UNIVERSITÄT ZU LÜBECK
INSTITUTE FOR SOFTWARE ENGINEERING
AND PROGRAMMING LANGUAGES

DEVELOPMENT OF A SAT SOLVER

ENTWICKLUNG EINES SAT-SOLVERS

Bachelorarbeit

im Rahmen des Studiengangs
Informatik
der Universität zu Lübeck

vorgelegt von
Jannis Harder

ausgegeben und betreut von
Prof. Dr. Martin Leucker

Lübeck, den 27. März 2014

Abstract

The boolean satisfiability problem (SAT) belongs to set of NP-complete problems. Nevertheless the advent of SAT solvers which are fast for many practical problem instances made reduction to SAT practical for different problems in a large number of areas. Besides the commonly used complete SAT solvers based on the DPLL procedure there also exists Stålmarck's method which received comparatively little attention in the literature. A recent work by Thakur and Reps "A Generalization of Stålmarck's Method" combines Stålmarck's method with concepts of abstract interpretation, thereby creating a new variant of Stålmarck's method. This thesis extends upon this work by introducing two new modifications to the generalized variant of Stålmarck's method. The first modification provides additional flexibility for heuristics when using a set of constraints as an abstract domain. The other modification allows for new exploration strategies of the search space. A prototype implementation is described including the design choices made. This prototype is evaluated and compared to existing DPLL based SAT solvers.

Zusammenfassung

Das Erfüllbarkeitsproblem der Aussagenlogik (SAT) gehört zu den NP-vollständigen Problemen. Dennoch hat das Aufkommen von SAT-Solvern, die das Problem in vielen in der Praxis auftretenden Fällen effizient lösen können, dafür gesorgt, dass Reduktion auf SAT für viele Probleme aus zahlreichen Bereichen praktikabel geworden ist. Neben den üblicherweise verwendeten SAT-Solvern die auf der DPLL Prozedur basieren existiert außerdem die Methode von Stålmarck welche, in der Literatur vergleichsweise wenig Beachtung gefunden hat. Eine kürzlich erschienene Veröffentlichung von Thakur und Reps „A Generalization of Stålmarck's Method" verknüpft die Methode von Stålmarck mit Konzepten der abstrakten Interpretation und beschreibt eine daraus entstehende Variante der Methode von Stålmarck. Diese Arbeit baut darauf auf und erweitert diese Methode durch zwei neue Erweiterungen. Die erste Erweiterung ermöglicht zusätzliche Flexibilität für Heuristiken bei der Verwendung von Constraintmengen als abstrakte Domäne. Die zweite Erweiterung erlaubt neue Erkundungsstrategien des Suchraums. Eine Prototypimplementation mit den dazugehörigen Entwurfsentscheidungen wird beschrieben. Dieser Prototyp wird ausgewertet und mit bestehenden, DPLL-basierten SAT-Solvern verglichen.

Erklärung

Hiermit erkläre ich an Eides statt, dass ich die vorliegende Arbeit ohne unzulässige Hilfe Dritter und ohne die Benutzung anderer als der angegebenen Hilfsmittel selbständig verfasst habe; die aus anderen Quellen direkt oder indirekt übernommenen Daten und Konzepte sind unter Angabe des Literaturzitats gekennzeichnet.

_____

Lübeck, den 27. März 2014

Acknowledgements

I want to thank Professor Dr. Martin Leucker for advising me and for giving me the opportunity to work on a topic of my choice.

I also want to thank Sophie Kowalski, Andreas Kordon, my brother and my parents without whose support and encouragement I could not have written this thesis.

Contents

1 Introduction

High performance solvers for the *boolean satisfiability problem* (SAT) have evolved into versatile tools in a large number of areas. SAT is one of the prototypical NP-complete problems and it admits efficient and direct encodings for a large number of different problems. Apart from the theoretical usefulness this turned out to be of great practical utility.

Reduction of problems to a SAT instance which is then solved has become a standard process in various areas of hardware and software verification, planning and scheduling, and combinatorial design [1]. SAT solvers have also been used as a tool for cryptanalysis [2], for theorem proving and as a base for further decision and search procedures as in satisfiability modulo theories (SMT) [3] or answer set programming (ASP) [4].

The field of competitive SAT solvers has converged to three different families of algorithms [5], [6]. These families are the two complete approaches *conflict driven clause learning* (CDCL), *look-ahead search* and the family of incomplete methods based on *local search* [1]. For application and combinatorial problem instances all winners of the SAT competition 2013 belong to the CDCL family of algorithms [5], [6]. This thesis will focus on complete methods, which are required for many applications, and thus will omit the details of local search based procedures.

The success of CDCL based solvers naturally places them in the focus of research for improving the performance of SAT solvers. And while there is no sign that the possible improvements to CDCL solvers are exhausted, it is worthwhile to explore alternatives, as there can be specific problem classes where they are superior to CDCL solvers, as is the case with look-ahead solvers for random SAT instances.

To compare the relative performance of different methods it is important to have optimized implementations. CDCL solvers have very sophisticated implementations that have been improved continously for over a decade. Competing methods that did not receive as much attention thus are at a disadvantage. Therefore it is important to carefully analyze and improve the methods with a focus on the implementation to enable a fair comparision.

This thesis aims to present the work needed to implement a SAT solver, not based on CDCL, from scratch. Stålmarck's method was chosen because recent work by Thakur and Reps [7] opens up many

ways to improve the basic method which itself saw successful industrial use. In addition this thesis presents some new techniques for extending Stålmarck's method.

Apart from developing a solver that can be used to compare variants of Stålmarck's method to CDCL based and other solvers, it can serve as a guideline of the steps needed to implement SAT solvers using even different techniques.

1.1 *Related work*

There is a large amount of literature available in the field of SAT solving. The "Handbook of Satisfiability" edited by Biere et al. [1] provides an extensive overview of the area. Many current SAT solvers are based on variants of the DPLL [8] procedure which is a modification of the DPP method [9]. This includes lookahead solvers [10]–[12] and conflict driven clause learning solvers [13]–[15]. A different algorithm is Stålmarck's method [16], [17] which is the basis for the algorithm implemented as part of this thesis. A short summary of these methods is given in chapter 2 of this thesis. Recent work has explored the combination of classic SAT solving techniques with abstract interpretation [18], a technique developed for static program analysis. There has been independent work on applying abstract interpretation to CDCL methods [19] and to Stålmarck's method [7], [20]. This thesis builds on the work combining abstract interpretation with Stålmarck's method.

1.2 *Structure of the Thesis*

The remainder of the Thesis is structured as follows:

Chapter 2 gives a brief overview of complete SAT solving methods. This includes the commonly used DPLL techniques as well as Stålmarck's method, the basis for this work.

Chapter 3 presents a generalization of Stålmarck's method as described by Thakur and Reps [7], [20]. It includes a brief summary of the used order theory and abstract interpretation basics.

Chapter 4 extends the method described in the previous chapter with a framework for handling arbitrary constraints and a framework for different search space exploration strategies.

Chapter 5 describes an attempt of implementing the extended method using reduced ordered binary decision diagrams (BDDs) to represent constraints. This attempt did not succeed in producing a usable SAT solver. Nevertheless part of the work done for this implementation, namely a conversion from a formula in conjunctive normal form (CNF) to a smaller set of BDD constraints, was successful and is evaluated.

Chapter 6 describes a different implementation of the extended method. This time simpler constraints are used to avoid the problems that occurred for the implementation described in chapter 5. The architecture and implementation details of the SAT solver are described.

Chapter 7 evaluates the implementation described in chapter 6 and

compares it to solvers that implement other methods.

The final chapter 8 summarizes the results obtained and gives an outlook of possible enhancements to the presented method and its implementation.

2
An Overview of Satisfiability Solver Algorithms

2.1 Boolean Satisfiability

The *boolean satisfiability problem* (SAT) asks whether a formula of propositional logic has a satisfying model. A propositional formula, or boolean expression, can be inductively defined as a variable, a constant truth value ($\mathtt{t}$ or $\mathtt{f}$), the negation of a formula ($\neg$), or the connection of two formulas using a connective ($\wedge$, $\vee$, $\rightarrow$ or $\leftrightarrow$).

Grammar of a propositional formula:

$$\phi ::= \mathtt{t} \mid \mathtt{f} \mid v_i \mid \phi_1 \wedge \phi_2 \mid \phi_1 \vee \phi_2 \mid \phi_1 \rightarrow \phi_2 \mid \phi_1 \leftrightarrow \phi_2 \mid \neg\phi$$

Given a variable assignment β, a function that maps every variable to a truth value, it is possible to recursively define a function $\hat{\beta}$ that assigns a matching truth value to any propositional formula.

A variable assignment β that assigns $\mathtt{t}$ to a formula ϕ, e.g. $\hat{\beta}(\phi) = \mathtt{t}$, is called a model of ϕ. In this case we write $\beta \vDash \phi$. If all possible variable assignments are a model of ϕ, so that ϕ is a tautology, we write $\vDash \phi$. Satisfiability can then be defined as the existence of a model $\exists\beta : \beta \vDash \phi$. As $\vDash \phi \Leftrightarrow \neg\exists\beta : \beta \vDash \phi$ a tautology is exactly a formula which has an unsatisfiable negation.

In addition to deciding whether a model exists, a SAT solvers task usually includes finding a concrete model in the case of satisfiability. There are also SAT solvers that are able to output an unsatisfiability proof or an unsatisfiable core, a minimal subset of clauses that are unsatisfiable. An extensive account of satisfiability and its history is given in [1].

2.2 Conjunctive Normal Form

While the grammar of propositional logic is suited for manual formula manipulation, a simpler structure is advantageous for algorithmic manipulation. The most common choice for SAT solving is the *conjunctive normal form* (CNF).

Grammar of a formula in CNF:

$$\phi_{CNF} ::= \mathtt{t} \mid C \mid C \wedge \phi_{CNF}$$
$$C ::= \mathtt{f} \mid L \mid L \vee C$$
$$L ::= v_i \mid \neg v_i$$

The usefulness of the conjunctive normal form for SAT solving was first highlighted by Davis and Putnam in [9]. A formula in conjunctive normal form consists of a conjunction of *clauses*. A clause in turn is a disjunction of *literals* and a literal is a variable or its negation. This allows a simple representation as a set of sets of literals.

The set representation of

$$A \wedge (B \vee \neg C) \wedge \mathtt{f}$$

is

$$\{\{A\}, \{B, \neg C\}, \varnothing\}$$

While any propositional formula can be converted to an equivalent formula in conjunctive normal form by applying simple rewrite rules, this causes some formulas to grow exponentially in length. An example

for a family of such formulas is $(A_1 \wedge B_1) \vee (A_2 \wedge B_2) \vee ... \vee (A_n \wedge B_n)$ which has a corresponding CNF with 2^n clauses. For SAT solving though it is not necessary to use an equivalent formula. Using a weaker notion, an *equisatisfiable* formula, is sufficient. Equisatisfiability allows introduction of new variables. For CNF conversion the new variables are usually constrained to take the value of a sub-formula. This enables a conversion with the output length polynomially bounded by the input length [21].

Splitting the clause

$$(A \vee B \vee C \vee D)$$

into two clauses

$$(A \vee B \vee X) \wedge (C \vee D \vee \neg X)$$

where X is a fresh variable is an example of a transform that preserves satisfiability but not equivalence.

2.3 The DPLL Procedure

There are several methods in the literature that form the basis for most SAT solver implementations. The first described satisfiability testing procedure that can be seen as a predecessor to modern SAT solvers is the Davis-Putnam procedure (DPP) [9], which also introduced the conjunctive normal form into satisfiability testing [1]. DPP consists of three rules, which modify a set of CNF clauses without changing its satisfiability. Repetitive application reduces the set to either the empty set, if satisfiable, or to a set containing an empty clause, if unsatisfiable. These rules are:

Unit-Clause Rule When a clause consists of a single literal, that literal must be true to satisfy the formula. Assuming true for the literal's value, all clauses containing that literal are satisfied and can be removed. All clauses containing the literal's negation are modified by removing the negated literal, which cannot make a clause's value become true.

Pure-Literal Rule When a literal is present but there are no clauses containing its negation it is safe to assume the literal's value is true, as this cannot cause any clauses to become unsatisfied. All clauses containing this literal can thus be removed.

Resolution Rule This rule can be used to eliminate a variable from the set of clauses. All clauses containing the variable are grouped into the set of literals with a positive occurrence and with a negative occurrence. All those clauses are then replaced with the disjunction of both sets. When this disjunction is converted back into the conjunctive normal form it becomes the set of all disjunctions formed by taking a clause of either set.

