujita, and Ryuzo Hasegawa. QMaxSat: A Partial Max-SAT Solver. *Journal on Satisfiability, Boolean Modeling and Computation*, 8(1/2):95–100, 2012.

[88] Stephan Kottler. Description of the SApperloT, SArTagnan and MoUsSaka solvers for the SAT-competition 2011, 2011.

[89] Mark H. Liffiton and Ammar Malik. Enumerating Infeasibility: Finding Multiple MUSes Quickly. In *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems - CPAIOR 2013 - 10th International Conference*, volume 7874 of *LNCS*, pages 160–175. Springer, 2013.

[90] Mark H. Liffiton, Michael D. Moffitt, Martha E. Pollack, and Karem A. Sakallah. Identifying conflicts in overconstrained temporal problems. In *Nineteenth International Joint Conference on Artificial Intelligence - IJCAI 2005*, pages 205–211. Professional Book Center, 2005.

[91] Mark H. Liffiton and Karem A. Sakallah. Algorithms for Computing Minimal Unsatisfiable Subsets of Constraints. *Journal of Automated Reasoning*, 40(1):1–33, 2008.

[92] Mark H. Liffiton and Karem A. Sakallah. Generalizing Core-Guided Max-SAT. In *Theory and Applications of Satisfiability Testing - SAT 2009 - 12th International Conference*, volume 5584 of *LNCS*, pages 481–494. Springer, 2009.

[93] Fangzhen Lin and Yuting Zhao. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence*, 157(1-2):115–137, 2004.

[94] Inês Lynce and João Marques-Silva. Haplotype Inference with Boolean Satisfiability. *International Journal on Artificial Intelligence Tools*, 17(2):355–387, 2008.

[95] Inês Lynce and Joël Ouaknine. Sudoku as a SAT problem. In *International Symposium on Artificial Intelligence and Mathematics - ISAIM 2006*, 2006.

[96] Inês Lynce and João P. Marques Silva. On Computing Minimum Unsatisfiable Cores. In *Theory and Applications of Satisfiability Testing - SAT 2004 - 7th International Conference*, volume 3542 of *LNCS*, pages 305–310. Springer, 2004.

[97] Seth M. Malitz. Genus g graphs have pagenumber $O(\sqrt{g})$. *Journal of Algorithms*, 17(1):85–109, 1994.

[98] Seth M. Malitz. Graphs with E edges have pagenumber o(sqrt(e)). *Journal of Algorithms*, 17(1):71–84, 1994.

[99] João Marques-Silva, Federico Heras, Mikolás Janota, Alessandro Previti, and Anton Belov. On Computing Minimal Correction Subsets. In *23rd International Joint Conference on Artificial Intelligence - IJCAI 2013*. IJCAI/AAAI, 2013.

[100] João Marques-Silva, Mikolás Janota, and Anton Belov. Minimal sets over monotone predicates in boolean formulae. In *Computer Aided Verification - CAV 2013 - 25th International Conference*, volume 8044 of *LNCS*, pages 592–607. Springer, 2013.

[101] Fabio Massacci and Laura Marraro. Logical Cryptanalysis as a SAT Problem. *Journal of Automated Reasoning*, 24(1/2):165–203, 2000.

[102] Shuichi Miyazaki, Kazuo Iwama, and Yahiko Kambayashi. Database Queries as Combinatorial Optimization Problems. In *Cooperative Database Systems for Advanced Applications - CODAS 1996 - 1st International Symposium*, pages 477–483. World Scientific, 1996.

[103] A. Morgado and J. Marques-Silva. A Pseudo-Boolean Solution to the Maximum Quartet Consistency Problem. In *WCB08 - Workshop on Constraint Based Methods for Bioinformatics*, France, 2008.

[104] António Morgado and João Marques-Silva. Combinatorial Optimization Solutions for the Maximum Quartet Consistency Problem. *Fundamenta Informaticae*, 102(3-4):363–389, 2010.

[105] Matthias Müller-Hannemann and Stefan Schirra, editors. *Algorithm Engineering: Bridging the Gap between Algorithm Theory and Practice [outcome of a Dagstuhl Seminar]*, volume 5971 of *LNCS*. Springer, 2010.

[106] Alexander Nadel. Boosting minimal unsatisfiable core extraction. In *Formal Methods in Computer-Aided Design - FMCAD 2010 - 10th International Conference*, pages 221–229. IEEE, 2010.

[107] Alexander Nadel, Vadim Ryvchin, and Ofer Strichman. Efficient MUS extraction with resolution. In *Formal Methods in Computer-Aided Design - FMCAD 2013*, pages 197–200. IEEE, 2013.

[108] Nina Narodytska and Fahiem Bacchus. Maximum Satisfiability Using Core-Guided MaxSAT Resolution. In *Twenty-Eighth AAAI Conference on Artificial Intelligence*, pages 2717–2723. AAAI Press, 2014.

[109] Jaroslav Nesetril and Patrice Ossona de Mendez. *Sparsity - Graphs, Structures, and Algorithms*, volume 28 of *Algorithms and combinatorics*. Springer, 2012.

