The present work was submitted to the LuFG Theory of Hybrid Systems

Bachelor of Science Thesis

Automated Exercise Generation for three Satisfiability checking algorithms

Greda Eshiba-Emir

Examiners:
Prof. Dr. Erika Ábrahám
Prof. Dr.-Ing. Ulrik Schroeder

Supervisor:
Prof. Dr. Erika Ábrahám

Aachen, 4.07.2022

Abstract

The great interest in solving the first ever NP-complete problem, i.e. the satisfiability problem, has brought many groundbreaking innovations. Among these innovations are SAT solvers which are tools that can be used to solve many real-life problems algorithmically. Since their invention, there has been a large body of researchers and other interested parties who have worked diligently on refining these tools even in the context of competitions. Although there is a large commitment in research to the satisfiability problem the subject matter of it is not so prominent in the university context hence there is little teaching material. In this bachelor thesis, we present our exercise generator, which introduces one of the most innovative techniques in modern SAT solvers namely conflict resolution. Furthermore, we discuss the question of what makes a "good" exercise and analyze our generated exercises under this aspect.

To give a small outlook on other innovations that have emerged in the context of the research area In satisfiability Checking, we look at other algorithms and tools and provide examples for exercises for satisfiability-modulo-theories solving (SMT) and cylindrical algebraic decomposition (CAD).

Erklärung

Hiermit versichere ich, dass ich die vorgelegte Arbeit selbstständig verfasst und noch nicht anderweitig zu Prüfungszwecken vorgelegt habe. Alle benutzten Quellen und Hilfsmittel sind angegeben, wörtliche und sinngemäße Zitate wurden als solche gekennzeichnet.

Greda Eshiba-Emir
Aachen, den 4.07.2022

Acknowledgements

I would like to start by thanking my supervisor Prof. Dr. Erika Ábrahám, not only for giving me the opportunity to write this bachelor thesis with her, but also for the countless support and advice she has given me throughout this process. Furthermore, I have also been very pleased with the uplifting and educational conversations during the supervision sessions. I would also like to thank Prof. Dr.-Ing. Ulrik Schroeder for agreeing on and taking the time to be my second examiner. Many thanks also to Prof. Dr. Erich Grädel for inspiring and encouraging me to do more in the field of theoretical computer science and for recommending me to extend my knowledge in the field with the lecture *Satisfiability Checking* after having attended his exciting lecture *Mathematical Logic*. Special thanks also go to my friends, especially to Linecker B. and Philipp P., who have given me valuable mental support during difficult stages of my bachelor. Last but not least, I would like to thank my family, my godmother Dorle, Thomas, Heide and also Jannis who have been patient with me during this whole journey and have given me the space and trust to grow and who have never given up on me.

The only way to walk
a journey of a thousand miles
is to take one step at a time.

BARBARA OAKLEY

Contents

Chapter 1

Introduction

In the fight against the COVID-19 pandemic drastic measures were taken that made major disruptions in people's lives. These measures included lockdowns and not least bans on large events and closures of major institutions, including universities.

For the latter, this meant that lectures and even exams had to be moved to the homes of students and lecturers, and online teaching had become a new standard. With this sudden change, many problems and challenges arose for both students and lecturers. While the absence of lecture halls for lectures was a sacrifice that was met with compromise, it was a problem in the case of exams. The lecturers now no longer had 100% control over what happened during the exam, since the exam writers were not gathered in one place to write the exam. It was therefore no longer completely comprehensible whether the students had worked on their exams fairly without secretly accessing aids that were not allowed. Even if the exams were proctored, there was not always a 100% guarantee that the entire exam situation could be properly monitored. This vulnerable situation offered scope for cheating of various kinds during the exam.

To counteract this there are only a few options that are not completely detrimental to the exam writers because they often contain restrictions. One possible approach, to prevent especially collaborative cheating and at the same time still maintaining a relatively high learning effect on the study side is to individualize exams from student to student. In order to be able to realize this approach of individualized tasks even for larger events with hundreds of students it is helpful to resort to an automated mechanism for this since doing it in the traditional way, which is by hand, is too time consuming and inefficient.

In this thesis we want to present such an automated mechanism for generating exercises, that can be used for exam preparation.

The context in which these exercises are to be generated is the lecture *Satisfiability Checking* held by Prof. Dr. Erika Ábrahám, which is offered as an elective course for computer science students at the RWTH Aachen University. In the course of this lecture the students will learn about the *satisfiability problem (SAT)* and are familiarized with decision procedures that can solve it and their implementation in tools which are termed *SAT solvers*. To get a better gist of the functioning of SAT solvers the students are given several tasks where they have to simulate how SAT solvers decide instances of the satisfiability problem with pen and paper.

Our implementation, which is written entirely in the programming language Java, will include the implementation of the DPLL+CDCL SAT solving algorithm for checking the satisfiability of propositional logic formulas. In addition we will also present exercises for *less lazy satisfiability-modulo-theories (SMT) solving for equality logic* and *the real root isolation* in the *cylindrical algebraic decomposition (CAD)* to solve real arithmetic problems, which also are taught in this lecture, but for which we do not provide an implementation.

This thesis will be structured as follows: we first clarify the context in which the exercises, that we generate with our exercise generator can be embedded in the preliminary Chapter 2. Then we give a short description of the algorithm addressed in the exercises entering Chapter 3. This is then followed by an analysis of the quality characteristics and discussion of alternative design options for the exercises. We then present an example of the generated exercises, which when freshly generated will be available for the students in the form of the TeX data format and can be parsed to a PDF file, e.g. by using PDFLATEX. For the *theory solver for less lazy satisfiability-modulo-theories (SMT) for equality logic* and the *real root isolation* in the context of the *cylindrical algebraic decomposition (CAD)*, we do not give a discussion, since we only focus on the SAT solver in our implementation, which is why the rest should only serve as an outlook for the time being.

For the implementation of our exercise generator we will only provide a description in this paper but the original JAVA source code can be accessed via the following link `https://github.com/Greda96/SATcheckExercises`

Chapter 2

Preliminaries

In this chapter we want to lay the foundation for the concepts that will be used in the subsequent chapters. At the same time, we want to provide the context in which the satisfiability checking algorithms incorporated in the exercises we want to create can be embedded. In doing so, we enter the terrain of *satisfiability checking* and clarify central terminologies such as the *satisfiability problem*, then we also introduce the *DPLL* algorithm by giving an insight into the historical background and explaining the functioning of the algorithm. We will also examine the main driving force of state-of-the-art solvers namely *Conflict-Driven Clause Learning* and its combination with the DPLL SAT solver. Furthermore, we learn about *satisfiability-modulo-theories (SMT) solving* while focussing on the *lazy* approach. We will then conclude the chapter with the *cylindrical algebraic decomposition (CAD)* and briefly explain how it works while focusing on one particular part of the *projecting phase*, which is the *real root isolation* by also providing an example exercise.

2.1 Fundamentals & Notation

For the further course of this reading, we assume a general knowledge of propositional logic and the fundamentals of mathematics. However at the beginning, we want to recapitulate some terms that will recur frequently in this work, not least also to establish an agreement for the notation used throughout this thesis. For the latter and also for Definitions, unless otherwise specified, we will mainly rely on [Ábr22] and [KS08].

Definition 2.1.1 (Literal, set of variables, set of literals)**.** *A literal is either a variable x or its negation $\neg x$. Let φ be a propositional logic formula then we denote the set of variables contained in φ as $AP(\varphi)$ and the set of literals of φ as $lit(\varphi)$.*

Definition 2.1.2 (Clause, Conjunctive normal form (CNF))**.** *A formula is in conjunctive normal form (CNF) if and only if it is a conjuntion of clauses, where a clause is a disjunction of literals. More precisely, for a formula to be in CNF it must be of the form*

$$\bigwedge_i \left(\bigvee_j l_{ij} \right),$$

where l_{ij} represents the j-th literal of the i-th clause.

Example 2.1.1. *Let $\varphi := (\neg a \vee b \vee \neg c) \wedge (a \vee d) \wedge (\neg c)$. Then φ is a propositional logic formula in CNF consisting of the three clauses $(\neg a \vee b \vee \neg c)$, $(a \vee d)$ and $(\neg c)$. Furthermore the set of variables and literals of φ are respectively given by $AP(\varphi) = \{a, b, c, d\}$ and $lit(\varphi) = \{\neg a, a, b, \neg c, d\}$.*

Definition 2.1.3 (Assignment)**.** *An assignment for a propositional logic formula φ is a potentially partial function $\alpha : AP \rightharpoonup \{false, true\}$ (where AP denotes the set of propositions or boolean variables). It can be convenient to write α as the set of the literals that are mapped to true. An assignment α is said to be a full assignment for φ if all variables of φ are assigned a truth value by α, otherwise we call α a partial assignment.*

Definition 2.1.4 (Satisfiability, contradiction, validity)**.** *Given a formula φ, φ is satisfiable if there exists an assignment that evaluates φ to true. An satisfying assignment is also called model. If such an assignment does not exist φ is unsatisfiable, in this case we refer to φ as a contradiction. If φ is satisfied under all possible assignments we say φ is valid (or a tautology). We write $\alpha \models \varphi$ to denote that α satisifies φ and $\alpha \not\models \varphi$, if α does not satisify φ. If φ is a tautology we shortly write $\models \varphi$.*

Example 2.1.2. *Let φ be the formula presented in Example 2.1.1, then φ is satisfiable, because the following assignment $\alpha : \{a,b,c,d\} \rightarrow \{false,true\}$ with $\alpha(c) = false$ and $\alpha(a) = \alpha(b) = \alpha(d) = true$, satisfies φ. The formula φ is not valid because every assignment $\alpha : \{a,b,c,d\} \rightarrow \{false,true\}$ with $\alpha(c) = true$ evaluates φ to false.*

Definition 2.1.5 (Resolution, resolvent)**.** *Assume literals $a_1, \ldots, a_n$, $b_1 \ldots, b_m$ and a variable x, then we call the following inference rule resolution:*

$$\frac{(a_1 \vee \ldots \vee a_n \vee x) \qquad (b_1 \vee \ldots \vee b_m \vee \neg x)}{(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)} \text{ (Resolution)}$$

Furthermore we call $(a_1 \vee \ldots \vee a_n \vee b_1 \vee \ldots \vee b_m)$, the result of the resolution of $(a_1 \vee \ldots \vee a_n \vee x)$ with $(b_1 \vee \ldots \vee b_m \vee \neg x)$, the resolvent of both clauses.