The names of the rules given here are as used in recent literature and differ from the names given in the original publication.

Unit-Clause Example:

$$\begin{matrix}\{\mathbf{A}\}\\ \{\mathbf{A}, \neg B\}\\ \{\neg\mathbf{A}, \neg C\}\\ \{\neg B, C, D\}\end{matrix} \Rightarrow \begin{matrix}\{\neg C\}\\ \{\neg B, C, D\}\end{matrix}$$

Pure-Literal Example:

$$\begin{matrix}\{\mathbf{A}, B, \neg D\}\\ \{\neg C, D\}\\ \{\mathbf{A}, \neg B\}\\ \{\neg B, C\}\end{matrix} \Rightarrow \begin{matrix}\{\neg C, \neg D\}\\ \{\neg B, C\}\end{matrix}$$

Resolution Example:

$$\begin{matrix}\{\mathbf{A}, B, \neg C\}\\ \{\mathbf{A}, \neg B, C\}\\ \{\neg\mathbf{A}, \neg D\}\\ \{\neg\mathbf{A}, E\}\\ \{B, C, D, \neg E\}\end{matrix} \Rightarrow \begin{matrix}\{B, \neg C, \neg D\}\\ \{\neg B, C, \neg D\}\\ \{B, \neg C, E\}\\ \{\neg B, C, E\}\\ \{B, C, D, \neg E\}\end{matrix}$$

The resolution rule is problematic as it generates a large number of clauses, quickly exhausting the available amount of memory. This prompted Loveland and Logemann to replace it with a *splitting rule* that successively explores the conclusion of assuming a variable to be true or false, thereby creating a recursive algorithm [8]. This is the DPLL procedure which still forms the basis for most recent complete SAT solvers.

The split rule on a variable v is implemented by recursively invoking the DPLL procedure twice, once with the variable assumed true and once assumed false. This is equivalent to adding a unit clause with the literal v or $\neg v$, which will trigger the unit-clause rule in the recursive call.

The resulting structure of the DPLL procedure is shown in pseudocode below:

```
1:  procedure DPLL(φ)
2:      φ ← Propagate(φ)
3:      if φ = ∅ then
4:          return sat
5:      else if ∅ ∈ φ then
6:          return unsat
7:      Choose v ∈ Vars(φ)
8:      s_t ← DPLL(φ[v = t])
9:      if s_t = sat then
10:         return sat
11:     s_f ← DPLL(φ[v = f])
12:     return s_f
```

Algorithm 2.1: DPLL Procedure

Propagate is a function that applies the unit-clause and pure-literal rules

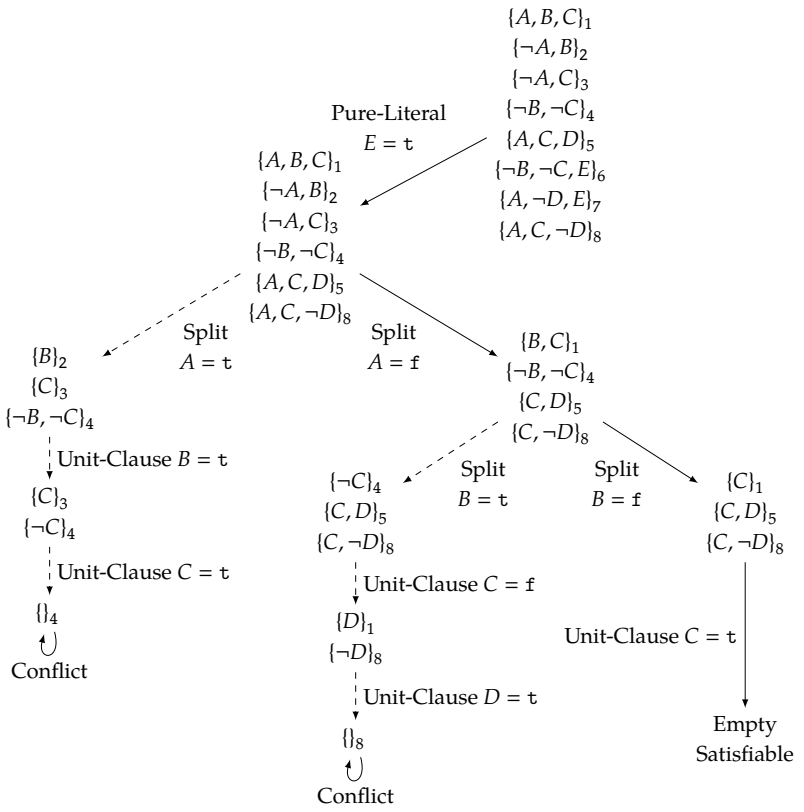


Figure 2.1: Example of execution flow and rules used during the DPLL execution

2.4 *Lookahead*

The DPLL procedure does not specify on which variable to branch. The choice of the decision variable has a substantial influence on the overall performance of the procedure. Lookahead solvers try to make good choices here by exploring the consequences of branching on a variable. This is realized by a lookahead procedure which invokes the propagate procedure for both possible assignments of a variable.

The variables are then ranked by a score computed from the resulting CNF formulas. If a conflict occurs while assuming a literal during the lookahead procedure, a so called *failed literal*, the assumed variable must have the negated value. [1]

Algorithm 2.2: DPLL with Lookahead

DecisionHeuristic estimates how much the formula is simplified by branching on a variable. DirectionHeuristic estimates the most likely assignment for a variable.

```
1:  procedure DPLL(φ)
2:      φ ← Propagate(φ)
3:      φ, v ← Lookahead(φ)
4:      if φ = ∅ then
5:          return sat
6:      else if ∅ ∈ φ then
7:          return unsat
8:      b ← DirectionHeuristic(φ, v)
9:      s_b ← DPLL(φ[v = b])
10:     if s_b = sat then
11:         return sat
12:     s_¬b ← DPLL(φ[v = ¬b])
13:     return s_¬b
14: procedure Lookahead(φ)
15:     choose a set of variables P
16:     for each v ∈ P do
17:         φ_t ← Propagate(φ[v = t])
18:         if ∅ ∈ φ_t then
19:             φ ← Propagate(φ[v = f])
20:             continue
21:         φ_f ← Propagate(φ[v = f])
22:         if ∅ ∈ φ_f then
23:             φ ← Propagate(φ[v = t])
24:             continue
25:         H_v ← DecisionHeuristic(φ, φ_t, φ_f)
26:     return φ, arg max_v H_v
```

The first SAT solver using this method, developed in 1995, was `posit` [10]. Current implementations using lookahead are the variants of the `march` solver [11], [12] which are competitive for random SAT instances.

2.5 *Conflict Driven Clause Learning*

Another way to improve the DPLL procedure is by dynamically adding new clauses to the formula to guide the search to a solution. This is done by analyzing conflicts during the recursion and adding clauses that avoid the reason for the conflict in the future. This approach is called conflict driven clause learning (CDCL). [1]

To find a clause that avoids the reason for an occurred conflict an *implication graph* is used. The implication graph is a directed graph of all literals considered true under the current assumptions. Whenever the unit-clause rule is applied, new edges pointing to the literal of the unit clause are added from each negated literal of the original clause

that became unit.

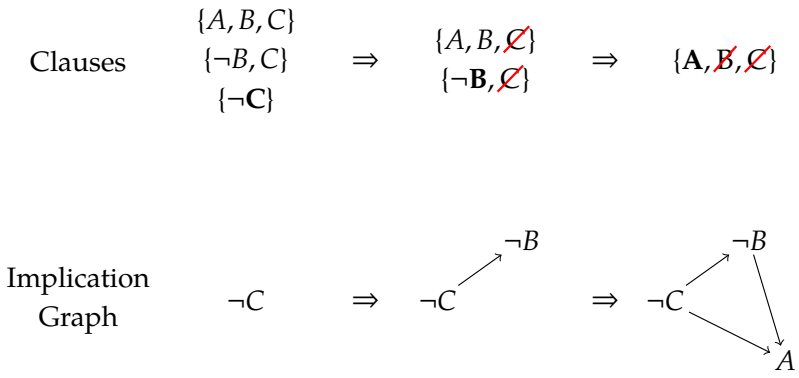


Figure 2.2: Example for building an implication graph

A conflict occurs when the negation of every literal of a clause is present in the implication graph. To find a clause that avoids this conflict in the future, a graph cut between the assumed literals and the literals causing the conflict is chosen. A clause containing all negated literals of nodes with an outgoing edge crossing the cut is then added to the formula. This makes sure that at most all but one of the reasons for the conflict occur simultaneously in the future. The choice of the graph cut is made heuristically. The implication graph can be represented implicitly by recording the clause that implies a literal.

When such a clause is added to the formula, without backtracking all its literals are false. Depending on the graph cut used, it is necessary to backtrack multiple levels until the new clause becomes unit. The new clause can be unit for multiple levels and in practice backtracking to the lowest level where the clause is unit, known as *non-chronological backtracking* or *backjumping* has been shown to be effective [13].

As backtracking is done multiple levels at the same time CDCL is often implemented iteratively with an explicit stack of assumed literals. This results in the structure shown in the pseudocode of algorithm 2.3.

The first SAT solver that introduced this method, along with non-chronological backtracking, was GRASP [13]. Another notable solver is Chaff [14] as it introduced a very efficient data structure for unit-propagation called *watched literals*. A complete walk-through for the development of a CDCL solver is given in [15], which describes the successful solver `minisat`.

2.6 *Stålmarck's Method*

Another SAT solver procedure is Stålmarck's method. While a preliminary version was patented in 1990 [16] and later refined, it received little attention compared to DPLL based procedures. Nevertheless it was industrially successful [22] and received new attention when the patent expired [7]. A complete description is given in [17].

Like DPLL Stålmarck's method keeps track of a current formula and information deduced about the variables. While DPLL uses a partial assignment, Stålmarck's method as described in [17] goes one step further and uses an equivalence relation on the literals and the

```
1:  procedure CDCL(φ)
2:      level ← 0
3:      loop
4:          if φ[assumptions] = ∅ then
5:              return sat
6:          else if ∅ ∈ φ[assumptions] then
7:              if level = 0 then
8:                  return unsat
9:              c ← AnalyzeConflict(assumptions, reasons)
10:             φ ← φ ∪ {c}
11:             backtrack to smallest level so that |c[assumptions]| = 1
12:         else if c ∈ φ with c[assumptions] = {l} then
13:             assumptions_level ← assumptions_level ∪ {l}
14:             reasons_l ← c
15:         else
16:             level ← level + 1
17:             chose unassigned literal l
18:             assumptions_level ← assumptions_level ∪ {l}
```

Algorithm 2.3: CDCL

AnalyzeConflict computes a conflict avoiding clause from the implication graph.

constant values `t` and `f`. Another difference is in the representation of the formula, where Stålmarck's method uses a conjunction of terms of the form $x \leftrightarrow (y \rightarrow z)$, called triplets.

The procedure works by simplifying the current formula, thereby refining the equivalence relation on the literals, until either a contradiction is reached or until the current formula becomes the empty conjunction. This is realized by a set of *simple rules*, which apply local reasoning on the triplets, and a *dilemma rule* which is used to build a recursive search procedure.

The *simple rules* apply to a single triplet and all equivalences between the literals of its variables. If a new equivalence follows from that, it is added to the relation. The rules are shown in figure 2.3. The form of the triplets ensures that once a basic rule can be applied, the resulting equivalences completely constrain the variables to satisfy the triplet. This means that a triplet can be removed once it triggered a rule.