[110] Takao Nishizeki and Norishige Chiba. *Planar graphs : theory and algorithms*. Annals of discrete mathematics. North-Holland, 1988.

[111] Alexander Nöhrer, Armin Biere, and Alexander Egyed. Managing SAT inconsistencies with HUMUS. In *Variability Modelling of Software-Intensive - VAMOS 2012 - Sixth International Workshop*, pages 83–91. ACM, 2012.

[112] Yoonna Oh, Maher N. Mneimneh, Zaher S. Andraus, Karem A. Sakallah, and Igor L. Markov. AMUSE: a minimally-unsatisfiable subformula extractor. In *Design Automation Conference - DAC 2004*, pages 518–523. ACM, 2004.

[113] L.T. Ollmann. On the book thicknesses of various graphs. In *Combinatorics, Graph Theory, and Computing - CGTC 73 - 4th Southeaster Conference*, page 459. Utilitas Mathematica Publ. Inc, 1973.

[114] Shannon Overbay. Graphs with Small Book Thickness. *The Missouri Journal of Mathematical Sciences*, 19(2):121–130, 2007.

[115] David A. Plaisted and Steven Greenbaum. A Structure-Preserving Clause Form Translation. *Journal of Symbolic Computation*, 2(3):293–304, 1986.

[116] Alessandro Previti and João Marques-Silva. Partial MUS Enumeration. In *Twenty-Seventh AAAI Conference on Artificial Intelligence*. AAAI Press, 2013.

[117] Raymond Reiter. A Theory of Diagnosis from First Principles. *Artificial Intelligence*, 32(1):57–95, 1987.

[118] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, 1965.

[119] Sébastien Roch. A Short Proof that Phylogenetic Tree Reconstruction by Maximum Likelihood Is Hard. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 3(1):92–94, 2006.

[120] Arnold L. Rosenberg. The Diogenes Approach to Testable Fault-Tolerant Arrays of Processors. *IEEE Transactions on Computers*, 32(10):902–910, 1983.

[121] Peter Sanders. Algorithm Engineering - An Attempt at a Definition. In *Efficient Algorithms, Essays Dedicated to Kurt Mehlhorn on the Occasion of His 60th Birthday*, volume 5760 of *LNCS*, pages 321–340. Springer, 2009.

[122] Tian Sang, Fahiem Bacchus, Paul Beame, Henry A. Kautz, and Toniann Pitassi. Combining Component Caching and Clause Learning for Effective Model Counting. In *Theory and Applications of Satisfiability Testing - SAT 2004 - 7th International Conference*, 2004.

[123] Stefan Schlobach, Zhisheng Huang, Ronald Cornet, and Frank van Harmelen. Debugging incoherent terminologies. *Journal of Automated Reasoning*, 39(3):317–349, 2007.

[124] Charles Semple and Mike A. Steel. A characterization for a set of partial partitions to define an X-tree. *Discrete Mathematics*, 247(1-3):169–186, 2002.

[125] João P. Marques Silva. The Impact of Branching Heuristics in Propositional Satisfiability Algorithms. In *9th Portuguese Conference on Artificial Intelligence: Progress in Artificial Intelligence*, EPIA '99, pages 62–74. Springer, 1999.

[126] João P. Marques Silva. Minimal unsatisfiability: Models, algorithms and applications (invited paper). In *40th IEEE International Symposium on Multiple-Valued Logic - ISMVL 2010*, pages 9–14. IEEE Computer Society, 2010.

[127] João P. Marques Silva and Inês Lynce. On Improving MUS Extraction Algorithms. In *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th International Conference*, volume 6695 of *LNCS*, pages 159–173. Springer, 2011.

[128] João P. Marques Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In *DAC*, pages 675–680, 2000.

[129] Helmut Simonis. Sudoku as a Constraint Problem. In *The Fifth Workshop on Modeling and Reformulating Constraint Satisfaction Problems*, pages 13–27, 2006.

[130] Patrik Simons, Ilkka Niemelä, and Timo Soininen. Extending and implementing the stable model semantics. *Artificial Intelligence*, 138(1-2):181–234, 2002.

[131] Carsten Sinz. Towards an optimal CNF encoding of boolean cardinality constraints. In *Principles and Practice of Constraint Programming - CP 2005 - 11th International Conference*, volume 3709 of *LNCS*, pages 827–831. Springer, 2005.

[132] Carsten Sinz, Andreas Kaiser, and Wolfgang Küchlin. Formal methods for the validation of automotive product configuration data. *AI EDAM*, 17(1):75–97, 2003.

[133] Michael Steel. The complexity of reconstructing trees from qualitative characters and subtrees. *Journal of Classification*, 9(1):91–116, 1992.

[134] G. Stertenbrink and J.C. Meyrignac. 100 Sudoku problems (http://magictour.free.fr/top100), 2005.

[135] Harold S. Stone. *Introduction to computer organization and data structures.* McGraw-Hill computer science series. McGraw-Hill, New York, Maidenhead, 1972. Title page imprint: London.

