SAT and CP - Parallelisation and Applications

Thorsten Ehlers

Dissertation
zur Erlangung des akademischen Grades
Doktor der Ingenieurwissenschaften
(Dr.-Ing.)
der Technischen Fakultät
der Christian-Albrechts-Universität zu Kiel
eingereicht im Jahr 2017

Kiel Computer Science Series (KCSS) 2017/3 dated 2017-05-30

URN:NBN urn:nbn:de:gbv:8:1-zs-00000333-a0
ISSN 2193-6781 (print version)
ISSN 2194-6639 (electronic version)

Electronic version, updates, errata available via https://www.informatik.uni-kiel.de/kcss

The author can be contacted via the@informatik.uni-kiel.de

Published by the Department of Computer Science, Kiel University

Dependable Systems Group

Please cite as:

▻ Thorsten Ehlers. *SAT and CP - Parallelisation and Applications* Number 2017/3 in Kiel Computer Science Series. Department of Computer Science, 2017. Dissertation, Faculty of Engineering, Kiel University.

```
@book{DissThorstenEhlers2017,
  author    = {Thorsten Ehlers},
  title     = {SAT and CP - Parallelisation and Applications},
  publisher = {Department of Computer Science, CAU Kiel},
  year      = {2017},
  number    = {2017/3},
  series    = {Kiel Computer Science Series},
  note      = {Dissertation, Faculty of Engineering,
               Kiel University.}
}
```

About this Series

The Kiel Computer Science Series (KCSS) covers dissertations, habilitation theses, lecture notes, textbooks, surveys, collections, handbooks, etc. written at the Department of Computer Science at Kiel University. It was initiated in 2011 to support authors in the dissemination of their work in electronic and printed form, without restricting their rights to their work. The series provides a unified appearance and aims at high-quality typography. The KCSS is an open access series; all series titles are electronically available free of charge at the department's website. In addition, authors are encouraged to make printed copies available at a reasonable price, typically with a print-on-demand service.

Please visit `http://www.informatik.uni-kiel.de/kcss` for more information, for instructions how to publish in the KCSS, and for access to all existing publications.

1. Gutachter:	Prof. Dr. Dirk Nowotka Christian-Albrechts-Universität Kiel
2. Gutachter:	Prof. Dr. Mike Codish Ben Gurion University of the Negev

Datum der mündlichen Prüfung: 22.05.2017

Acknowledgements

I'm rolling thunder pouring rain
I'm coming on like a hurricane
My lightning's flashing across the sky

AC/DC

This work would not have been possible without the help, encouragement, support and advise of many people whom I would like to thank at this place.

First of all I would like to thank my family, especially my parents Manfred and Susanne, for growing me, making me the person I am now, and supporting me through all decisions, especially the hard ones.

Next, I thank my supervisor, Dirk Nowotka, for his guidance, patience, and support, and for providing me the freedom and funding to do research in different directions.

Thanks to my current and former colleagues from the Dependable Systems Group in Kiel, who supported, questioned me. Namely, this is Gesa Walsdorf, Philipp Sieweck, Tim Grebien, Mike Müller, Florin Manea, Robert Mercaş, Maike Bradler, Kamellia Reshadi, Mitja Kulczynski, Danny Poulsen, Joel Day, Pamela Fleischmann, Max Friese, Yvonne Küstermann, Anneke Twardzik and Karoliina Lehtinen.

Furthermore, I would like the members of my examining committee, Mike Codish, Steffen Börm and Manfred Schimmler for their efforts.

I had a great time in Melbourne, Australia, when spending six month at the Melbourne University in 2015. Here, I owe thanks especially to Peter Stuckey for hosting me, the many fruitful discussions we had, and the advise he gave me. This stay surely changed my understanding of constraint programming and SAT solving. A big thanks also to my colleagues there, Graeme Gange, Diego de Uña, Ignasi Abio, Valentin Mayer-Eichberger and Geoffrey Chu. Furthermore, I would like to thank Mark, Sigrid, Oliver, Kylie, Linda, all the guys from the Graduate House and my football gang for the pleasant time.

Science is collaboration; I would like to thank my co-authors that were not mentioned so far, Peter Schneider-Kamp, Luís Cruz-Filipe, Miro Spönemann, Reinhard von Hanxleden, Ulf Rüegg and Johannes Traub.

I also had some fun debugging the software I worked on with Max Bannach and Sebastian Berndt in the field of tree decompositions, hopefully we will manage to publish some results shortly!

I would also like to thank my friends and flatmates, for their support and patience, for example when I was writing a paper while we were renovating the flat.

Last but not least I want to thank Maria for her patience and advice when I felt overstrained by this work, and for just having a good time with me.

Zusammenfassung

Diese Dissertation befasst sich mit der Parallelisierung von Programmen welche eine beliebige, oder eine optimale Lösung zu Problemen suchen, die auf bestimmte, formale Arten spezifiziert werden. Wir beschreiben Parallelisierungsansätze für zwei verschiedene Arten von Lösern, sowie einen Anwendungsfall.

In dem ersten Kapitel beschäftigen wir uns mit SAT, dem Erfüllbarkeitsproblem der Aussagenlogik, und Algorithmen, welche die Erfüllbarkeit oder Unerfüllbarkeit aussagenlogischer Formeln entscheiden. Wir beginnen mit einer kurzen Einführung in Grundlagen der Beweistheorie, welche dann in Bezug zu der Stärke verschiedener algorithmischer Ansätze gesetzt wird. Desweiteren diskutieren wir Implementierungsdetails aktueller SAT Löser, und zeigen Verbesserungen. Zuletzt wird eine Parallelisierung dieser Löser diskutiert, wobei ein Schwerpunkt auf der Kommunikation von Zwischenergebnissen innerhalb eines parallelen Lösers, dem Austausch gelernter Klauseln, liegt.

In dem zweiten Kapitel betrachten wir Constraint Programing (CP) mit Lernmechanismen. Im Gegensatz zu klassische Techniken werden hier Lernmechanismen, wie sie bei SAT Lösern zum Einsatz kommen, übernommen. Wir präsentieren Ergebnisse einer Parallelisierung von CHUFFED, einem lernenden CP Löser. Da dieser sowohl Charakteristiken eines klassischen CP-Lösers als auch eines SAT-Lösers aufweist, ist es nicht klar, welche Parallelisierungsansätze hier am besten funktionieren.

Im letzten Kapitel diskutieren wir Sortiernetzwerke, Sortieralgorithmen deren Vergleichsoperationen a priori, also unabhängig von der Eingabe, festgelegt werden. Aufgrund dieser Datenunabhängigkeit können Sortiernetzwerke effizient parallel implementiert werden. Wir betrachten die Frage nach der minimalen Anzahl von parallelen Sortierschritten, welche für die Sortierung von bestimmten Eingabegrößen benötigt werden, und zeigen untere und obere Schranken für mehrere Fälle.

Abstract

This thesis is considered with the parallelisation of solvers which search for either an arbitrary, or an optimum, solution to a problem stated in some formal way. We discuss the parallelisation of two solvers, and their application in three chapters.

In the first chapter, we consider SAT, the decision problem of propositional logic, and algorithms for showing the satisfiability or unsatisfiability of propositional formulas. We sketch some proof-theoretic foundations which are related to the strength of different algorithmic approaches. Furthermore, we discuss details of the implementations of SAT solvers, and show how to improve upon existing sequential solvers. Lastly, we discuss the parallelisation of these solvers with a focus on clause exchange, the communication of intermediate results within a parallel solver.

The second chapter is concerned with Contraint Programing (CP) with learning. Contrary to classical Constraint Programming techniques, this incorporates learning mechanisms as they are used in the field of SAT solving. We present results from parallelising CHUFFED, a learning CP solver. As this is both a kind of CP and SAT solver, it is not clear which parallelisation approaches work best here.

In the final chapter, we will discuss Sorting networks, which are data oblivious sorting algorithms, i. e., the comparisons they perform do not depend on the input data. Their independence of the input data lends them to parallel implementation. We consider the question how many parallel sorting steps are needed to sort some inputs, and present both lower and upper bounds for several cases.

Contents

List of Figures

List of Tables

List of Algorithms

Chapter 1

Introduction

"It's a dangerous business, Frodo, going out your door. You step onto the road, and if you don't keep your feet, there's no knowing where you might be swept off to."

BILBO BAGGINS

This thesis presents results from three different fields. Still, they are connected to each other. In the first two chapters, we will discuss procedures which automatically create proofs for the satisfiability or unsatisfiability of propositions given in a certain form, propositional formulas in Chapter 2, and constraint satisfaction problems in Chapter 3. In both cases, we will especially consider the parallelisation of these procedures.

Modern algorithms for these satisfiability problems perform a sophisticated version of a backtracking algorithm. Whenever a backtrack has to be done, a learning mechanism is invoked which learns a reason for this backtrack. This no-good is stored and used to prune the search space in subsequent search. Moreover, if the solver proves the formula unsatisfiable, this proof is made of these no-goods. These learning techniques are the foundation of the impressive performance of state-of-the-art solvers, computer programs which decide the satisfiability of such formulas. Besides experimental results, this can also be seen by an analysis of proof systems that these learning techniques can be related to. However, they are also a burden on the parallelisability of these procedures.

In Chapter 2, we will give an overview over techniques used in nowadays SAT solvers, and relate them to proof systems. Understanding these proof systems is beneficial both for understanding the challenges in parallelising

these solvers, and the SAT encoding of some specific problem we discuss in Chapter 4.

Next, we present and discuss the changes we made to the SAT solver GLUCOSE for the SAT Competition 2016. With these changes, our solver won a gold medal in this competition.

Finally, we discuss the parallelisation of SAT solvers. Here, we especially consider the exchange of no-goods recorded during search between SAT solvers running in parallel. On the one hand, it is crucial for the performance of parallel SAT solvers to avoid redundant work, which can be achieved by communicating information about failed search among the parallel solvers. On the other hand, one has to consider the amount of information exchange, especially when scaling to larger numbers of parallel processing units used.

Next, we consider the parallelisation of CHUFFED, a CP solver with learning, in Chapter 3. As this solver is both kind of a SAT solver and a CP solver, it is not clear how to parallelise it. After giving an overview of some ideas and techniques from the field of constraint programming, we compare approaches from the fields of SAT and CP, respectively. When considering optimisation problems, the parallel solver achieves a super-linear speed-up.

Chapter 4 is concerned with sorting networks, a model of data-oblivious sorting algorithms. We will consider sorting networks which are restricted in the sense that the number of sorting steps they perform is bounded. The existence of such networks can be encoded in formulas in propositional logic. We present a technique which takes advantage of symmetries in sorting networks, and derive an optimisation problem which allows for minimising the size of these propositional formulas. Furthermore, we extend encodings used in previous research on this topic by some predicates, which allow for smaller formulas, and reduce the size of proofs created by a SAT solver.

Chapter 2

SAT

"Romanes eunt domus."

BRIAN

The satisfiability problem of propositional logic, SAT, was the first problem proven to be NP-complete [77]. On the one hand, this implies that it is computationally challenging, and intractable unless $P = NP$. On the other hand, many problems from NP can be encoded as propositional formulas, which makes SAT an interesting foundation.

In the past two decades, SAT solvers, i. e., procedures which decide the satisfiability of a formula in propositional logic, have become extremely fast for a broad range of formulas, which in turn has attracted interest in this field.

One of the first applications was the verification of hardware. In Bounded Model Checking [46], LTL formulas describing some execution steps of the hardware under consideration are encoded as a SAT formula. Contrary to previous approaches based on BDDs, this allows for using state-of-the-art SAT solvers [47]. This technique has proven extremely useful, and won the prize for being the "most influential paper in the first 20 years of TACAS" 2014. Another technique for verifying properties of transition systems like hardware is IC3 ("Incremental Construction of Inductive Clauses for Indubitable Correctness") [54], based on SAT encodings.

Analogously, SAT solvers are used as core technology in software verification. The tool Blast [42] searches for execution paths which lead to some error location, a part of the code that should be unreachable. The feasibility of these paths is encoded in SAT: if the formula is satisfiable, the location is reachable, and the SAT-certificate contains input values that lead to this error. A similar approach is chosen in KLEE [60], which uses a SAT solver to generate inputs for a program which make the program execute along some path. These inputs

are then used as test cases, yielding good code coverages. These techniques are used both in academia and in software companies [26]. Furthermore, SAT is used for the analysis of termination of programs [146].

In cryptanalysis, SAT solvers were used to check the security of cryptographic procedures. While they cannot prove the security, successful attacks on a cryptographic procedure would imply that it is unsafe. In [158], SAT solvers are used to do the "laborious" work in attacks on MD4 and MD5. CryptominiSAT, developed by Mate Soos [201], is a SAT solver which is tuned towards cryptographic problems, and extends normal SAT encodings to XOR constraints.

Lately, SAT solvers have been used to compute lower bounds for some combinatorial problems, among them Ramsey numbers [75], the pythagorean triples problem [120], and bounds on some properties of sorting networks [56, 70]. The last case will be handled in more detail in Chapter 4.

Lastly, SAT solvers are used as foundation of solvers for more complex logic. In "SAT modulo theories" (SMT), boolean variables are connected with a meaning with respect to some background theory. The DPLL(T)-framework uses SAT solvers as core technology, and checks consistency with the background theory lazily [167]. This framework is used in state-of-the-art SMT solvers like Z3 [163], BARCELOGIC [48] or CVC4 [31]. Additionally, clause learning CP solvers like CHUFFED [62] use an architecture which is similar to DPLL(T). We will discuss the parallelisation of such solvers in Chapter 3.

In this chapter, we will discuss the satisfiability problem of propositional logic and algorithms to decide it. As some concepts are also relevant for the subsequent chapters, the foundations explained here are more complete than actually required for this chapter. We present some formalisms first, and then discuss proof systems for deciding the unsatisfiability of a propositional formula, especially with a focus on the size of the proofs they produce. This is relevant as SAT solvers — software programs, which create proofs according to some proof system — are limited by lower bounds of the respective proof system: A solver implementing a weak proof systems cannot yield short proofs.

In this context, we also show changes made to GLUCOSE [19] in the SAT competition 2016, with which we won the gold medal as best GLUCOSE hack [96].

We then discuss the parallelisation of SAT solvers, especially with a focus on the exchange of information inside a parallelised SAT solver. These results were published in [97].

2.1 Preliminaries

SAT is the common abbreviation for the satisfiability problem of propositional logic. It is also called Boolean Logic, after George Boole, who formalised its main concepts in 1847 [52]. Propositional logic deals with boolean propositions — they are either true (**1**), or false (**0**). Boolean variables are such that may take exactly one of these values. Propositional formulas are built from boolean variables and constants, and connections between them [59]. It is sufficient to consider the logical "OR" and negation, as we will see shortly. The "OR", written as $\vee$, is defined as

$$\mathbf{0} \vee \mathbf{0} = \mathbf{0}$$
$$\mathbf{0} \vee \mathbf{1} = \mathbf{1}$$
$$\mathbf{1} \vee \mathbf{0} = \mathbf{1}$$
$$\mathbf{1} \vee \mathbf{1} = \mathbf{1}$$

As can be seen by this definition, the logical "OR" is commutative. The negation is represented by the symbol $\neg$, and defined as

$$\neg \mathbf{1} = \mathbf{0}$$
$$\neg \mathbf{0} = \mathbf{1}$$

A literal is either a variable or its negation.

2.1.1 Propositional Formulas

Definition 2.1 (Propositional formula)**.** The set of formulas in propositional logic, F_{PL}, can be defined recursively as

$\mathbf{1} \in F_{PL}$	
$x \in F_{PL}$	If x is a boolean variable
$(f) \in F_{PL}$	$f \in F_{PL}$
$\neg f \in F_{PL}$	$f \in F_{PL}$
$f \vee g \in F_{PL}$	$f, g \in F_{PL}$

It is common to use further connectives for "AND" $\wedge$, implications $\Rightarrow$,

equivalences $\Leftrightarrow$ or "XOR" $\oplus$. These can be defined by the above definitions as

$$\begin{aligned} \mathbf{0} &= \neg \mathbf{1} \\ f \wedge g &= \neg(\neg f \vee \neg g) \\ f \Rightarrow g &= \neg f \vee g \\ f \Leftrightarrow g &= (f \Rightarrow g) \wedge (g \Rightarrow f) \\ f \oplus g &= \neg(f \Leftrightarrow g) \end{aligned}$$

A conjunction of propositional formulas is their connection by $\wedge$, and their disjunction is the connection by $\vee$. The definition of $\wedge$ by $\vee$ and $\neg$ is one of De Morgan's laws [85].

As an example for propositional encodings, consider the node colouring problem on graphs.

Problem 2.2 (Node colouring). Let $G = (V, E)$ denote a graph, and $c \in \mathbb{N}$. The node colouring problem asks for a mapping $\xi : V \mapsto [1, c] \cap \mathbb{N}$ such that each pair of adjacent nodes is coloured with different colours, i. e.,

$$\xi(u) \neq \xi(v) \qquad \forall \{u, v\} \in E$$

Node colouring is one of Karp's 21 NP-complete problems [131], but it has also practical applications. For example, consider the problem of assigning frequencies to radio stations such that stations which are close to each other do not interfere [161], which can be modeled as a node colouring problem: Radio stations are represented by nodes in the graph. For every pair of radio stations that must not use the same frequency, the respective nodes are connected by an edge. Then, the colours represent the frequencies used.

Node colouring can be encoded as a SAT problem as follows.

Example 2.3. We may encode the proposition $\xi(v) = i$ by a boolean variable x_i^v. In this encoding, we encode the implicit constraint from the problem definition that ξ is a function: Each node has to be coloured with exactly one colour, which has to be encoded explicitly. Furthermore, we add the colouring-constraint that each two adjacent nodes must be coloured with different colours, and yield the following encoding.

$$\begin{aligned} \neg x_i^v \vee \neg x_j^v \qquad & \forall 1 \leqslant i \leqslant j \leqslant c, v \in V \\ \neg x_i^u \vee \neg x_i^v \qquad & \forall 1 \leqslant i \leqslant c, \{u, v\} \in E \end{aligned}$$

$$\bigvee_{i=1}^{c} x_i^v \qquad \forall v \in V$$

The first set of clauses forbids that a node v is coloured both with different colours i and j. The second set forbids colouring adjacent nodes u and v with the same colour, whereas the last clauses encode that each node has to be coloured with at least one colour. □

2.1.2 Satisfiability of Propositional Formulas

Given a propositional formula ϕ and an assignment of its variables to true or false, we may ask for the evaluation of the formula under this assignment. The variables of a formula can be defined as follows.

Definition 2.4 (Variables of a formula)**.** The variables of a formula are

$$\begin{aligned} \text{vars}(\mathbf{1}) &= \{\} \\ \text{vars}(x) &= \{x\} \\ \text{vars}((f)) &= \text{vars}(f) \\ \text{vars}(\neg f) &= \text{vars}(f) \\ \text{vars}(f \vee g) &= \text{vars}(f) \cup \text{vars}(g) \end{aligned}$$

Let $\beta : \text{vars}(\phi) \mapsto \{\mathbf{1}, \mathbf{0}\}$ denote a mapping of the variables of some formula ϕ to true and false values. With this, we evaluate the truth value of ϕ.

Definition 2.5 (Evaluation of a propositional formula)**.** Let f denote a function which evaluates a formula ϕ and an assignment to the variables of ϕ to either **1** or **0**. It can be defined as follows.

$$\begin{aligned} f(\mathbf{1}, \beta) &= \mathbf{1} \\ f(x, \beta) &= \beta(x) \\ f((\phi), \beta) &= f(\phi, \beta) \\ f(\neg\phi, \beta) &= \neg f(\phi, \beta) \\ f(\phi \vee \phi', \beta) &= f(\phi, \beta) \vee f(\phi', \beta) \end{aligned}$$

Definition 2.6 (Satisfiability of a propositional formula)**.** A propositional formula ϕ is satisfiable if and only if there exists an assignment β such that $f(\phi, \beta) = \mathbf{1}$.

A formula is called unsatisfiable if it is not satisfiable, i.e. there is no satisfying assignment. The negation of an unsatisfiable formula, which is satisfied by all variable assignments, is denoted a tautology.

Let ϕ and ψ denote propositional formulas, $\beta : \text{vars}(\phi) \cup \text{vars}(\psi) \mapsto \{\mathbf{1}, \mathbf{0}\}$ and f an evaluation function. ϕ entails ψ, written $\phi \models \psi$, iff for every β it holds that if β satisfies ϕ, it also satisfies ψ, i. e., $f(\phi, \beta) \Rightarrow f(\psi, \beta)$. The formulas are equivalent, denoted $\phi \equiv \psi$, iff $\phi \models \psi$ and $\psi \models \phi$. They are equisatisfiable if either both, or none are satisfiable [155].