Example 2.1.3. *Resolution involving the two clauses $c_0 := (\neg a \vee b \vee \neg c)$ and $c_1 := (a \vee d)$ gained from our CNF formula in Example 2.1.1 yields*

$$\frac{(\neg a \vee b \vee \neg c) \qquad (a \vee d)}{(b \vee \neg c \vee d)}$$

where $(b \vee \neg c \vee d)$ is the resolvent of c_0 and c_1.

2.2 Satisfiability Checking

2.2.1 The satisfiability problem

The first ever NP-complete problem, the *satisfiability problem*, also known as *SAT problem* or just *SAT* [Coo71, ST12], poses the question whether a given propositional logic formula is satisfiable.

A problem instance of the SAT problem, also called *SAT instance* is usually represented by a CNF formula (see Definition 2.1.2). This choice of representation for the SAT instances is made for efficiency and simplicity reasons [DP60, FM09]. It should be noted that this is not a restriction, because any propositional logic formula can be transformed into a logically equivalent formula in CNF [KS08]. An algorithm that performs a satisfiability equivalent transformation in polynomial time, and is therefore efficient is Tseitin's encoding [Tse83].

2.2.2 SAT solvers

To check the satisfiability of a SAT instance a so-called *SAT solver* can be used. A SAT solver implements an algorithm which receives a SAT instance as input and tries to solve it by assigning suitable truth values to its variables which together should satisfy the formula. If the SAT solver determines that the SAT instance is satisfiable, it returns a satisfying assignment as output, otherwise it returns that the instance is unsatisfiable (see Definition 2.1.4).

SAT solvers have reached a level of development that allows us to check large SAT instances with up to millions of variables for satisfiability. This development has been driven by the desire to solve real-life problems algorithmically, which has led to a lot of attention being paid to research on the satisfiability problem [KS08, ST12]. Since then, the areas of application of SAT solvers have included cryptography, bioinformatics, planning, and digital circuit design, just to mention a few [MSLM21, Ábr22].

In general, contemporary SAT solvers can be divided into two classes. On the one hand SAT solvers which are organized according to the *DPLL* paradigm and on the other hand SAT solvers which are built according to the *stochastic search* approach [KS08].

In our work we will focus on the first class, since the SAT solver we implemented for the exercise generator is based on the DPLL SAT solver. The following subsection is therefore dedicated to the DPLL paradigm.

2.2.3 The DPLL paradigm

Algorithm 1 The DPLL algorithm

```
function DPLL(CNF formula $\varphi$)
    if $\neg$BCP() then
        return UNSAT
    end if
    while true do
        if $\neg$DECIDE() then
            return SAT
        end if
        while $\neg$BCP() do
            if $\neg$BACKTRACK() then
                return UNSAT
            end if
        end while
    end while
end function
```

Historical background

The theory underpinning modern SAT solvers dates back to the 1960s. Since computers did not have so much computing power at that time, one had to fight with scalability issues at the beginning when trying to solve the SAT problem. But also the algorithm which was applied at the early stage was quite naive, mainly enumerating all possible assignments for a problem instance. This corresponds to setting up a truth table which is known to grow exponentially with the number of variables, with a formula with n variables having a size of 2^n in the worst case [FM09].

To increase efficiency, the computer scientist Martin Davis and the philosopher Hilary Putnam came up with the idea of using CNF formulas for SAT solving, and at that time they proposed rules that are now fundamental mechanisms for modern SAT solvers [DP60]. These rules included the *unit clause rule* which we will explain in more detail in Section 2.2.3. Also, the algorithm of Davis and Putnam, *DPP* for short, already included resolution (see Definition 2.1.5) as a subroutine.

Later the computer scientists Donald Loveland and George Logeman refined the DPP algorithm by choosing a recursive approach where they assigned the variables of the formula the truth values *true* or *false* which led to the problem being divided into subproblems [DLL62].

This approach is equivalent to traversing a binary search tree where the nodes represent the variables and where assignments are made along the branches of the tree which are then understood as partial assignments. These assignments are initially only guesses, since the truth values *true* or *false* are chosen randomly for a variable, which is why this assignment of values in this manner is also called a *decision.* With each such decision, a level of the tree at which it was made will associate what is called a *decision level.* After each decision, *propagation* is used as a lightweight method to detect implications, i.e. consequences of the previous assignment with the new decision in form of further variable assignments. While recursively assigning the values *true* or *false* along the branches, and propagating their decisions a *conflict* can occur, this is when the current assignment evaluates the formula to *false*. To resolve this conflict the last decision has to be flipped, such that the alternative guess is made. If both truth values were tried unsuccessfully on the variables, one goes back to the previous decision. This procedure is referred to as *chronological backtracking*.

This extension of the DPP algorithm, which is presented as pseudocode in Algorithm 1, is known ever since as the classical DPLL algorithm, where the acronym "DPLL" stands for *Davis-Putnam-Logeman-Loveland* [FM09].

Main characteristics of DPLL

After we have made an excursion into the history of the origin of the DPLL algorithm, we want to go into more detail and elaborate on the functioning of the algorithm by describing its main methods. But before we do so, we will take a look at the following definition.

Definition 2.2.1 (Status of a clause under a partial assignment)**.** *Given a partial assignment and a clause c, then c can have the following statuses:*

- *Satisfied, if at least one of the literals in c is assigned the value true.*
- *Unsatisfied or conflicting, if all literals in c are assigned the value false.*
- *Unit, if all literals in c but one, which is unassigned, are assigned the value false.*

If c does not have any of the above statuses, we say c is unresolved.

Example 2.2.1. *Let* $\alpha = \{a, \neg b, d\}$ *be a partial assignment. Then the following holds:*

- $(a \vee b \vee d)$ *is satisfied,*
- $(\neg a \vee \neg d)$ *is conflicting,*
- $(\neg a \vee b \vee c)$ *is unit and*
- $(c \vee e)$ *is unresolved.*

Now we can turn to the functioning of Algorithm 1. The DECIDE() method chooses an unassigned variable and assigns it a truth value. As we have already mentioned in the preceding subsection this is also called a decision. In order to determine which of the unassigned variables to choose next for a decision and which value to assign to it, a *decision heuristic* can be established, which represents an ordering on the unassigned variables. After having made a decision we internally count these, and increment the decision counter everytime we make another decision. If DECIDE() returns *false*, which is when all variables are assigned we return SAT, which means that the formula is satisfied. With the BCP() method, also called *boolean constraint propagation* or *unit propagation* the unit clause rule is iteratively being applied [MSLM21]. The unit clause rule states that, if we find a unit clause we need to set the only unassigned variable in it to *true* in order to be able to satisfy the clause. We say that this literal is propagated because it is an implied assignment. We call BCP() in the beginning of the algorithm, because of the fact that a formula in its original form, before being processed in the SAT solver, can already contain unit clauses, in this context meaning clauses that contain only one literal. An example for such a formula is the formula shown in Example 2.1.1 which contains the unit clause $(\neg c)$.

If we find unit clauses in the very beginning the decision level associated with this propagation is the *ground level* or in other words decision level 0 [KS08, MSLM21]. Furthermore we call the BCP() method after each decision, since an assignment made by DECIDE() can trigger a clause to become unit (see Definition 2.2.1) and therefore imply a forced assignment. If the BCP() method reports a conflict, which is only the case when an unsatisfied clause is detected, and we are not at the ground level, the BACKTRACK() method is invoked. In the BACKTRACK() method we set the decision level to the last decision level and erase the assignment at the current branch by undoing it. After that we flip the assignment that is not yet being flipped and call BCP() again. If BCP() detects a conflict at decision level 0, we return UNSAT which means the formula is unsatisfiable.

In the following we introduce a notation that gives us a description of the state of a literal that is being assigned during the SAT solving process [Ábr22].

Notation 2.2.1 (Antecedent). *For a literal l that is being assigned by a (partial) assigment α we write antecedent(l) to indicate the reason for the assignment of l. The antecedent of l can either be a clause c which implied the assignment of l due to the unit-clause rule, in this case we will write antecedent$(l) = c$, or if the assignment of l is owed to a decision we write antecedent$(l) = nil$.*

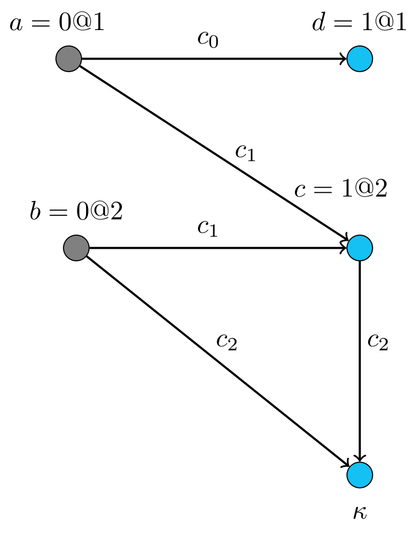
One way to visualize a partial assignment and to get a structural overview of the repeated use of the unit-clause rule during SAT solving is to use an *implication graph* [Ábr22, MSLM21, ST12].