Figure 2.3: Simple rules of Stålmarck's method

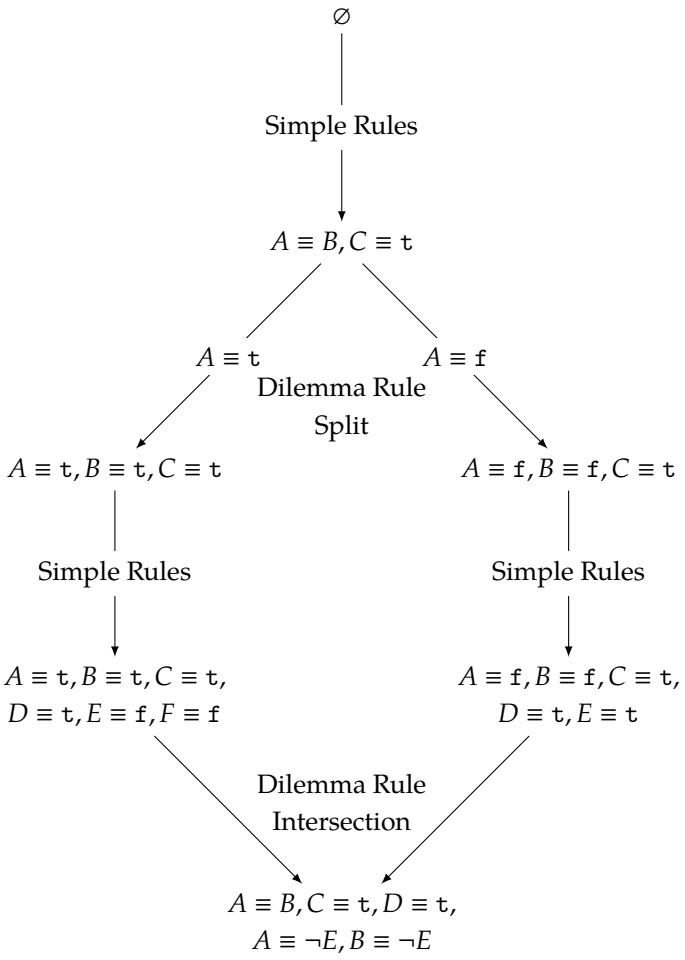
R1:	$\mathtt{f} \leftrightarrow (y \rightarrow z)$	$\Rightarrow$	$y \equiv \mathtt{t}, z \equiv \mathtt{f}$
R2:	$x \leftrightarrow (y \rightarrow \mathtt{t})$	$\Rightarrow$	$x \equiv \mathtt{t}$
R3:	$x \leftrightarrow (\mathtt{f} \rightarrow z)$	$\Rightarrow$	$x \equiv \mathtt{t}$
R4:	$x \leftrightarrow (\mathtt{t} \rightarrow z)$	$\Rightarrow$	$x \equiv z$
R5:	$x \leftrightarrow (y \rightarrow \mathtt{f})$	$\Rightarrow$	$x \equiv \neg y$
R6:	$x \leftrightarrow (x \rightarrow z)$	$\Rightarrow$	$x \equiv \mathtt{t}, z \equiv \mathtt{t}$
R7:	$x \leftrightarrow (y \rightarrow y)$	$\Rightarrow$	$x \equiv \mathtt{t}$

While the formula representation is different from the CNF representation used by DPLL, these basic rules can be adopted to work with conjunctions of any kind of terms.

The essential innovation of Stålmarck's method is the *dilemma rule*. The dilemma rule allows branching on a variable (or possibly on an equivalence of two variables). Unlike the basic DPLL procedure's splitting rule, the dilemma rule reconciles the two considered cases. The rule branches on the variable, by assuming an equivalence for one case and its negation for the other, and deducing more equivalences for both cases. It continues by combining the equivalence relations from both branches. An example is shown in figure 2.4. If we view the relations as a set of tuples, they are combined by computing their intersection. This is equivalent to the disjunction of the equivalences. The dilemma rule is consistent as everything deduced is an equivalence

that holds in all cases. A conflict will result in an equivalence relation with a single equivalence class containing all literals. Thus in case of a conflict in one branch, the dilemma rule simply continues with the other.

Figure 2.4: Dilemma Rule Example



Stålmarck's method tries to find a shallow sequence of dilemma rule applications that leads to a contradiction or empty formula. The shallowest sequence does not use the dilemma rule at all. This means that only the simple rules are applied until no further application is possible. This is done by a procedure called 0-Saturation. $(k+1)$-Saturation is then defined as applying the dilemma rule repeatedly, cycling through all variables, until no more progress is made. The two cases of the dilemma rule are processed with k-Saturation. Stålmarck's method is then defined as trying 0-Saturation, 1-Saturation, and so on, until a solution or conflict is found. This is a complete method as with k set to the number of variables present in the formula, k-Saturation will exhaustively try all possible assignments.

```
1:  procedure Stålmarck(φ)
2:      k ← 0
3:      R ← ∅
4:      loop
5:          φ, R ← k-Saturation(φ, R)
6:          if t ≡ f ∈ R then
7:              return unsat
8:          k ← k + 1
9:  procedure 0-Saturation(φ, R)
10:     for all t ∈ φ matching a simple rule r given R do
11:         φ ← φ \ {t}
12:         R ← (R ∪ r(t, R))^≡
13:     if t ≡ f ∈ R then
14:         return ∅, {t ≡ f}
15:     else if v ≡ t or v ≡ f for all variables v then
16:         return sat from procedure Stålmarck
17:     return φ, R
18: procedure (k + 1)-Saturation(φ, R)
19:     repeat
20:         R′ ← R
21:         for variable v with v ≡ t ∉ R and v ≡ f ∉ R do
22:             φ_f, R_f ← k-Saturation(φ, (R ∪ {v ≡ f})^≡)
23:             φ_t, R_t ← k-Saturation(φ, (R ∪ {v ≡ t})^≡)
24:             R ← R_f ∩ R_t
25:             φ ← φ_f ∪ φ_t
26:     until R′ = R
27:     return φ, R
```

Algorithm 2.4: Stålmarck's Method

$X^{\equiv}$ is the reflexive transitive symmetric closure of X, i.e. the smallest equivalence relation containing X.

3
The Generalized Method of Stålmarck

Recent work of Thakur and Reps explores the possibilities of generalizing Stålmarck's method, and describes the theoretical foundations for doing so [7]. They reformulate the method in the terminology of *abstract interpretation*. Different variants of the generalized method can then be obtained by switching out one *abstract domain* for another. They also made an extended technical report available [20], containing proofs omitted in [7].

Abstract domains are an order theoretic tool for working with approximate knowledge to simplify otherwise prohibitively large computations. The key observation made by Thakur and Reps is that the use of an equivalence relation as an approximation of the possible satisfying assignments is an abstract domain. This implies that different methods can be realized by choosing different abstract domains. Indeed the preliminary variant of Stålmarck's method described in the patent [16] does not use an equivalence relation but instead a partial assignment, which, in the formulation of Thakur and Reps, is simply the use of a different abstract domain.

3.1 Order Theory and Abstract Interpretation

Abstract interpretation uses approximation to reduce computational costs. Nevertheless it produces sound results and can be used to derive accurate bounds or even exact results. To ensure soundness abstract interpretation makes careful use of over- or under-approximations. This is formalized using *order theory*. This section summarizes the basic order theory needed in this thesis. An introduction to order theory is given in [23].

The essential construction of order theory is that of a *partially ordered set* or *poset*. A partial order is a a reflexive, antisymmetric and transitive relation and a poset $(P, \sqsubseteq)$ is a set P together with a partial order $\sqsubseteq$ over its elements.

Given a poset $(P, \sqsubseteq)$ and a subset of its elements $S \subseteq P$ an element $l \in P$ is called lower bound if $\forall s \in S : l \sqsubseteq s$. The dual notion, an upper bound, is an an element $u \in P$ so that $\forall s \in S : s \sqsubseteq u$. A subset can have many, one or no upper or lower bounds. If an upper bound of a subset S is a lower bound for the set of all upper bounds of S it is called the *least* upper bound or *join* and written $\bigsqcup S$. If a subset has a least upper

bound, it is unique. The dual notion is called the *greatest* lower bound or *meet*, written $\bigsqcap S$.

If a poset contains an upper bound for all its elements it is called the *greatest* element or *top*, written $\top$. The *least* element or *bottom*, written $\bot$ is defined dually.

If every pair of elements of a poset has a join (or dually meet), also written $a \sqcup b$ instead of $\bigsqcup\{a, b\}$, the poset is called a join-semilattice (or meet-semilattice). If a poset is a join- and meet-semilattice it is called a lattice. If the least and greatest element exist, it is called a bounded lattice. The poset together with the binary meet and join operations define an algebraic structure. Both operations are commutative, associative and idempotent. In addition to that, both operations are connected through an absoprtion law $a \sqcup (a \sqcap b) = a$ and $a \sqcap (a \sqcup b) = a$. If the lattice is bounded an additional identity law holds $a \sqcup \bot = a$ and $a \sqcap \top = a$.

For every set S there exists a *power set lattice* $(\mathscr{P}(S), \subseteq)$ with $\cup$ as join and $\cap$ as meet operation. This lattice is bounded and has $\top = S$ and $\bot = \varnothing$.

For every lattice $(P, \sqsubseteq)$ there is an *order dual* lattice $(P, \sqsupseteq)$ with meet and join, $\top$ and $\bot$, etc. exchanged.

If every subset of a lattice has a meet and join it is called a *complete* lattice. This implies boundedness, as the meet and join of all elements are the least and greatest element. Every finite lattice is trivially complete as it is possible to construct the meet or join of a finite subset from the corresponding binary operation.

For the task of SAT solving, we are only concerned with finite lattices, and thus will limit the description of abstract domains to complete lattices.

Given two posets $(P, \sqsubseteq)$ and $(Q, \preceq)$ and a function $f : P \to Q$, we call f *monotone*, *isotone* or *order-preserving* when $\forall x, y \in P : x \sqsubseteq y \Rightarrow f(x) \preceq f(y)$.

Given a poset $(P, \sqsubseteq)$ and an arbitrary set X the functions from X to P have a partial order $\sqsubseteq$, where $f \sqsubseteq g$ iff $\forall x \in X : f(x) \sqsubseteq g(x)$.

A function $f : P \to P$ is called reductive if $f \sqsubseteq \mathrm{id}_P$, i.e. its output is bounded above by the input. A function $g : P \to P$ is called extensive if $\mathrm{id}_P \sqsubseteq g$, i.e. its output is bounded below by the input.

For any fixed $x \in P$ the function $f(y) = y \sqcap x$ is reductive and $g(y) = y \sqcup x$ is extensive.

A function which is monotone, reductive and idempotent is called a *lower closure operator* and a function which is monotone, extensive and idempotent is called an *upper closure operator*.

Given a complete lattice $(P, \sqsubseteq)$ and a closure operator $f : P \to P$ the image of the lattice under the closure operator $(f[P], \sqsubseteq)$ is also a complete lattice [23, Proposition 7.2].

Two posets $(P, \sqsubseteq)$ and $(Q, \preceq)$ with two monotone functions $\alpha : P \to Q$ and $\gamma : Q \to P$ so that $\forall p \in P,\ q \in Q : \alpha(p) \preceq q \Leftrightarrow p \sqsubseteq \gamma(q)$ form a *Galois connection*. This is written $(P, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Q, \preceq)$ where α and γ are called *lower* and *upper adjoint* respectively.

Galois connections can be composed by composing the upper and

lower adjoints, i.e. when $(P, \sqsubseteq) \underset{\alpha_1}{\overset{\gamma_1}{\leftrightarrows}} (Q, \preceq)$ and $(Q, \preceq) \underset{\alpha_2}{\overset{\gamma_2}{\leftrightarrows}} (R, \preceq)$ then $(P, \sqsubseteq) \underset{\alpha_2 \circ \alpha_1}{\overset{\gamma_1 \circ \gamma_2}{\leftrightarrows}} (R, \preceq)$.

For a Galois connection $(P, \sqsubseteq) \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (Q, \preceq)$ the functions $\alpha \circ \gamma : P \to P$ and $\gamma \circ \alpha : Q \to Q$ are respectively lower and upper closure operators [23, Corollary 7.27 on p.160, 18, Corollary 5.3.0.5].

The composition of the lower adjoint with the upper closure operator $\alpha \circ (\gamma \circ \alpha)$ is equal to the lower adjoint α and dually $\gamma \circ (\alpha \circ \gamma) = \gamma$ [23, Lemma 7.26 on p.259].

Given an ordered set $(P, \sqsubseteq)$ and an upper closure operator $f : P \to P$ there is a Galois connection $(f[P], \sqsubseteq) \underset{\mathrm{id}}{\overset{f}{\leftrightarrows}} (P, \sqsubseteq)$ [23, Corollary 7.28 on p.160]. Dually, given a lower closure operator $g : P \to P$ there is a Galois connection $(P, \sqsubseteq) \underset{g}{\overset{\mathrm{id}}{\leftrightarrows}} (g[P], \sqsubseteq)$.

We can use a Galois connection between lattices to define an *abstract domain* [18]. When $(P, \sqsubseteq)$ is a so called *concrete set* that is part of the problem to solve with abstract interpretation, $(Q, \preceq)$ is called the *abstract set* and is usually chosen to have a computationally simpler structure than P. The function $\alpha : P \to Q$ is called the *abstraction function* and the function $\gamma : Q \to P$ the *concretization function*.