2.1.3 Conjunctive Normal Form

There are different normal forms for SAT formulas, as NNF, DNF [141] or DDNF[82]. Most SAT solvers expect the input formula to be in conjunctive normal form (CNF).

Definition 2.7 (CNF)**.** A propositional formula ϕ in conjunctive normal form is a conjunction over disjunctions of (possibly negated) boolean variables. These disjunctions are denoted as clauses, and $|c_i|$ denotes the number of literals in a clause c_i. Furthermore, $|\phi|$ denotes the number of clauses in ϕ.

$$\begin{aligned}\phi &= \bigwedge_{i=1}^{n} c_i \\ &= \bigwedge_{i=1}^{n} \bigvee_{j=1}^{|c_i|} l_{i,j} | l_{i,j} \in \{x, \neg x\}, x \in \text{vars}(\phi)\end{aligned}$$

This restriction does not restrict the power of these solvers, as every propositional formula can be transformed into a formula in CNF [205]. The so-called Tseitin Transformation considers the input formula as a boolean circuit, and introduces auxiliary variables for the outputs of each gate. These are bound to the correct value by a constant number of clauses for each gate — thus, the size of the resulting formula is linear in the size of the input formula. We exemplify this procedure in the following example.

Example 2.8. Consider the formula

$$\phi = ((A \wedge B) \vee C) \wedge (\neg A \vee (\neg B \wedge D))$$

We represent the formula as a boolean circuit, and introduce new variables $x_1, \ldots, x_5$ for the gate outputs, as shown in Figure 2.1. Note that a satisfying

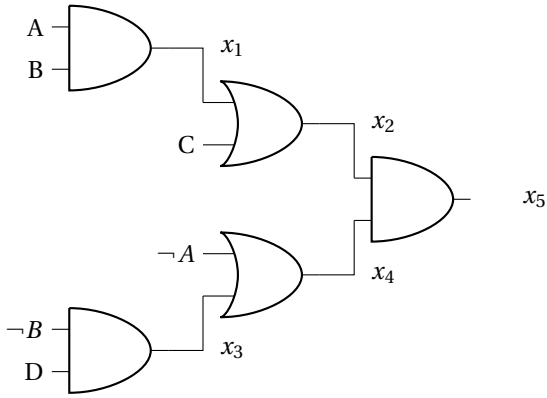


Figure 2.1. The formula $((A \wedge B) \vee C) \wedge (\neg A \vee (\neg B \wedge D))$, represented as boolean circuit.

assignment for ϕ must satisfy the newly introduced variable x_5.

The output variables $x_1, \ldots, x_5$ are defined by the following equivalences.

$$\begin{aligned} x_1 &\leftrightarrow (A \wedge B) \\ x_2 &\leftrightarrow (x_1 \vee C) \\ x_3 &\leftrightarrow (\neg B \wedge D) \\ x_4 &\leftrightarrow (\neg A \vee x_3) \\ x_5 &\leftrightarrow (x_2 \wedge x_4) \end{aligned}$$

These equivalences can easily be transformed into CNF by replacing each bi-implication by two implications, and representing these by disjunctions. The resulting CNF consists of 16 clauses.

$$\begin{aligned} (\neg x_1 \vee A) \qquad & (\neg x_1 \vee B) \\ (x_1 \vee \neg A \vee \neg B) \qquad & (x_2 \vee \neg x_1) \\ (x_2 \vee \neg C) \qquad & (\neg x_2 \vee x_1 \vee C) \\ (\neg x_3 \vee \neg B) \qquad & (\neg x_3 \vee D) \\ (x_3 \vee B \vee \neg D) \qquad & (\neg x_4 \vee \neg A \vee x_3) \\ (x_4 \vee A) \qquad & (x_4 \vee \neg x_3) \\ (\neg x_5 \vee x_2) \qquad & (\neg x_5 \vee x_4) \end{aligned}$$

$$(x_5 \vee \neg x_2 \vee \neg x_4) \qquad\qquad (x_5)$$

□

Subsumption of clauses is a useful notion when dealing with formulas in CNF. Consider a formula ϕ containing clauses c_1 and c_2. The clause c_1 subsumes clause c_2 if all its literals are contained in c_2. Abusing notation we may write $c_1 \subseteq c_2$. Every satisfying assignment for ϕ must assign at least one literal in c_1 to **1**, which also satisfies c_2. Thus, c_2 may be removed from ϕ without changing the satisfiability.

2.2 Proofs and Complexity

In this section, we consider procedures to prove the satisfiability, or unsatisfiability, of a given propositional formula. Furthermore, we discuss the structure of proofs especially for the unsatisfiability of SAT formulas. Determining the satisfiability of a propositional formula, or SAT in short, is obviously in NP: Given an assignment of its variables to **1** or **0**, it is easily verifiable in polynomial time if it satisfies the formula, or not. Conversely, proving the unsatisfiability of a SAT formula is more involved. We will discuss different proof systems in Section 2.2.5 and relate them to the algorithms we discuss before.

We first introduce some notation, and then discuss algorithms and proof systems for SAT. We will restrict ourselves to formulas in CNF for this. It is common to handle such formulas as a set of sets of literals. Thus, given a clause c, we simply write $c \in \phi$ to denote that c is a conjunct of ϕ. As common, we define the empty conjunction as true, and the empty disjunction as false.

$$\bigwedge \bigvee \varnothing = \mathbf{1}$$
$$\bigwedge \bigvee \{\varnothing, \ldots\} = \mathbf{0}$$

In the following algorithms, we will often need the set of clauses that contain some literal ℓ. As as shorthand, we will denote this set by

$$\phi_\ell = \{c \in \phi : \ell \in c\}$$

If a literal is assigned to true, it satisfies all clauses containing it. All clauses which contain its negation must be satisfied by other literals. The residual

Algorithm 1: Enumeration of all assignments

```
Data: Propositional formula φ in CNF
Result: 1 if φ is satisfiable, and 0 otherwise
for β : vars(φ) ↦ {0,1}^n do
    if f(φ,β) = 1 then
        return 1;
return 0;
```

formula with respect to some partial assignment can be defined as follows.

Definition 2.9 (Formulas under partial assignment)**.** For a formula ϕ, $x \in \text{vars}(\phi), \ell \in \{x, \neg x\}$, let $\phi_{|\ell}$ denote the residual formula with ℓ set to true.

$$\phi_{|\ell} = \left(\phi \backslash (\phi_\ell \cup \phi_{\neg\ell})\right) \cup \{c : c \cup \{\ell\} \in \phi\}$$

This is, all clauses containing ℓ are removed, as they are satisfied, and $\neg\ell$ is removed from all clauses containing it as they cannot be satisfied by it.

The naïve approach to determining the satisfiability of a formula on n variables is the enumeration of all 2^n possible assignments, as depicted in Algorithm 1.

This algorithm can be seen as the deterministic version of the behaviour of a non-deterministic Turing Machine which guesses assignments, and checks if they satisfy the formula. It can easily be implemented to run in $\mathcal{O}(2^n|\phi|)$ time. Next, we will discuss some algorithms that cannot be proven to be faster on all formulas, but perform better in practice. Afterwards, we consider the underlying proof systems.

2.2.1 Resolution

Resolution [84, 187] is the underlying principle of most SAT solving algorithms. It is an inference system which only knows the rule

$$\frac{(C \vee x) \qquad (D \vee \neg x)}{(C \vee D)}$$

The intuition behind this rule is that in a satisfying assignment, x is either set to true, and hence D must be satisfied, or x is set to false, and C must be satisfied.

A SAT formula is unsatisfiable if and only if the empty clause can be derived by repeated applications of this rule. This is, there is a sequence of clauses $c_0, \ldots, c_k$ such that every clause is derived from clauses either from the original formula, or a clause derived before, ending with the empty clause.

We will first consider its application in SAT solving algorithms. Afterwards, we discuss the strength of different restrictions of resolution, and relate these restricted versions to different algorithms.

2.2.2 The DP Algorithm

The DP algorithm was developed by Davis and Putnam in 1960 [84]. It is based on the resolution principle, and was intended to improve the speed of proofs for theorems given in propositional logic. Therefore, they extended the resolution algorithm by further rules to speed up the computation. These rules seek to simplify the formula as much as possible and reduce the blowup during the resolution proof.

The "Rule for the Elimination of One-Literal Clauses" considers clauses of length 1. If a formula contains such a clause $c = (l_i)$ — also called unit clause — l_i must be assigned **1** in every satisfying assignment. Thus, all clauses containing l_i can be removed, as in every satisfying assignment, they would be satisfied by l_i. Clauses containing $\neg l_i$ cannot be satisfied by this literal, hence, it can be removed. This step can be seen as the removal of all clauses containing l_i except for c, as they are subsumed by c. Then, resolving on the variable of l_i yields the same result. The repeated application of this rule is denoted Unit Propagation (UP) or Boolean Constraint Propagation (BCP).

If a variable occurs only positive (or only negative) in the formula, it may safely be set to **1** (to **0**), thus all clauses containing it may be removed. Again, this step basically applies subsumption.

If none of these rules apply, a variable is chosen, and eliminated by pairwise resolving all clauses containing its positive and negative literal, respectively. For efficiency reasons, one may discard tautological clauses derived in this step, and remove subsumed clauses. The drawback of this resolution is its memory requirement, as the intermediate formula may become exponentially large.

Algorithm 2: The DP algorithm

Data: Propositional formula ϕ in CNF
Result: **1** if ϕ is satisfiable, and **0** otherwise

```
1  if φ = ∅ then
2      return 1;
3  if ∅ ∈ φ then
4      return 0;
   /* Unit Propagation                                   */
5  if {ℓ} ∈ φ then
6      return DP(φ_|ℓ);
   /* Pure Literal                                       */
7  if ∃ℓ.φ_¬ℓ = ∅ then
8      return DP(φ_|ℓ);
9  Choose Variable x ∈ vars(φ);
10 φ' := φ ∪ {(c1 ∪ c2)\{x, ¬x}, c1 ∈ φ_x, c2 ∈ φ_¬x}\(φ_x ∪ φ_¬x);
11 return DP(φ');
```

2.2.3 The DPLL Algorithm

The DPLL algorithm, developed by and named after Davis, Putnam, Logemann and Loveland, was developed to overcome the exponential blowup of the DP algorithm [83]. Therefore, the resolution step of the DP algorithm is replaced by backtracking. If no leaf of the search tree has been reached, and neither the unit literal rule nor the pure literal rule apply, the algorithm picks a variable from the formula, and branches on it. The pseudocode shown in Algorithm 3 only produces the answers SAT and UNSAT, respectively. It can be extended to produce a satisfying assignment — if such exists — by keeping track of the variable assignments during backtracking. Most deterministic SAT algorithms that are used nowadays are extensions of the DPLL algorithm, only the pure literal rule is hardly used.

Example 2.10. Consider the formula

$$\phi = \{\{\neg x_1, x_2\}, \{\neg x_1, x_3\}, \{\neg x_2, \neg x_3, x_4\},$$
$$\{x_4, x_5\}, \{x_4, \neg x_5\}, \{\neg x_4, x_5\}, \{\neg x_4, \neg x_5\}\}$$

Algorithm 3: The DPLL algorithm

Data: Propositional formula ϕ in CNF
Result: **1** if ϕ is satisfiable, and **0** otherwise

```
1  if φ = ∅ then
2      return 1;
3  if ∅ ∈ φ then
4      return 0;
   /* Unit Propagation                                   */
5  if {ℓ} ∈ φ then
6      return DPLL(φ_{|ℓ=1});
   /* Pure Literal                                       */
7  if ∃ℓ.φ_{¬ℓ} = ∅ then
8      return DPLL(φ_{|ℓ});
9  Choose Variable x ∈ vars(φ);
10 if DPLL(|φ_{x=1}) = 1 then
11     return 1;
12 else
13     return DPLL(|φ_{x=0});
```

Branching on $x_1 = \mathbf{1}$, we get the residual formula

$$\phi_{x_1} = \{\{x_2\}, \{x_3\}, \{\neg x_2, \neg x_3, x_4\},$$
$$\{x_4, x_5\}, \{x_4, \neg x_5\}, \{\neg x_4, x_5\}, \{\neg x_4, \neg x_5\}\}$$

Applying the unit clause rule repeatedly sets x_2 and x_3 to **1**, which yields a new unit clause x_4. Another step of unit propagation ends with the unit clauses x_5 and $\neg x_5$, which is a contradiction. Hence, the algorithm backtracks and branches on $\neg x_1$. The search tree for a run of the DPLL algorithm is shown in Figure 2.2, where the branching order x_1, x_2, x_3, x_4 is used. The nodes are labelled with the respective branching variables. If unit propagation yields a contradiction, this is marked by $\Rightarrow_{UP} \mathbf{0}$. □

Example 2.10 shows the application of the DPLL algorithm on a formula. Here, its main weakness becomes obvious: The formula in the example is unsatisfiable due to the clauses $\{x_4, x_5\}, \{x_4, \neg x_5\}, \{\neg x_4, x_5\}, \{\neg x_4, \neg x_5\}$, a so-called

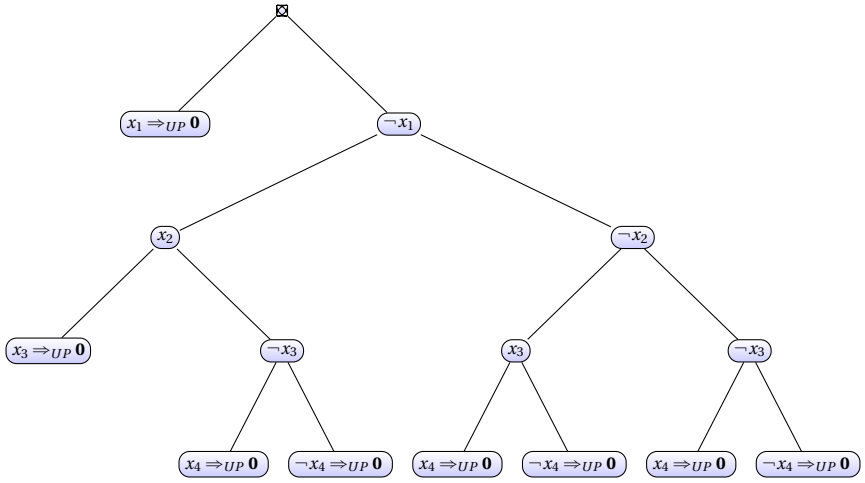


Figure 2.2. Example for the search tree of a run of the DPLL algorithm.

unsatisfiable core. With the chosen variable ordering, the DPLL algorithm is not capable of taking advantage of this fact. Instead, the search visits similar parts of the search space several times.

2.2.4 CDCL

In 1999, Marques-Silva and Sakallah published GRASP (Generic seaRch Algorithm for the Satisfiability Problem) [196]. It seeks to overcome the weaknesses of the DPLL algorithm. Assume that the DPLL algorithm derives the empty clause after branching and unit propagating. It will then backtrack and try another variable assignment, but it will not re-use any information about the failed search. The idea of conflict driven clause learning (CDCL) is to derive a no-good which explains why the search failed. This no-good can then be used to prune the search space, and prevent the solver from running into similar conflicts in other parts of the search space.

In case the search algorithm finds a leaf of the search tree which yields an empty clause, the reason for this conflict is analysed. To do this, the algorithm keeps track of all unit propagations that occurred during search, and the clauses causing them. In case a conflict occurs, i.e. all literals of one clause — the

Algorithm 4: Conflict clause generation

Data: Propositional formula ϕ in CNF, implication graph, clause in conflict c
Result: backjump level ℓ, learned clause cl

```
/* Initialisation                                          */
```
1 cl := c;
2 ℓ := 0;
3 **while** *cl contains more than one literal assigned at current decision level* **do**
4 $\quad$ ℓ := literal from cl that was assigned last;
5 $\quad$ reason := clause that propagated $\neg\ell$;
6 $\quad$ v := var(ℓ);
7 $\quad$ cl := resolve(cl, reason, v);
8 ℓ := second highest decision level in cl;

conflict clause — are assigned to false, the reason for this situation is compiled into a new clause as shown in Algorithm 4.

We will first discuss the algorithm in terms of an example, and then discuss properties of the derived clauses.

Example 2.11. Consider the formula ϕ which consists of the following clauses.

$c_1 = (\neg a \vee d)$ $\quad$ $c_2 = (\neg b \vee e)$
$c_3 = (\neg c \vee f)$ $\quad$ $c_4 = (\neg a \vee \neg b \vee \neg f \vee m)$
$c_5 = (\neg m \vee \neg n)$ $\quad$ $c_6 = (\neg d \vee \neg e \vee \neg g \vee n)$
$c_7 = (\neg e \vee \neg f \vee g)$

Assume the solver branches on a. As this is the first decision, the assignment and everything implied by it are said to be on decision level 1. The first branch implies d, as the clause c_1 becomes a unit clause under this partial assignment. Similarly, branching on b on decision level 2 implies e. The third branch, c, triggers several propagations, and yields a conflict, as both n and $\neg n$ are implied.

Figure 2.3 shows an implication graph which describes both the decisions and their implications. This is a directed acyclic graph where nodes denote variable assignments, and edges between nodes are labelled with the clauses that caused propagations. This representation shows that branching on f instead of c on the third decision level would have led to the same conflict. The

node labelled with f is called a unique implication point (UIP) [196], as it is on all paths from the decision on the last decision level, c, to the conflicting assignments.

Here, we explain the derivation of a learned clause in terms of resolution, which fits the implementation of current SAT solvers [89].

We therefore resolve the clauses which implied n and $\neg n$.

$$\frac{(\neg m \vee \neg n) \qquad (\neg d \vee \neg e \vee \neg g \vee n)}{(\neg d \vee \neg e \vee \neg g \vee \neg m)}$$

This clause still contains the literals g and m which were assigned at the last decision level. We resolve it with the reason for m:

$$\frac{(\neg d \vee \neg e \vee \neg g \vee \neg m) \qquad (\neg a \vee \neg b \vee \neg f \vee m)}{(\neg a \vee \neg b \vee \neg d \vee \neg e \vee \neg f \vee \neg g)}$$

Now there are still two literals in the resolved clause, f and g, which were assigned at decision level 3. Another resolution step with c_7, the clause which implied g, yields

$$\frac{(\neg a \vee \neg b \vee \neg d \vee \neg e \vee \neg f \vee \neg g) \qquad (\neg e \vee \neg f \vee g)}{(\neg a \vee \neg b \vee \neg d \vee \neg e \vee \neg f)}$$

This clause was derived by resolution, therefore it is logically implied by ϕ, and we can safely add it to ϕ. □

Clauses learned by the conflict analysis have some important properties.

- ▻ All of their literals are assigned to **0** in the actual solver state, as they are resolved from a conflict. This can be seen by an inductive argument: In the conflicting clause, all literals are assigned to **0** by definition. When this clause is resolved with the clause that caused the latest unit propagation, all literals but one from this clause are assigned to **0**, as otherwise no unit propagation would have been triggered. The only literal which is assigned to **1** is the one that is deleted in the resolution step.

- ▻ They were derived by resolution, thus, they can be added to the input formula without changing its satisfiability.

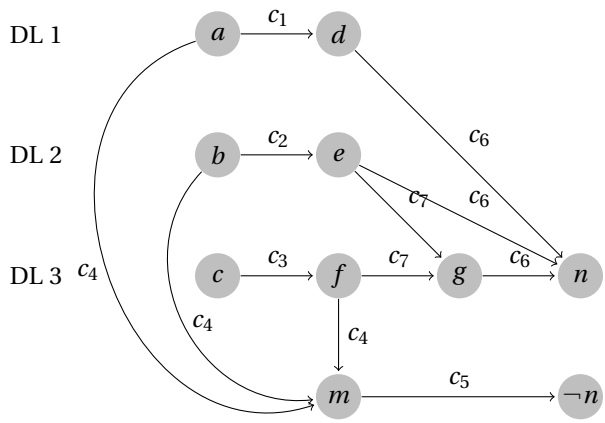


Figure 2.3. Implication Graph after branching a in Example 2.11. The edge labels denote the clauses involved in the propagation. For example, m is propagated by c_4 as a, b, and f are assigned **1**.