Definition 2.2.2 (Implication graph). *An implication graph is a labeled directed acyclic graph $G = (V, E, L)$, where V is the set of nodes, which contains the currently assigned variables as well as an additional node κ, for the case we detect a conflicting clause c_{confl}. L is a labeling function that assigns a label to each node. A node n that represents that a variable x is assigned a value $v \in \{0,1\}$ (0 denoting false and 1 denoting true) at a decision level d, is labeled with $L(n) = (x = v@d)$. We define literal$(n) = x$ if $v = 1$ and literal$(n) = \neg x$ if $v = 0$. Furthermore the edge relation E is defined as*

$$\begin{aligned} E = \{(n_i, n_j) \mid n_i, n_j \in V, n_i \neq \mathit{literal}(n_i) \in \mathit{Antecedent}(\mathit{literal}(n_j))\} \\ \cup \{(n, \kappa) \mid n, \kappa, \neg \mathit{literal}(n) \in c_{confl}\} \end{aligned}$$

representing the set of directed edges where each edge (n_i, n_j) is labeled with Antecedent$(\mathit{literal}(n_j))$ if $n_j \neq \kappa$ and with c_{confl} otherwise.

Example 2.2.2. *Consider the CNF formula* $\varphi := \underbrace{(a \vee d)}_{c_0} \wedge \underbrace{(a \vee b \vee c)}_{c_1} \wedge \underbrace{(b \vee \neg c)}_{c_2}$ *and assume further the following decisions* $\{a = 0@1, b = 0@2\}$*. The resulting implication graph is now given by*



the implication graph shows that a conflict occurred at decision level 2 *because the clause* c_2 *became conflicting under the current assignment.*

2.2.4 Conflict-Driven Clause Learning

Based upon the DPLL SAT solver *Conflict-Driven Clause Learning SAT solvers* were invented. These SAT solvers extend the original DPLL SAT solver by many new techniques including *clause learning* and *non-chronological backtracking* which allows for speeding up the search process [MSLM21, ST12]. The two techniques mentioned above are combined in a subroutine of the SAT solving process called *conflict resolution*, to which we will devote a separate section in this thesis. In the previous section we have seen a representation of a partial assignment that was created during the SAT solving process as an acyclic graph, another way to represent a partial assignment that is used in CDCL SAT solvers is the so called *trail*. A trail is a stack in which the assignments of literals that are assigned the value *true* during the SAT solving process are stored. Beside the literals also the reason of the assignment, i.e. the antecedent (see Notation 2.2.1), is append to the trail. If a literal is taken from the trail then this corresponds to undoing the assignment of *true* to this literal [Knu06, HB20, GV20]. Another important concept that makes CDCL SAT solvers so efficient are *unique implication points*. Assuming an implication graph with a conflict node κ a unique implication point, or *UIP* for short, is a node in the implication graph other than κ for which all paths from the decision node at the last decision level to the conflict node, pass through it (the decision node is by definition also a UIP). A UIP represents an alternative assignment to the most recent decision level that would cause the same conflict. We are talking about a first Unique implication point when we reach the first UIP from the viewpoint of the conflict node. If we consider the implication graph from Example 2.2.2 then $a = 0@1$, $b = 0@2$ since they are decision nodes are UIPs. However, the implication graph does not contain a first unique implication point, since not every path starting from the decison node $b = 0@2$ at the current decision level goes through one and the same node [MSLM21]. The advantage of UIPs is that with them we learn small clauses [MSLM21], which we will explain in more detail in the next section.

2.2.5 Conflict resolution

As mentioned previously conflict resolution is part of the CDCL algorithm and is used when a conflict occurs during SAT solving. A conflict occurs when a clause under the current assignment is evaluated to *false* but must be *true* for the entire formula to become *true*. To resolve the conflict, the resolution method is used. The resolution involves the conflicting clause (see Definition 2.2.1) and a *conflict clause*. The conflict clause is the antecedent of the literal, that was last assigned before the conflict occured. We then perform resolution on the two clauses, eliminating the variable corresponding to that literal. The goal of the resolution is to generate an *asserting clause*, i.e. a clause that contains only one variable from the current decision level. If the resolvent does not yet yield an asserting clause, it is processed in a further resolution step. Just like in the initial resolution step we now combine the resolvent, which is also a conflict clause, with the clause that implied the literal that was assigned last in it. This process is then repeated until the termination criterion is met, i.e., when the generated conflicting clause is asserting [KS08]. Reaching the asserting clause, which is now a unit clause, corresponds to what is called to having "learnt" a clause, this clause is then added to our original formula which we know now the asserting clause is implied by [ST12, MSLM21]. In terms of the implication graph we are located at the first unique implication point when learning the asserting clause [MSLM21]. After the last resolution step, the decision level we need to backtrack to is determined. In the state-of-the-art CDCL solvers in the case of the current decision level being larger than 0, we backtrack to the second highest decision level *in* the asserting clause, i.e. to a decision level other than the current decision level that contains at least one of the variables in the asserting clause [KS08, ST12]. So we essentially backtrack "relative" to the asserting clause, and eventually need to skip decision levels, this kind of backtracking is referred to as *non-chronological backtracking* [KS08]. Doing conflict resolution until reaching the first unique implication point assures us that we backtrack far enough in the search tree [MSLM21]. If we have determined the level we need to backtrack to we erase all decision levels that follow that backtracking level. In the case we detect a conflict at decision level 0 the SAT solver returns *false*. After backtracking we now need to propagate the learnt clause [MSLM21, Ábr22, ST12]

2.3 Satisfiability-Modulo-Theories Solving

Seeing the success of SAT solvers which have the power to decide real-life problems encoded in propositional logic, and having the need to solve ever more critical problems, researchers sought the opportunity to extend to more expressive logics, and this is when the research field of *satisfiability-modulo-theories solving*, *SMT solving* for short, was born. The logics SMT solving comprises are for example *linear* and *non-linear arithmetic*, *bit-vector arithmetic* and the theory of equalities and uninterpreted functions and many more [ÁK17]. In this paper we will take a closer look at SMT solving in the context of *theory of equalities and uninterpreted functions*. Just like in

SAT solving, also in SMT solving we try to decide the satisfiability problem, but with the crucial difference that the formulas we are checking are first-order logic formulas over some theories.

Being an extension of propositional logic by theories, the formulas in satisfiability-modulo theories also look structurally different and now also statements about equations and inequalities can be made, which can also be combined with the logical operators, which we already know from propositional logic. Unlike propositional logic formulas the variables in the theories, which can now come from structures (for example $\mathbb{R}$) in , can be quantified, i.e. the existence quantifier ($\exists$) and the all quantifier ($\forall$) appear in them. Linear arithmetic formulas even allow for function operators like multiplication, and also comparison predicates like $>$ (to be interpreted here as the usual "larger than"). An example of such a formula would be
$4 \cdot x^3 + 7 \cdot x^2 \leq 0 \land 20 \cdot x + 22 > 50 \, \text{with} \, x \in \mathbb{R}$. Equality logic extends propositional logic with equalities and can introduce *uninterpreted functions* which we will examine in the next section. We call equalities and inequalities that are combined with boolean operators *constraints* [ÁK17].

These extensions make the language of the theories so expressive. With this expressiveness we can now decide even more complex problems, because we are able to encode even more [BKM14].

SMT solving can be divided into two branches in terms of the approach to the solving process, namely *Eager SMT solving* and *Lazy SMT solving* [ÁK17]. In the following we want to concentrate on the workings in *Lazy SMT solving*, which is said to be the "dominating" approach in SMT solving according to [Seb07] and [ÁK17].

2.3.1 Lazy SMT solving

Lazy SMT solving can also be subdivided into two different branches: *less* and *full lazy SMT solving* [ÁK17], however, we will lay our focus on less lazy SMT solving. But before we can do that we first want to define the theory for which we want to decide the formulas. Since we want to create exercises for less lazy SMT solving for *equality logic with uninterpreted* functions we explain what uninterpreted functions are and then define the syntax and semantic of equality logic with uninterpreted functions.

Uninterpreted functions (UF)

Uninterpreted functions are, as the name suggests, function symbols that are not interpreted, they have no meaning in the sense of semantics for mathematical functions and are therefore not interpreted as part of a model of a formula [KS08]. Uninterpreted functions must however fulfill a rule they must be *consistent*, i.e. for two input values x,y with $x = y$ and an uninterpreted function F must hold $x = y \rightarrow F(x) = F(y)$. This rule is known as *functional congruence* [KS08, Ábr22]. Roughly speaking, one uses uninterpreted Functions to reduce a problem to a smaller problem by abstracting a part of it, in this case the functions, (making them a "black box" so to speak), which one hopes is irrelevant [ÁK17]. Quite often the meaning of functions in proofs is ignored to simplify the proof, so uninterpreted functions have a practical use, but it should be emphasized that they make formulas weaker, a consequence of this is that a formula that is a tautology is suddenly no longer a tautology after replacing the functions in it with uninterpreted functions. To check whether a formula with uninterpreted functions is a tautology a reduction technique called *Ackermann's reduction* is used. We will not discuss this here, but we refer to [KS08] for a more detailed explanation of the reduction and other special properties of uninterpreted functions.