These functions can be used to approximate operations on the concrete set using functions on the abstract set. Given a function $f : P \to P$ the function $\hat{f} : Q \to Q$ is a *correct* or *sound* upper approximation iff $\forall x \in Q : \alpha(f(\gamma(x))) \preceq \hat{f}(x)$ [18, Theorem 7.1.0.2] The best or most *precise* approximation is $\alpha \circ f \circ \gamma$, which is monotone [18, Corollary 7.2.0.4].

For a SAT solver an interesting *concrete set* is the power set lattice of boolean assignments $\boldsymbol{C} = (\mathscr{P}(V \to \{\mathtt{t}, \mathtt{f}\}), \subseteq)$. Its elements can represent the set of all assignments satisfying either the complete formula or satisfying a part of it. Directly using this representation to compute the set of all satisfying assignments is equivalent to an exhaustive search of the solution space. What DPLL and the preliminary variant of Stålmarck's method instead use is a partial assignment. Together with the special value bottom $\bot$ this happens to be an abstract domain for the family of sets of boolean assignments, namely the *Cartesian domain*. The abstraction function produces a partial assignment where only those variables are assigned, which have the same value in all concrete assignments. The concretization function returns a set of all assignments where the variables agree to those assigned in the partial assignment. A special case occurs for the empty set of assignments, i.e. a conflicting input formula, which is corresponds to $\bot$ for the abstraction and concretization function.

3.2 *Applying Abstract Interpretation to Stålmarck's Method*

An essential part of Stålmarck's method is the equivalence relation constraining the set of assignments considered in the search for a satisfying assignment. Similarly to the partial assignment, this also is an abstract domain for the concrete domain $\boldsymbol{C} = (\mathscr{P}(V \to \{\mathtt{t}, \mathtt{f}\}), \subseteq)$. For each single assignment of the concrete set there is an equivalence

relation over the literals and the constants `t` and `f` with two equivalence classes containing all literals set to `t` and to `f` respectively. The intersection of all these equivalence relation is the corresponding abstract set. The concrete set for a given equivalence relation contains an assignment for each possible way to map the equivalence classes to $\{\mathtt{t}, \mathtt{f}\}$, so that the equivalence classes containing the constants `t` and `f` are mapped to the contained constant. A special case is when a literal and its negation are in the same equivalence class, which represents a conflict. For the abstract set all conflicting equivalence relations are considered to be equal and to be the bottom element $\bot$.

The observation that an abstract domain is at the core of Stålmarck's method, prompted Thakur and Reps to generalize it to use different abstract domains. For this they describe a corresponding abstract interpretation operation for each step of Stålmarck's method [7].

The simple rules of Stålmarck's method used for the 0-Saturation update the equivalence relation to eliminate more assignments which cannot satisfy the given triples. The corresponding operation on a concrete set would directly eliminate all assignments violating the constraints defined by the triples. This is a lower closure operator and as such can be lifted to a lower closure operator in the abstract domain using the concretization and abstraction functions [24, Theorem 1]. This has to be approximated in an efficiently implementable way. The technique used is *local decreasing iterations*, introduced in [24]. Local decreasing iterations approximate a lower closure operator using a set of reductive operators that approximate the closure operator. These operators are repeatedly applied sequentially until a fixed point or another termination criteria is reached. All elements of this sequence are sound approximations of the closure operator's result [24, Theorem 2]. The set of reductive operators is built from the set of constraints. For each constraint a reductive operator is defined that approximates the effect of the single constraint in isolation.

When an equivalence relation is used as abstract domain and triplets as constraints the usual simple rules would be a suitable set of reductive operators. The iteration until a fixed point is reached is exactly equivalent to 0-Saturation in this case. For the Cartesian domain and a CNF formula the one-literal-rule of the DPLL procedure would fit this framework. Furthermore Thakur and Reps describe a way to mechanically derive reductive operators for arbitrary abstract domains and constraint types by considering only a fixed number of variables simultaneously.

The remaining missing piece is the dilemma rule. The dilemma rule requires splitting the search space into two branches. It also needs a way to extract the common information of both branches. Splitting the search space in the abstract domain is realized by defining an *acceptable splitting set* and computing meets with the elements. An acceptable splitting set contains abstract elements satisfying two conditions. Each splitting element a has a companion b, with $\gamma(a) \cup \gamma(b) = \gamma(\top)$, so that two branches, each with a meet of them, explore the complete search space. The second condition is that for each concrete element which is

a singleton set $X = \{x\}$, there is a set of splitting elements M_x, called the *cover* of x, so that their meet has the singleton set as concretization $\gamma(\bigsqcap M_x) = X$. This ensures that nested application of the splitting rule can explore every single assignment of the search space. While not explicitly stated by Thakur and Reps we also require 0-Saturation of $\bigsqcap M_x$ to return $\bot$ if x is not a valid assignment.

When the abstract domain is as expressive as the Cartesian domain, which means that it has an abstract element for each element of the Cartesian domain so that both have the same concretization, the single variable assignments form an acceptable splitting set.

A dilemma rule split then simply uses a splitting set element for one branch and its companion for the other branch. Each branch then computes the meet with the splitting set element. After the two branches are completed a join operation is used to merge the branches. This approximates a join operation in the concrete domain which would remove exactly the assignments ruled out by both branches.

Where Stålmarck's method iterates over all unassigned variables, the generalized variant accomplishes the same by iterating over all splitting set companion pairs a, b so that $a \not\sqsubseteq A$ and $b \not\sqsubseteq A$ where A is the current abstract domain element. This completes the generalization of Stålmarck's method.

Stålmarck's Method	Generalized Variant
Equivalence relation	Abstract domain element
Simple rules	Sound reductive operators
0-Saturation	Local decreasing iterations
Dilemma rule split	Meet with two splitting set companions in two branches
Dilemma rule intersection	Join of the two branches

Table 3.1: Summary of the generalized variant of Stålmark's method as described by Thakur and Reps

Thakur and Reps have analyzed sufficient conditions for an abstract domain to be suitable for their generalized variant of Stålmarck's method:

1. There exists a Galois connection $\boldsymbol{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \boldsymbol{\mathcal{A}}$ between the concrete set $\boldsymbol{C} = (\mathscr{P}(V \to \{\mathtt{t}, \mathtt{f}\}), \subseteq)$ and the chosen abstract set $\boldsymbol{\mathcal{A}} = (A, \sqsubseteq)$.
2. The expressiveness of the abstract domain surpasses the Cartesian domain, which means that for each x of the Cartesian domain there is a corresponding y of the chosen abstract domain so that $\gamma_{Cart}(x) = \gamma_{\mathcal{A}}(y)$.
3. The join operation of the chosen abstract set can be computed.
4. The meet operation of the chosen abstract set can be computed.
5. There exists an acceptable splitting set.

Algorithm 3.1: Generalized Method of Stålmarck

1: **procedure** Stålmarck(ϕ)
2: $k \leftarrow 0$
3: $A \leftarrow \top$
4: **loop**
5: $A \leftarrow k$-Saturation(ϕ, A)
6: **if** $A = \bot$ **then**
7: **return** `unsat`
8: $k \leftarrow k + 1$
9: **procedure** 0-Saturation(ϕ, A)
10: **reduce** $A \sqsupseteq \alpha(\gamma(A) \cap \{\beta \in V \to \{\mathtt{t}, \mathtt{f}\} \mid \beta \vDash \phi\})$
11: **if** $A \sqsubseteq M_x$ **for some** x **with** $x \vDash \phi$ **then**
12: **return** `sat` **from procedure** Stålmarck
13: **return** A
14: **procedure** $(k+1)$-Saturation(ϕ, A)
15: **repeat**
16: $A' \leftarrow A$
17: **for each companions** $X, Y \in S$ **with** $X \not\sqsupseteq A$ **and** $Y \not\sqsupseteq A$ **do**
18: $A_X \leftarrow k$-Saturation($\phi, A \sqcap X$)
19: $A_Y \leftarrow k$-Saturation($\phi, A \sqcap Y$)
20: $A \leftarrow A_X \sqcup A_Y$
21: **until** $A' = A$
22: **return** A

4
Extending Stålmarck's Method

This chapter presents new work that builds upon the method by Thakur and Reps described in the previous chapter. The goal is to develop further extensions of the method to aid the practical implementation.

4.1 Efficient Abstract Domains for Sets of Assignments

For implementing Stålmarck's method using abstract domains it is important to use a domain with an efficient join operation and a meet operation that is efficient for splitting elements. This gets more complicated for more expressive abstract domains.

For the Cartesian domain even a naive join operation runs in linear time with respect to the number of variables, while a meet with a splitting element takes constant time. This can be improved to linear in the number of newly assigned variables by keeping track of the assigned variables for each branch. Equivalence relations can be handled efficiently using a disjoint set data structure [25] but are already more expensive.

A framework which can guide the efficient implementation of different abstract domains and allows combinations of different domains is desirable. This section derives such a framework suitable for many possible domains. This framework is the basis for the first implementation attempt described in chapter 5 as well as the implementation described in chapter 6.

All abstract domains considered so far share the property that their elements can be represented as a set of constraints, where each constraint limits a small subset of variables to a subset of their possible assignments. Thakur and Reps also give implications (or inequalities) and affine relations of three variables which fit into this pattern as examples. We can generalize this to allow arbitrary sets of constraints.

Let S be the set of all single variable assignments $\{v \equiv b \mid v \in V, b \in \{\mathtt{t}, \mathtt{f}\}\}$ For a given set of constraints $C \supseteq S$ we first consider the order dual of its power set lattice as abstract domain $\boldsymbol{\mathcal{A}} = (\mathscr{P}(C), \sqsubseteq)$ with $\sqsubseteq = \supseteq$. It is necessary to use the inverse of the inclusion order, as more constraints reduce the number of satisfying assignments. This lattice has set union as meet operation and set intersection as join operation.

Lemma 1. *The abstract domain* $\boldsymbol{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathscr{P}(C), \sqsubseteq)$ *with* $C \supseteq S$ *and* S *as*

splitting set fulfills the conditions of the generalized Method of Stålmarck.

Proof. Both lattice operations are simple set operations and can be implemented very efficiently, satisfying conditions 3 and 4. As the single variable assignments are among the allowed constraints the resulting abstract domain generalizes the Cartesian domain and satisfies conditions 2 and 5.

We can also show the existence of a Galois connection $\boldsymbol{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} \boldsymbol{A}$ between the abstract and the concrete set, the remaining condition 1. We define the concretization function γ to produce all assignments satisfying all constraints, that is

$$\gamma(X) = \{y \in \mathscr{P}(V \to \{\mathtt{t}, \mathtt{f}\}) \mid \forall x \in X : x(y)\}.$$

This is equal to the intersection or concrete meet of the sets of all assignments satisfying a single constraint

$$\gamma(X) = \bigcap_{x \in X} \gamma(\{x\}).$$

Also the abstract meet of two abstract elements is the union of their constraints $A \sqcap B = A \cup B$. From this it follows that the concretization function preserves meets, as the concretization of a meet and the meet of concretizations are both the intersection of constraint satisfying assignments of all involved constraints

$$\begin{aligned}\gamma(A \sqcap B) = \gamma(A \cup B) &= \bigcap_{x \in A \cup B} \gamma(\{x\}) \\ &= \bigcap_{x \in A} \gamma(\{x\}) \cap \bigcap_{x \in B} \gamma(\{x\}) = \gamma(A) \cap \gamma(B).\end{aligned}$$

A meet preserving map between complete lattices is the upper adjoint of a uniquely defined Galois connection [23, Proposition 7.34]. The corresponding lower adjoint, which is the matching abstraction function, is the meet of all elements which have a concretization bounded below by the concrete element:

$$\alpha(Y) = \bigsqcap \{X \mid Y \subseteq \gamma(X)\}.$$

In this case it is the union (abstract meet) of all constraint sets which do not rule out any assignment of the concrete element. This is exactly the set of constraints satisfied by all assignments of the concrete element

$$\alpha(Y) = \{x \in C | x(Y)\}. \qquad \square$$

The problem with this approach is that all interaction between constraints is ignored. With equivalences as constraints this would mean that $\{x \equiv y, y \equiv z\} \sqcup \{x \equiv z\} = \varnothing$. This is clearly unsatisfactory. For the examples given by Thakur and Reps they define a meet operation that for each input set adds all constraints implied by the other constraints in the set. We can formalize this for arbitrary constraints by using the image of the previously described abstract lattice under its Galois connection's lower closure operator $(\alpha \circ \gamma[\mathscr{P}(C)], \sqsubseteq)$.