- They contain exactly one literal which has been assigned at the highest decision level. If the size of the learned clause equals 1, the solver can backtrack to decision level 0 by undoing all decisions and their implications, and propagate this unit clause. Otherwise, the solver backtracks to the second highest decision level at which a variable from the clause was assigned. By construction, all but one literal are assigned to **0** on this decision level. Thus, the solver may continue by propagating the unassigned literal, denoted the asserting literal, from this clause instead of branching, as depicted in Algorithm 5.
- They contain the 1UIP, which is the UIP closest to the conflict. It has empirically been found that using this clause is the best choice with respect to the solver performance [216].

The whole CDCL algorithm is shown in Algorithm 5. Although this algorithm does not appear too complicated, it is highly interesting both from a practical and theoretical perspective. There exists a plethora of literature on efficient implementations and improvements of the original algorithm, which

Algorithm 5: The CDCL algorithm

```
   Data: Propositional formula φ in CNF
   Result: 1 if φ is satisfiable, and 0 otherwise
   /* Initialisation                                          */
1  DL=0;
2  while true do
3      if propagate() = Conflict then
4          if D L =0 then
5              return 0;
6          else
7              (newDL, learned clause) := analyse_conflict();
8              Add learned clause;
9              backtrack(newDL);
10             DL := newDL;
11     else
12         if Free variables remaining then
13             ℓ := find_branching_literal();
14             DL := DL+1;
15             assign ℓ = 1;
16         else
               /* All variables are assigned without a conflict  */
17             return 1;
```

will be discussed in Section 2.3. When run on unsatisfiable formulas, CDCL-based SAT solvers create a proof of unsatisfiability based on the clauses they learn during search, i. e., a resolution proof. This clause learning process is not directed, as in the DP algorithm, it is rather guided by the search. The next section will present some foundations for the structure of proofs generated in this way. Afterwards, details concerning the implementation of propagation and branching are explained, which were only sketched in the above presentation.

2.2.5 Underlying Proof Systems

In this section, we will consider proof systems for propositional logic. This is of interest for several reasons. Firstly, analysing the behaviour of an actual solver implementation is hard due to the heuristics used in it. Therefore, it is useful to relate the proofs generated by a solver to some proof system $\mathcal{S}$: If it can be shown that proofs generated by the solver on some formula ϕ correspond to proofs in $\mathcal{S}$, then the proof generated by the solver cannot be smaller than the smallest proofs derivable in $\mathcal{S}$.

Secondly, analysing the weaknesses of such a proof system may yield insights in weaknesses of solver implementations. This topic will also be relevant in Chapter 3.

Fourthly, it is a fundamental problem in computer science. If there was a system that generates proofs for the unsatisfiability of a formula such that the size of these proofs is polynomially bounded by the size of the input formula, and they could be verified by a deterministic Turing machine in polynomial time, this would imply that NP is closed under complementation, and therefore that $NP = coNP$ [78].

Formally [34], a proof system is a polynomial-time computable predicate $\mathcal{S}$ such that

$$F \in TAUT \leftrightarrow \exists p.\mathcal{S}(F,p),$$

i. e., F is tautological if and only if there is a proof p that is recognised by $\mathcal{S}$. In the literature, the notion of proof systems is sometimes used interchangeably with inference systems that can be used to create a proof [33, 34, 50].

This section will be concerned with the proofs of unsatisfiability for a propositional formula, which implies that its negation is tautological; thus, the above notation is also applicable.

We will discuss only proof systems based on resolution, as these form the foundation of the proofs generated by DP, DPLL or CDCL SAT solvers.

Comparison of proof systems

When discussing the relative strength of proof systems, it is crucial to have a notion of whether one proof system is stronger than another. The first definition gives a notion that one proof system is at least as strong as another one.

Definition 2.12 (p-simulation [209])**.** Given two proof systems $\mathcal{S}_1$ and $\mathcal{S}_2$, $\mathcal{S}_1$ p-simulates $\mathcal{S}_2$ if there exists a polynomial-time computable function f such that for every unsatisfiable formula ϕ and proof p,

$$\mathcal{S}_1(\phi, f(p)) \leftrightarrow \mathcal{S}_2(\phi, p).$$

Given a resolution refutation, there are different metrics to estimate its complexity like the number of symbols required to encode it into a string [79], or its width which is defined as the maximum size of a clause in it [38]. Here, we will consider the complexity in terms of resolution steps performed.

Definition 2.13 (Complexity of a resolution refutation [113])**.** Let $\Gamma = c_0, \ldots, c_k$ denote a resolution refutation of a formula $\phi = \bigwedge_{i=0}^{n} c_i$. Then, the complexity of Γ is the number of clauses generated for the refutation i. e., $k - n$.

With this notion of the complexity of a proof, proof systems can be compared to each other. A common tool is to consider families of formulas which are easy to prove for one, but hard to prove for another proof system.

Definition 2.14 (Separation of proof systems [6])**.** Let $\mathcal{F}$ denote a family of unsatisfiable SAT formulas, and $\mathcal{S}_1$, $\mathcal{S}_2$ be two proof systems such that $\mathcal{S}_1$ p-simulates $\mathcal{S}_2$. Furthermore, let $p_1(\phi)$ ($p_2(\phi)$) denote a shortest proof for some formula $\phi \in \mathcal{F}$ accepted by $\mathcal{S}_1$ ($\mathcal{S}_2$). If there exists a function f such that $|p_1(\phi)| \leqslant f(|p_2(\phi)|)$ for all $\phi \in \mathcal{F}$, then f separates $\mathcal{S}_1$ from $\mathcal{S}_2$. If f is exponential, then there is an exponential separation between $\mathcal{S}_1$ and $\mathcal{S}_2$.

Given this notion, we discuss the relative strength of different restricted versions of resolution, and relate them to algorithms which yield proofs in the respective proof system.

In order to characterise these different resolution refinements, it is helpful to describe the resolution proof as a directed, acyclic graph [58]. The nodes are labelled with clauses derived in the proof and clauses from the input formula. For each clause derived by resolution, there are two edges pointing to the clauses that it was derived from. These edges are labeled with the variable that was removed in the respective resolution step.

General Resolution

General resolution proofs are sometimes also called unrestricted [109]: Here, the proof is just a sequence of clauses, beginning with the original formula,

such that every clause generated for this proof is the resolvent of two clauses which either belong to the original formula, or have been derived before. Haken proved in 1985 that there is a family of formulas for which the shortest general resolution proof has exponential size [113]. These formulas are often referred to as "pigeonhole principle", as they seek for an one-to-one mapping of $n+1$ pigeons to n holes. Alternatively, this can be seen as the question for a n-colouring of a complete graph K_{n+1}. It has been shown that clause learning SAT solver are as strong as general resolution [181].

Accordingly, every SAT solver based on the CDCL algorithm requires an exponential running time on the pigeon hole formulas.

Next, we will consider restricted forms of resolution refutations, discuss their respective strengths, and relate them to the DPLL and DP algorithm, respectively.

Tree Resolution

In tree-like resolution refutations, the DAG of resolutions steps without the clauses of the original formula is a tree [37]. This kind of resolution refutations is sometimes called Davis-Logemann-Loveland, as it is closely related to refutations given by the DPLL algorithm. Each propagation performed during a run of the DPLL algorithm can be explained in terms of a resolution step, as depicted in Example 2.15. Thus, a tree-like resolution proof can be generated from the run of the DPLL algorithm [37]. Hence, the running-time of the DPLL algorithm is lower-bounded by the shortest refutation in tree-like resolution.

Example 2.15. Consider the following, unsatisfiable formula on 4 variables.

$$\begin{aligned}(\neg a \vee b) \wedge (a \vee \neg b) &\qquad \wedge (\neg b \vee c)\\ (b \vee \neg c) \wedge (\neg c \vee d) &\qquad \wedge (\neg c \vee \neg d)\\ (c \vee d) \wedge (c \vee \neg d)&\end{aligned}$$

Branching on a yields the propagations $a \Rightarrow b$, $b \Rightarrow c$, $c \Rightarrow d$ and $c \Rightarrow \neg d$, which is a contradiction. Backtracking and branching on $\neg a$ yields $\neg a \Rightarrow \neg b$, $\neg b \Rightarrow \neg c$, $\neg c \Rightarrow d$ and $\neg c \Rightarrow \neg d$, a contradiction. Figure 2.4 shows a tree-like resolution proof which follows the branching of the DPLL algorithm. For example, consider the right branch starting at the root of the tree, labeled with $\varnothing$, which derives the clause (a). Here, (a) is derived from (b) and $(a \vee \neg b)$,

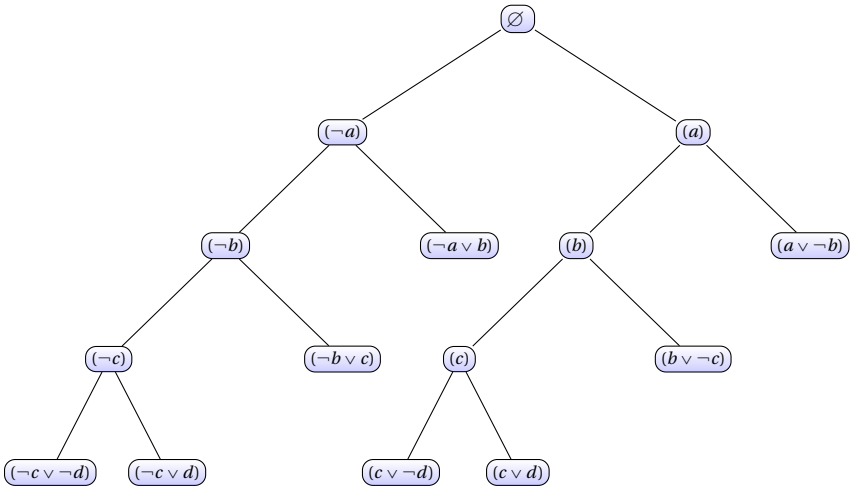


Figure 2.4. Tree-like resolution proof represented as DAG

which can be related to the unit propagation $a \Rightarrow b$ in the run of the DPLL algorithm. □

The weakness of tree-like resolution refutations can be seen analogously to the weakness of the DPLL algorithm, which cannot learn from failed search in one part of the search space, and therefore repeatedly performs similar work. Tree-like resolutions may not re-use clauses from one part of the derivation tree in another part.

There are families of formulas for which tree-like resolution proofs have size $2^{\Omega\left(\frac{n}{\log n}\right)}$, where n denotes the size of the input formula. On the contrary, there exist general resolution proofs of size $\mathcal{O}(n)$ for these formulas [37]. As every tree-like resolution proof is also a general resolution proof, general resolution p-simulates tree-like resolution, hence general resolution is strictly stronger than tree-like resolution.

We will reconsider this gap between the respective proof systems in Chapter 3, as backtracking algorithms for CP solving have the same weakness as the DPLL algorithm, and are thus outperformed by clause learning solvers.

Regular Resolution

Regular resolution proofs (REG) are a subset of general resolution proofs. Here, on every path in the respective DAG which starts from the empty clause, every variable may appear at most once as label of an arc.

This can be further restricted to ordered resolution proofs [108], where the variables occurring on paths starting from the empty clause are ordered according to the same ordering. These proofs are also called Davis-Putnam, as they correspond to proofs generated by the DP algorithm. There exists a family of formulas for which proofs in ordered resolution have size $n^{\mathcal{O}(\log(\log n))}$, whereas general resolution proofs of size $\mathcal{O}(n^4)$ exist [108]. This result was improved by Bonet et al., and later Urquhart, who gave an exponential separation between regular and general resolution [50, 208]. As ordered resolution proofs are also regular, this implies an exponential separation between ordered and general resolution, which may be seen as a theoretical explanation for the better performance of CDCL based SAT solvers, compared to the DP algorithm.

Extended Resolution

Contrary to the previously described resolution refinements, extended resolution (ER) extends general resolution. It is based on the idea by Tseitin of introducing new variables, and relating them to formulas built from variables of the input formula [205], denoted the extension rule.

Example 2.16. Let ϕ denote a SAT formula such that $x, y \in vars(\phi)$, and $z \notin vars(\phi)$. Then, one may define

$$z \leftrightarrow (x \vee y)$$

which can be expressed by the clauses

$$(\neg x \vee z)$$
$$(\neg y \vee z)$$
$$(\neg z \vee x \vee y)$$

□

Formally, an extended resolution proof for some formula ϕ is a general resolution proof for ϕ', where ϕ' is obtained from ϕ by repeated application of the extension rule.

Interestingly, this yields a stronger proof system. Intuitively, this corresponds to the size of SAT encodings: By the Tseitin encoding, every propositional formula can be encoded in conjunctive normal form without asymptotically increasing its size by introducing new variables, whereas the encoding of simple formulas may become exponentially large if no new variables are used. In resolution proofs without the extension rule, only clauses are derived, but no addition of new variables is allowed.

Cook proved that there exist polynomial size proofs for the pigeon hole principle using extended resolution [76]. However, if there were polynomial size extended resolution refutations for every unsatisfiable SAT formula, this would imply $NP = coNP$ [50]. Thus, there has been work both in deepening the understanding of extended resolution [142] and in proving super-polynomial lower bounds on the proof size [206].

The strength of ER has attracted interest in using it in SAT solvers. Audemard et al. modified their SAT solver GLUCOSE, allowing it to apply the extension rule [14]. While successful on some formulas, it appears hard to decide when to apply the extension rule in general. Furthermore, the solver has to be forced to preferably use the newly introduced literals in propagations and conflict analysis.

Example 2.17. Consider a formula ϕ which contains the constraint $(a \vee b) \Rightarrow (c \wedge d)$ represented by the clauses $(\neg a \vee c)$, $(\neg a \vee d)$, $(\neg b \vee d)$ and $(\neg b \vee d)$. One may seek to introduce a new variable x which indicates that independently of whether a or b are set to true, this implies c and d. x can be defined by terms of the extension rule $(\neg x \vee a \vee b)$, $(\neg a \vee x)$ and $(\neg b \vee x)$. Then, resolution yields

$$\dfrac{(\neg b \vee x) \qquad \dfrac{(\neg a \vee c) \qquad (\neg x \vee a \vee b)}{(\neg x \vee b \vee c)}}{(\neg x \vee c)}$$

Equivalently, $x \Rightarrow d$ can be derived. Once a clause learning solver detects a conflict during its search, it may use x in the newly learned clause rather than a or b, which is a more general no-good. However, propagation is typically implemented as a breadth-first search [89], thus, setting a or b to $\mathbf{1}$ will propagate c and d without considering x, a problem that was encountered in [14]. □

Chu et al. used problem-specific knowledge to find good definitions for which they applied extended resolution in [65].

Furthermore, it has been suggested to use the extension rule as inspiration of a preprocessing step, denoted by Bounded Variable Addition (BVA) [154].

2.3 Techniques & Implementations

We will now consider techniques which were presented in seminal papers, and are used in practically all modern SAT solvers. These techniques consider learning clauses, efficient data structure for propagations, branching, restarting and the deletion of learned clauses.

2.3.1 Preprocessing via Bounded Variable Elimination

The DP algorithm is hardly used anymore, mainly due to the large number of resolvents it produces. However, it is noteworthy that many SAT solvers run it for some iterations as a preprocessing step called bounded variable elimination (BVE) [88, 204].

Let

$$\begin{aligned}\phi' &= res(\phi, x)\\ &= (\phi \backslash \phi_x) \cup \left\{c' = c_1 \cup c_2 \backslash \{x, \neg x\} | x \in c_1, \neg x \in c_2, \text{c' is no tautology}\right\}\end{aligned}$$

denote the formula obtained by adding all non-tautologous clauses generated by resolving on x, and removing all clauses containing x or $\neg x$.

Let ϕ denote a SAT formula in CNF, and $\phi_\ell, \phi_{\neg\ell}$ denote the clauses containing ℓ and $\neg\ell$ for some literal ℓ. As in the DP algorithm, all resolvents $c = c_1 \cup c_2 \backslash \{\ell, \neg\ell\}$ for $c_1 \in \phi_\ell$ and $c_2 \in \phi_{\neg\ell}$ are computed and added to ϕ unless they are tautologies. In this way, some variables can be eliminated from the input formula, which yields an equisatisfiable formula.

As the DP algorithm is only used as a preprocessor here, the blow-up of the formula size can be avoided: If the number of newly created clauses exceeds the number of clauses that may be removed by a fixed value, the variable is not removed. Interestingly, on some industrial formulas many of the resolved clauses are tautologies and thus do not have to be added.

The resulting formula is equisatisfiable to the input formula. Let ϕ denote the input formula, and $\phi' = res(\phi, x)$ for some variable x. Furthermore, assume that ϕ' is satisfiable, and let β' denote a satisfying assignment for ϕ'. A satis-

Algorithm 6: Bounded Variable Elimination

Data: Propositional formula ϕ in CNF, integer k
Result: An equisatisfiable formula

```
if φ = ∅ then
    return φ;
if ∅ ∈ φ then
    return φ;
if {ℓ} ∈ F then
    return BVE(φ_{|ℓ=1});
if ∃ℓ∀c ∈ φ.ℓ ∉ c then
    return BVE(φ\{c|ℓ ∈ c});
Choose Variable x ∈ vars(φ) s.t. |res(φ, x)| ⩽ |φ| + k;
if Such variable exists then
    return BVE(res(φ, k));
else
    return φ;
```

fying assignment β for ϕ can be derived from β' efficiently, if during BVE the eliminated clauses were stored. Assume that $(C \vee D)$ was derived by resolving $(C \vee x)$ and $(D \vee \neg x)$. Now all variables from C and D are assigned to **1** or **0** by β'. As β' is satisfying, at least one of the literals in $(C \vee D)$ is assigned to **1**. If both C and D are satisfied by β', the assignment for x can be chosen independently of them. Otherwise, only either C or D can be unsatisfied, therefore, x can be chosen such that the other clause is satisfied.

2.3.2 Watched Literal Scheme

It is crucial for the performance of a SAT solver to achieve a high speed in terms of decisions made per time. Both DPLL and CDCL interleave branching with the repeated application of the unit clause rule, which is interrupted when either a clause is found to be in a conflict state, or all literal assignments implied by the branches have been found. This propagation process takes up to 90% of the overall running time, thus, it is important to implement it efficiently. A common approach is the use of lazy data structures, as suggested

Algorithm 7: Naïve BCP

Data: Propositional formula ϕ in CNF, literal ℓ to propagate
Result: Conflicting clause c if conflict occurred, and null otherwise

```
1  prop_queue := {ℓ};
2  while prop_queue is not empty do
       /* Take next literal from queue                          */
3      Take ℓ ∈ prop_queue;
4      prop_queue := prop_queue\{ℓ};
5      val(ℓ) := 1;
       /* Check all clauses containing ¬ℓ                       */
6      for c ∈ φ_¬ℓ do
7          if ∃ℓ' ∈ c.val(ℓ') = 1 then
               /* Clause is already satisfied                   */
8              continue;
           /* Count number of literals assigned to false        */
9          falseLits := |{ℓ' ∈ c|val(ℓ') = 0}|;
10         if falseLits = |c| then
               /* Clause is in conflict                         */
11             return c;
12         else if falseLits = |c| − 1 then
               /* Unit propagate                                */
13             Choose ℓ' as unassigned literal from c;
14             prop_queue := prop_queue ∪{ℓ'};
15             val(ℓ') := 1;
16 return null;
```

in [162]. A naïve implementation may keep a list for each literal, containing the clauses in which the negation of this literal is contained. Once the literal is set, either by branching or implication, each of the clauses in this list is checked: If all but one literals in it are set to **0**, the remaining literal is implied and propagated, unless it is set to true already. If all literals are assigned to false, a conflict is detected. The pseudocode for the propagation of one literal is given in Algorithm 7.

Here, we assume a function val which returns the current valuation of a

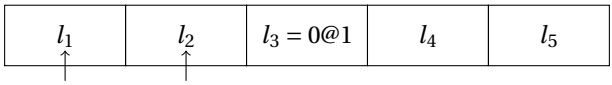


Figure 2.5. One clause with watched literals l_1 and l_2. When assigning l_3 to 0, this clause is not considered by the propagation algorithm, as both l_1 and l_2 are not assigned any value yet.