In the following we want to combine uninterpreted functions with the theory equality logic.

Definition 2.3.1 (Equality logic with uninterpreted functions (EQ+UF))**.**
Syntax: *The signature contains*

- *variables x over an arbitrary (but "large enough") domain D,*
- *constants c from the same domain D,*
- *function symbols F for functions of the type $D^n \to D$,*
- *equality as predicate symbol*

$$\begin{array}{lllllll} Term & t ::= & c & | & x & | & F(t, \ldots, t) \\ Formulas & \varphi ::= & t = t & | & (\varphi \land \varphi) & | & (\neg\varphi) \end{array}$$

Semantics
as defined for first-order logic (note the uninterpreted nature of functions)

2.3.2 Less lazy SMT solving for EQ+UF

In less lazy SMT solving, the interaction of an SAT solver and at least one theory solver takes place within the SMT solving process. A theory solver is practically the pendant to the SAT solver for propositional logic, the only difference being that the theory solver decides problem instances from a theory, i.e. in the case of equality logic with uninterpreted functions a theory solver, which we will refer to here as EQ+UF-theory solver, solves equalities of uninterpreted functions [Ábr22, ÁK17, Seb07]. In a preprocessing step, an input formula φ^{EQ+UF} is converted into a readable form for the SAT solver. This encoding is done by replacing inequalities with fresh propositional logic variables. We also call this abstracted formula *boolean abstraction* or *boolean skeleton* and write φ_{abs}^{EQ+UF} in this context. Now that φ^{EQ+UF} has been put into a form readable by the SAT solver, it is fed into the SAT solver to decide it. If φ_{abs}^{EQ+UF} is satisfiable, then the SAT solver returns SAT as in the usual case, returning a fulfilling assignment as a solution. Viewed in isolation, any assignment of a variable in the boolean abstraction with true means that the associated constraint is addressed that is underlying it, an assignment with false means the negation of this equality. After the SAT solving process is completed, the less lazy EQ+UF theory solver is consulted, which then checks the consistency of these inequalities, i.e. whether they are non-contradictory in the theory. In the case of equalities of uninterpreted functions, we check whether function congruence is guaranteed. If the less lazy EQ+UF-theory solver determines that there is no conflict, then it asserts the satisfiability of our input formula φ^{EQ+UF}. If the less lazy EQ+UF theory solver finds a conflict then it generates an explanation for this conflict. It sets up a so-called *minimal infeasible subset.* From this minimal infeasible subset, we can then read the conflicting clause by assembling the boolean variable corresponding to each inequality contained in the infeasible subset, into a clause, which then forms our conflicting clause. As with SAT solving, we now make a conflict resolution to resolve this conflict and finally add the asserting clause to our Boolean abstraction [Ábr22].

2.4 Cylindrical Algebraic Decomposition

When talking about *cylindrical algebraic decomposition (CAD)*, two things can be associated with it. On the one hand the mathematical object, i.e. the decomposition of the n-dimensional real space $\mathbb{R}^n$ in the sense of the properties *cylindrical* and *(semi)-algebraic* (see Definition 2.4.3 and Definition 2.4.2), on the other hand the algorithm for the construction of such a decomposition [Kre20]. Both the mathematical object and the algorithm can be traced back to the mathematician and computer scientist George E. Collins [Col75]. Inspired by the work of the mathematician Alfred Tarski, who tried to develop a decision procedure to perform quantifier elimination over real closed fields, but which was not practicable due to its too poor runtime and therefore remained only as a theoretical relic, Collins tried to work on a more effective solution [Tar98, Kre20]. The CAD method originates as a sub algorithm of this alternative solution by Collins [Jir95]. Although the CAD algorithm had its original use in real closed fields, it could later be applied in the field of algebraic geometry. Because the formulas over real closed fields were also well suited to solve problems in algebraic geometry, the use of the CAD method, which is known to be complete, turned out to be very lucrative for this area of mathematics. According to [Kau10] Collin's quantifier elimination method can even be regarded as a *"general tool"* to solve problems related to special subsets of $\mathbb{R}^n$, namely *semi-algebraic* sets which are also called *cells* (see Definition 2.4.1). The CAD algorithm is able to put the constructions of these cells into a standard form, which as a side effect opens the possibility of solving further, more complex, problems [Kau10].

Definition 2.4.1 (Decomposition of $\mathbb{R}^n$). *A decomposition of $\mathbb{R}^n (n \geq 1)$ is a finite set $\mathcal{C}$ of pairwise disjoint, regions (i.e. non-empty connected subsets of $\mathbb{R}^n$) in $\mathbb{R}^n$ with $\bigcup_{C \in \mathcal{C}} = \mathbb{R}^n$. We call $\mathcal{C}$ a cell.*

Definition 2.4.2 (Semi-algebraic). *A decomposition $\mathcal{C}$ of $\mathbb{R}^n$ is semi-algebraic if each $C \in \mathcal{C}$ can be constructed by a finite union, intersection and complementation of solution sets of polynomial constraints $p \sim 0$ where $p \in \mathbb{Q}[x_1, \ldots, x_n]$ and $\sim \in \{\leq, <, =, \neq, >, \geq\}$.*

Definition 2.4.3 (Cylindrical). *A decomposition $\mathcal{C}$ of $\mathbb{R}^n$ is cylindrical if either $n = 1$ or the set of projections of the regions in $\mathcal{C}$ to the first $n-1$ dimensions is a cylindrical decomposition of $\mathbb{R}^{n-1}$.*

A decomposition of $\mathbb{R}^n$ is now called a cylindrical algebraic decomposition if it has the characteristics described in Definition 2.4.3 and Definition 2.4.2. A CAD for an input set $P = \{p_1, \ldots, p_m\} \subset \mathbb{Q}[x_1, \ldots, x_n]$ of polynomials is now a CAD of $\mathbb{R}^n$ where the polynomials of each cell have a constant sign, i.e. either positive, negative or zero. [Ábr22, CRE18].

2.4.1 Cylindrical algebraic decomposition method

The CAD construction is done in two phases, the *projection* and the *lifting phase.* In the projection phase, we iteratively apply a projection operator to our set of polynomials, starting with $P_n = \{p_1, \ldots, p_n\} \subset \mathbb{Q}[x_1, \ldots, x_n]$. At each successive iteration in the projection phase, we then start with a polynomial set that emerges from the previous one reduced by one dimension through elimination of a variable. This kind of working, using the previous result for the computation in the next iteration is also called *incremental* [KÁ19]. We continue the projection until we end up with the set $P_1 \subset \mathbb{Q}[x_1]$ containing only univariate polynomials. At this point it should be mentioned that it is from that projection phase that the major flaw of CAD stems. The repeated projection until reaching P_1 leads to an asymptotic complexity which is double exponential in the number of variables [Kre20, KS08]. The lifting phase is then initiated by generating a CAD of $\mathbb{R}$, i.e. for each $p \in P_1$. The exact explanation of the construction of the CAD for the univariate case, where the so-called *real root isolation* is used, will be postponed to the next section. The CAD created by the real root isolation consists of the real roots of each $p \in P_1$ and the open intervals between them. Proceeding from there the CAD for $\mathbb{R}$ is used to build the CAD of $\mathbb{R}^n$, which we do not discuss in detail in this thesis, because our main focus is on the construction of the CAD for $\mathbb{R}$ in terms of the real root isolation. For the interested reader we refer for example to [Jir95] for further reading on this topic.

2.4.2 Real root isolation

In the following we want to examine the construction of the CAD for univariate polynomials using the real root isolation. But first we need to define our tool set.

Theorem 2.4.1 (Sturm's theorem)**.** *Assume a square-free (no square factors, i.e., no repeated roots) univariate polynomial*

$$p = a_k x^k + a_{k-1}x^{k-1} + \ldots + a_1 x + a_0 \in \mathbb{Q}[x]$$

with $lc(p) \neq 0$. For the Sturm sequence $p_0, p_1, \ldots, p_l$ with:

- $p_0 = p$,
- $p_1 = p'$ *(where p' is the derivative of p),*
- $p_i = -\mathsf{rem}(p_{i-2}, p_{i-1})$ *for $i = 2, \ldots, l$ (where* rem *is the remainder of the polynomial division of p_{i-2} by p_{i-1})*
- $\mathsf{rem}(p_{l-1}, p_l) = 0$

Let $\sigma(\zeta)$ denote the number of sign changes (ignoring zeroes) in the sequence $p_0(\zeta), p_1(\zeta), p_2(\zeta), \ldots, p_l(\zeta)$. Then for each $a,b \in \mathbb{R}$ with $a < b$ the number of distinct real roots of p in $(a,b]$ is $\sigma(a) - \sigma(b)$.

Definition 2.4.4 (Cauchy bound)**.** *Given a univariate polynomial*

$$p = a_k x^k + a_{k-1}x^{k-1} + \ldots + a_1 x + a_0 \in \mathbb{Q}[x]$$

with $lc(p) \neq 0$. If $\zeta \in \mathbb{R}$ is a (real) root of p (i.e. $p(\zeta) = 0$) then

$$|\zeta| \leq \underbrace{1 + \max_{i=0,\ldots,k-1} \frac{|a_i|}{|a_k|}}_{:=C},$$

where C is called Cauchy bound.