Lemma 2. *The abstract domain* $\boldsymbol{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\alpha \circ \gamma[\mathscr{P}(C)], \sqsubseteq)$ *with* $C \supseteq S$ *and* $S' = \alpha \circ \gamma[S]$ *as the splitting set fulfills the conditions of the generalized Method of Stålmarck.*

Proof. The meet operation is still the intersection of constraints, satisfying condition 4. The join operation requires computation of all implied constraints. While not practical, a possible algorithm is to compute the join in the concrete domain, thus satisfying condition 3. As $\gamma(\alpha \circ \gamma(A)) = \gamma(A)$ the splitting set contains an element corresponding to each element of the Cartesian domain, satisfying conditions 2 and 5.

The Galois connection, needed for the remaining condition 1, is constructed by composing

$$\boldsymbol{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathscr{P}(C), \sqsubseteq)$$

and

$$(\mathscr{P}(C), \sqsubseteq) \underset{\mathrm{id}}{\overset{\alpha\circ\gamma}{\leftrightarrows}} (\alpha \circ \gamma[\mathscr{P}(C)], \sqsubseteq)$$

obtaining

$$\boldsymbol{C} \underset{\mathrm{id}\circ\alpha}{\overset{\gamma\circ(\alpha\circ\gamma)}{\leftrightarrows}} (\alpha \circ \gamma[\mathscr{P}(C)], \sqsubseteq)$$

which can be simplified to

$$\boldsymbol{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\alpha \circ \gamma[\mathscr{P}(C)], \sqsubseteq). \qquad \square$$

Even for some possible two-variable constraints, as in the case of the equivalence relation, computing all implied constraints results in quadratic runtime and storage requirements, suggesting that in general this construction is not practical.

Nevertheless, the power set lattice $(\mathscr{P}(C), \sqsubseteq)$ and its image under the lower closure operator $(\alpha \circ \gamma[\mathscr{P}(C)], \sqsubseteq)$ define a useful upper and lower bound. This allows for a trade off between complexity and expressiveness by applying a function to the inputs of a meet that computes only some implied constraints. Also when a set of complex constraints is equivalent to a set of simpler constraints it can be useful to not only add the simple constraints but also to remove the complex constraints. Arbitrary heuristics, including consideration of the other input set, can be used to guide the generation and removal of constraints.

Algorithm 4.1 presents the resulting method as pseudocode. It is an instantiation of algorithm 3.1 using the power set lattice with the addition of calls for the heuristics that add or remove constraints.

Theorem 1. *The generalized method of Stålmarck with the abstract domain* $\boldsymbol{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\mathscr{P}(C), \sqsubseteq)$ *extended with addition of some implied constraint and removal of some non splitting set constraints is sound and complete.*

Proof. We can prove the soundness of this algorithm by showing that k-Saturation$_{ext}$ of this variant is always bounded below by k-Saturation using the domain $\boldsymbol{C} \underset{\alpha}{\overset{\gamma}{\leftrightarrows}} (\alpha \circ \gamma[\mathscr{P}(C)], \sqsubseteq)$ and the best approximation for 0-Saturation, which by lemma 2 is sound. To avoid confusion we

```
1:  procedure Stålmarck_ext(φ)
2:      k ← 0
3:      A ← {}
4:      loop
5:          A ← k-Saturation(φ, A)
6:          if A = ⊥ then
7:              return unsat
8:          k ← k + 1
9:  procedure 0-Saturation_ext(φ, A)
10:     reduce A ⊆ α(γ(A) ∩ {β ∈ V → {t, f} | β ⊨ φ})
11:     if A ⊇ M_x for some x with x ⊨ φ then
12:         return sat from procedure Stålmarck
13:     return ConstraintHeuristic(A)
14: procedure (k + 1)-Saturation_ext(φ, A)
15:     repeat
16:         A′ ← A
17:         for each companions X, Y ∈ S with X ⊈ A and Y ⊈ A do
18:             A_X ← k-Saturation_ext(φ, A ∪ X)
19:             A_Y ← k-Saturation_ext(φ, A ∪ Y)
20:             A ← MergeHeuristic(A, A_X, A_Y)
21:     until A′ = A
22:     return A
```

Algorithm 4.1: Stålmarck's Method using Constraint Sets with Heuristics for Addition and Removal of Constraints

use $\hat{A}$ for the solver state of k-Saturation and A for the solver state of k-Saturation$_{ext}$.

We can use induction on k prove the soundness of k-Saturation$_{ext}$. For the base case 0-Saturation begins with

$$\hat{A} \leftarrow \alpha(\gamma(\hat{A}) \cap \{\beta \in V \to \{\mathtt{t}, \mathtt{f}\} \mid \beta \vDash \phi\}),$$

as we chose to use the best approximation for the bound. 0-Saturation$_{ext}$ begins with

$$A \subseteq \alpha(\gamma(A) \cap \{\beta \in V \to \{\mathtt{t}, \mathtt{f}\} \mid \beta \vDash \phi\})$$

which in the lattice order is bounded below by 0-Saturation. The procedure 0-Saturation$_{ext}$ also invokes ConstraintHeuristic which can add implied constraints or remove constraints. Removing constraints is extensive and thus cannot violate the lower bound. Adding some implied constraints is bounded below by $\alpha \circ \gamma$ which adds all implied constraints. As $\alpha \circ \gamma$ is idempotent and order preserving and the bound is the result of an application of $\alpha \circ \gamma$ the bound is also valid for the result of ConstraintHeuristic.

For the inductive case we show that after every iteration of the loop in $(k+1)$-Saturation$_{ext}$ the solver state A is bounded below by the solver state of $(k+1)$-Saturation assuming k-Saturation$_{ext}$ is bounded below by k-Saturation. The requirement for MergeHeuristic is to compute $A \cup (A_X \cap A_Y)$ followed by possible addition of implied and removal of constraints. In the case of $(k+1)$-Saturation constraints are never

removed, thus $\hat{A}_X \cap \hat{A}_Y \supseteq \hat{A}$. Together with $\hat{A}_X \sqcup \hat{A}_Y = \alpha \circ \gamma(\hat{A}_X \cap \hat{A}_Y)$ we obtain $\hat{A} = \alpha \circ \gamma(\hat{A} \cup (\hat{A}_X \cap \hat{A}_Y)$. Again, here we use that $\alpha \circ \gamma$ is a lower bound for addition of implied constraints and obtain $\hat{A}$ as a lower bound for A given that $\hat{A}_X$ and $\hat{A}_Y$ are lower bounds for A_X and A_Y.

The completeness of the generalized method of Stålmarck depends only on the elements of the splitting set which ensure that every assignment will eventually be considered. As neither ConstraintHeuristic nor MergeHeuristic are allowed to remove constraints which are part of the splitting set the same argument is still valid for this variant. □

4.2 Nonuniform Depth Search

A problem with Stålmarck's method for difficult problem instances is the exponential increase in solving time with search depth. This is caused by an effectively breadth first exploration of the search space. The other extreme is the depth first exploration approach taken by the DPLL procedure. This leads to the question whether an exploration strategy that lies in between those extremes is possible. The rationale for using such an exploration strategy is the huge potential saving in runtime when a problem instance can be solved in many different ways using $k + 1$ nested applications of the dilemma rule but not using k nested applications.

To answer that, we will first show how a depth first exploration equivalent to DPLL is possible using the framework provided by Stålmarck's method. The key observation is that in case of a conflict in the first branch of the dilemma rule, it is equivalent to the DPLL procedure. When the dilemma rule is applied recursively without any depth limit it will eventually find a solution or reach a conflicting state. This means that ∞-Saturation is equivalent to DPLL.

To implement alternative exploration strategies we can replace the iteratively increased depth limit with a more elaborate heuristic that decides whether to continue applying the dilemma rule or not. Desirable properties of such a heuristic are the following:

- Every application of the dilemma rule explores related areas of the search space.
- The computation needed for each recursive application of the dilemma rule is cheap.
- It is possible to deepen the search with each iteration.

We suggest a framework to implement such a heuristic based on a per-variable search depth. This per variable search depth can be computed by a possibly expensive heuristic as it is only updated at the outermost recursion level. This heuristic is implemented using the NDS-Saturation procedure which takes a depth limit and a mapping of variables to search depths as additional parameter. As a first step it applies 0-Saturation and should the depth limit be zero it returns after that. After that it repeatedly iterates over all variables which have a

depth larger than or equal to the current depth limit and applies the dilemma rule using the current variable until a fixed point is reached. For the branches of the dilemma rule the NDS-Saturation procedure is recursively invoked with a depth limit reduced by one. All variables that come before the current variable in the fixed point iteration step as well as the current variable are removed from the mapping passed to the recursive invocation. This ensures that permutations of the same assignments are only explored once. It is important that the initial mapping contains consecutive depths as otherwise the recursion can terminate without exploring the variables which have a small gap.

Algorithm 4.2: The NDS-Saturation procedure

```
1:  procedure NDS-Saturation(φ, A, d, δ)
2:      A ← 0-Saturation(φ, A)
3:      if d = 0 then
4:          return A
5:      repeat
6:          A′ ← A
7:          δ′ ← δ
8:          for each variable v with (v ≡ t) ∉ A and (v ≡ f) ∉ A do
9:              δ′(v) ← 0
10:             if δ(v) ≥ d then
11:                 A_t ← NDS-Saturation(φ, A ⊓ (v ≡ t), d − 1, δ′)
12:                 A_f ← NDS-Saturation(φ, A ⊓ (v ≡ f), d − 1, δ′)
13:                 A ← A_t ⊔ A_f
14:     until A′ = A
15:     return A
```

5
Binary Decision Diagarams as Constraints

To use the general framework described in the last chapter different abstract domains need to be implemented. Here a trade off between simple domains and more complex domains can be made. On the simple end there is the Cartesian domain, corresponding to partial assignments, and the 2-BAR domain, corresponding to an equivalence relation, as used in the classic variants of Stålmarck's method. On the complex end sets of arbitrary constraints are allowed.

5.1 Processing of BDD Constraints

This chapter explores the possibility of implementing the generalized method of Stålmarck using arbitrary constraints represented as *binary decision diagrams*. Reduced, ordered binary decision diagrams, often just called binary decision diagrams (BDD), are a compact and unique representation of arbitrary multivariate boolean functions [26], [27]. A BDD is a directed acyclic graph where each node has an out-degree of at most 2, thus we can measure the size of a BDD as the number of nodes in the graph. There are polynomial time algorithms for common operations including conjunction, disjunction, symmetric difference, variable abstraction and many more. For the rest of this chapter we will treat BDDs mostly as a black box, as we make use of the cudd library [28] that implements BDDs along with all related data structures and algorithms needed.

A constraint over a set of variables can be represented as a boolean function of the variables that is true iff the variables satisfy the constraint. Such a function can in turn be represented by a BDD.

With BDDs it is possible to directly encode an input formula in CNF as a set of constraints. This means that from the beginning the concretization of the set of constraints is exactly the set of satisfying assignments. As the meet of the current constraint set with a cover of an unsatisfying assignment is bottom in this case, it has the interesting effect that 0-Saturation can be skipped.

Using more complex constraints allows the use of a smaller number of constraints as multiple smaller constraints can be combined by replacing them with their conjunction. This is allowed as it results in an equivalent set of constraints. Combining two constraints with overlapping input variables can make new information available or

even detect a conflict if the constraints were contradictory. Thus combining of constraints should be performed regularly. In general though, the size of BDDs grows exponentially with the number of variables [27, theorem U]. This means that constraints should only be combined up to a certain limit. Without this limit combining the BDDs would already be a complete but inefficient SAT solving procedure and the generalized Stålmarck's method would not be needed.

Constraints that are a single variable assignment are handled as a special case. Fixing an input variable of a BDD can only make the resulting BDD smaller. Thus it makes sense to have a separate partial assignment to keep track of the single variable constraints and to always substitute all assigned variables in all constraints. Again this operation produces a set of constraints that is equivalent to the initial set of constraints and is thus allowed.

The more complex the constraints the less likely it is that two dilemma rule branches produce the same constraint. To still get progress even when no branch was conflicting it is necessary to find common implied constraints. Common implied constraints can be found by computing the disjunction of two constraints, one of each branch. As the amount of possible implied constraints is quadratic, only a subset of them should be generated.