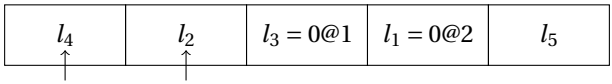


Figure 2.6. The clause from Figure 2.5 after assigning $l_1 = 0$. As this is watched, a new watched literal is sought, in this case l_4, and swapped to the front of the clause.

literal as **1**, **0** or "undefined". The drawback of this routine is that it considers every clause containing the currently propagated literal, and visits the literals within each clause repeatedly. Early implementations cached the number of literals set to **1** and **0**, respectively [162]. However, this still requires visiting all clauses, and resetting the counters during backtracking. Moskewicz et. al therefore suggested a lazy approach. It is based on the observation that clauses will not participate in any propagation or conflict unless at most one of their literals is unassigned, and all others are assigned to false. Therefore, they suggest to watch two literals from each clause, the "watched literals", cf. Figure 2.5. During propagating a literal, only clauses are considered which are watched by this literal. If one of the watched literals is set to false, and the other one is not set to true, a new watched literal is sought, which, on success, replaces the old one, as depicted in Figure 2.6. Otherwise, unit propagation or conflict analysis, respectively, are triggered.

This technique brings several advantages. Firstly, if a variable is assigned a value either by branching or propagation, it is not necessary to check all clauses which contain it or its negation. Secondly, it is not necessary to reset any data structures for backtracking, as the choice of watched literals remains feasible. Thirdly, this approach takes advantage of the CPU caches, as the watched literals are stored as the first two literals in a clause, thus, they fit into one cache line [62].

Currently, solvers do not use this scheme in a strict sense, as it does not make sense for binary clauses [179]. Here, special data structures are used,

where the watch lists contain the other literals in the respective clause rather than a pointer to the clause.

2.3.3 Branching

The DPLL algorithm described in Section 2.2.3 contains the line "Choose variable x..." in which the decision is made how to branch next. There is no explanation here how to make this choice. A deterministic polynomial-time algorithm that, when given a propositional formula ϕ, returns a literal ℓ such that $\phi_{|\ell}$ is equisatisfiable to ϕ would imply $P = NP$.

Thus, several heuristics have been suggested to find good literals for branching. Jeroslaw and Wang suggested in [129] to choose a literal ℓ which maximises $\sum_{c \in \phi : \ell \in c} 2^{-|c|}$, where ϕ is the input formula. The idea behind this choice is that picking a literal which occurs often on short clauses will satisfy them, and leave some other literals in long clauses such that it is likely that these can be satisfied as well. One may either compute these values statically, or dynamically before every branching decision [196]. The latter heuristic, dynamic largest individual sum (DLIS) considers the fact that branching and subsequent unit propagations may significantly change the importance of different literals. For example, occurrences in a clause should only be counted if the clause is not already satisfied under the current partial assignment.

These heuristics come with some drawback: Either it is computationally expensive to compute the ranking of each literal before every decision, or it is static and does not consider learned clauses.

The solver ZChaff [162] introduced a new technique, called variable state independent decaying sum (VSIDS). Here, there is a counter for every literal. Whenever a (learned) clause is added to the database, the counter for each literal in it is incremented. When branching, the literal with the highest counter value is chosen. To emphasise on conflict clauses learned recently, these counters are divided by some constant $c > 1$ periodically. As the counter values can be stored in a priority queue, implemented as a heap, it is significantly faster to find a new branching literal. Furthermore, it proved beneficial for SAT solvers to focus on recent conflicts. Most current SAT solvers use variations of this technique [17, 44, 89]. Here, the counter is often called activity of a variable. These variations include the choice of literals for which the activity is increased [17], the amount of increase, or the frequency of decrease [89]. Interestingly, VSIDS

seems to be able to identify important variables in the formula, and lead the solver to consider these preferably [134, 149].

Another important extension is phase saving. Assume a formula $\phi = \phi' \wedge \phi''$ with $\text{vars}(\phi') \cap \text{vars}(\phi'') = \varnothing$. Furthermore assume that a solver has found a partial assignment β' satisfying ϕ'. Then, a bad branching decision which is not compatible with β' may force the solver to perform unnecessary work to find a satisfying assignment for ϕ' again. Therefore, Pipatsrisawat and Darwiche suggested to store the last polarity that a variable was assigned to, and re-use it in the next branching decision [180].

2.3.4 Conflict Driven Clause Learning

The concept of CDCL was briefly introduced in Section 2.2.4. In theory, a SAT solver may resolve any two fitting clauses, and add them to its database, which would yield an algorithm similar to DP, except for the order in which clauses are learned. However, CDCL solvers run a search to find a solution for the input formula, and learn clauses during this process. Thus, when the result is UNSAT, this is proven by a resolution proof which was led by the search for a SAT result. This works surprisingly well, and current SAT solvers implement several techniques to increase their efficiency. In this section, we will review some techniques related to the learning process.

Clause Minimisation

The minimisation of learned clauses was introduced in [202]. Consider again the learned clause from Example 2.11. It contains all literals set at the decision levels 1 and 2, which in fact is not necessary. The literal d was set due to the binary clause $(\neg a \vee d)$. Now $\neg a$ is also contained in the learned clause, thus, we may remove $\neg d$ from it. The reason for this is just a resolution step

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

This is called self-subsuming resolution, as the result subsumes the learned clause. Thus, the subsuming clause can be added instead of the firstly learned clause. Furthermore, this step can be repeated for e, gaining

$$\frac{(\neg a \vee \neg b \vee \neg e \vee \neg f) \qquad (\neg b \vee e)}{(\neg a \vee \neg b \vee \neg f)}$$

In this example, only binary clauses were used, but this technique can also handle non-binary clauses.

Literal Block Distance

The literal block distance (LBD) was introduced in [19] to estimate the value of learned clauses. Consider a learned clause in the moment when it was learned, i. e., before a backjump was performed. Then, all literals in this clause are assigned to **0** at different decision levels. The LBD value is defined as the number of different decision levels on which literals from this clause were assigned.

The idea is that literals which were assigned at the same decision level, called a block, are likely to be linked together directly. Thus, a learned clause creates some connection between the blocks in it, which is considered helpful. The authors call learned clauses with a LBD of 2 "glue clauses", as in these clauses there is exactly one literal from the decision level on which the conflict occurred; all other literals were assigned at the same decision level. Therefore, this creates a strong link between the asserting literal and the remaining literals. These so-called "glue clauses" inspired the name of the solver GLUCOSE.

The LBD value of learned clauses is used for determining which clauses are worth being kept in the clause database of a CDCL solver, as we will discuss in the next section. Furthermore, it is used to trigger solver restarts (cf. Section 2.3.5) and to control communication in parallel solvers (cf. Section 2.5.1).

Clause Deletion Strategies

CDCL based SAT solvers learn many clauses during a solver run. On the one hand, this is beneficial as these learned clauses allow for more propagations during search, and they are the foundation of UNSAT proofs, as shown in the previous section. On the other hand, keeping a lot of learned clauses slows down the propagation process, and may even cause the solver to run out of memory. Therefore, solvers clean their learned clause database regularly and delete some of the learned clauses [19, 111, 126, 169].

MINISAT uses an activity value for each clause, similar to the activities used for variables in the VSIDS heuristic. The activity of clauses seen in the

analysis of conflicts is increased by some value, and this value is multiplied by a constant $c > 1$ after every conflict analysis, which emphasises on recently learned clauses. If the size of the learned clause database reaches some threshold, which depends on the number of clauses in the input formula, half of the clauses with smaller activities are deleted if their size is larger than two. Binary clauses are kept forever.

On some large formulas, MINISAT allows for a large amount of learned clauses. In the case of formulas which require a large proof this behaviour is extremely beneficial, as we will show in Chapter 4.

On the contrary, GLUCOSE is quite successful using a totally different strategy. Here, clauses are ranked by their LBD value as primary key. "Glue clauses", i. e., clauses with a LBD value of 2, are never deleted. Furthermore, cleaning of the clause database is triggered according to some slowly increasing sequence which is independent of the size of the input formula. On many formulas, this idea works very well. One reason can be seen in the fact that many learned clauses propagate once — just after they have been learned — and are hardly used afterwards [18, 19]. Therefore, it may be a reasonable decision to reduce the clause database aggressively.

In [169], both strategies were mixed. The solver keeps clauses with very low LBD value forever, whereas clauses with higher LBDs are managed based on their activities.

Another approach is the freezing of clauses [15]. The authors suggest a dynamic measure, denoted by "phase saving based quality measure" (psm) to estimate the importance of a learned clause in the current part of the search space. Given a set of assigned literals $\mathcal{P}$, the psm of a clause c is defined as $psm(c, \mathcal{P}) = |\mathcal{P} \cap c|$. If this value is large, then it is considered unlikely that the clause will propagate, as many of its literals are likely to be satisfied. Hence, it is also unlikely that it will be involved in conflict analysis, thereby contributing to a proof. If the psm value is small, clauses are not deleted immediately, they are set to an intermediate state, which the authors call frozen, and can be reactivated if the psm value becomes smaller again. Clauses are only deleted if they have not been reactivated for some time.

LBD seems to be a rather strong measure especially on formulas created from Bounded Model Checking (BMC) problems. However, it does not always help: In some cases, considering the activity or size of learned clauses works better. The solver we submitted to the SAT Competition 2016 used the size of

learned clauses rather than their LBD value, unless they were glue clauses, as we will discuss in Section 2.4.

Furthermore, the decision of how, and how often, to clean the clause database is non-trivial already for sequential solvers. The portfolio solvers we will discuss in Section 2.5.1 can learn a very large amount of good clauses, and it is not clear how to manage them.

2.3.5 Restarts

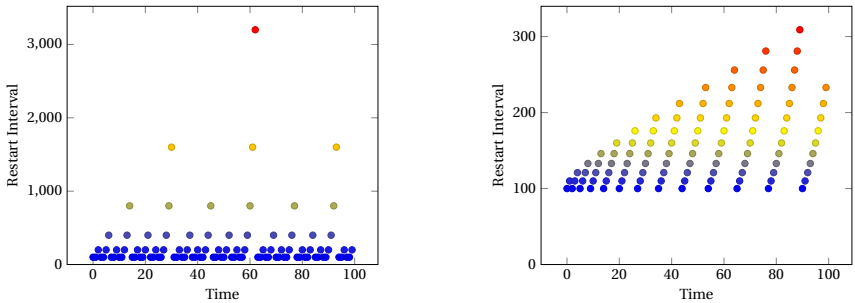
Modern SAT solvers restart their search after some time for different reasons. Assume again that ϕ is a SAT formula, and ℓ a literal occurring in ϕ. If a solver makes a bad decision during branching, i. e., it branches on ℓ such that $\phi \wedge \ell$ is unsatisfiable, whereas ϕ itself is satisfiable, then the solver must prove that $\phi \wedge \ell$ is unsatisfiable before searching for a solution for $\phi \wedge \neg\ell$. Thus, it may be beneficial to restart the search after a while. Here, restarting means a backtrack to decision level 0, i. e., undoing all branches. Other information gained during search like learned clauses, variable activities or phase information is not deleted. Thus, the solver continues its search by branching on the variable with the highest VSIDS activity. This behaviour has also proven helpful on unsatisfiable formulas, as it allows the solver to focus on hard parts of the formula [19].

MINISAT uses the Luby sequence to trigger restarts [89]. The Luby sequence is given by

$$t_i = \begin{cases} 2^{k-1} & \text{, if } i = 2^k - 1 \\ t_{i-2^{k-1}+1} & \text{, if } 2^{k-1} \leqslant i < 2^k - 1 \end{cases}$$

This is, after i restarts, MINISAT triggers a new restart after ct_i conflicts, were c is a constant which equals 100 in MINISAT.

Figure 2.7(a) shows the resulting restart intervals. Heule found that using a smaller constant reduces the average number of conflicts required to solve a formula [112, 183]. Biere used another restart-scheme, called “inner-outer-restarts". It uses two exponentially growing sequences, where the “inner" sequence determines the length of one restart interval. The “outer” sequence is used to reset the values of the inner sequence in order to favor short intervals between restarts, as shown in Figure 2.7(b). After k restarts, the pair of inner



(a) Luby sequence

(b) Inner-outer-scheme

Figure 2.7. Intervals between subsequent restarts based on (a) the Luby sequence, and (b) the inner-outer-scheme.

and outer values is given by

$$(i,o)^0 = (i_0, o_0)$$

$$(i,o)^k = \begin{cases} (i * c_i, o) & \text{, if } i \leqslant o \\ (i_0, o \cdot c_o), & \text{, else} \end{cases}$$

In PICOSAT the constants are chosen as $i_0 = o_0 = 100$ and $c_i = c_o = 1.1$ [45].

These rapid restarts improved the performance of SAT solvers; however, they use a static scheme which may restart the solver just before finding a solution. A later version of PICOSAT therefore comes with a dynamic strategy [43]. It measures how often variables are assigned to a value which is the negation of the value stored for phase saving. If this happens often, the solving process is considered agile, and no restart is necessary. Conversely, if the solver is stuck on a small subproblem, this agility value is likely to be small, thus PICOSAT triggers a restart.

Audemard et al. used different heuristics to dynamically trigger or block restarts [20]. Their solver GLUCOSE keeps track of the LBD values of learned clauses. As long as these are low, the solver is considered to make progress, as it learns good clauses. On the contrary, if average LBD values increase quickly, a restart is triggered. Furthermore, they consider the decision levels the solver reaches. If these rise to high values, it is likely that the solver is just about to

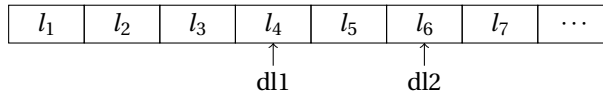


Figure 2.8. The trail of a SAT solver. The arcs indicate the literals which were branching decisions, whereas the other literals were propagated.

find a solution, and therefore restarts are blocked.

After this introduction, we now present the first result we obtained on sequential SAT solving. Afterwards, we will turn to the parallelisation of SAT solvers.

2.4 SAT Competition 2016

The SAT competition is an event which takes place every second year, interleaved with SAT races[1]. Participants submit their SAT solvers, which are tested on a wide range of different benchmarks from different categories, and winners are determined for each of them. We submitted a solver to the "GLUCOSE hack track". Here, participants are asked to "hack" the solver GLUCOSE 3.0[2] and improve its performance as much as possible. In order to enforce small changes, the Levenshtein distance [147] between original and submitted source files is upper-bounded by 1,000. The idea behind this limitation is to encourage people to participate without writing a complete solver themselves. Moreover, this allows to come up with small but good ideas that might be beneficial for the whole SAT community. We applied two changes to the solver we submitted, which will next be discussed in detail.

2.4.1 Refining the Restart Strategy

As discussed in Section 2.3.5, GLUCOSE blocks restarts if the solver seems to be deeper in the search space than on average, as this might indicate that it is close to finding a solution. To define "deep in the search space", Audemard et al. used the trail size as an indicator. An example of a trail is shown in Figure 2.8. It

[1] A history of competitions and races is available at `http://www.satcompetition.org/`.

[2] Available from `http://www.labri.fr/perso/lsimon/glucose/`.

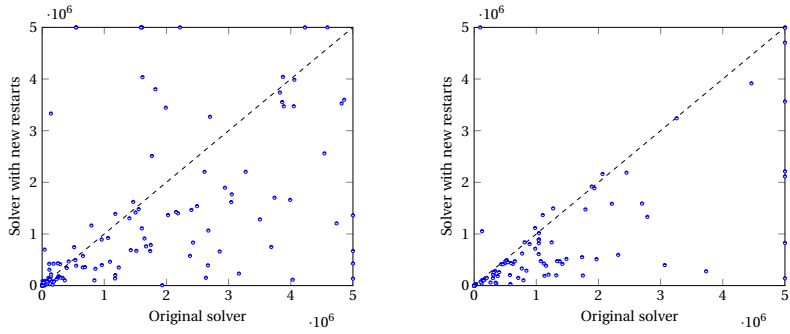
contains the literals that are assigned to some value in the order in which they were assigned. In this example, the literals l_1, l_2 and l_3 were either contained in the input formula or learned as unit clauses, they are fixed. The variables set at decision level 1 are l_4 and l_5, where l_4 was a branching decision, and l_5 was propagated. The trail size is simply the number of literals on the trail.

Whenever a conflict occurs, GLUCOSE stores the trail size. Furthermore, it computes the average of the last 5,000 stored trail sizes. If the solver reaches a trail size which is “deep" in the search space, this is considered interesting, and restarts are blocked for a while. In the default settings, this is the case if the trail size exceeds the average of stored trail sizes by a factor of 1.4 [20]. This strategy has helped GLUCOSE to overcome the weaknesses on satisfiable benchmarks that the early versions suffered from.

However, many benchmarks contain a lot of unit clauses, and even more can be found quickly by the solver. Therefore, the trail consists of a static part — the literals l_1, l_2 and l_3 in Figure 2.8 — and a dynamic part. We adapted the restart blocking strategy of GLUCOSE to consider the size of the dynamic part rather than the overall trail size, allowing the solver to decide more precisely when to block restarts. It especially becomes more likely that restarts are blocked, as we kept the same constants.

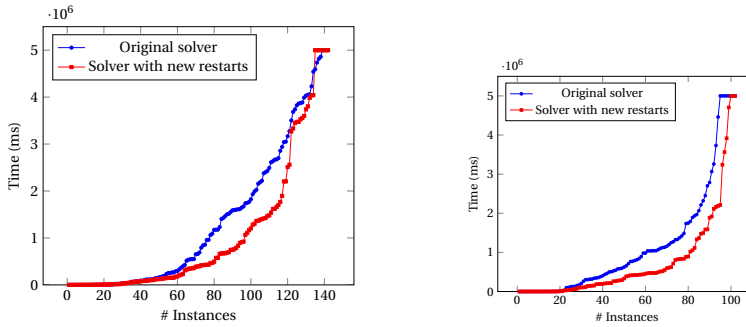
We tested the modified solver on the 300 benchmarks used in the main track of the SAT Race 2015. According to the argumentation in [20], one might expect that the modified restart behaviour would lead to an improved performance on satisfiable formulas, and decreased performance on UNSAT cases. Interestingly, this is not the case. Figures 2.9(a) and 2.9(b) show scatter plots which compare the performances of the original version of GLUCOSE, and the modified version, on satisfiable and unsatisfiable formulas. In the case of satisfiable inputs, the modified version is slightly faster on average, which can also be seen in the cactus plot in Figure 2.10(a).

On the contrary, the original solver can solve 5 benchmarks in the given time limit of 5,000 seconds that the modified version cannot solve, whereas the modified version solves only 4 benchmarks uniquely. The sum of execution times is slightly reduced from 173,294 to 146,908 seconds. Surprisingly, the modified solver is significantly stronger on unsatisfiable inputs, cf. the scatter plot 2.9(b) and the cactus plot 2.10(b). The overall running time for benchmarks that at least one of the solvers could solve is reduced from 79,617 to 53,161 seconds, and more instances can be solved. These results correlate



(a) SAT (b) UNSAT

Figure 2.9. Pairwise comparison: Original solver vs. solver with modified restart strategy



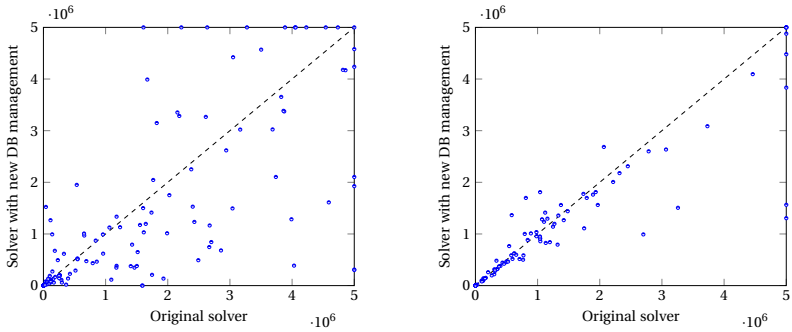
(a) SAT (b) UNSAT

Figure 2.10. The impact of the modified restart strategy as cactus plots.

with [43], where the author conjectures “frequent restarts may also be harmful, particularly on unsatisfiable crafted instances”.