Example 2.4.1. *Let $p = 4x^3 - 7x^2 + 20x + 22$. Then $lc(p) = a_3 = 4, a_2 = -7, a_1 = 20$ and $a_0 = 22$. The Cauchy bound of p is then given by*

$$C = 1 + \max\left\{\frac{|-7|}{|4|}, \frac{|20|}{|4|}, \frac{|22|}{|4|}\right\} = 1 + 5.5 = 6.5.$$

Furthermore we agree on the following notation from [Ábr22].

Notation 2.4.1 (Interval representation)**.** *An interval representation (of a real root) is a pair (p,I) of a univariate polynomial p with real coefficients and an open interval $I = (l,r) \subseteq \mathbb{R}$, $l,r \in \mathbb{Q} \cup \{-\infty, \infty\}$ such that I contains exactly one real root of p.*

$$(\underbrace{p}_{\in\mathbb{Q}[x]}, \underbrace{(l, r)}_{\textit{exactly one real root of } p \textit{ in the interval } (l,r)})$$

We now assume a set $P = \{p_1 \sim_1, \ldots, p_k \sim_k\}$ of univariate polynomial constraints where $p_i \in \mathbb{Q}[x_1]$ and $\sim_i \in \{\leq, <, =, \neq, >, \geq,\}$ for $1 \leq i \leq k$, that is freshly produced in the projection phase of CAD.

The interval $I = [-C,C]$ with Cauchy bounds (where C is the Cauchy bound of p_i contained in P) contains all real roots of the polynomials $p_1, \ldots, p_k$.

We start by splitting I into three sub-intervals $[-C, -C]$,$(-C,C)$,$[C,C]$, which now form our working set $I' := \{[-C,-C],(-C,C),[C,C]\}$. Then we successively pick and remove one interval from I' and count the real roots of p_i contained in the chosen interval using the Sturm sequence of p_i as described in Theorem 2.4.1.

It should be reminded that Sturm's theorem is defined for left open intervals only, but we can still count real roots of a polynomial in an right open interval by considering the following: for an interval (a,b), with $a,b \in \mathbb{R}$, if $p_i(b) > 0$ then the number of real roots contained in (a,b) equals the number of real roots contained in $(a,b]$. If $p_i(b) = 0$ then the number of real roots is $\sigma(a) - \sigma(b) - 1$, i.e. the number of real roots of p_i in $(a,b]$ reduced by 1.

For point intervals $[a,a]$ in I' we need to check if $p_i(a) = 0$, if this is true then we got a real root which is a. If for a polynomial p_i the number of real roots in an interval (a,b) is larger than 1, or if the interval contains two roots from two different polynomials, we choose m with $a < m < b$ and split that interval into three sub-intervals $(a,m), (m,m), (m,b)$ and set $I' = I' \cup \{(a,m), (m,m), (m,b)\}$.

If we end up with exactly one real root from p_i that is contained in an interval (a,b) we remember that real root, which we denote as $(p_i, (a,b))$ according to Notation 2.4.1. Note that for point intervals $[a,a]$ there is no real root representation, we simply write $[a,a]$, since it already contains the exact representation of the real root which we can explicitly designate as a. Otherwise we proceed counting the real roots contained in an interval from I' until $I' = \emptyset$, which implies that all real roots are isolated [Ábr22].

Chapter 3

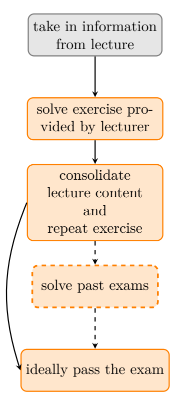
Automated Exercise Generation

Having laid the foundation for the current chapter, we can now turn to the heart of this thesis, namely the automated exercise generation.

3.1 Why an Exercise Generator?

As we have already explained in the Introduction, the exercise generator can be a solution to the problem of cheating in exams. In this thesis, however, the focus is mainly on exercises which we want to provide for exam preparation. The problem with the classical approach of many teachers and lecturers to provide only a few or even only one exercise sheet for learning is that this is not adapted to the workflow of each student. While for one student it is sufficient to access and grasp the methods to be learned with a single exercise sheet, another student needs more assignments to understand and consolidate the concepts of the lecture. Now this student could take this exercise sheet and solve it repeatedly, but solving it once and consulting the solution before solving it again can lead to the fact that the student has familiarized himself with only this task and has internalized it so far that the bias arises to have understood the exercise type and the underlying theory, whereas only this specific task has been internalized [BRIM14]. It would normally require further tasks to test whether the student is really able to solve the that particular exercise type correctly. With an exercise generator a student has the possibility and also the choice to generate tasks for practicing at any time, and as often as he wants so a personal workflow is guaranteed [AGK13].

In Figure 3.1 we demonstrate how the workflow of a learner with an exercise generator as exam preparation aid can look like compared to when it is not used.


take in information from lecture
solve exercise provided by lecturer
consolidate lecture content and repeat exercise
solve past exams
ideally pass the exam
(a) workflow without exercise generator

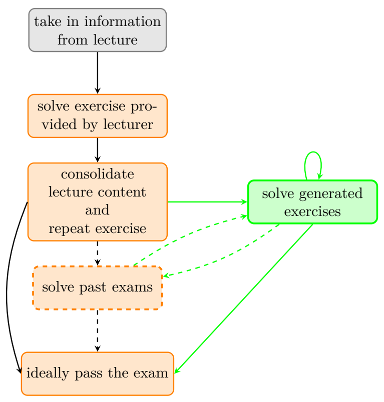

take in information from lecture
solve exercise provided by lecturer
consolidate lecture content and repeat exercise
solve generated exercises
solve past exams
ideally pass the exam
(b) workflow with exercise generator

Figure 3.1

3.2 Related Work

There are several papers that refer to the generation of automated exercises in different contexts, for example [NFSGS16], [AGK13] and [Gan18]. However, to our knowledge, there are hardly any papers on automated exercise generation for Satisfiability Checking. In fact, our research has shown that only Dr. rer. nat. Johannes Waldmann from HTWK Leipzig who implemented an exercise generator for his lecture Constraint programming, has addressed exercise generation in this context [Joh14].

3.3 Conflict Resolution in SAT Solving

3.3.1 Quality characteristics

The tasks we create are intended to practice and consolidate the lecture material and are therefore addressed to learners. We want to help the learner with our task and not work against him and in order to avoid the latter, it is necessary to work out some criteria in advance that contribute to the quality of the exercise and to examine them more closely. With increasing demands on the quality of an exercise, the criteria to be considered also become more diverse. Therefore, the creation of a *good* task requires more refinement and effort. In doing so, we define our objective and set requirements for the problem presented in the task, the task definition and the solution of the task.

3.3.2 Objective

In our opinion, the goal, which every author of an exercise should pursue, is that the exercise should provide a didactic surplus and help the learner to better understand the underlying lecture material. Therefore, it is important to avoid as much as possible any disturbing factors or obstacles that might arise from the exercise itself and distract from this goal. In the following we will list and explain more detailed the objectives for our exercise.

Time required

Our task should be solvable in a reasonable time frame. Normally, our task should be solvable in less than half an hour. We made this estimation after first checking some problem instances by hand. This estimation is also realistic due to the fact that the problem instances we produce for our exercises are not large. However it should be noted that learners who are not familiar with the topic of the exercise will need more time to look up the necessary information to solve the exercise.

Writing effort

The problem instances we use in our conflict resolution task are not large and not very complex and therefore do not produce long solutions, the writing effort is therefore reasonable.

Working material and tools

As mentioned earlier, we do not use complex problem instances for our exercise, which are of reasonable size, therefore the learner does not need to take any additional material such as another tool (SAT solver or similar) to help solve the task. The task is manageable by hand and in a reasonable amount of time. It only requires the lecture notes for reference and as an aid in case of a gap in understanding. The tasks are also printable since we deliver a PDF file and can therefore be solved on paper.

Learning target

Our exercises are intended to teach students one of the driving methods in modern DPLL+CDCL SAT solvers, namely the conflict resolution.

3.3.3 Problem

The problem that we present in our exercise should be one that is relevant. We present computational tasks or tasks where an algorithm is to be performed practically, which is why our problem does not include quiz questions about the theory underlying the conflict resolution. It is important to present problems that can be solved with the content presented in the lecture. The problem should be variable because the learner should get the intuition for solving the problem type no matter how the problem instance looks like. If the problem instances look the same from exercise to exercise, they are likely going to produce similar solution. Moreover variability ensures that learning does not become too monotonous. Concretely in our case this means that the formulas in our task should look different from task to task. The fact that some formulas can produce a solution of the same length is unavoidable but in the end the solution will not be exactly the same in most cases. This means that the learners should rarely get the same asserting clause several times in a row as solution to the conflict resolution. The problem should be realistic, i.e. the scenario presented in it should also be possible in the context of the DPLL+CDCL algorithm. The supposed conflict should therefore really be a conflict that could happen during the SAT solving process given the formula presented in the exercise. In order to ensure this we have used flags which during SAT solving only give us formulas that actually cause a conflict. In addition we have identified and removed formulas from our formula dictionary that superficially fulfill the criteria but are not useful because they produce too short calculation paths, e.g. through conflicts on decision level 0. This means that the solution should also be solvable which is not guaranteed by problem instances that are not considered for the conflict resolution, due to unsatisfiability.

3.3.4 Task definition

In order to introduce the problem to the learners, the problem must be presented in the context of the task. Ideally, the task definition should not leave any questions open. It should clearly state what needs to be done, this means in particular that the task should leave no room for interpretation. As we have explained in our objective, we want to avoid that obstacles arising from the task distract from the essential, namely from learning the material. All instructions given in the task should therefore be clear, precise and understandable. Last but not least, the instructions in the task should be recognizable, i.e. it must be obvious to the learner where the presentation of the problem ends and the instructions for solving the task begins. Since an exercise must contain (proper) instructions, the main requirement is that they are present where necessary.