A possible heuristic is to not use constraints as input for the implied constraint generation that were present before the dilemma rule application or are otherwise present in both branches. Also, to avoid generation of increasingly more complex constraints only disjunctions that are not larger than the input constraints should be added.

Algorithm 5.1: Dilemma Rule Merging Heuristic for BDD Constraints

```
1:  procedure MergeHeuristic(A, A_X, A_Y)
2:      A ← A ∪ (A_X ∩ A_Y)
3:      A_X ← A_X \ A
4:      A_Y ← A_Y \ A
5:      for each (c_X, c_Y) ∈ A_X × A_Y with |vars(c_X) ∩ vars(c_Y)| ≥ 1 do
6:          c ← c_X ∨ c_Y
7:          if |c| ≤ max{|c_X|, |c_Y|} then
8:              A ← AddConstraint(A, c)
9:      return A
10: procedure AddConstraint(A, k)
11:     for each c ∈ A with |vars(k) ∩ vars(c)| ≥ 2 do
12:         k' ← k ∧ c
13:         if |k'| ≤ max{|k|, |c|} then
14:             return AddConstraint(A \ {c}, k')
15:     return A ∪ {k}
```

5.2 *Initial Clustering of CNF Clauses*

The usual input format for a SAT solver is a formula in CNF. Thus it is desirable to also support this input format for our implementation.

A naive way to support this would be to turn every CNF clause into a single BDD constraint. The problem with this approach is, that it results in an unnecessarily large number of constraints.

As BDDs allow combining of constraints it is possible to reduce the number of constraints. Typical CNF formulas have a number of clauses from the few hundreds up to millions of clauses, thus an efficient heuristic for the initial merging of constraints is necessary.

To devise such an efficient heuristic, criteria for merging clauses have to be identified. As in general the size of a BDD grows exponentially with the number of input variables it makes sense to merge constraints that have an overlapping set of variables to minimize the growth. Usually for each clause there is a large number of clauses that have an overlapping set of variables. When merging two constraints a new constraint, still overlapping with the other constraints, is created. Thus the choice cannot be made in isolation when the total size of all constraints should be minimized.

For the implementation this was solved by using *divisive hierarchical clustering* [29]. The set of clauses is recursively partitioned into two sets constrained to have a similar size while minimizing the number of shared variables. The result of this process is a *dendrogram*, a tree with the clauses as leaves. It is then possible to merge the clauses along the tree in a bottom up order, stopping when a threshold in BDD complexity is crossed.

While clustering makes only use of the number of variables, the threshold for merging can use a more elaborate complexity measure. The constraints in the final set should be small, measured in BDD nodes, and have few variables. As merging of simple BDDs with many variables can quickly build up large BDDs it makes sense to heavily penalize the number of used variables. A complexity measure that turned out to work well in practice is the number of BDD nodes multiplied with a constant $\alpha > 1$ raised to the number of variables, i.e. $|c| \cdot \alpha^{|vars(c)|}$.

Partitioning of a set of clauses is realized using graph bipartitioning. The set of clauses is represented as a bipartite graph with the clauses and variables as nodes. There is an edge between a variable and a clause exactly when the clause contains the variable. Each node in the graph has two attributes, called *size* and *weight*. Variable nodes have a size of 1 and a weight of 0 while clause nodes have a size of 0 and a weight of 1. The graph partitioning library `metis` [30], version 5.1, is used to partition the graph into two parts. The objective function is chosen to minimize the *total communication volume* which is the sum of sizes of the nodes which have at least one cut edge. In this case it is the number of variables present in both parts. In addition to that the sum of the node weights of both parts is constrained to be about the same, so that the number of clauses is roughly balanced.

5.3 *Performance Problems*

A prototype of this method was implemented. While the conversion from a CNF formula to a set of BDD constraints worked as intended, applying Stålmarck's method to it did not. The prototype was evaluated with a selection from the problems of the SAT Competition 2013 and the "Small, Difficult Satisfiability Benchmarks" suite [31]. Depending on the heuristic used for generating implied constraints two runtime behaviors could be observed. In the first case, when the heuristic generates few or only small constraints, the procedure quickly stops making progress and no new common implied constraints are found when dilemma branches are merged. In this case the method degenerates to an exhaustive search that additionally does expensive computations in each step. In the second case, when the heuristic generates more constraints, the number of constraints, the size of constraints or both grow too fast. This causes a slow down large enough that even for small instances practically no progress is made. Even though it would be possible for a heuristic to exist which does not show either behavior, the attempts made to find such a heuristic were unsuccessful. Profiling showed that most of the time is spent inside the `cudd` library, therefore lack of optimizing the prototype did not cause the slow performance. While there are multiple places in the `cudd` source code where specific optimizations could lead to small improvements of performance for this use case, it does not explain the overall performance. Instead a different approach using simpler constraints was chosen and is described in the following chapter. Nevertheless to allow further analysis or to develop other algorithms based on it the evaluation of the described CNF to set of BDDs conversion is presented below.

5.4 *Evaluation of the CNF to Set of BDDs Conversion*

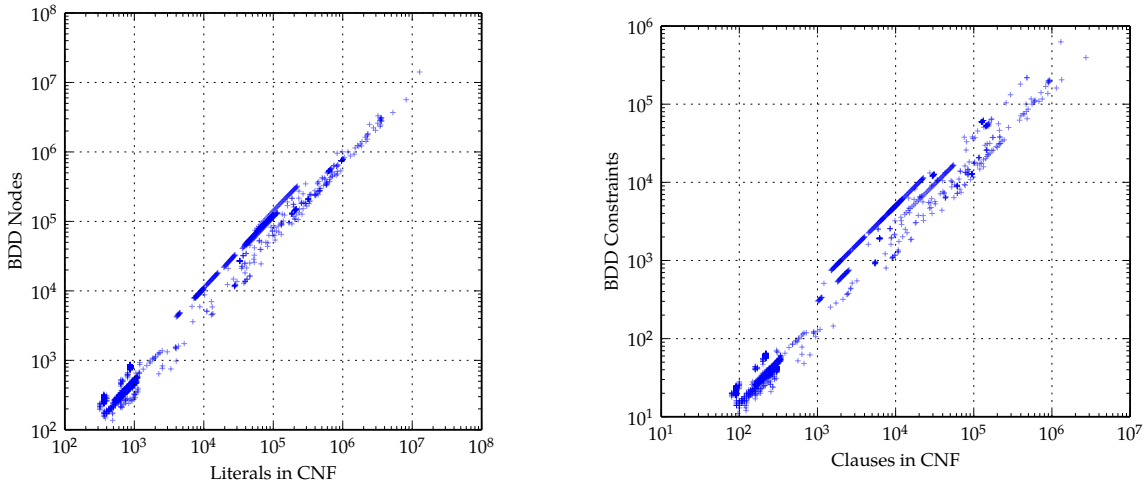


Figure 5.1: Formula Sizes of the CNF to Set of BDDs Conversion

The set of evaluation problems used, again instances from the SAT Competition 2013 and the "Small, Difficult Satisfiability Benchmarks"

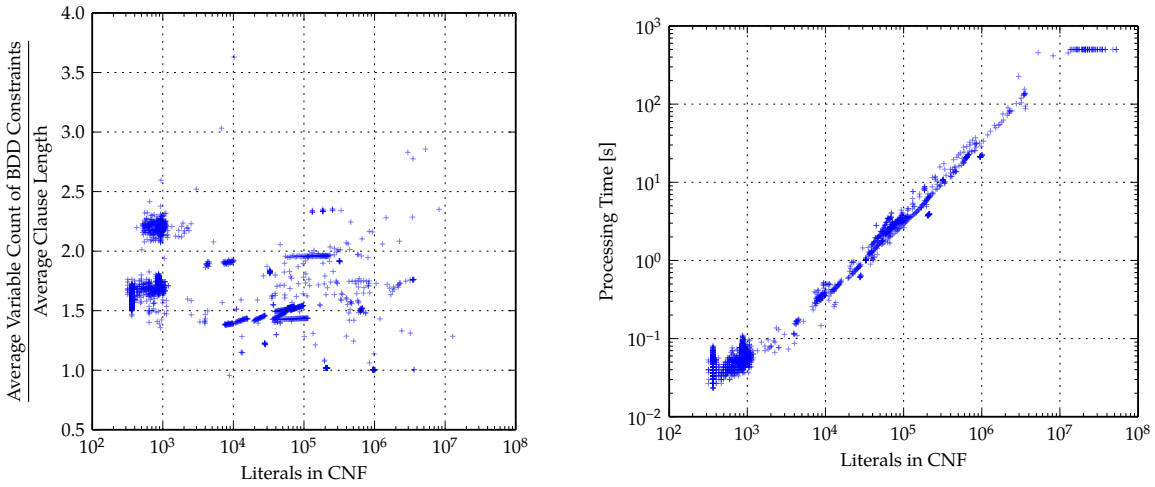


Figure 5.2: Constraint Size and Timing of the CNF to Set of BDDs Conversion

suite [31] contains 4651 CNF formulas. The computer used for the evaluation has an "Intel Core i7-2600K" CPU clocked at 3.40 GHz and is equipped with 16 GiB of DDR3-1600 RAM using 9-9-9 timing. The operating system used is Arch Linux 64-Bit with Linux kernel version 3.13.5. The parameters used are a threshold of 200 and $\alpha = 1.8$. A timeout of 500 seconds per instance was used. 62 instances could not be converted within this timeout. Another 225 instances caused the conversion to exceed hard-coded limitations of the `cudd` library and are thus excluded. The results for the other 4364 instances are shown in figure 5.1 and 5.2.

The diagrams of figure 5.1 shows the change of the instance size that results from the conversion. In the left diagram the size is measured by counting the smallest, constant sized elements of both representations: literals for CNF and nodes for BDDs. It shows that the overall size is only affected by a small constant factor near 1, i.e. stays mostly constant. Whether a slight growth or reduction in size can be observed seems to depend on the kind of instance used. The right diagram measures the size in number of constraints: CNF-clauses and BDDs. Here, again depending on the kind of instance, a reduction by a factor in the range of about 2 up to 10 can be observed. Another important metric is the number of variables for each individual constraint. The less variables a constraint has the more likely a variable's value or a contradiction can be derived from it. The left diagram of figure 5.2 shows the increase in the average number of variables per constraint depending on the problem size. While the increase varies a lot, it is not dependent on the problem size. If the increase would get larger with the problem size it would make the conversion useless beyond a certain problem size. The final diagram shows the time taken for the conversion. Excluding the timeout at 500 seconds and the very small instance sizes it shows a processing time mostly linear in the size of the problem.

This evaluation shows that the conversion is able to satisfy the goals of an efficient and scalable initial reduction in the number of

constraints.

6
Implementation

This chapter describes the implementation of the SAT solver `miplono` based on an extension of Stålmarck's method as described in chapter 4. It implements a nonuniform depth search and uses an abstract domain that is a set of constraints. As customary for SAT solvers, `miplono` processes problem instances in CNF. It also uses CNF as an internal formula representation. Currently it supports variable assignments and equivalences as constraints. The single variable assignments are also used as splitting set. The flexibility in adding implied and removing redundant constraints means it is possible to add constraint types to `miplono` by specifying the interaction with some of the existing constraint types.

6.1 Core Architecture

The core of `miplono` is the recursive search and the associated managing of the current solver state. The dilemma rule makes it necessary to keep the solver state of the first branch while exploring the second branch. Also there can be multiple completed first branches with multiple recursive invocations of the dilemma rule in the corresponding second branches.

A naive implementation would simply duplicate the solver state before computing the meets with the two companion splitting set elements. As the mutations done to the solver state are often small compared to the overall size of the state it is possible to do better by engineering a *persistent* data structure for the state. [32] One approach would be to use *immutable* data structures, which are inherently persistent. A different approach, taken by `miplono`, is to have a mutable active version of the state together with state *differences* or *deltas*. The different versions form a tree where the edges represent the state deltas. Any mutation to the active state also updates the deltas belonging to the edges connected to the active node. The currently active version can be changed by *rewinding* the delta of an incoming edge or by *replaying* the delta of an outgoing edge. Inactive leaf nodes can be dropped along with their incoming edge. In addition to that an inactive node connected with an outgoing edge of the active node can be merged into the active node by replaying the delta of the outgoing edge and recording the changes in the incoming edge.