2.4.2 Re-considering LBD

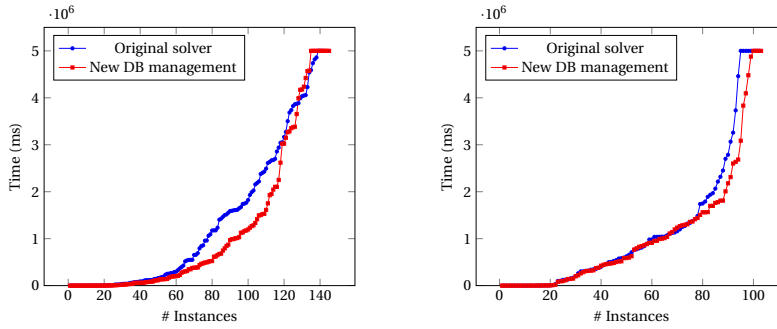
The second change we made was concerned with the management of the learned clause database. GLUCOSE used the LBD value of learned clauses as primary metric to decide which clauses to keep, and which to delete. We found



(a) SAT (b) UNSAT

Figure 2.11. Pairwise comparison: Original solver vs. solver with modified clause database management

that LBD is in fact a good measure, especially if the values are rather small, and especially on benchmarks from benchmarks considered "application". These are typically benchmarks with a clearly defined community structure [9, 10, 107]. Therefore, we kept this measure for "glue clauses", i. e., clauses with a LBD value of 2. For other clauses, we used the number of literals as a measure, and preferably kept short clauses. Additionally, we removed clauses whenever their activity dropped to 0, meaning that they have not been used in conflict analysis for some time. As this was too aggressive with the default settings, we increased the parameter "cla-decay" to $1-10^{-4}$. This parameter is used to control how fast the value added to the activity of a clause is increased over time, which in return emphasises the activity of clauses used recently. We reduced the increase, which means that the clause activities do not emphasise on recent conflicts too much. As the activities are only used for tie breaking, contrary to MINISAT [89] where the activity was an important measure, this appeared reasonable. If the activity of a clause is increased to a value larger than 10^{20}, all clause activities are multiplied by 10^{-20} in MINISAT. This is necessary as otherwise the activity values might grow larger than the value range of float variables. If a learned clause is not used between some rescaling steps, its activity may drop to 0. Thus, modifying the increase of learned clause activities lead to numerically smaller clause activities, which delayed the rescaling, and therefore increase



(a) SAT (b) UNSAT

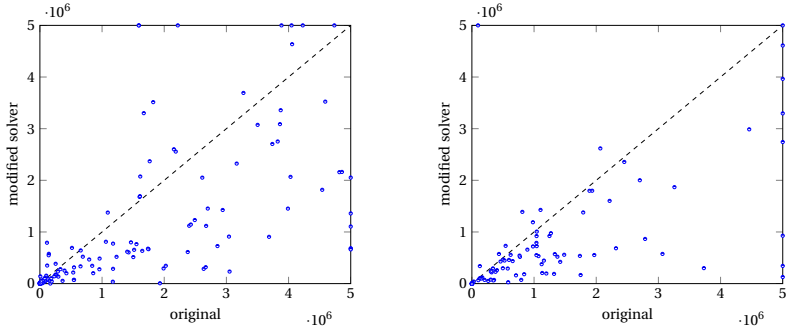
Figure 2.12. The impact of the modified clause database management as cactus plots.

the time until activities might eventually drop to 0.

The impact on satisfiable instances is shown in the scatter plot 2.11(a) and the cactus plot 2.12(a). The overall running time on benchmarks solved by at least one of the original and the modified version, decreased from 138,060 to 106,201 seconds. The modified solver is especially faster on formulas of medium hardness, but solves 4 less benchmarks. On the contrary, the overall running time on unsatisfiable inputs decreases only slightly from 79,617 to 74,157 seconds, but the modified solver is able to solve 5 more benchmarks. These results coincide with [12] where short clauses are found to be more useful on satisfiable formulas, and LBD is considered more important on unsatisfiable instances.

The performance of the submitted solver, compared with original GLUCOSE, on satisfiable and unsatisfiable formulas is shown in the scatter plots in Figs 2.13(a) and 2.13(b). Again, some diversification on satisfiable benchmarks can be observed, and an improved average running time. In the case of unsatisfiable benchmarks, the modified solver clearly outperforms the original version. The performance on both satisfiable and unsatisfiable instances is shown in Figure 2.14, with a clearly improved overall performance.

These performance gains could also be seen in the SAT Competition 2016 [27], where our solver won the gold medal as best GLUCOSE hack solver in the main track.



(a) SAT (b) UNSAT

Figure 2.13. Comparison between original GLUCOSE and the hack with both optimisations enabled.

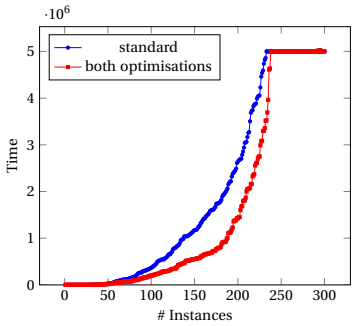


Figure 2.14. Comparison between original GLUCOSE and the hack with both optimisations enabled

Figure 2.15. Medal won at the SAT Competition 2016

After presenting this first result, we now turn to parallel SAT solvers. Firstly, we will give an overview over existing techniques for parallel SAT solving and actual implementations, before presenting results that were published in [97].

2.5 Parallel SAT

The success of SAT solvers has attracted interest in their parallelisation. This is not because they naturally lend themselves to parallelisation, in contrary, there is no obvious way to parallelise modern SAT solvers. With the improved performance of SAT solvers, they are more and more used to solve hard problems, which in return leads to an interest in even more improved performance, and more efficient usage of modern, especially parallel, hardware.

We will start by discussing an approach which uses additional computational resources to strengthen learned clauses. Next, we discuss the two most prominent parallelisation techniques, and pose some related research questions. Afterwards we discuss one result we published in [97], which tackles one of these questions.

In sequential SAT solving, learned clauses are minimised, and then added to the learned clause database, as discussed in Section 2.3.4. The authors of [213] suggest to use a “solver-reducer-architecture". The solver sends learned clauses to the reducer, which tries to drop some of the literals, i. e., derive a subsuming clause. If successful, this minimised clause is sent back to the solver. This approach improves the performance especially on unsatisfiable formulas, but its scalability appears limited.

Another way to parallelise a SAT solver is the Shannon Expansion [195][3]. Let ϕ denote a propositional formula, and $x \in \text{vars}(\phi)$. Then,

$$\phi \equiv (\phi \wedge x) \vee (\phi \wedge \neg x).$$

This is, if ϕ is satisfiable then there is at least one solution with x assigned to 0, or x assigned to 1. Thus, one may split the formula into two subformulas $\phi' = \phi \wedge x$ and $\phi'' = \phi \wedge \neg x$, and solve these two formulas in parallel. Obviously, this expansion can be used recursively on the subformulas ϕ' and ϕ''. The conjunctions that define the subformulas are often referred to as cubes [124], and the principle of decomposing a formula with cubes is called Cube&Conquer.

Early parallel SAT solvers used this splitting approach together with load balancing implemented as work stealing [49, 215]. The authors report an efficiency of 95% on 256 CPUs. These solvers were based on the DPLL algorithm, thus, such results can be expected as there are only efficiency issues if some CPUs are idle: As depicted in Section 2.2, the DPLL algorithm does not re-use

[3] This expansion was mentioned already in 1848 by George Boole [51].

any information gained in one part of the search space in other parts. Furthermore, on satisfiable formulas a super-linear speed-up can be expected from this approach, if the solutions are located in a small part of the search space [184].

By contrast, splitting the formula statically contradicts the techniques described in the previous section such as restarts and activity based search. Consider the formula $((a \vee b) \wedge \phi) \wedge ((a \vee \neg b) \wedge \phi)$ [13], where ϕ is unsatisfiable. A DPLL based solver will have to consider the cases $b = \mathbf{0}$ and $b = \mathbf{1}$ separately, which both simplify to $\phi \wedge a$, whereas a clause learning solver can solve one of the cases, and deduce unsatisfiability of the other case from its learned clauses easily. If b is used to split the formula, this advantage is lost unless the parallel solvers are allowed to exchange learned clauses. Furthermore, if a is used to split the formula, the subformula $\phi_{|\neg a}$ can be proven unsatisfiable by unit propagation. This imbalance in the hardness of the subformulas makes it necessary to use some work balancing technique like work stealing.

In order to avoid redundant work on similar subformulas, learned clauses can be exchanged between the solver processes. Unfortunately, this approach seems to be of limited success in current solver implementations. In [145], learned clauses of size at most 2 are exchanged. This threshold on the size of learned clauses is chosen dynamically in [153], but the performance gains are limited.

Interestingly, for some hard combinatorial problems this approach seems to work. The authors of [124] propose an approach where the expansion is performed such that the resulting subformulas become small after unit propagation, and every subformula is solved by a CDCL solver. They report a speed-up from this approach, even if a sequential solver is used to solve one subformula after another. Unfortunately, it is not clear if e. g., a clause database management which is tuned to hard, combinatorial formulas might have neglected this effect.

Another parallel solver which splits the search space is pMiniSAT [66]. It is based on a master-slave architecture, and the splitting is executed dynamically. If one solver is asked for a new cube, it creates one using the literal which it branched on the lowest decision levels. Assume it was working on the cube $\ell_1 \wedge \ell_2$, and its first branching decisions were ℓ_3 and ℓ_4. Then it may extend its own cube to $\ell_1 \wedge \ell_2 \wedge \ell_3 \wedge \ell_4$, and return the cubes $\ell_1 \wedge \ell_2 \wedge \neg\ell_3$ and $\ell_1 \wedge \ell_2 \wedge \ell_3 \wedge \neg\ell_4$. Thus the solver splits on variables that seemed interesting

to the VSIDS heuristic on the subformulas. Furthermore, the authors note that this approach yields many cubes which can be refuted with $1-10$ conflicts. Thus, they suggest to keep a queue of cubes at the master process, which allows for distributing new work quickly without waiting for the reply of another slave where new work is stolen from.

This approach reduces the idle time of slaves, but it comes with another drawback. If working on a satisfiable formula, the master may send cubes to the slave such that every subformula which is being processed is unsatisfiable, and the cubes defining subspaces which actually contain a solution are in the work queue. Such problems are overcome in solvers like AmPharos [16], which clearly outperforms other Cube&Conquer solvers on application benchmarks. Here, slaves work on one cube for a short time. If they neither find a solution nor prove it unsatisfiable, they pick another cube. This comes with the advantage that clauses which were learned when working on one part of the search space, are also used for other parts. Furthermore, cubes leading to a satisfiable subspace will eventually be considered.

These kind of problems do not arise with the other, currently more prominent approach in parallel SAT solving.

2.5.1 Portfolio-based Parallel SAT Solving

The portfolio approach for parallel SAT solving is based on the observation that the performance of sequential SAT solvers is very sensible to small changes on some parameters, e. g., concerning the restart strategy, or the increase of VSIDS activities.

This is exemplified by some results shown in Table 2.1 and Figure 2.16. We ran GLUCOSE on the benchmarks used in the main track of the SAT Competition 2016. In the first run, we used the default random seed, and in the second run, we ran it 64 times with different random seeds. In both cases, a timeout of 5000 seconds was used. When using a preprocessor and the default random seed, 61 satisfiable, and 83 unsatisfiable benchmarks could be solved. Without preprocessing, the results are slightly worse, with 59 and 79 solved instances. When using different random seeds and a preprocessor, 81 satisfiable benchmarks could be solved by at least one of the configurations. Without running the preprocessor, the number of solved satisfiable formulas increased from 59 to 86. In the case of unsatisfiable benchmarks, some performance gain can

Table 2.1. Number of benchmarks from the SAT Competition 2016 which were solved by GLUCOSE when running it repeatedly with different random seeds.

Configuration	Presolved		no Preprocessing	
	SAT	UNSAT	SAT	UNSAT
default	61	83	59	79
64 different random seeds	81	89	86	88

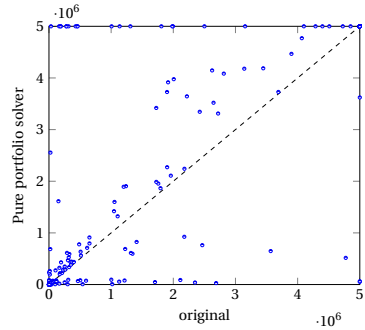


Figure 2.16. Comparison of default GLUCOSE and a portfolio solver with different random seeds. The running times are given in milliseconds.

be observed, but it is not as significant. This is not too surprising as in theory, satisfiable benchmarks can be solved efficiently if the solver starts with a fitting guess on variable assignments, whereas it always has to provide a resolution proof for unsatisfiable inputs.

The solver MANYSAT [116] won the parallel track of the SAT Race 2008[4], taking advantage of this instability. Here, the authors used different settings for the restart strategy, branching, and clause learning. Furthermore, they found that the exchange of learned clauses significantly improves the performance. Clauses that one of the parallel solvers learns can be added to the clause database of other solvers, as they are implied by the input formula. Experimentally, the authors found that for MANYSAT, the performance was best when clauses with a size of at most 8 were exchanged. Later, the authors observed that the clauses a solver learns tend to become longer over the time the solver is

[4] http://baldur.iti.uka.de/sat-race-2008/

running. Therefore, they suggested to adapt clause exchange dynamically [115]. If only few short clauses are learned, the solvers increase the export threshold, and vice versa.

Most current portfolio-based SAT solvers also consider the LBD values for determining which clauses to exchange. PLINGELING shares clauses with LBD of at most 8, and size of at most 30 [44]. Furthermore, the solvers PENELOPE and SYRUP, based on GLUCOSE, use several techniques concerned with the management of clause sharing. PENELOPE [12] implements freezing of clauses, as described in Section 2.3.4. Clauses which are imported from other solvers can either be set to frozen directly, or dynamically. Afterwards, freezing and reactivating are performed based on the psm measure.

SYRUP [18] performs clause exchange lazily. This is based on the observation that many learned clauses are seen only a very few times in conflict analysis. Furthermore, on unsatisfiable formulas many learned clauses are useless in the sense that they do not appear in the final proof of unsatisfiability [197]. Thus, the authors suggest to lazily export learned clauses when they are seen in conflict analysis for the second time, and their size and LBD are below the respective median values of all learned clauses. They also perform lazy clause import: Imported clauses are watched only by one literal. They are only promoted to the two-literal-scheme when they are in conflicting state, i. e., all of there literals are assigned to false. Thus, they are only promoted if they prune a part of the search space that was not pruned by other clauses from the learned clause database.

Other solvers work well, though they do not use as sophisticated techniques for clause exchange. In HordeSAT [29], the solvers exchange clauses once every second. Every solver exports clauses with a sum of sizes of at most 1,500 literals, where short clauses are preferred. Furthermore, it uses hashing to filter duplicate clauses. By this, they report a super-linear speed up for up to 512 processes, each of which running 4 solver threads. However, these values also depend on they way they are computed — with the evaluation scheme used in [97], the speedup on 2,048 threads would have only been 4.4, and lower than the speed up on 1,024 threads.

There are several challenges arising in the field of parallel SAT solving. Hamadi and Wintersteiger [117] named seven challenges, including the question whether the variations of the CDCL approach are an appropriate basis for parallel solvers, or if these should be based on a totally new paradigm.

We will state some more challenges in the field of parallel SAT solving, mainly focussed on the portfolio approach. Afterwards, we present results for one of them, which were published in [97].

11 Challenges in Massively-Parallel Portfolio SAT Solving

1. The first questions deals with the question of how to measure the speed-up of a parallel solver. It is common to evaluate solvers on benchmarks from SAT Competitions. These sets typically contain benchmarks that are easy, i. e., can be solved within less than a second by most sequential solvers, as well as hard benchmarks which can hardly be solved in a reasonable amount of time, even by parallel solvers. On the first kind of formulas, parallel solvers typically do no achieve better results than sequential ones, e. g., because they spend some time forking MPI processes. As parallel solvers are intended to solve hard problems, this is not necessarily bad, but it is important to be aware of this, and decide whether they should be considered in an evaluation with the same weight as harder formulas. In [97], we used the some of running times on our benchmark set for the computation of speedups. This correlates to the user-experience: It compares the time required to solve all benchmarks with a solver on n cores with the time required to solve these formulas with n sequential solvers, but allowing for parallel execution of solvers on different benchmarks. Thus, this approach is somewhat realistic, but it is still pessimistic, as it does not consider the fact that a timeout has to be used especially for the sequential reference solver. If the sequential solver cannot solve a given formula, but the parallel version solves it, this version of computing speed-ups is biased towards the sequential solver. On the contrary, the authors of HordeSAT [29] computed speed-ups based only on benchmarks that their parallel solver could solve. Furthermore, they computed the overall speed-up as the arithmetic mean of the speed-ups observed on the single inputs. By this measure, they achieved super-linear speedups on hard formulas, whereas according to our way speed-up computation, it would have only been 4.4 on 1,024 threads.

   We do not claim that our approach is the best, actually it is biased against parallel solvers. Still, it is clear that no arithmetic means should be used, as they emphasise good results on single tests.

   Question 1. What is a fair evaluation scheme for parallel SAT solvers on a

mix of easy and very hard benchmarks?

2. It is often claimed that the scalability of portfolio solvers is limited as the search of the single solvers synchronises over time [13]. However, it is not clear how this claimed synchronisation is defined. Considering VSIDS-activities may lead to a false-positive result: Comparing variables with high activities between the single solvers might lead to the insight that all solvers focus on a similar subset of variables. However, this due to the nature of VSIDS, which tends to pick central variables [134]. We conducted some tests with a student, Dennis Sen, in his Bachelor's Thesis [194]. Here, we considered the polarity vectors of the different solvers, and compared them to each other. In these tests, we found these vectors highly volatile, and could not find evidence of a synchronisation between solvers.

   Finally, one might consider learned clauses as a criterion for the possible synchronisation of solvers: If clauses learned by one solver are redundant for other solvers, i. e., do not prune the search space of other solvers, this might be seen as a hint for synchronisation. However, checking for redundancy is computationally expensive, and it is not clear if this effort pays off.

   In case synchronisation is detected, the first approach might be to add some randomisation. The easiest way for this is random branching. Solvers like GLUCOSE allow for both choosing variables randomly, and branching on a randomly chosen polarity. Using both sources of randomisation contradicts the idea of polarity vectors, and may, in the worst case, just result in superfluous work.

   Question 2. How can a lack of diversification in portfolio based SAT solving be defined and detected?

3. There is some evidence that the scalability of portfolio SAT solvers is limited by the structure of resolution proofs [210]. The author discusses the depth of resolution proofs in the sense that the proofs require learned clauses which can only be resolved from other learned clauses, which induces a sequential structure in the proof. This would imply a bound on the parallelisability of portfolio solvers [133]. However, it is not clear if this is only true for some artificially created formulas, or if it also concerns so-called "real-world" problems.

Question 3. How deep are proofs of relevant formulas?

4. Both sequential and parallel solvers clean their databases for learned clause regularly. In the case of sequential solvers, this process is fine-tuned towards benchmarks from competitions. However, parallel SAT solvers typically re-use these heuristics. Parallel portfolio solvers on hundreds of cores learn a huge amount of clauses, and share many of them. Still, they use the same heuristics as in the sequential case.

   It is not clear how to determine a good size of the clause database of each of the solvers. Furthermore, there is a dependancy between clause exchange and clause management strategies. Assume a solver removes 50% of its learned clauses whenever it cleans its clause database. Then, exchanging more learned clauses may influence the choice of clauses kept, i. e., a higher clause exchange allows the solver to keep more clauses of medium quality.

   Question 4. What is a good clause database cleaning strategy for massively-parallel portfolio solvers?

5. Which learned clauses are a good choice for exporting? This question was already considered in several papers [18, 115], and is one of the seven challenges in [117]. This question involves both the amount of clauses exchanged, and the actual choice of a subset of learned clauses. If solving unsatisfiable formulas, clause exchange seems to have a larger positive impact [29], which coincides with results from sequential SAT solving [169], and results from our tests [97]. Most portfolio solvers export short clauses clauses with small LBD values, i. e., based on static measures, whereas PENELOPE export clauses lazily if they are involved in conflict analysis, and thus seem interesting. However, these measures relate the clauses to set of learned clauses the exporting solver is operating on; It is not clear whether this is a good measure for the usefulness for other solvers.

   Question 5. What is a good clause export policy, both for satisfiable and unsatisfiable formulas?

6. Restarts are crucial for the performance of sequential CDCL solvers [20, 43, 96, 112]. However, most portfolio based SAT solvers just re-use the restart policies of the sequential solvers they are based on.

(a) The dynamic restart policy of GLUCOSE triggers restarts if the average LBD of learned clauses increases. More aggressive restarts might lead to more clauses with small LBD, thus turning the solvers into clause producers [20], which in turn create clauses that “glue” the formula. Is such an increased amount of learned clauses helpful?