3.3.5 Solution

A task should of course also have a solution in order to maximize the learning effect. Without a solution the student cannot understand if he has solved the task correctly or why he was not able to finish it. In addition, this prevents the learner from teaching himself wrong things, because he is led to believe that he has solved the task correctly, because without a solution it is often difficult to determine every mistake. And since sources of error can be caused by carelessness as well as by gaps in knowledge, it is important that the learner can clearly identify them in order to get a chance to eliminate the source of error even before he is put to the test in the form of an exam and they happen there. The solution should be correct and complete, so that the student does not learn anything wrong which can also have fatal consequences and above all negatively affect the learning effect. Also, a good readability is conducive to better understand the proposed solution approaches, so the solution should be well structured and it should be easy to follow how the task should have been solved. To make sure that the students have solved the task correctly, they can consult our given solution.

3.3.6 Presentation of the exercise

In the following, we will present an example of our generated exercise for conflict resolution, explain the task definition and the structure of the task sheet, and discuss our design decisions. Subsequently, we present the solution sheet.

SATcheck - Exercise#6

Topic: Conflict Resolution, DPLL+CDCL

RWTH Aachen University – July 3, 2022

Exercise Timer

Starting Time	End Time	Duration
		. . . min

(How long did the exercise take you?)

Task

Consider the following propositional logic formula in CNF:
$c_0 : (A \vee \neg B) \wedge c_1 : (C \vee \neg D \vee \neg A) \wedge c_2 : (C \vee D) \wedge c_3 : (B \vee \neg C \vee A) \wedge c_4 : (A \vee \neg D)$

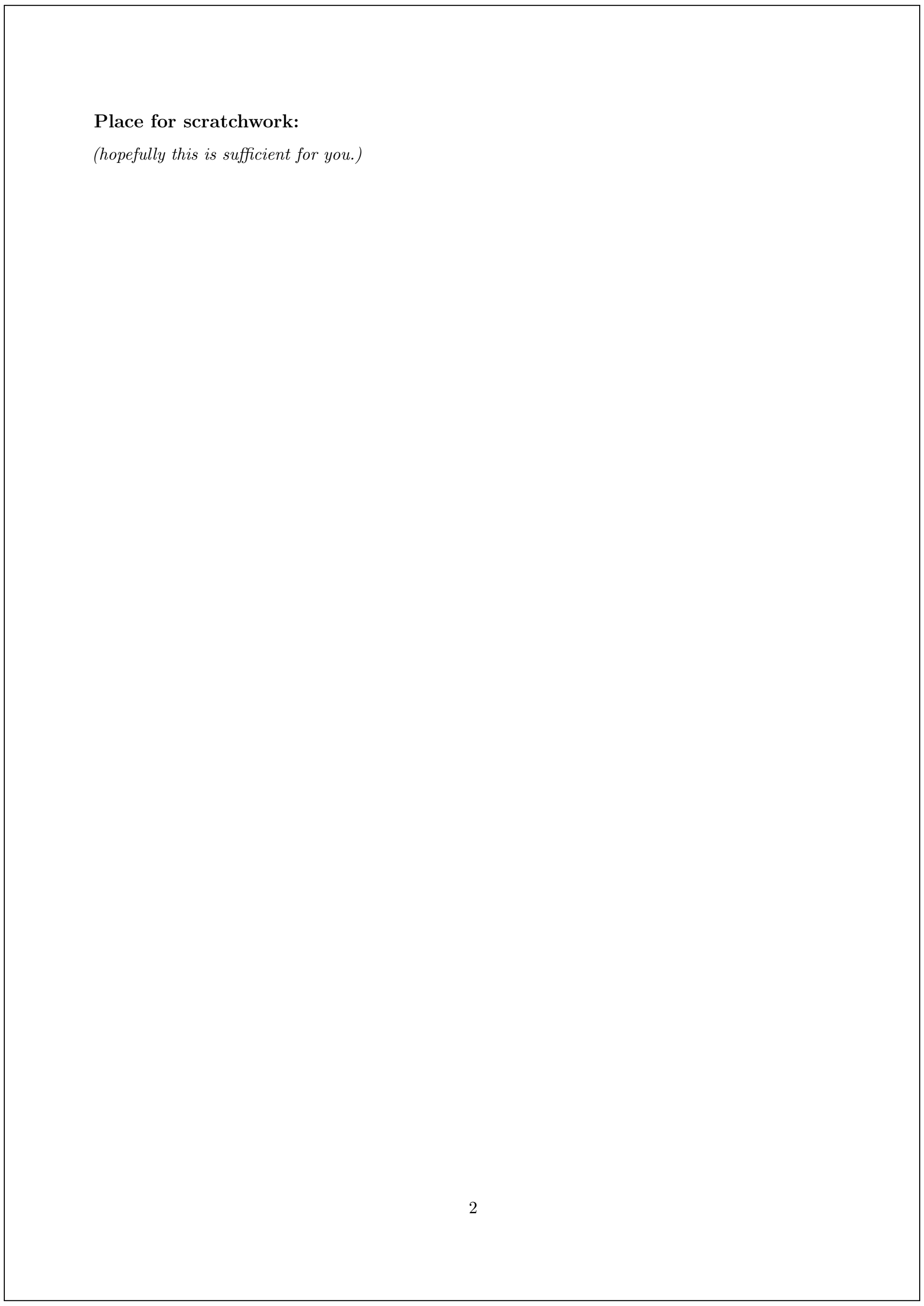
Furthermore assume the following trail:
$DL0 : -$
$DL1 : \neg A : nil, \neg B : c_0, \neg C : c_3, D : c_2$

We have encountered a conflict at the current decision level. Apply conflict resolution to c_4 *till the first unique implication point.* How many new clauses (i.e. clauses that are not already contained in the original formula), are generated during the whole resolution process? Write down the clauses in a row separated by a comma e.g.: clause1, clause2, . . . ,clauseN.

Hint(s)

- Read the task carefully.
- Recall the definitions of the terms *conflicting clause*, *conflict clause*, *first unique implication point*.

Your solution goes here:

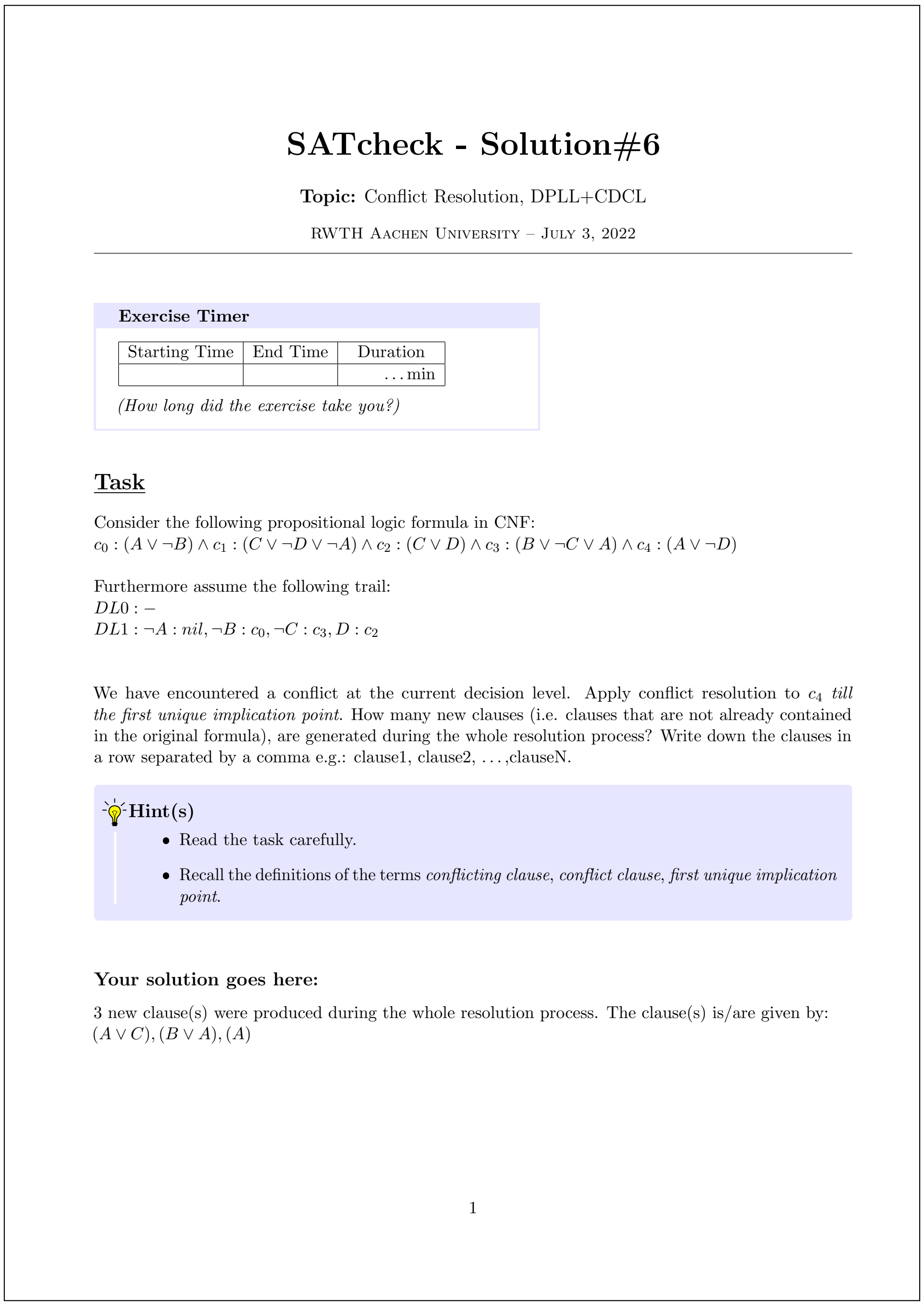


Place for scratchwork:

(hopefully this is sufficient for you.)