[136] Yusuke Suzuki. Optimal 1-planar graphs which triangulate other surfaces. *Discrete Mathematics*, 310(1):6–11, 2010.

[137] Robert Endre Tarjan. Sorting Using Networks of Queues and Stacks. *Journal of the ACM*, 19(2):341–346, 1972.

[138] Emina Torlak, Felix Sheng-Ho Chang, and Daniel Jackson. Finding Minimal Unsatisfiable Cores of Declarative Specifications. In *Formal Methods - FM 2008 - 15th International Symposium*, volume 5014 of *LNCS*, pages 326–341. Springer, 2008.

[139] Emina Torlak, Mandana Vaziri, and Julian Dolby. Memsat: checking axiomatic specifications of memory models. In *Programming Language Design and Implementation - PLDI 2010*, pages 341–350. ACM, 2010.

[140] J.N.M. van Loon. Irreducibly inconsistent systems of linear inequalities. *European Journal of Operational Research*, 8(3):283 – 288, 1981.

[141] Hans van Maaren and Siert Wieringa. Finding Guaranteed MUSesFast. In Hans Kleine Bning and Xishun Zhao, editors, *Theory and Applications of Satisfiability Testing - SAT 2008 - 11th International Conference*, volume 4996 of *LNCS*, pages 291–304. Springer, 2008.

[142] Miroslav N. Velev. Using Rewriting Rules and Positive Equality to Formally Verify Wide-Issue Out-of-Order Microprocessors with a Reorder Buffer. In *Design, Automation and Test in Europe - DATE 2002 - Conference and Exposition*, pages 28–35. IEEE Computer Society, 2002.

[143] Miroslav N. Velev and Randal E. Bryant. Effective use of Boolean satisfiability procedures in the formal verification of superscalar and VLIW microprocessors. *Journal of Symbolic Computation*, 35(2):73–106, 2003.

[144] Miroslav N. Velev and Ping Gao. Efficient SAT Techniques for Relative Encoding of Permutations with Constraints. In *Advances in Artificial Intelligence - AI 2009 - 22nd Australasian Joint Conference*, volume 5866 of *LNCS*, pages 517–527. Springer, 2009.

[145] Wei Wei and Bart Selman. A New Approach to Model Counting. In *Theory and Applications of Satisfiability Testing - SAT 2005 - 8th International Conference*, volume 3569 of *LNCS*, pages 324–339. Springer, 2005.

[146] Siert Wieringa. Understanding, Improving and Parallelizing MUS Finding Using Model Rotation. In *Principles and Practice of Constraint Programming - CP 2012 - 18th International Conference*, volume 7514 of *LNCS*, pages 672–687. Springer, 2012.

[147] Avi Wigderson. The Complexity of the Hamiltonian Circuit Problem for Maximal Planar Graphs. Technical Report TR-298, EECS Department, Princeton University, 1982.

[148] Gang Wu, Ming-Yang Kao, Guohui Lin, and Jia-Huai You. Reconstructing phylogenies from noisy quartets in polynomial time with a high success probability. *Algorithms for Molecular Biology*, 3, 2008.

[149] Gang Wu, Jia-Huai You, and Guohui Lin. A Lookahead Branch-and-Bound Algorithm for the Maximum Quartet Consistency Problem. In *5th International Workshop Algorithms in Bioinformatics - WABI 2005*, volume 3692 of *LNCS*, pages 65–76. Springer, 2005.

[150] Gang Wu, Jia-Huai You, and Guohui Lin. Quartet-Based Phylogeny Reconstruction with Answer Set Programming. *IEEE/ACM Transactions on Computational Biology and Bioinformatics*, 4(1):139–152, 2007.

[151] Guohui Xiao and Yue Ma. Inconsistency measurement based on variables in minimal unsatisfiable subsets. In *20th European Conference on Artificial Intelligence - ECAI 2012*, volume 242 of *Frontiers in Artificial Intelligence and Applications*, pages 864–869. IOS Press, 2012.

[152] M. Yannakakis. Embedding planar graphs in four pages. *Journal of Computer and System Sciences*, 38(1):36–67, 1989.

Author's Publications

[153] Michael A. Bekos, Michael Kaufmann, and Christian Zielke. The Book Embedding Problem from a SAT-Solving Perspective. In *Graph Drawing - GD 2015 - 24th International Symposium*, to appear, 2015.

[154] Johannes Dellert, Christian Zielke, and Michael Kaufmann. MUStICCa: MUS Extraction with Interactive Choice of Candidates. In *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th International Conference*, volume 7962 of *LNCS*, pages 408–414. Springer, 2013.

[155] Stephan Kottler, Christian Zielke, Paul Seitz, and Michael Kaufmann. CoPAn: Exploring Recurring Patterns in Conflict Analysis of CDCL SAT Solvers - (Tool Presentation). In *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th International Conference*, volume 7317 of *LNCS*, pages 449–455. Springer, 2012.

[156] Christian Zielke and Michael Kaufmann. A New Approach to Partial MUS Enumeration. In *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference*, volume 9340 of *LNCS*, pages 387–404. Springer, 2015.

List of Figures

List of Tables

List of Algorithms