(b) Let us consider the work performed by one solver between two subsequent restarts. The solver branches on some literals $\ell_1, \ell_2, \ldots, \ell_k$, and is faced with conflicts. If k' is the lowest decision level on which a conflict occurs, this may be seen as the refutation of the cube $\ell_1 \wedge \ell_2 \wedge \ldots \wedge l_{k'}$. In some still unpublished experiments we found that this can be used as a hint whether portfolio solvers are better suited for some formula than Cube&Conquer solver or vice versa: If k' is small, this indicates that a Cube&Conquer solver might be a good choice, whereas large values for k' indicate that the formulas is not well suited for being split by short cubes. We think that this is worth a more profound research: does it make sense to dynamically switch between portfolio and Cube&Conquer solving?

(c) The activity values of the VSIDS heuristic are highly volatile [18]. Is this also a good choice for massively-parallel portfolio solvers, or does it make sense to slightly decrease the volatility? On the one hand, forcing solvers to remain in one part of the search space might lead to learning more clauses which are relevant for this particular subspace, thus allowing them to learn stronger lemmas. On the other hand, this may lead to learning longer clauses [190] or clauses with a larger LBD value, which would reduce the amount of exported clauses.

Question 6. Are common restart-strategies well-suited for portfolio solvers? Can information about the solving progress be used to guide the parallel solver?

7. Wieringa and Heljanko suggested a parallel solver on two cores in which one solver is just used to strengthen the clauses learned by the other solver [213]. Given one learned clause, the “reducer" branches its literals to 0, and uses BCP and the analyzeFinal-routine of MINISAT to check for a subsuming clause such that its negation unit-propagates to a conflict. This technique, which is also referred to as asymmetric branching or clause vivification [178],

or a similar technique might be use to minimise learned clauses before exporting them. However, the search for a subsuming clause somewhat contradicts the core idea of 1UIP learning which aims at encoding the reason for a conflict rather than just the decisions which, together with their implications, led to the conflict. Still, we consider it interesting to strengthen learned clauses before sharing them with potentially hundreds of other solver threads.

Question 7. Does it pay off to spend more effort in clause minimisation?

8. The notion of “clause sharing" is somewhat misleading, as portfolio solvers typically send copies of these clauses to other solver incarnations. Thus, if several solvers run on the same node, clauses are stored multiply on this node. On the contrary, rather old solvers like MIRAXT [148] and SARTAGNAN [140] physically share clauses between threads on a shared memory machine. This comes with several advantages.

   - ▻ If the solver runs on a hard formula, proofs may become large, which may become a problem if several solvers are executed on the same node. Therefore, a physically shared memory reduces the memory footprint significantly. Furthermore, the amount of memory available on hardware like Intel’s Xeon Phi is limited, thus this may even be mandatory when running parallel SAT solvers on this kind of hardware.
   - ▻ Communication between solver threads is easier on a shared memory machine, which can be used to exchange information about e. g., subsumed clauses, as described in [140].
   - ▻ Consider a hierarchical portfolio solver which consists of portfolio solvers which work on a physically shared clause database on shared memory systems, and communication between nodes is performed via MPI. Then, a learned clause could be placed in the shared clause database first, and be sent to other nodes if it is considered relevant by other solver threads, which would result in a less local version of the lazy clause exchange policy described in [18].

   On the other hand, using a physically shared clause database prohibits some of the techniques used to improve cache efficiency [63, 89, 156], especially

the dynamic reordering of literals in a clause. We still believe that a carefully designed implementation can be fast enough, such that the benefits outweigh the decreased single thread performance.

Question 8. How can the benefits of a physically shared clause database be taken advantage of in a (hierarchical) portfolio solver?

9. How can the usefulness of a received clause be checked? In sequential SAT, a learned clause explains a conflict, and may render formerly learned clause useless. Even there, it's not clear

   ▻ How to check if old clauses can be removed efficiently, except for subsumption checks.

   ▻ How, in the case of unsatisfiability, the finally proof is made, and which clauses contribute to it [197].

   Question 9. How can the usefulness of both a single clause and clause exchange in general be detected during a solver run?

10. Preprocessing [88, 154, 204] and inprocessing [128, 155] are important techniques, especially in sequential SAT solving. Parallel bounded variable elimination was implemented and evaluated on a shared memory machine in [104]. However, this approach requires a synchronisation between the preprocessing-threads which is likely to prevent this approach from scaling beyond shared memory machines. However, bounded variable elimination is not only useful because of the decrease in number of variables and clauses, but also as it derives subsuming clauses which improve the propagations possible by BCP. It might be an interesting research direction to exchange short clauses derived in the process of variable elimination rather than actually eliminating the variable on all nodes.

    We conjecture that techniques which add clauses or variables are better suited for distributed parallel SAT solving.

    Question 10. Which preprocessing or inprocessing techniques are helpful in a (massively-parallel) portfolio solver?

11. Most portfolio solvers perform communication in an all-to-all manner: A clause learned by one solver is, if so, sent to all other solvers. For a large

scale of parallelism this will lead to a prohibitive amount of communication. In [97], we suggested to limit the communication in terms of a communication graph such that only some pairs of solvers are allowed to exchange clauses, which will be described in the remainder of this section.

Question 11. How can the communication in a portfolio solver be controlled such that the solver still benefits from learned clauses, and the amount of communication does not become prohibitively large?

TOPOSAT: Communication Structures for Portfolio SAT Solvers

In this section, we present and discuss experiments which aim at answering Question 11. If a portfolio solver is run on n processes, and every solving process is allowed to send some of its learned clauses to all of its neighbours, the amount of information exchanged grows by $\theta(n^2)$. For large n, this becomes prohibitive both due to the amount of information transmitted through the connecting network, and the limited ability of the single solvers to handle an arbitrary amount of clauses. This information exchange can be limited by adjusting thresholds on when to export a clause, e. g., the maximum size or LBD value. Another approach is to create some locality in the parallel solver, and exchange information locally rather than globally. This is especially advantageous if the network is heterogeneous, for example if two fast networks are connected by a relatively slow interconnect. We therefore limited the communication by terms of a communication graph. Let $p_1, \ldots, p_n$ denote the processes a portfolio solver is made up from. For some graph $G = (V, E)$ with $V = \{p_1, \ldots, p_n\}$, we let the solver process p_i send its clauses to the processes it is connected to by an edge in G, i. e., $N_G(p_i)$, if the LBD is at most 4, a value which is quite small compared to the clause exchange policies e. g., used by PLINGELING [44], but seems reasonable [169]. In the case of unary and binary clauses, i. e., clauses of size at most 2, we allowed solvers to forward them if they were not known already, which can be checked by a simple look-up in $\mathcal{O}(1)$ time for unit clauses, and $\mathcal{O}(\log(n))$ for binary clauses. In this way, we made sure that these clauses, which we believe to significantly contribute to the solving process, eventually reach all solver processes. Furthermore, we considered it unlikely that the exchange of these clauses would overload the network.

In our solver, denoted TOPOSAT after the underlying graph topology which determines its communication, each solver processes runs two threads, one of

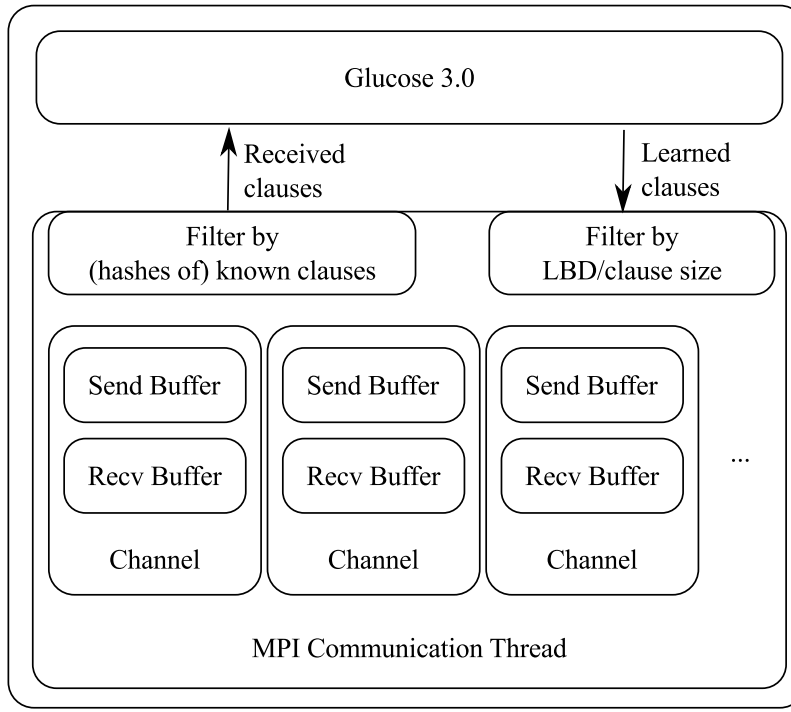


Figure 2.17. Architecture of a TOPOSAT process.

which performs the actual solving, whereas the other one handles the communication via MPI, as depicted in Figure 2.17[5].

The communication threads synchronise in configurable intervals, and exchange clauses, we used 5 seconds in our experiments. Each solver is allowed to send at most 2^{14} literals in each communication cycle. If there are too many clauses in the send buffer, clauses with small LBD are preferred, ties are broken by the clause sizes. Note that HordeSAT [29] uses a similar scheme, but limits the send buffer to 1,500 literals. When clauses are received, their hash value is computed, and compared to the hash values of clauses received earlier. In

[5]The design of the processes and communication cycles was implemented by Philipp Sieweck, PhD student at the Dependable Systems Group, Kiel University.

Table 2.2. Results for pure portfolio, running times are given in minutes.

cores	time(min)		speedup		timeouts	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
1	2789	2324	1.00	1.00	32	25
2	2379	2258	1.17	1.03	23	25
4	1961	2207	1.42	1.06	15	25
8	1693	2152	1.64	1.08	15	23
16	1498	2046	1.86	1.14	14	22
32	1136	1994	2.45	1.17	10	22

case the hash has been seen already, the clause is not added to the clause database. This approach worked well in our experiments, but comes with some difficulties we well discuss in the end of this section. In order to diversify the search, the random seed of the solver with MPI rank i is $9.16483e+07+i$; we just use the sum of GLUCOSE's default random seed and the MPI rank.

Experimental Results

We extensively tested TOPOSAT with different topologies and clause exchange policies on a subset of the benchmarks used in the SAT Competition 2013, a list of the files used is given in the Appendix A.1. We did not use the full set for two reasons: Firstly, cryptographic like the ones provided by Vegard Nossum [25, 168] are extremely hard for SAT solvers, and it is not clear if clause exchange is helpful on them. Second, we only chose some benchmarks from families of benchmarks created in a similar manner, as we did not expect too many insights from highly similar formulas. Thus, we preferred to use our computational resources for more tests on different solver settings rather than running the solver repeatedly on benchmarks it probably will not solve.

As a baseline, we ran GLUCOSE 3.0 32 times with random seeds $(1,2,\ldots,32)$ on this benchmark set, analogously to the experiments we presented in the beginning of Section 2.5.1. Here, we used a timeout of 1 hour. In order to simulate what we called a "pure" portfolio solver, i. e., a solver which does not exchange clauses at all, let $t_{i.b}$ denote the time required by the solver run with random seed i on benchmark b. Then, we estimated the running time of a

Table 2.3. TOPOSAT: Grid topology, sharing only unit clauses.

cores	time(min)		speedup		timeouts	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
1	1974	2069	1.00	1.00	19	18
2	1595	1688	1.23	1.22	17	18
4	1400	1542	1.40	1.34	12	15
8	1199	1438	1.64	1.44	12	14
16	917	1342	2.15	1.54	8	13
32	604	1241	3.26	1.67	3	10
64	547	1137	3.60	1.82	4	9
128	321	1050	6.10	1.97	1	6
256	269	1003	7.33	2.06	1	6

pure portfolio solver with n processes on a benchmark b as the minimum of the first n random seeds,

$$T_{n,b} = \min_{1 \leqslant i \leqslant n} t_{i,b}$$

The results are given in Table 2.2. The first column shows the number of cores used, and the next two columns show the sum of running times $\sum_b T_{n,b}$ both for satisfiable and unsatisfiable formulas. The speedup, given in Columns 3 and 4, is quite limited, especially in the case of unsatisfiable benchmarks. However, using 32 cores only 10 satisfiable benchmarks remain unsolved, which is a clear improvement compared to 32 unsolved benchmarks on 1 core. As discussed in the beginning of this section, this is the expected behaviour on satisfiable formulas. In the case of unsatisfiable benchmarks, 3 more cases can be solved within the given time limit when comparing the configurations on 32 and 1 cores, respectively.

However, in both cases the results are rather disappointing.

We slightly increased the amount of information exchange in the next experiment, for which TOPOSAT was used. Here, we allowed for the exchange of unit clauses only. The communication structure was chosen as a 2-dimensional grid, and solvers forwarded unit clauses. The results are given in Table 2.3. Interestingly, TOPOSAT was significantly faster than GLUCOSE 3.0 also when

Table 2.4. TOPOSAT: Results for 2-dimensional grid and clauses with LBD $\leqslant 4$ exchanged.

cores	time(min)		speedup		timeouts	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
1	1974	2069	1.00	1.00	20	19
2	1460	1641	1.35	1.42	16	14
4	1182	1053	1.67	1.96	9	10
8	938	889	2.10	2.32	7	5
16	676	626	2.92	3.30	5	3
32	542	557	3.64	3.70	4	2
64	483	328	4.09	6.31	0	2
128	279	414	7.08	4.30	0	1
256	273	397	7.23	5.46	0	1

using only one core. There are mainly two reasons for this: The tests were run on a SGI UV system, equipped with Intel E5-4640 CPUs. In the first experiments, we ran 8 GLUCOSE solvers in parallel on each CPU, hence the solvers had to share the memory bus and caches, whereas in the second experiment, the solver could use the hardware alone. Second, TOPOSAT uses a different random seed which may have been a better choice for these benchmarks. Comparing the configuration on 32 cores, TOPOSAT solves 7 more satisfiable, and 12 more unsatisfiable benchmarks. Furthermore, the more solvers are used, the more benchmarks can be solved, and some speedup can be observed.

Next, we increased the amount of communication. We left the communication topology unchanged, but increased the thresholds on exported clauses, we allowed clauses with LBD at most 4 to be exported. The results are given in Table 2.4. In the case of 32 cores, this configuration is 10% faster on satisfiable inputs, but one more formula remains unsolved. On unsatisfiable benchmarks, the speedup is 4 compared to 1.67 when only exchanging unit clauses, and 8 more formulas can be solved. On satisfiable benchmarks, the increased amount of communication has some, but limited, positive impact. This configuration is slightly faster in all but the case of 256 cores, and solves more benchmarks in all but the case of 32 cores. The impact is much stronger on unsatisfiable formulas, where e. g., the configuration on 128 cores is 2.5 times faster than the

Table 2.5. Results with a complete graph as communication topology.

cores	time(min)		speedup		timeouts	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
1	1974	2069	1	1	20	19
2	1460	1641	1.35	1.26	16	14
4	1253	1095	1.58	1.89	9	10
8	992	842	1.99	2.46	10	4
16	594	584	3.32	3.54	6	2
32	449	372	4.39	5.56	2	0
64	330	241	5.98	8.59	1	0
128	302	356	6.54	5.81	1	1
256	-	-	-	-	-	-

previous one.

Thus, we sought to improve the performance on unsatisfiable formulas by increasing the amount of clause exchange. We therefore changed to topology to a complete graph, allowing solvers to send their learned clauses to all other solvers. Again, a threshold of 4 on the LBD of exported clauses was used. As expected, this came with a clear benefit on unsatisfiable formulas, as shown in Table 2.5.

Here, the configuration on 32 cores was able to solve all unsatisfiable formulas in an overall time of 372 minutes, faster than the solver on 256 cores in a grid topology, which required 397 minutes, and left one formula unsolved. On satisfiable inputs, the results are mixed, the increased clause exchange does not seem to pay off. The scatter plot in Figure 2.18 compares the results for the grid, and complete all-to-all topology, both on satisfiable and unsatisfiable inputs, for 64 cores. This plot emphasises on the positive impact of increased communication on unsatisfiable formulas, and the mixed results for satisfiable cases.

The last row in Table 2.5 does not give values for the test on 256 cores, as TopoSAT was not able to handle the amount of communication in this case. Table 2.6 shows the time required to complete one communication cycle on the benchmark SAT_dat.k80.cnf. On 64 cores, the maximum time for one cycle was 32.9 seconds, and the average time 1.5 seconds. Here, the benchmark

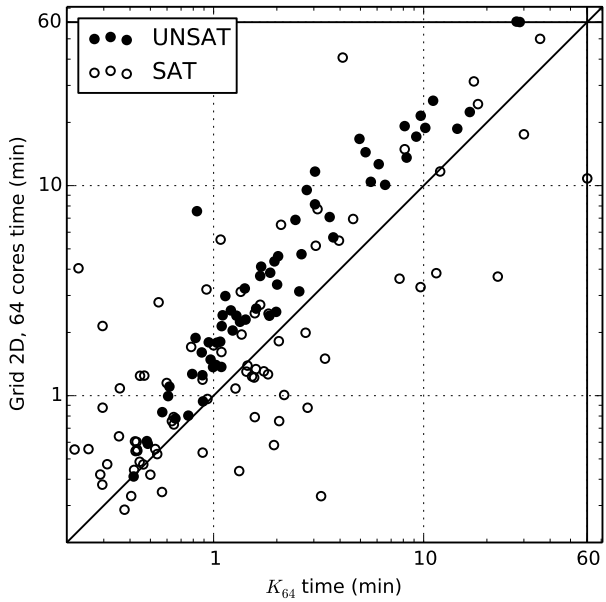


Figure 2.18. Scatter plot for grid and complete topology.

could be solved within 10 minutes. On a larger number of cores, the communication times became prohibitive, and prevented the solver from solving the benchmark.

So far, the results are as expected: The exchange of learned clauses is beneficial, especially on unsatisfiable formulas. On satisfiable benchmarks, it does not harm unless the amount of communication causes extremely large latencies. We thus sought a configuration which is still scalable, and allows for enough clause exchange to speed up the creation of large proofs for unsatisfiable formulas.

First, we considered a configuration in which the degree of each node in the communication graph depends on the total number of cores used. Every solver was connected to $\mathcal{O}(\log(n))$ neighbours. For the solver with MPI rank i,

Table 2.6. Durations of communication cycle of TOPOSAT on the formula SAT_dat.k80.cnf

		Cycle time (msec)		
Procs.	K_n edges	max.	avg.	Runtime (min)
64	2016	32885	1500	10.18
128	8128	400000	336630	-
256	32640	-	-	-

we used the neighbours with MPI ranks

$$N(i) = \{(i + j) \mod n | -n \leqslant j \leqslant n, |j| \leqslant \log(n), j \neq 0\}$$

Again, unit clauses were forwarded. This sparse topology was especially fast for large numbers of solver processes, and satisfiable formulas. On 128 cores, all satisfiable formulas could be solved in 202 minutes, which is a speed-up of 1.62 compared to the configuration on 64 cores, and the fastest configuration for satisfiable benchmarks presented so far. Using 256 cores, this configuration was slower on satisfiable formulas than when using 128 cores. As all these tests were run only once, we conjecture that this is due to the volatility of execution times on satisfiable benchmarks rather than a clear insight.

On unsatisfiable benchmarks, the solver using 64 cores is slower than its pendant on a complete communication graph. On the contrary, it scales when more cores are used, and outperforms the solver using a complete communication graph even on unsatisfiable formulas on 128 and 256 cores.

Next, we slightly increased the amount of communication, and used 16 and 32 neighbours for each node, if sufficiently many solver processes were used. For small numbers of cores, this thus corresponds to a complete communication graph. Thus, we did not run the tests repeatedly, and the results equal those from Table 2.5. Furthermore, for 256 cores the configuration with maximum node degree $\Delta(G) = 16$ equals the above configuration with $\mathcal{O}(\log(n))$ neighbours.