2

In the title of the sheet, we have also printed a number (next to "Exercise"), this number is consecutive and is incremented with each freshly generated task. The purpose of assigning such an ID to the exercise sheets is to distinguish them from other exercise sheets. If the student generates several exercise sheets at once, he should have the possibility to manage them. Since the corresponding solution sheet has the same ID, it is also easier to assign the tasks to the solution sheet correctly and quickly. In addition, the numbering also has advantages on the developer side, because if a student contacts us with an error, we can look up the formula that produced this error in the formula dictionary where the generated exercises are stored during runtime of our program, and check it again, which simplifies the bug search. Below the header of the sheet in the left corner is a table designed to mimic a timer. In this timer, the learner can enter when he started the exercise sheet and when he finished or stopped it. In addition, he can also enter the time in minutes that he took to solve the task. This has the advantage that if the student records the time for several tasks, he can track exactly how long it took him from exercise sheet to exercise sheet. In addition, a student can also report a problem instance that, contrary to our wishes, could not be solved in a reasonable time or was impossible to solve. The "timer" is then followed by the description of the problem. In our task, we provide a formula in CNF and the associated trail at the time of the conflict that occurred during SAT solving. This is followed by the problem statement, which briefly states the scenario, i.e. that a conflict has occurred and which clause is conflicting. Then the question is formulated and the learner is also told what to do and how to indicate the solution. We then conclude the task with a box with hints where we remind the student of the terms that are important and which he should recall to be able to solve the exercise. At the end, we also give the student some space to write on the exercise sheet and clearly mark the place designated for writing down the solution and intermediate steps. The solutions of the exercises should not be so long that they fill the entire space, but the student should of course be given the necessary space in case of mistakes and in the case that he needs to restart the formulation of his solution or to make further remarks.



SATcheck - Solution#6

Topic: Conflict Resolution, DPLL+CDCL

RWTH Aachen University – July 3, 2022

Exercise Timer

Starting Time	End Time	Duration
		... min

(How long did the exercise take you?)

Task

Consider the following propositional logic formula in CNF:
$c_0 : (A \vee \neg B) \wedge c_1 : (C \vee \neg D \vee \neg A) \wedge c_2 : (C \vee D) \wedge c_3 : (B \vee \neg C \vee A) \wedge c_4 : (A \vee \neg D)$

Furthermore assume the following trail:
$DL0 : -$
$DL1 : \neg A : nil, \neg B : c_0, \neg C : c_3, D : c_2$

We have encountered a conflict at the current decision level. Apply conflict resolution to c_4 *till the first unique implication point.* How many new clauses (i.e. clauses that are not already contained in the original formula), are generated during the whole resolution process? Write down the clauses in a row separated by a comma e.g.: clause1, clause2, ...,clauseN.

Hint(s)

- Read the task carefully.
- Recall the definitions of the terms *conflicting clause*, *conflict clause*, *first unique implication point.*

Your solution goes here:

3 new clause(s) were produced during the whole resolution process. The clause(s) is/are given by:
$(A \vee C), (B \vee A), (A)$

1

3.4 Implementation

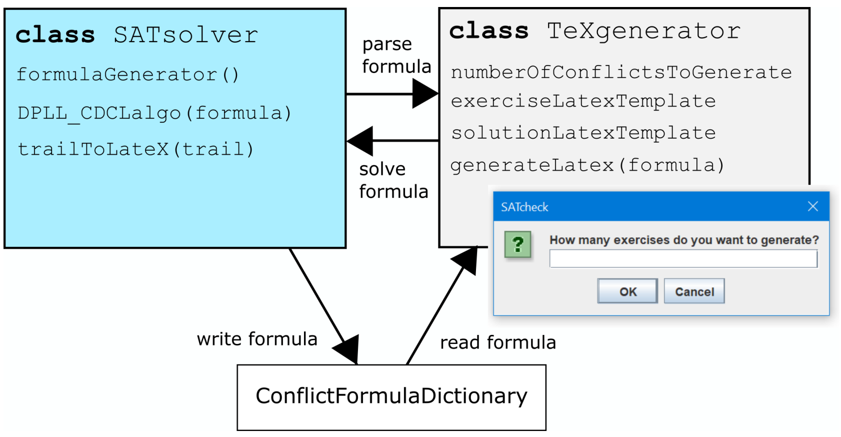


Figure 3.2: Organization of the exercise generator

Our exercise generator, written entirely in the programming language Java consists of two classes: `SATsolver` and `TeXgenerator`.

`SATsolver` contains our SAT solver `DPLL_CDCLalgo`, with whose help we obtain the problem instances for our exercises. In our task we want to practise conflict resolution, for this purpose we provide a formula that produces a conflict during SAT solving, by now we will refer to those formulas as *conflict formulas*, for the sake of simplicity. By using a boolean variable as flags that we put into the `BCP` method, because that is where the conflicts are first detected, we remember the formula. At the point where the flag is set, we store the formula in a .txt file which serves as "dictionary" of conflict formulas. In `TeXgenerator`, the SAT solver is invoked and a conflict formula is passed to it which is retrieved from the conflict formula dictionary. The formula as well as the produced trail are then translated into LATEX code and written into our LATEX template which we have defined within `TeXgenerator` for the exercise and solution. When running the code inside `TeXgenerator` class, the exercise and solution are simultaneously generated and stored as .tex files to the relative path of the users file system in the computer. The produced LATEX files have a default name which is "ExerciseX" for the exercise sheet and "SolutionX" for the solution sheet, where "X" represents the number of the generated sheet.

In Figure 3.2 we have visualized the dependency between the `SATsolver` and `TeXgenerator` class.

3.4.1 `DPLL_CDCLalgo`

For the implementation of our SAT solver, we have followed the basic structure of the DPLL+CDCL SAT solver (see Algorithm 2) presented in the lecture *Satisfiability Checking* held by Prof. Dr. Erika Ábrahám [Ábr22].

Algorithm 2 The CDCL algorithm

```
1: function CDCL(CNF formula $\varphi$)
2:     if $\neg$BCP() then
3:         return UNSAT
4:     end if
5:     while true do
6:         if $\neg$DECIDE() then
7:             return SAT
8:         end if
9:         while $\neg$BCP() do
10:            if $\neg$RESOLVE_CONFLICT() then          ▷ Conflict resolution
11:                return UNSAT
12:            end if
13:        end while
14:    end while
15: end function
```

In the following, however, we will explain the internal of the main building blocks of our implementation of the SAT solver: `BCP()` and `decide()` and `trail`.

`BCP()`

For our `BCP()` method we follow the approach of [Zha96] and propagate unit clauses as follows: if the formula contains a unit clause u then delete all clauses that contain the only literal $l \in u$, we thus imitate the assignment of the truth value *true* to this literal. We remove the clauses since they are already satisfied with the assignment of l to *true* (recall Definition 2.2.1). In addition, delete $\neg l$ in each clause in which it occurrs. We thus imitate the assignment to the truth value *false*, because $\neg l$ can no longer contribute to the satisfiability of the formula and is therefore being discarded.

`decide()`

The decision heuristic we choose in the `decide` method is fixed and static. The variable order arranges the unassigned variables in ascending or lexicographic order. For decisions we always make assignments to the truth value *false*, this value is constant and does not change during SAT solving. For assignments we proceed in the same manner as in the `BCP` method but in reversed order, since for decisions we only make assignments to variables. We encode the assignment of a variable d to the value *false* in the case of a decision by deleting d in every clause in which it occurs, at the same time we remove every clause in which $\neg d$ occurs because the assignment of d to the value *false* implies that its negation $\neg d$ now has the value of *true* which we want to encode accordingly.

`trail`

For our assignments we mantain a trail which is represented as a stack. To the trail only literals that have been assigned the value *true* are appended along with the reason for the assignment, i.e. the antecedent of the literal.

3.4.2 `formulaGenerator()`

Located inside the `SATsolver` class our formula generator `formulaGenerator()` generates formulas randomly. The length of the clauses as well as the length of the formulas vary, whereby we specify a range for both. The literals, which we model internally as integers can take the value from -5 to 5, which are encoded as upper case alphabetic letters, i.e. $A \dots E$. This encoding happens inside `SATsolver` class where the LaTeX code for the trail is parsed as well as in the `TeXgenerator` class for building the LaTeX formulas. The minimum size of a clause is 1 and the maximum size is 6. The formulas are then encoded in the `TeXgenerator` class.

Note that the following exercises, for less lazy SMT solving for equality logic and the real root isolation for CAD, which are inspired by [Ábr22], are not exercises generated by our exercise generator, they are just drafts and serve as an outlook. For this reason we do not showcase them using the generated exercise sheet template (like shown in Section 2.2.5) to make this difference clear.