Besides, during the merging of dilemma rule branches all mutations of the state are done on leaf nodes. This means that only deltas of incoming edges have to be updated. The merging of dilemma rule branches requires mutation of a state which has two leaf children. Nevertheless the updating of deltas of outgoing edges can be avoided as the children are dropped along with the deltas directly after the dilemma rule merge. The only remaining challenge is that the merging requires querying of the two inactive child states. In general the persistence technique used would require activating each state before it can be queried. This would result in poor performance for dilemma rule merging as it would rapidly cycle the active state between the two children and their parent. Fortunately the state deltas can be used to our advantage to implement a dilemma rule merging that is more efficient than it would be in the non-persistent setting. The key observation to this is that the intersection of the child states' constraint sets contains many constraints that are already among the parent state's constraints. Thus only computing the intersection of the newly added constraints can be cheaper. This amounts to rewriting $A \leftarrow A \cup (A_X \cap A_Y)$ to $A \leftarrow A \cup ((A_X \setminus A) \cap (A_Y \setminus A))$ where $A_X \setminus A$ and $A_Y \setminus A$ can be queried directly from the state deltas.

During the recursive search the tree of state deltas always maintains a structure invariant that allows an efficient representation. A non-leaf node either has a single child, belonging to the first branch of the dilemma rule or it has two children where the child belonging to the first branch is a leaf child. This allows storing the state deltas on two stacks where only the top can be accessed. One stack, the *main stack* contains the deltas on the path of the active state to the root, with the root at the bottom. The other stack, the *side stack* contains the deltas to the remaining states, with the delta nearest to the active state at the top. The only remaining information needed to reconstruct the tree, whether a state has a single or two children, is tracked implicitly by the execution flow of the algorithm.

6.2 *Variable Assignments*

The single variable assignment constraints of the active state are stored in a linear array with an entry for each variable of the formula. Each entry has the value `true`, `false` or `unassigned`. The state delta simply stores a list of assigned literals.

The unit-literal rule is used for 0-Saturation. To efficiently perform unit propagation the watched literals technique, introduced by the SAT solver Chaff [14], is used. This technique was developed for the DPLL procedure and is used by most CDCL solvers. The idea behind the technique is to mark two non-false literals of every clause as watched. For every possible literal there is also a list of clauses in which it is watched. Whenever a variable is assigned only the clauses containing the literal that becomes false need to be considered as the other clauses possibly containing that literal have at least two other non-false literals. The affected clauses are then updated to maintain this invariant. New

variable assignments are performed for all clauses that became unit. The implementation of watched literals directly follows the description given in [15].

For a dilemma rule merge a flag is set for all literals stored in the delta of one branch. All literals in the other branch which have the flag set are added to the parent state.

Rewinding simply sets all variables of the delta's literals to unassigned. The watched literal data does not need to be updated as no new literals can become false during rewinding. When a delta is replayed the watched literal data is modified as it would be for any other assignment.

6.3 Nonuniform Depth

The solver implements the nonuniform depth search described in chapter 4. There are many possible ways to construct a heuristic that assigns a depth for each variable. It was observed that exploring some parts of the search space to a larger depth is an improvement mostly independent of which part is explored. Thus for the initial implementation a simple heuristic that explores a random part of the search space was chosen. This is done by selecting a *depth level* for each iteration. This level is increased when no progress was made in the last two iterations and decreased when there was progress for both of the last two iterations. From the depth level a *depth envelope* is computed. This is a sequence of depths, where each element has the same value or is one less than its predecessor. The first element, i.e. the largest depth, increases with the depth level, thereby guaranteeing that eventually the complete search space will be explored. The shape of the envelope is computed in a way that there are many elements with a small depth and only few with a large depth.

The variable depths are then determined by choosing a sequence of variables and assigning the depth of the matching element in the envelope. The first variable is picked randomly from a uniform distribution of all variables. The following variables are picked by a distribution with probabilities proportional to the reciprocal distance in the graph containing clauses and variables, thereby giving higher weight to variables likely to interact with the already chosen variables.

6.4 Equivalences

Equivalence constraints are implemented using a variant of the disjoint set data structure [25]. This data structure represents a partition of a set into equivalence classes as a forest where each class is a tree. Each non-root element points to a parent element and a root element points to itself. The root element is the representative of class and the representative for the class of each element can be found by following the parent pointers. An equivalence can be added that joins two classes into one. The operations update the parent pointers and an additional rank variable in a way that ensures that the representative can be found

efficiently.

As equivalences are between literals and not variables and there is a certain symmetry as $l_1 = l_2$ implies $\neg l_1 = \neg l_2$. To accommodate this, the data structure is slightly modified. The variables are used as set elements but the representative is a literal. The operations of the disjoint set structure are then modified to keep track of negations.

In addition to the parent pointer all elements of a class are kept in a cyclic singly linked list. This is necessary allow efficient iteration over all members of a class. Again the next pointer of a variable is a literal to handle equivalences between a positive and a negative literal. When an equivalence between previously non-equivalent literals is added the linked list can be updated by exchanging the next pointers and possibly negating them.

To allow rewinding all changes to the parent pointer, the next pointer and the rank variable are recorded. To allow replaying all added equivalences between the literals that were previously non-equivalent are also recorded.

There is no direct interaction between the formula and equivalence constraints. Instead equivalence constraints are created during the dilemma rule merge from equivalences implied by the variable assignments and conversely new variable assignments are created from the equivalence constraints when a member of a class is assigned. Equivalences between assigned variables are not explicitly tracked and their disjoint set and linked list data ignored.

During a dilemma rule merge all literals that are assigned true in one branch and false in the other are placed in the same equivalence class. Also the intersection of all new equivalence classes are added as equivalences.

Additionally the dilemma rule merge procedure for single variable assignments is modified to only consider variables that are representatives in the parent state. The result is a possible speedup linear in the size of the equivalence classes. This is valid as any assignment inside a branch would assign the same value to all literals in the class and a single resulting constraint is also enough to assign all these literals in the parent state.

7 Evaluation

This chapter presents an initial evaluation of the SAT solver `miplono`, described in the previous chapter. The evaluation is done on the instances of the "Small, Difficult Satisfiability Benchmarks" suite [31]. This benchmark contains 3608 formulas in CNF. Again, the computer used for the evaluation has an "Intel Core i7-2600K" CPU clocked at 3.40 GHz and is equipped with 16 GiB of DDR3-1600 RAM using 9-9-9 timing. The operating system used is Arch Linux 64-Bit with Linux kernel version 3.13.5. A timeout of 100 seconds per instance is used. For comparision the CDCL solver `minisat` [15] in version 2.2.0 and the lookahead solver `march` [12] in verison `march_dl` (08-03-2005) are used.

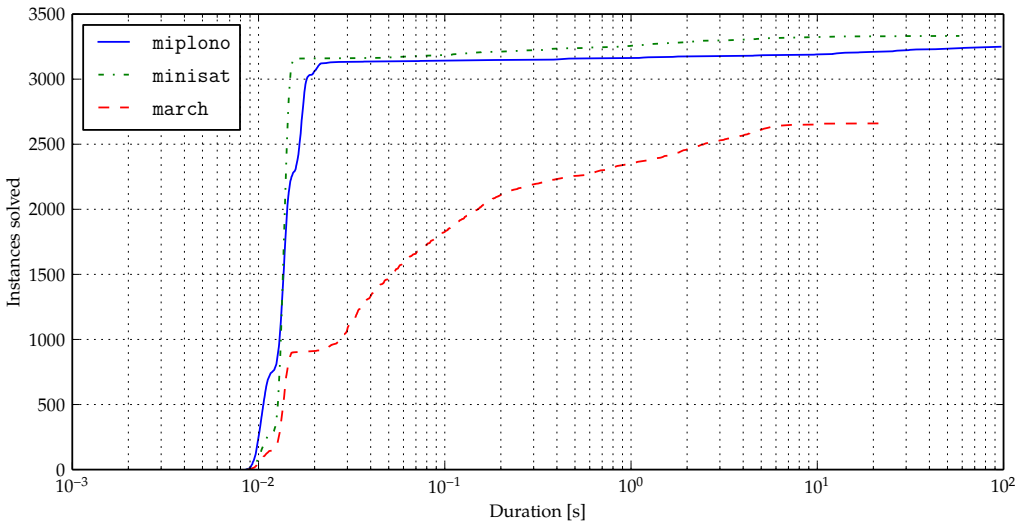


Figure 7.1: Number of instances solved by `miplono`, `minisat` and `march` within a given time limit

The overall performance of each solver can be seen in figure 7.1. It shows the commulative number of instances which individually can be solved within a given time limit for each solver. We can see that `miplono` solves more easy instances at the beginning than `minisat`. After that it slightly falls behind and also solves fewer instances in the total time limit of 100 seconds. The initial head start of `miplono` does not imply an advantage of its method, as a possible explanation is that `miplono` does not do any initial analysis or optimization of the CNF formula,

while minisat does. Nevertheless compared to a state of the art CDCL solver this prototype implementation shows promising performance. Compared to march there is a significant performance advantage and miplono is always ahead.

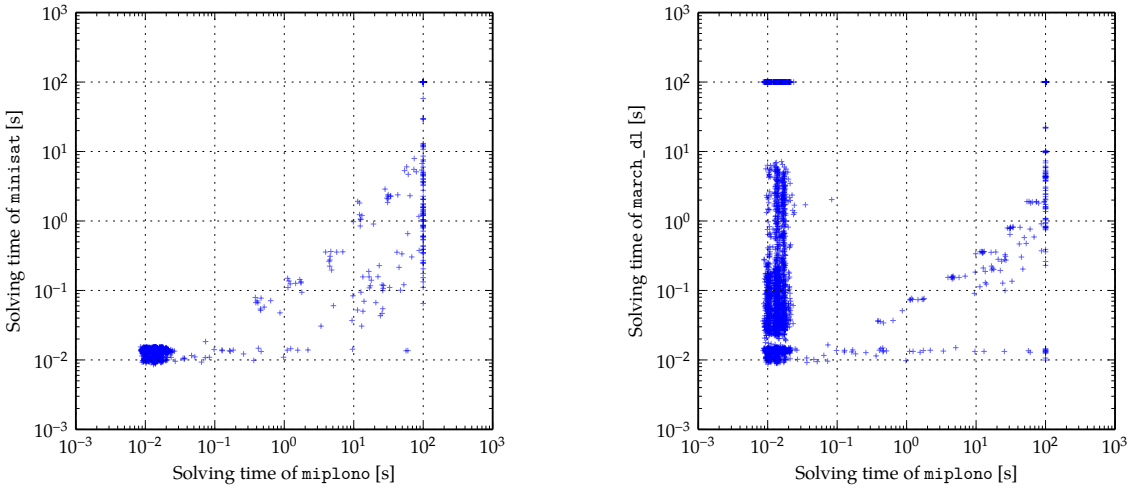


Figure 7.2: Comparision of miplono solving times to minisat and march

The relative performance on a per instance basis can be seen in figure 7.2. Here we can see that, apart from some instances solved very quickly by both, minisat is always faster than miplono. There are instances along the complete range where miplono is only 10 times slower than minisat, i.e. there is no sign that the relative performance gap generally increases with more difficult problems. The factor between the runtimes seems to vary more with the type of the instance than with the difficulty of the problem, indicating that the performance characteristics of this method and CDCL are simply different.

Comparing miplono to march we can see that there is a large number of instances that miplono solves very quickly, that take a long time for march to solve. In contrast, the problems that also take a long time for miplono to solve are solved faster by march. For these problems the performance gap is larger than it was the case in the comparison to minisat. Not only is the performance gap larger, it increases for more difficult problems. This indicates that there is a class of problems where march has superior asymptotic performance compared to miplono, which we did not see in the comparison to minisat.

The data analyzed so far indicates different relative performances dependent on the type of problem. Partitioning the benchmark results by instance type gives a better view of this behavior. The SDSB benchmark contains instances of 14 different types. (The number of instances per type varies.) Figure 7.3 shows a box plot of the solver runtimes for the 14 types and 3 solvers benchmarked and clearly shows different relative performance for the different instance types.