The configuration with 32 neighbours is faster in almost all cases, except for the configuration on 128 cores, where it is slightly slower than the one with 16 neighbours for each solver. Comparing the solver with $\Delta(G) = 16$ to the one

Table 2.7. Results for a topology with $\Delta(G) = \mathcal{O}(\log(n))$.

cores	time(min)		speedup		timeouts	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
1	1974	2069	1.00	1.00	20	19
2	1460	1641	1.35	1.26	16	14
4	1255	1179	1.57	1.75	10	9
8	852	900	2.31	2.30	9	4
16	712	674	2.77	3.07	7	3
32	517	466	3.82	4.44	3	2
64	327	358	6.03	5.78	1	0
128	202	272	9.77	7.60	0	0
256	209	221	9.45	9.36	0	0

Table 2.8. Results for a topology with $\Delta(G) = 16$.

cores	time(min)		speedup		timeouts	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
1	1974	2069	1.00	1.00	20	19
2	1460	1641	1.35	1.26	16	14
4	1253	1095	1.58	1.89	9	10
8	992	842	1.99	2.46	10	4
16	594	584	3.32	3.54	6	2
32	542	413	3.64	5.01	3	1
64	345	329	5.72	6.29	2	0
128	247	287	8.00	7.21	1	0
256	167	275	11.82	7.52	0	0

with $\Delta(G) = \mathcal{O}(\log(n))$, which uses a very similar communication graph on 128 cores, and the same graph on 256 cores, shows the volatility of the running times. On 256 cores, the first one is significantly faster on satisfiable, but slower on unsatisfiable formulas.

The different topologies discussed in this section are compared in Figure 2.19 for all, and in Figures 2.20 and 2.21 for satisfiable and unsatisfiable

Table 2.9. Results for a topology with $\Delta(G) = 32$.

cores	time(min)		speedup		timeouts	
	SAT	UNSAT	SAT	UNSAT	SAT	UNSAT
1	1974	2069	1.00	1.00	20	19
2	1460	1641	1.35	1.26	16	14
4	1253	1095	1.58	1.89	9	10
8	992	842	1.99	2.46	10	4
16	594	584	3.32	3.54	6	2
32	449	372	4.39	5.56	2	0
64	251	265	7.87	7.80	0	0
128	270	224	7.23	9.24	0	0
256	147	209	13.4	9.90	0	0

formulas. The grid topology performs significantly worse than the others on large numbers of processes used. The densest topology used, a complete communication graph, performs well on up to 64 cores, but does not scale further. On larger numbers of processes, the topologies with a large, but limited amount of communication perform best.

We may conclude that communication is helpful, as long as it does not overload the communication threads.

Furthermore, it is noteworthy that the execution times of our parallel solver are quite volatile. We exemplify this volatility by a comparison of the running times seen for the configuration on $\mathcal{O}(\log(n))$ neighbours for each node using 128 and 256 cores. For each benchmark b, we considered the speedup $\frac{t_b^{128}}{t_b^{256}}$, where t_b^{128} and t_b^{256} denote the running time of the solver on 128 and 256 cores, respectively. The arithmetic mean of these speed-ups is 1.44 on satisfiable, and 1.12 on unsatisfiable benchmarks, cf. Table 2.10. However, on satisfiable benchmarks, the minimum speed-up is 0.03. and the maximum 19.7, with a variance of 3.9. For unsatisfiable instances, the variance is significantly lower.

This volatility on some formulas is also shown in Figures 2.22(a) and 2.22(b), which compare the running times on 64, 128 and 256 cores. In both cases, some speed-up can be observed on the vast majority of benchmarks, and some instances on which one of the configurations significantly outperforms the

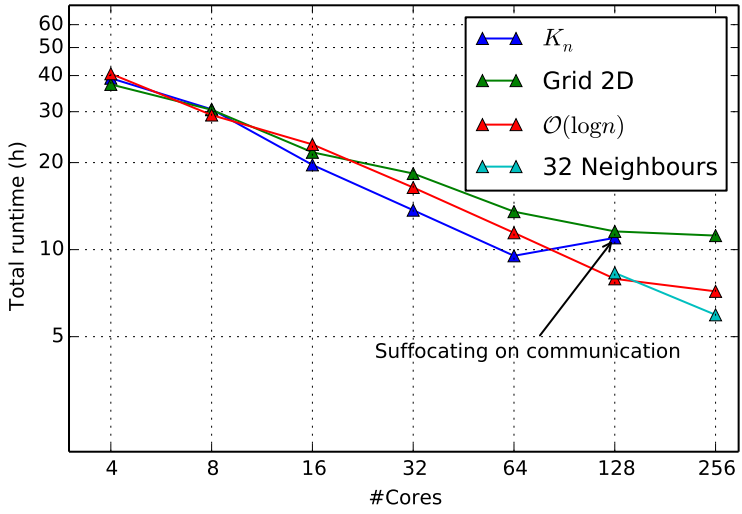


Figure 2.19. Comparison of topologies.

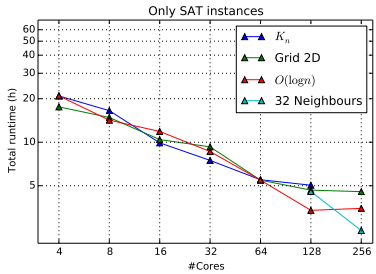


Figure 2.20. Comparison of topologies on satisfiable inputs.

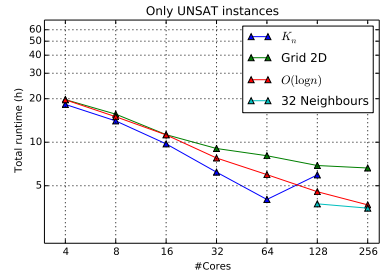
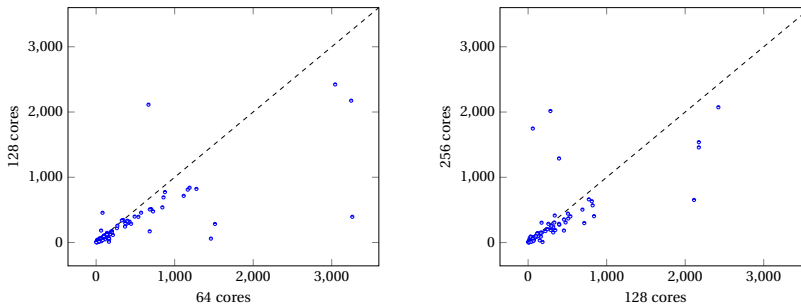


Figure 2.21. Comparison of topologies on unsatisfiable inputs.

other.

Table 2.10. Comparison of the performance on 128 and 256 cores: Speedups on SAT and UNSAT instances.

cores	average	min	max	variance
SAT	1.44	0.03	19.7	3.9
UNSAT	1.12	0.89	1.54	0.02
ALL	1.32	0.03	19.7	2.42



(a) 64 vs. 128 cores

(b) 128 vs. 256 cores

Figure 2.22. TOPOSAT: Comparisons using $\mathcal{O}(\log(n))$ neighbours.

A Note on Clause Hashing

In the above experiments, the hash values for received clauses were computed by sorting the literals within the clauses increasingly, and applying the hash function of Boost[6] afterwards. Although this worked well, we found later on that there is some serious concern with this approach. We exemplify it by the following experiment: Given three integers $0 \leqslant x_0 < x_1 < x_2 < n$, we computed the hash values for all triples and different values of n. It is in the nature of hashing the hash collisions occur in such an experiment. However, we found that the hash function used resulted in a huge amount of collisions. Table 2.11 shows the number of triples considered, and the number of distinct hash values. It can clearly be seen that the hashing creates a huge number of collisions, which in return may result in the unintended rejection of short clauses. In

[6] cf. `http://www.boost.org/doc/libs/1_38_0/doc/html/hash/reference.html`.

Table 2.11. Distinct hash values

n	triples	hashes
100	161,700	141,588
200	1,313,400	541,532
300	4,455,100	948,578
500	20,708,500	1,746,207
1,000	166,167,000	3,796,138

HordeSAT [29], the authors compute 4 distinct hash values for each received clause, and reject clauses only if each of the computed hashes has been seen before.

2.5.2 Subsequent Implementations

It is expected that communication will become a major concern for sufficiently large numbers of processes used, and on dense communication topologies independent of the implementation used. However, it is noteworthy that portfolio solvers like the prototype we submitted to the SAT competition 2016 [96] scale further. We conjecture that there are two problems with the approach used in TopoSAT. Firstly, TopoSAT imports received clauses similar to ManySAT [116]. If clauses are received between two restarts, they are imported, which involves choosing appropriate watched literals, and backtracking in case one of the imported clauses is in conflicting state. As found in subsequent experiments, it is not necessary to import clauses as fast as possible. Therefore it appears sufficient to import clauses when the solver preforms a restart. This is especially interesting as in the implementation of TopoSAT, the solver thread and the communication thread compete for the lock protecting the receive buffer.

Second, the communication threads in TopoSAT synchronise each time communication is performed. Thus, all communication is performed at the same time, resulting in much traffic at some times, whereas the MPI network is idle at other times, which may decrease the overall throughput.

2.6 Conclusion and Open Questions

In this chapter, we gave an overview on SAT, the satisfiability problem in propositional logic, and discussed proof systems. This discussion emphasised the importance of clause learning for the performance of SAT solvers, which can be related to the difference in strength of tree-like and unrestricted resolution. This is relevant both for the subsequent results and questions presented in this chapter, and also for the next chapter which deals with the parallelisation of CP solvers which use a learning scheme similar to the one used in SAT solvers. Furthermore, extended resolution was discussed. It is hard to implement it in SAT solvers, therefore, we introduced predicates in hard formulas in Chapter 4 before handing them to a SAT solver, which significantly reduced the size of the proofs created there.

Next, we discussed core techniques used in state-of-the-art SAT solvers. We presented the changes we made in our hack version of GLUCOSE 3, which won the gold medal of the GLUCOSE Hack Track of the SAT Competition 2016, and discussed the impact of each of the changes.

Afterwards, we turned to the parallelisation of SAT solvers, with a strong focus on the portfolio approach. After discussing the state of the art, we posed 11 questions to point at further research directions in this field. One of them was considered in the remainder of this section, where we presented and discussed the impact of communication on the performance of a portfolio solver on up to 256 cores. The results showed that the exchange of learned clauses strictly pays off, and that a speed-up is achievable also on unsatisfiable problems. When using more than 64 cores, communication in an all-to-all manner resulted in an amount of traffic that TOPOSAT was not able to handle, thus, it became necessary to limit the amount of communication, which we did by restricting clause exchange to some local neighbourhood of each solver process.

Future Work

Due to the versatility of SAT, problems from different domains are encoded in propositional formulas, which makes it hard to find a configuration for a SAT solver that works well on all possible inputs. This concerns both sequential and parallel solvers, but we conjecture that the difference is even larger in the

Table 2.12. Running times of a portfolio solver on the benchmark snw_17_9_CCSpreOptEncpre.cnf

cores	running time	learned clauses
4	10104	5358847
8	4643	5438918
16	2176	5337328
32	1234	5676733
64	1212	6495350

parallel case. Table 2.12 shows results when running the parallel solver which we submitted to the SAT Competition 2016 on a certain formula which could not be solved by any of the participating solvers, both in the sequential and parallel tracks. This formula encodes the non-existence of a sorting network with specific properties, which we will consider in Chapter 4.

Here, we disabled cleaning the clause database, and exchanged clauses with LBD at most 30, and size at most 100, effectively sharing about 90% of the learned clauses. This configuration is an extremely bad choice for most formulas, but worked surprisingly well here, gaining a linear speed-up on up to 32 cores. Furthermore, the overall number of clauses learned by all of the solvers involved remains stable up to 32 cores. Thus, the learned clauses are fit together to a proof of unsatisfiability without enlarging the proof in this case. It would be interesting to detect such good strategies automatically.

The 11 questions from Section 2.5.1 point to further research directions in portfolio SAT solving. Out of these, we especially consider the design of a hierarchical solver, which uses a shared clause database on nodes with shared memory, for interesting, as it allows for different further optimisations.

Furthermore, a good clause exchange strategy will be crucial when scaling the solvers on a higher number of cores.

Chapter 3

Parallelising Constraint Programming with Learning

"Those who cannot remember the past are condemned to repeat it."

JORGE AUGUSTÍN NICOLÁS RUIZ DE SANTAYANA

3.1 Introduction

Constraint Programming (CP) is a declarative way to describe solutions of problems [188]. For example, the set of constraints

$$\begin{aligned} &\text{allDifferent}(x_1, \ldots, x_n) \\ &\quad x_i \geqslant 1 && \forall 1 \leqslant i \leqslant n \\ &\quad x_i < n && \forall 1 \leqslant i \leqslant n \\ &\quad x_i \in \mathbb{N} && \forall 1 \leqslant i \leqslant n \end{aligned}$$

describes the problem of finding a set of n natural numbers between 1 and $n-1$ which are pairwise non-equal. The possibility to state constraints like "allDifferent" without encoding it to e.g. SAT formulas or linear programs makes CP an easily accessible technique. Furthermore, in many cases this kind of encoding keeps high-level structure which might be lost when translating to SAT [203]. Thus, Constraint Programming has become an active field of both research and applications, e. g., in artificial intelligence, planning and operations research.

The above problem describes the pigeon hole problem PHP_n^{n+1}. As seen in the previous chapter, this problem is extremely hard for SAT solvers. On the contrary, CP solvers like GECODE [105] can solve this problem efficiently, as they use specialised propagators — algorithms designed to handle constraints like "allDifferent" efficiently.

In this chapter we will describe the parallelisation of the CP solver CHUFFED [62], which is a CP solver with learning, i. e., it uses clause learning techniques as used in CDCL SAT solvers. The results from this research were published in [100].

We will begin with an introduction on Constraint Programming and solvers for it in Section 3.2. Next, we discuss the lazy clause generation (LCG)-approach in Section 3.3, which extends CP solvers by clause learning. Finally, we discuss our work in parallelising CHUFFED.

3.2 Constraint Programming

Constraint Programming includes the description of Constraint Satisfaction Problems (CSP) as well as search strategies for finding solutions for these. We will firstly introduce CSPs, give some motivating examples of how they are used, and then turn to techniques for finding solutions efficiently.

3.2.1 Constraint Satisfaction Problems

We will mostly follow the notation from [188] in this section. A Constraint Satisfaction Problem (CSP) is given by a triple $\mathcal{P} = (X, D, C)$ where $X = (x_1, \ldots, x_n)$ is a n-tuple of variables. The domains of these variables are $D = (D_1, \ldots, D_n)$. The set of constraints C is a t-tuple $(C_1, \ldots, C_t)$. Each constraint $C_i = (R_{S_i}, S_i)$ is a tuple where $S_i = scope(C_i)$ is the set of variables appearing in C_i, and R_{S_i} is a relation on the variables in S_i.

Example 3.1. Consider the CSP $\mathcal{P} = (X, D, C)$ with two variables and two constraints.

$$X = (x_1, x_2)$$
$$D = (\{0, \ldots, 5\}, \{2, 3\})$$
$$C = (x_1 \geqslant 1, x_1 + x_2 \leqslant 4)$$

The scope of the constraints is given by

$$\text{scope}(x_1 \geqslant 1) = \{x_1\}$$
$$\text{scope}(x_1 + x_2 \leqslant 4) = \{x_1, x_2\}.$$

□

A solution to a CSP is a n-tuple $A = (a_1, \ldots, a_n)$ such that

$$a_i \in D_i \qquad \forall 1 \leqslant i \leqslant n$$
$$A \models R_{S_i} \qquad \forall 1 \leqslant i \leqslant t$$

where the second constraint denotes that the projection of A on the variables in S_i satisfies R_{S_i}. In most cases, the constraints are given implicitly. The set of all solutions to $\mathcal{P}$ is denoted solns($\mathcal{P}$). In Constraint Programming, we may consider the decision problem solns($\mathcal{P}$) $\neq \varnothing$, compute the set of all solutions, or search a solution which maximises some objective function $\sigma : X \mapsto \mathbb{Q}$.

It can be seen that $CSP \in NP$ if, given a solution, every constraint can be checked in polynomial time. Furthermore, NP-complete problems as SAT can be encoded as CSP easily, thus, solving the decision problem of a CSP is NP-hard.

3.2.2 Applications

Constraint Programming is used in a wide field of applications, among which there are lots of real-world problems. Nurse rostering [177] asks for shifts for nurses, and is considered in a plethora of research papers. Other problems are vehicle routing [110] both in general of specific settings. For example, the optimisation of routes used for snow plowing in Pittsburgh is solved using constraint programming [135].

Earth observation management [36] manages the pictures taken by satellites in order to minimise overlaps. This is a computationally challenging problem which was solved using constraint programming.

The train lines connecting Australian coal mines with harbours were analysed and optimised in [118]. More examples like network planing or scheduling can be found in [188].

MiniZinc is language which is widely used in constraint programming [165]. Listing 3.1 shows the pigeon hole problem PHP_n^{n+1} in MiniZinc.

Listing 3.1. PHP_n^{n+1} in MiniZinc.

```
include "globals.mzn";
int: n;
int: pigeons=n+1;
int: holes=n;
array[1..pigeons] of var 1..holes: ar;

constraint alldifferent(ar);

% n can be defined here, or in a data file
n=5;
solve::int_search(ar, input_order, indomain_min, complete) satisfy;
```

First, integer variables for the number of pigeons and holes are created. Next, an array “ar” variables for the hole that pigeons are assigned to. The constraint “alldifferent” assures that all variables in this array take pairwise different values, so no two pigeons may be located in the same hole. The next line defines the value n. This may also be done in a separate data file, which allows for separating the problem description from the concrete data. In the last line, the desired solving process is given. The keyword “int_search” describes a backtracking search on integer variables. In this case, variables from the array “ar” are branched on. The order of branchings is “input_order”, so variables are chosen according to their order in the array. The first value chosen for branching is the minimum value in their domain, denoted by “indomain_min”. Complete is a parameter describing a complete search — no other value is actually supported here [35]. Finally, the keyword satisfy makes this search a decision problem.

MiniZinc serves as modeling language which is easy to understand for users. When the modeling is done, it is translated to FlatZinc, a low-level language that many different solvers can parse. During this compilation process, global constraints like “allDifferent” can either be rewritten to a trivial encoding, or be replaced by a constraint the respective solver supports. The translation of the above PHP-example is shown in Listing 3.2. Here, the constraint “alldifferent” has been replaced by “g12fd_int_all_different”, as the libraries of the g12 solver were used in the compilation process.

Listing 3.2. PHP_5^6 in Flatzinc.

```
predicate g12fd_int_all_different(array [int] of var int: x);
var 1..12: X_INTRODUCED_0;
var 1..12: X_INTRODUCED_1;
var 1..12: X_INTRODUCED_2;
var 1..12: X_INTRODUCED_3;
var 1..12: X_INTRODUCED_4;
var 1..12: X_INTRODUCED_5;
var 1..12: X_INTRODUCED_6;
var 1..12: X_INTRODUCED_7;
var 1..12: X_INTRODUCED_8;
var 1..12: X_INTRODUCED_9;
var 1..12: X_INTRODUCED_10;
var 1..12: X_INTRODUCED_11;
var 1..12: X_INTRODUCED_12;
array [1..13] of var int: ar:: output_array([1..13]) = [X_INTRODUCED_0,
    X_INTRODUCED_1,X_INTRODUCED_2,X_INTRODUCED_3,X_INTRODUCED_4,
    X_INTRODUCED_5,X_INTRODUCED_6,X_INTRODUCED_7,X_INTRODUCED_8,
    X_INTRODUCED_9,X_INTRODUCED_10,X_INTRODUCED_11,X_INTRODUCED_12];
constraint g12fd_int_all_different(ar);
solve :: int_search(ar,input_order,indomain_min,complete) satisfy;
```

3.2.3 Consistency and Propagators

An important concept in constraint programming is consistency. As we will discuss in the subsequent section, it is crucial for the performance of CP solvers.

In Example 3.1, the domain of variable x_i is given by $\{0\ldots,5\}$. However, the constraint $x_1 \geqslant 1$ forbids the assignment $x_1 = 0$, thus, the value 0 can be deleted from the domain of x_1 without deleting solutions. Formally, a CSP is denoted node consistent if for every unary constraint, i. e., constraints $c_k = (R_{S_k}, S_k)$ with $\text{scope}(c_k) = \{x_i\}$ for some i, the domain D_i is a subset of the values allowed by the relation R_{s_k}, i. e.,

$$\text{scope}(C_k) = \{x_i\} \Rightarrow D_i \subseteq R_{S_k}$$

holds. A CSP $\mathcal{P} = (X, D, C)$ easily be transformed in a node consistent CSP $\mathcal{P}' = (X, D', C)$ by removing all values from variable domains which cannot be part of a solution due to some unary constraint. Formally, D' can be computed

as follows.

$$D'_i = D_i \cap \bigcap \{R_k : \text{scope}(C_k) = \{x_i\}\}$$

Example 3.2. The CSP from Example 3.1 can be turned into a node consistent CSP $\mathcal{P}' = (X, D', C)$ by removing the value "0" from the domain of x_1.

$$\begin{aligned} D'_1 &= \{0,\ldots,5\} \cap \{c \in D_1 . c \geqslant 1\} \\ &= \{1,\ldots,5\} \end{aligned}$$

□

A stronger form of consistency is arc consistency. In this case, binary constraints are considered. Let C_k be a constraint on variables x_i, x_j. If there exists a value in $c \in D_i$ such that C_k cannot be satisfied when assigning c to x_i, c can be removed from D_i without reducing the set of solutions of the respective CSP, as depicted in the following example.