3.5 Less Lazy SMT Solver

Example Exercise

Consider the propositional logic formula with equalities:

$$\begin{aligned}\varphi^{EQ} := x_2 = x_1 \wedge (\neg(x_4 = x_3) \vee x_1 = x_2) \wedge \\ x_4 = x_3 \wedge (x_1 = x_3 \vee x_4 = x_2)\end{aligned}$$

Furthermore the Boolean abstraction of φ^{EQ} is given by:

$$a_1 \wedge (\neg a_2 \vee a_3) \wedge a_3 \wedge (a_4 \vee a_5)$$

Now a less lazy SMT solver solves the formula for satisfiability as presented in the lecture. If the SAT solver makes a decision, it chooses the unassigned variable a_i with the lowest index and assigns it false. After which decision level in the SAT solver does the Theory solver receive the first (in)equalities that need to be checked for consistency?

Write down the number of the decision level.

3.6 Real Root Isolation

Example Exercise

Consider the polynomial $p = x^2 + x + 1$ and its sturm sequence p_0, p_1, p_2 specified in the table of sign changes below.

Sturm sequence	values at	
	-2	2
$p_0 = x^2 + x + 1$		$+7$
$p_1 = 2x + 1$	-3	$+5$
$p_2 = -\frac{3}{4}$		
# sign changes $\sigma(\cdot)$		

We started filling out the table of sign changes for p but couldn't finish it. Please help us count the real roots in the interval $(-2; 2]$.

Chapter 4

Conclusion

4.1 Discussion

Designing a good exercise is not easy per se and requires considerable effort. This difficulty often arises after the assignment has been already handed out for practice, when students have problems, decoding the instructions in the exercise. What was considered a well thought exercise for the author of the exercise may still raise questions in a student. Reasons for this are, for example, that the student does not understand a definition of a term or a mathematical concept that is being mentioned in the exercise but was not defined within the lecture, but which the task author assumes that the student should know because it should be "fundamental knowledge", for example. Therefore, one has to try to anticipate what the majority of the students is likely to know, which is not easy. To be on the save side, these terms can be defined in the problem definition or an example can be given. However, we must be aware that students may have different levels of knowledge, which is why it is difficult to make the exercise clear to each student, in terms of assessing the quality criteria, this means that "good" here can only be measured approximately, meaning that we may not be able to meet the requirement that the exercise should leave no questions unanswered for each student, which is our main goal. However, for a task that raises "too many" questions for a larger group of students, we can better determine how high its quality is.

4.2 Future Work

We can improve our exercise generation overall on different levels. First of all we could allow for more variability. This concerns in particular our problem instances. For example we could allow different encodings for variable names like a,b,c,d,e and x_1, x_2, x_3, x_4, x_5 (or capitalized X_1,X_2,X_3, X_4, X_5). We could also mix up different variable encoding inside a formula like y_1, y_2, z_3, z_4, z_5. Remains reserved nevertheless are the names $c_0, c_1, c_2, c_3 \ldots$ because they are encoding the clauses of the formulas. Boolean variables are first of all only "names" and a different name does not change the algorithm in which the formula containing the variables plays a role, i.e. conflict resolution. In addition, students should understand the principle behind conflict resolution and not be put off by the fact that a variable is not a letter from the alphabet as usual. We could now discuss whether we should extend the length of the formulas in order to introduce more variability and thus also allow more variables, but then the question arises whether this still brings a didactic surplus if the students have to work with longer formulas. As mentioned at the beginning, we want to help the students to understand the lecture material and not to burden them with a long task that could lead to frustration, since longer formulas inevitably lead to more errors when doing the conflict resolution by hand. Longer formulas could be generated to be included e.g. in context of bonus tasks related to a recent lecture. Because in this case it is about the fact that the students should make an effort to gain these points, which are usually added to the students' exam score. Though not brought to realisation yet, we started examinining how to add incrementality to our formula generator by re-using already generated conflict formulas. We made, for instance, the observation that if we delete a literal that occurs only once in an entire formula conflict formula, the new formula that results from it still produces a conflict during SAT solving. Furthermore importing and exporting the formulas from the conflict formula dictionary has a negative effect on runtime. An improvement would be to parse the formulas directly after importing them from the `SATsolver` class instead of first fetching them from the formula dictionary and passing them on to `SATsolver` again from inside `TeXgenerator`. We can provide preparatory tasks to introduce conflict resolution. More general we could provide exercises of different levels of difficulty and parameterize the exercise generator accordingly. In the first difficulty level, we can give problem to test and strengthen the fundamental knowledge of the task type and help the students to learn the necessary definitions to solve the exercises asking to do a conflict resolution. In this "preparatory level", assuming the conflict formula and the trail at the time of the conflict are given, we ca ask the student what the conflicting and conflict clause are, because this is the starting point of the conflict resolution. This has the advantage that if the students make the same mistake over and over again in the actual conflict resolution exercise, they may realize that they lack knowledge about how to proceed in the conflict resolution and that he has not yet understood the definitions correctly. The introduction of such levels also contributes positively to the personal workflow of the learner because he can either start with simple tasks or possibly fall back on them if they realize that they still have gaps in their knowledge about the conflict resolution [AGK13].

Bibliography

[Ábr22] Prof. Dr. Erika Ábrahám. Lecture notes Satisfiability Checking. RWTH Aachen University, LuFG Theory of Hybrid Systems, 2021-2022.

[AGK13] Umair Z. Ahmed, Sumit Gulwani, and Amey Karkare. Automatically generating problems and solutions for natural deduction. In *Proceedings of the Twenty-Third International Joint Conference on Artificial Intelligence*, IJCAI '13, page 1968–1975. AAAI Press, 2013.

[ÁK17] Erika Ábrahám and Gereon Kremer. Smt solving for arithmetic theories: Theory and tool support. In *2017 19th International Symposium on Symbolic and Numeric Algorithms for Scientific Computing (SYNASC)*, pages 1–8, 2017.

[BKM14] Clark Barrett, Daniel Kroening, and Thomas Melham. *Problem solving for the 21st century: Efficient solver for satisfiability modulo theories.* Knowledge Transfer Report, Technical Report 3. London Mathematical Society and Smith Institute for Industrial Mathematics and System Engineering, June 2014.

[BRIM14] P. C. Brown, Roediger, H. L. III, and M. A. McDaniel. *Make it stick: The science of successful learning.* Belknap Press of Harvard University Press, 2014.

[Col75] George E. Collins. Quantifier elimination for real closed fields by cylindrical algebraic decomposition. *Lecture Notes in Computer Science*, 1975.

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

[CRE18] Alexander Imani Cowen-Rivers and Matthew England. Summer research report: Towards incremental lazard cylindrical algebraic decomposition. 2018.

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

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. 7, 1960.

[FM09] John Franco and John Martin. A history of satisfiability. *Frontiers in Artificial Intelligence and Applications*, 185, 01 2009.

[Gan18] Mikulas Gangur. Automated generation of statistical tasks. 12 2018.

[GV20] Vijay Ganesh and Moshe Y. Vardi. On the unreasonable effectiveness of sat solvers. In *Beyond the Worst-Case Analysis of Algorithms*, 2020.

[HB20] Randy Hickey and Fahiem Bacchus. *Trail Saving on Backtrack*, pages 46–61. 06 2020.

[Jir95] Mats Jirstrand. *Cylindrical algebraic decomposition-an introduction.* Linköping University, 1995.

[Joh14] Johannes Waldmann. Automated exercises for cosntraint programming. `https://www.imn.htwk-leipzig.de/~waldmann/talk/14/wlp/auto/`, 2014.

[KÁ19] Gereon Kremer and Erika Ábrahám. Fully incremental cylindrical algebraic decomposition. *Journal of Symbolic Computation*, 100, 07 2019.

[Kau10] Manuel Kauers. How to use cylindrical algebraic decomposition. *Séminaire Lotharingien de Combinatoire*, 65, 01 2010.

[Knu06] Donald E. Knuth. *The Art of Computer Programming, Volume 4, Fascicle 4: Generating All Trees–History of Combinatorial Generation (Art of Computer Programming).* Addison-Wesley Professional, 2006.

[Kre20] Gereon Kremer. Cylindrical algebraic decomposition for nonlinear arithmetic problems. 2020.

[KS08] Daniel Kroening and Ofer Strichman. *Decision Procedures: An Algorithmic Point of View.* Springer Publishing Company, Incorporated, 1 edition, 2008.

[MSLM21] Joao Marques-Silva, Ines Lynce, and Sharad Malik. Chapter 4: Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, Frontiers in Artificial Intelligence and Applications, pages 133–182. IOS Press BV, 2021.

[NFSGS16] Valentin Nentwich, Nicolai Fischer, Andreas C. Sonnenbichler, and Andreas Geyer-Schulz. Computer aided exercise generation - a framework for human interaction in the automated exercise generation process. In *ICE-B*, 2016.

[Seb07] Roberto Sebastiani. Lazy satisfiability modulo theories. *Journal of Satisfiability, Boolean Modeling and Computation*, 3, 12 2007.

[ST12] Uwe Schöning and Jacobo Torán. *Das Erfüllbarkeitsproblem SAT: Algorithmen und Analysen.* Mathematik für Anwendungen. Lehmanns Media, 2012.

[Tar98] Alfred Tarski. A decision method for elementary algebra and geometry. In Bob F. Caviness and Jeremy R. Johnson, editors, *Quantifier Elimination and Cylindrical Algebraic Decomposition*, pages 24–84, Vienna, 1998. Springer Vienna.

[Tse83] G. S. Tseitin. On the complexity of derivation in propositional calculus. In *Automation of Reasoning: 2: Classical Papers on Computational Logic*, pages 466–483. Springer Berlin Heidelberg, 1983.

[Zha96] Hantao Zhang. An efficient algorithm for unit propagation. *Ai Magazine*, 1996.