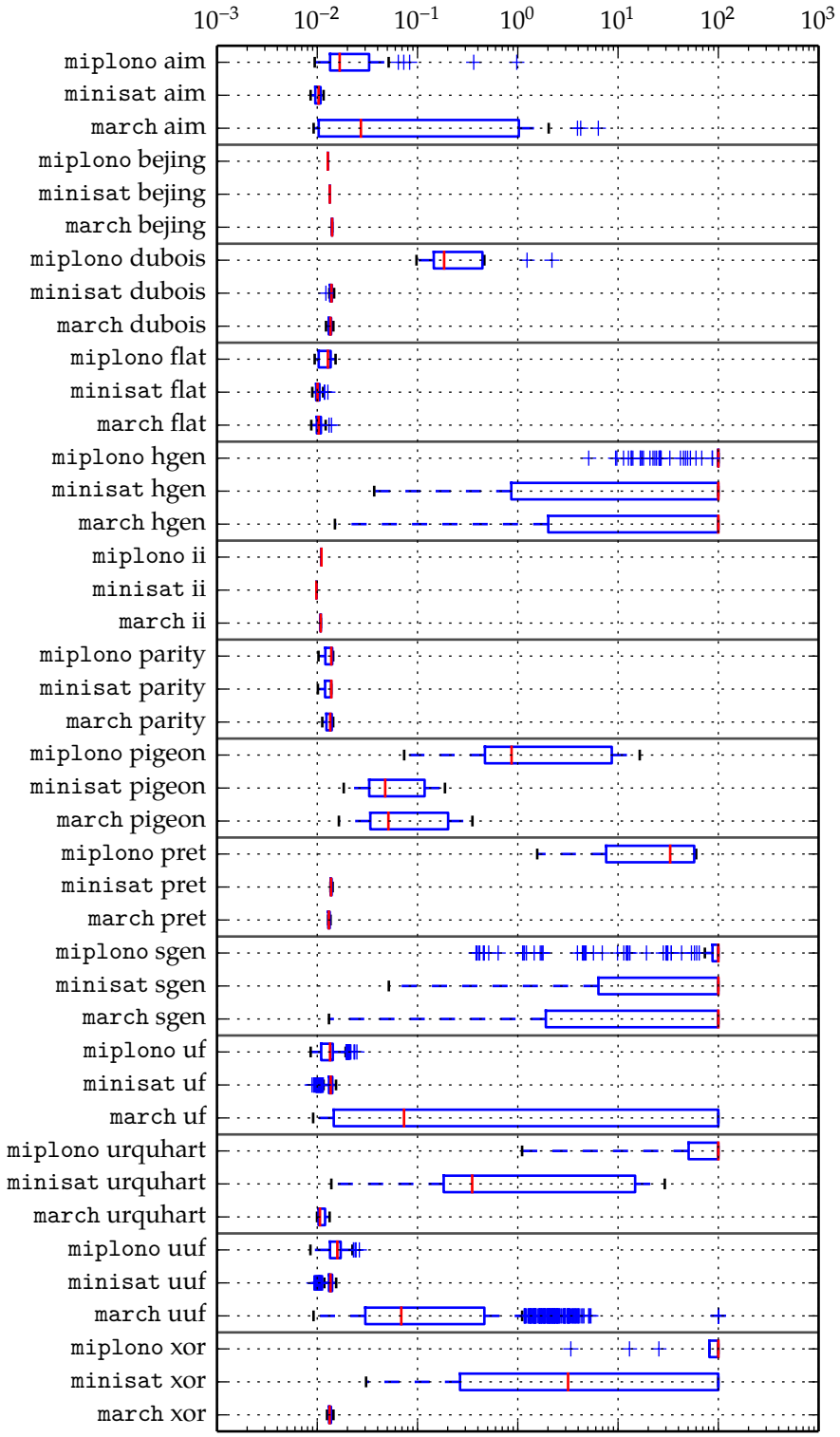


Figure 7.3: Comparison of `miplono`, `minisat` and `march` solving times for the different SDSB benchmarks

8
Conclusion and Outlook

This thesis presented the development of a SAT solving method based on the preceding work by Thakur and Reps [7], [20] which in turn is based on Stålmarck's method. The method extends existing work by providing a framework for using arbitrary constraints and arbitrary heuristic reasoning between constraints as well as a more flexible search strategy. To test the feasibility of the described approach two attempts at a prototype implementation were made. The first attempt uses binary decision diagrams to represent complex constraints. While this attempt was unsuccessful a conversion of CNF formulas to sets of BDDs was developed as part of it and is evaluated. As a second prototype implementation the solver `miplono` was developed which uses simple constraints to avoid the problems of the first prototype.

This implementation consists of a flexible core architecture which can accommodate any kind of constraints representable using persistent data structures supporting certain described operations. The data structures needed for single variable assignments and equivalences between literals are described. A comparison to state of the art methods shows some potential, especially considering the prototype state of the implementation.

There are many areas in which improvement of the method itself as well as on the implementation are possible.

The implementation of data structures used in the prototype is not fully optimized to reduce memory allocation. Performing such optimizations as well as other low level optimizations could possibly improve the solving time by a significant constant factor.

So far only two kind of constraints are implemented. Candidates for other kinds are implications, linear equations, quasi boolean constraints or binary arithmetic. Data structures to represent them and heuristics for constraint addition and removal are necessary for this. The heuristics also need to handle the interaction between different kinds of constraints.

The heuristics used in the prototype were chosen mainly for their simplicity. A careful evaluation and benchmarking of different heuristics, especially when more complex constraints are added, is important.

The dilemma rule provides a natural opportunity for parallelization as it evaluates two branches independently. The core architecture could be extended to handle distribution of work and synchronization of the

solver state by distributing messages containing state deltas.

The use of CNF as formula representation makes a combination of CDCL and this variant of Stålmarck's method possible. Nothing would prevent the addition of learned clauses for the conflicts that naturally occur during the recursive search. Whether this provides an advantage over either method can be evaluated.

The input format can be extended from CNF formulas to a conjunction of constraints other than disjunctive clauses. This can lead to more efficient descriptions of some problem instances which in turn can lead to a faster solving of these instances.

All this also requires a more thorough evaluation performed on a cluster with a larger set of instances and a larger timeout.

We have seen promising initial results of a new variant of Stålmarck's method described in this thesis and outlined several possibilities to improve on it. It remains to be seen if the remaining performance gap to state of the art methods can be closed.

Bibliography

[1] A. Biere, M. Heule, H. van Maaren, and T. Walsh, Eds., *Handbook of Satisfiability*. Amsterdam, Netherlands: IOS Press, 2009.

[2] M. Soos, K. Nohl, and C. Castelluccia, "Extending SAT solvers to cryptographic problems," in *Theory and Applications of Satisfiability Testing - SAT 2009*, ser. Lecture Notes in Computer Science, O. Kullmann, Ed., Springer Berlin Heidelberg, Jan. 2009, pp. 244–257.

[3] L. de Moura and N. Bjørner, "Z3: an efficient smt solver," in *Tools and Algorithms for the Construction and Analysis of Systems*, ser. Lecture Notes in Computer Science, C. R. Ramakrishnan and J. Rehof, Eds., Springer Berlin Heidelberg, Jan. 2008, pp. 337–340.

[4] F. Lin and Y. Zhao, "ASSAT: computing answer sets of a logic program by SAT solvers," *Artificial Intelligence*, vol. 157, no. 1–2, pp. 115–137, Aug. 2004.

[5] "Solver and benchmark descriptions," in *Proceedings of SAT Competition 2013*, A. Balint, A. Belov, M. J. Heule, and M. Järvisalo, Eds.

[6] *Results of the SAT competition 2013*. [Online]. Available: `http://satcompetition.org/2013/results.shtml` (visited on 01/19/2014).

[7] A. Thakur and T. Reps, "A generalization of Stålmarck's method," in *Static Analysis*, ser. Lecture Notes in Computer Science, A. Miné and D. Schmidt, Eds., vol. 7460, Springer Berlin Heidelberg, 2012, pp. 334–351.

[8] M. Davis, G. Logemann, and D. Loveland, "A machine program for theorem-proving," *Commun. ACM*, vol. 5, no. 7, pp. 394–397, Jul. 1962.

[9] M. Davis and H. Putnam, "A computing procedure for quantification theory," *J. ACM*, vol. 7, no. 3, pp. 201–215, Jul. 1960.

[10] J. W. Freeman, "Improvements to propositional satisfiability search algorithms," PhD thesis, University of Pennsylvania, Philadelphia, PA, USA, May 1995.

[11] M. Heule, M. Dufour, J. van Zwieten, and H. van Maaren, "March_eq: Implementing additional reasoning into an efficient look-ahead SAT solver," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, H. H. Hoos and D. G. Mitchell, Eds., Springer Berlin Heidelberg, Jan. 2005, pp. 345–359.

[12] M. J. H. Heule and H. van Maaren, "March_dl: Adding adaptive heuristics and a new branching strategy," *Journal on Satisfiability, Boolean Modeling and Computation*, vol. 2, pp. 47–59, Mar. 2006.

[13] J. P. Marques Silva and K. A. Sakallah, "GRASP—a new search algorithm for satisfiability," in *Proceedings of the 1996 IEEE/ACM international conference on Computer-aided design*, ser. ICCAD '96, Washington, DC, USA: IEEE Computer Society, 1997, pp. 220–227.

[14] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, "Chaff: Engineering an efficient SAT solver," ser. DAC '01, New York, NY, USA: ACM, 2001, pp. 530–535.

[15] N. Eén and N. Sörensson, "An extensible SAT-solver," in *Theory and Applications of Satisfiability Testing*, ser. Lecture Notes in Computer Science, E. Giunchiglia and A. Tacchella, Eds., Springer Berlin Heidelberg, Jan. 2004, pp. 502–518.

[16] G. M. N. Stålmarck, *Method and apparatus for checking propositional logic theorems in system analysis, European Patent EP0403454A1*, 1990.

[17] M. Sheeran and G. Stålmarck, "A tutorial on Stålmarck's proof procedure for propositional logic," *Formal Methods in System Design*, vol. 16, no. 1, pp. 23–58, Jan. 2000.

[18] P. Cousot and R. Cousot, "Systematic design of program analysis frameworks," ser. POPL '79, New York, NY, USA: ACM, 1979, pp. 269–282.

[19] V. D'Silva, L. Haller, and D. Kroening, "Satisfiability solvers are static analysers," in *Static Analysis*, ser. Lecture Notes in Computer Science, A. Miné and D. Schmidt, Eds., Springer Berlin Heidelberg, Jan. 2012, pp. 317–333.

[20] A. Thakur and T. Reps, "A generalization of Stålmarck's method," CS Dept., Univ. of Wisconsin, Madison, WI, Tech. Rep., 2012.

[21] D. A. Plaisted and S. Greenbaum, "A structure-preserving clause form translation," *Journal of Symbolic Computation*, vol. 2, no. 3, pp. 293–304, Sep. 1986.

[22] A. Borälv, "The industrial success of verification tools based on Stålmarck's method," in *Computer Aided Verification*, ser. Lecture Notes in Computer Science, O. Grumberg, Ed., Springer Berlin Heidelberg, Jan. 1997, pp. 7–10.

[23] B. A. Davey and H. A. Priestley, *Introduction to Lattices and Order*, 2nd ed. Cambridge University Press, Apr. 2002, p. 316.

[24] P. Granger, "Improving the results of static analyses of programs by local decreasing iterations," in *Foundations of Software Technology and Theoretical Computer Science*, ser. Lecture Notes in Computer Science, R. Shyamasundar, Ed., Springer Berlin Heidelberg, Jan. 1992, pp. 68–79.

[25] J. E. Hopcroft and J. D. Ullman, "Set merging algorithms," *SIAM Journal on Computing*, vol. 2, no. 4, pp. 294–303, Dec. 1973.

[26] R. E. Bryant, "Graph-based algorithms for boolean function manipulation," *IEEE Transactions on Computers*, vol. 35, no. 8, pp. 677–691, 1986.

[27] D. E. Knuth, *The Art of Computer Programming, Volume 4, Fascicle 1: Bitwise Tricks & Techniques; Binary Decision Diagrams*. Addison-Wesley Professional, 2009.

[28] F. Somenzi, "CUDD: CU decision diagram package-release 2.5.0," *University of Colorado at Boulder*, 2012.

[29] A. K. Jain and R. C. Dubes, *Algorithms for clustering data*. Englewood Cliffs, New Yearsey: Prentice Hall, 1988.

[30] G. Karypis and V. Kumar, "A fast and high quality multilevel scheme for partitioning irregular graphs," *SIAM Journal on Scientific Computing*, vol. 20, no. 1, pp. 359–392, Jan. 1998.

[31] I. Spence, *Small, difficult benchmarks for the satisfiability problem*. [Online]. Available: `http://www.cs.qub.ac.uk/~i.spence/sdsb/` (visited on 03/05/2014).

[32] H. Kaplan, "Persistent data structures," English, in *Handbook of data structures and applications*, D. P. Mehta and S. Sahni, Eds., Boca Raton, Fla.: Chapman & Hall/CRC, 2005, ch. 31.