Example 3.3. Consider again the CSP from Example 3.2. From the second constraint, we can derive

$$\begin{aligned} x_1 &\leqslant 4 - x_2 \\ &\leqslant 2 \qquad\qquad \text{because } x_2 \geqslant 2 \end{aligned}$$

and so the CSP $\mathcal{P}'' = (X, (\{1,2\},\{2,3\}), C)$ has the same solutions as $\mathcal{P}$. □

There are further, stronger forms of consistency. Obviously, the concept of node- and arc consistency can be extended to constraints of arbitrary cardinality. More forms of consistency together with a detailed analysis and algorithms can be found in [188].

With the concept of consistencies, we may now consider propagators. A propagator is a function $f(X, D, C) = (X, D', C)$ such that the set of solutions remains unchanged, and the domains are not extended [170]. Thus, propagators create some form of consistency.

$$\begin{gathered} D'_i \subseteq D_i \\ \text{solns}(X, D, C) = \text{solns}(X, D', C) \end{gathered}$$

The drawback of this definition is that the identity function is also a valid propagator. Furthermore, a propagator may also be a decision procedure which returns empty domains for all variables if the problem is unsatisfiable. Thus, this definition includes both trivial and coNP-hard problems.

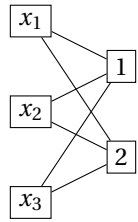


Figure 3.1. Bipartite graph for an alldifferent constraint.

There exist propagators which consider more than one constraint. For example, [4] uses a preprocessing step to incorporate implications into propagators for linear inequalities. In the following, we will restrict our viewpoint to local consistencies. This is, propagators f_c are connected to some constraint c, and only check if variable domains are consistent with respect to this constraint.

For example, for the alldifferent constraint there exist propagators which are based on graph matchings[87]. Figure 3.1 shows a bipartite graph for the constraint alldifferent(x_1, x_2, x_3) with $x_1, x_2, x_3 \in \{1,2\}$, which is PHP_2^3.

While it is extremely hard for SAT solvers to prove unsatisfiability of this problem, CP solvers can handle it efficiently. The core idea is that every feasible solution is a maximum matching on a this graph which covers all variable nodes. By the theorem of Hall [114], such a matching cannot exist in the above case as the cardinality of the left-hand side is larger than the one of the right-hand side of the graph. Some CP solvers as GECODE [105] come with very strong propagators for this constraint, allowing them to massively prune variable domains. In the case of pigeon hole problems PHP_n^{n+1}, GECODE can prove unsatisfiability for $n = 10^6$ in 3 seconds on a standard computer, whereas $n = 15$ is intractable for SAT solvers.

Another global constraint which is frequently used is a linear inequality $\sum_i c_i x_i \leqslant k$ for constants c_i and variables x_i. The MiniZinc homepage lists numerous global constraints [1]. Furthermore, very specialised constraints exist. For example, [207] discusses propagators for finding Steiner trees on graphs.

Algorithm 8: Solving a CSP by Backtracking

Data: CSP $\mathcal{P} = (X, D, C)$, Propagators F
Result: SAT and a solution, if there is a solution, UNSAT and $\varnothing$ otherwise

```
1  if ∅ ∈ D then
2      return (UNSAT, ∅);
3  if |d_i| = 1 ∀ d_i ∈ D then
4      return (SAT, D);
   /* Check if some propagator applies                    */
5  if ∃ f ∈ F. f(P) = (X, D', C) ∧ D' ⊊ D then
6      return backtrack((X, D', C));
   /* Branch, if no propagator applied                    */
7  Choose x_i with |D_i| > 1 ;
8  Choose c ⊂ D_i with |c| > 0;
9  if backtrack((X, D, C ∪ {x_i ∈ c}) = (SAT, A) then
10     return (SAT, A);
11 return backtrack(X, D, C ∪ {x_i ∉ c});
```

3.2.4 Backtracking

We now turn to algorithms for finding solutions to CPs. In the following, we will assume finite variable domains. In this case, one might check all $\Pi_{x_i \in X} |D_i|$ possible variable assignments until a solution is found, or all assignments have been checked. Thus, if the constraints can be checked in polynomial time, this variant of CP is NP-hard. This approach is only feasible for small numbers of variables, and small variable domains. Hence, backtracking algorithms for constraint programming interleave the search, performed by the backtracking, with reasoning in the form of propagations, similarly to the DPLL algorithms described in Section 2.2.3.

Algorithm 8 shows the concept of interleaving search and reasoning. It performs propagations until no more variable domain can be shrunk, i. e., a fixed point has been reached. If this does not yield neither a solution nor unsatisfiability, a branch is done by splitting the domain of one variable in two disjoint subdomains, and searching for solutions in both of them. In this example, it is not specified which variable is chosen to branch on. The above-mentioned

Algorithm 9: Solving a CSP by Branch&Bound

Data: CSP $\mathcal{P} = (X, D, C)$, Propagators F, objective function o
Result: SAT if there is a solution, UNSAT otherwise

```
1 (s, a) := backtrack(P, F, o);
2 while s ≠ UNSAT do
3     objective := o(a);
      /* Tighten the bound on the objective value                    */
4     P := (X, D, C ∪ {o(x) < objective});
5     (s', a') := backtrack(P, F, o);
6     if s' ≠ UNSAT then
7         (s, a) := (s', a');
8 return (s, a);
```

rule “in_order" would choose x_i such that $|D_j| = 1$ for all $j < i$. Other rules may e. g., choose a variable with minimum domain size. Analogously, there are different strategies to split the variable domains. In Figure 3.1, the rule “indomain_min" branches by the assignment $x_i = \min D_i$, assuming that there is some ordering on D_i. The effectiveness of this approach heavily depends on the ability of the propagators to prune the search space. Using sufficiently strong propagators, some problems can be solved without any search, e. g., the above mentioned problem PHP_n^{n+1}. Still, this approach suffers from the same weakness as the DPLL algorithm. After searching one branch of the search tree, no information learned during this search is re-used when continuing search. Techniques which allow for overcoming this weakness will be discussed in Section 3.3.

3.2.5 Backtracking and Branch&Bound

In the previous section, we considered the question whether there exists a solution for a CSP $\mathcal{P} = (X, D, C)$. Such algorithms can easily extended to deal with optimisation problems. Here, an objective function $o : X \mapsto \mathbf{Q}$ is given, and a solution which minimises o is sought. Maximisation problems can be turned into minimisation problems by minimising $-o(x)$. An optimum solution x for a COP is a solution such that for all other solutions x', $o(x) \leqslant o(x')$.

```
1 var 1..5: a;
2 solve maximize a;
```

```
1          a = 1;
2          ----------
3          a = 2;
4          ----------
5          a = 3;
6          ----------
7          a = 4;
8          ----------
9          a = 5;
10         ----------
11         ==========
```

Figure 3.2. Example for slow convergence of the Branch&Bound routine. The left-hand side shows a tiny MiniZinc example of an optimisation problem, and the right-hand side the output of the G12lazy solver when run on it.

A simple algorithm for solving constraint optimisation problems (COP) by Branch&Bound is given in Figure 9. Here, the backtrack procedure is called repeatedly. If it finds a solution, a tightened bound on the objective value is added to ensure a convergence towards solution closer to the optimum. In practice, the tightened bound may be added to the CSP within the backtracking algorithm. This prevents the backtracking routine from searching parts of the search space repeatedly[1].

However, also the convergence speed towards an optimum solution may be quite slow, as the solver only solves decision problems repeatedly without considering the objective value. Figure 3.2 shows an artificial example for this. By default, the solver searches for solutions by branching on minimum values within the variable domain. In this case, the complete domain for variable “a" is traversed, and the bound on the objective value is decreased by 1 in every iteration.

Figure 3.3 shows data collected during a run of CHUFFED on an optimisation benchmark from the MiniZinc Challenge 2013 [100][2]. In both plots, the horizontal axis represents the time, and the vertical axis shows the current

[1] Cf. the source code of solvers like CHUFFED (`https://github.com/geoffchu/chuffed`) or GECODE `http://www.gecode.org/`.

[2] see also `http://www.minizinc.org/challenge.html`.

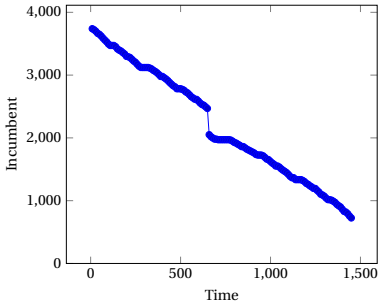


Figure 3.3. cargo: slow convergence towards optimal result.

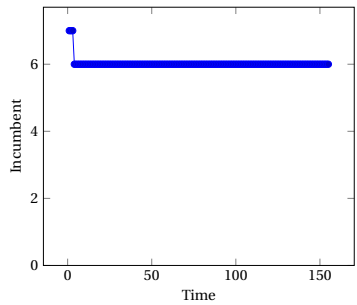


Figure 3.4. mqueens: slow proof of optimality.

bound on the objective value. In the example on the left-hand side, denoted `cargo`, many solutions are found, and the objective value is improved in many small steps. Finding an optimum solution and proving its correctness takes almost 1,500 seconds. The example on the right-hand side, `queens`, is different. Here, a solution with objective value 6 is found in 3 seconds. Afterwards, proving optimality of this solution, i. e., traversing the search space and concluding that no better solution exists, takes another 150 seconds.

A reason for these convergence behaviours can be seen in the following experiment. For both problems, we injected an artificial bound on the objective value. Then, we solved the respective decision problem. Figure 3.5 shows the running times depending on the injected bounds for the first problem, cargo. Here, the maximum solving time is 30 seconds, many calls to the solver terminated within at most 3 seconds. Furthermore, the proving that no solution with objective value better than 714 exists is extremely fast. This matches the result seen in Figure 3.3, where the solver terminates short after having found an optimum solution.

Figure 3.6 shows the results for this experiment on the benchmark “mqueens”. Here, finding some solution is fast. On the contrary, proving that no solution with objective value at most 6 exists is the hardest problem.

These different hardnesses make it hard to give a good strategy for a Branch&Bound solver. In the first example, it appears promising to speed up the solving process by conducting a binary search on the objective value. However, when solving a formula similar to the second example this may be a

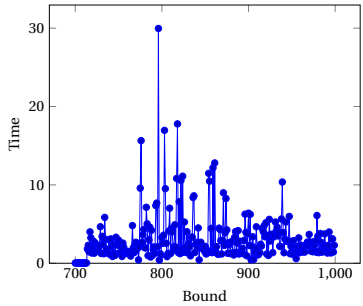


Figure 3.5. `cargo`: time for decision problem, depending on bound on objective value.

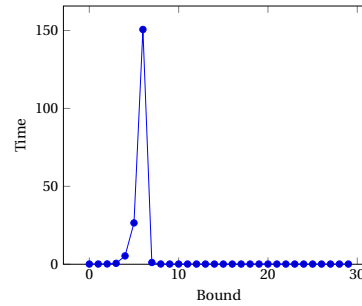


Figure 3.6. `queens`: time for decision problem, depending on bound on objective value.

bad approach, as one might hit the hardest, and unsatisfiable case first. In the worst case, the solver might time out without finding any solution, although finding suboptimal solutions is not too hard in many cases. Running time distributions as in the second example can be found in many applications, e. g., [39, 150]. We will show solutions for this problem in Section 3.5.

3.3 Constraint Programming with Learning

Supplementing Constraint Programming with learning is a very active field of research. We restrict this section to a brief overview about some techniques and results as necessary for the subsequent sections.

The methods described so far — interleaving search with propagations — allow for an efficient traversal of the search space. Nevertheless, it suffers from the same weakness as the DPLL procedure for propositional logic. The solver does not learn from its search, and thus repeats searching similar parts of the search space. In Chapter 2, the CDCL technique (Conflict Driven Clause Learning) was described as a way to overcome this weakness. Similar approaches were considered in Constraint Programming more than two decades ago [86]. They were used in two different ways. Firstly, new constraints were added whenever the solver had to backtrack, trying to avoid repeated search. Second, it is crucial to remember which parts of the search space were visited already when

restarting the solver. This can e. g., be done to change the variable ordering, or to switch to a different search strategy [192]. Unfortunately, it is hard to find good reasons for conflicts, and store them efficiently [86, 132]. Hence, Codish et al. [170] suggested to use a resolution based approach as in CDCL clause learning. The naïve approach would be a direct translation of the respective CSP to SAT in CNF, which in some cases is prohibitive because the SAT encoding may grow extremely large. Furthermore, high-level structures of a problem may be lost by this translation.

Instead, they suggested a lazy translation, denoted LCG (Lazy Clause Generation). Similarly to the architecture in SMT solving [167], propagators are provided for each constraint. Whenever branching or propagations yield variable domains that allow for deriving implied bounds from the respective constraints, these propagators shrink variable domains accordingly. Furthermore, when a conflict occurs, these propagators are capable of explaining their propagations in terms of clauses.

Whenever a conflict occurs, these clauses are used to derive a conflict clause by resolution. Consider the following example from [203].

Example 3.4. Let x, y, z denote integer variables such that $x, y, z \in [0, 6]$, and boolean variables b, c. Furthermore, let the following constraints be given.

$$z \geqslant y \qquad (1)$$
$$b \Rightarrow y \neq 3 \qquad (2)$$
$$c \Rightarrow y \geqslant 3 \qquad (3)$$
$$c \Rightarrow x \geqslant 6 \qquad (4)$$
$$4x + 10y + 5z \leqslant 71 \qquad (lin)$$

Figure 3.7 shows the implication graph of a possible run of a CP solver on this problem. The solver first branches on $[x \geqslant 5]$, marked in blue letters. This causes the propagator for the linear inequality "lin" to propagate $[y \leqslant 5]$. The next branch, b, implies $[y \neq 3]$. Next, the third branch on c triggers further propagations, which yield $[y \geqslant 4]$, $[x \geqslant 6]$ and $[z \geqslant 4]$, which is infeasible with respect to the linear inequality.

As in Section 2.2.4, this implication graph can be analysed to create a 1UIP conflict clause. Here, a possible conflict clause is $(c \wedge [y \neq 3]) \rightarrow false$, which is $(\neg c \vee [y = 3])$ in clausal form. The solver may then backtrack by one decision, and propagate the newly learned clause to $\neg c$.

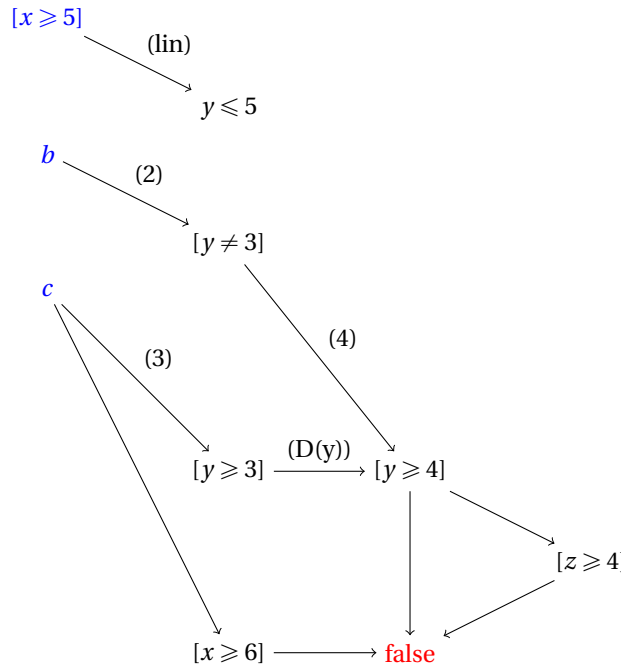


Figure 3.7. Implication graph and conflict analysis.

Sufficiently smart propagators may even yield stronger conflict clauses, see e. g., in [101]. Here, the linear propagator for the linear inequality can derive a stronger explanation with respect to the branch $x \geqslant 5$ as follows:

$$\begin{aligned} 4x + 10y + 5z &\leqslant_{(y \leqslant z)} 4x + 15y \\ &\leqslant_{(x \geqslant 5)} 15y + 20 \\ &\leqslant 71 \end{aligned}$$

Thus, one may derive that $y \leqslant \lfloor \frac{51}{15} \rfloor = 3$, and the nogood $x \geqslant 5 \Rightarrow y \leqslant 3$ can be learned, which allows for a backtrack to the first decision level, and propagating $y \leqslant 3$. □

Figures 3.8 and 3.9 show architectures for this scheme. In the first case, the SAT solver performs the search, as in the DPLL(T) approach [167]. Decisions

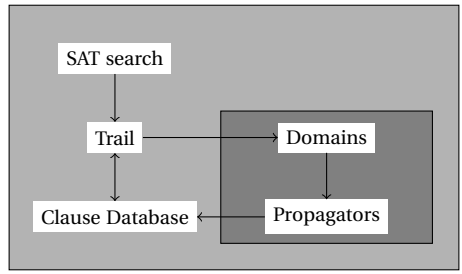


Figure 3.8. Search driven by SAT engine.

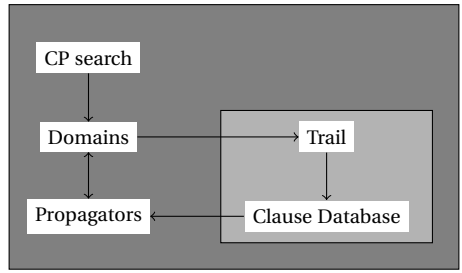


Figure 3.9. Search driven by FD engine.

and propagations of the SAT solver are given to the CP solver, which in return computes propagations and conflicts, respectively. Feydy et al. changed the roles, and made the CP solver[3] control the search process [102]. Thus, their solver can use search strategies given by the user, and uses the SAT solver only as a propagator for learned clauses. CHUFFED is more flexible. Here, the search can be driven by both solvers, and this choice can be flipped during the solver run [192].

In order to use these learned clauses, the embedded SAT solver must understand its predicates. LCG solvers therefore introduce boolean variables $x \geqslant c$, i. e., they use an unary encoding. If the variable domains are large, this can create a huge amount of variables and constraints. Therefore, solvers like CHUFFED create these variables lazily [102].

[3] In their paper the solver is called FD (Finite Domain) solver. For consistency, we denote it CP solver here.

Among other solvers, this approach is implemented in CHUFFED [62], which we will use in the subsequent sections.

Despite the elegance of the lazy propagation approaches, both in SMT and LCG, it is important to note that the implementation of efficient propagators is highly non-trivial, and in some cases, using pure propagations is even harmful. Propagators only explain themselves in terms of clauses. Even for simple constraints like XOR, a purely clause-based encoding has exponential size, cf. Section 2.1.3. Thus, the propagator may return an exponential number of different clauses as explanations for its propagations. Therefore, it is important to choose whether to encode a constraint directly, or to propagate it [5]. Some solvers try to automate this choice: They start with a propagation based approach. If a the propagator for a constraint is used often, a direct SAT encoding for this single constraint is added lazily [3].

3.4 Parallel Constraint Programming

There are different approaches to parallelising constraint programming solvers. We will first consider some approaches for parallel implementations of classical CP solvers, i. e., solvers without learning, and then briefly repeat the core insights of parallel SAT solvers, as they use learning mechanisms. Afterwards, we present some results on parallel CP with learning in the next section.

A quite straightforward approach in parallelising CP solvers is the portfolio approach: Run different solvers in parallel, and abort once the first solver has finished. In case there are more solvers available than computational resources, [7] discusses approaches to choose a good subset of the available solvers. In the context of optimisation problems, the solvers running in parallel may exchange bounds on the objective value, as it is done in [8]. Additionally, the authors showed that some solvers are faster than other in different phases of their run: Some are faster in finding some solutions, whereas others are faster in proving optimality.

Another prominent approach is to split the search space, and run solver instances in parallel on the respective subspaces. This can be done by creating sets of cubes, i. e., conjunctions of predicates. For example, let x denote an integer variable with lower and upper bounds $lb(x)$ and $ub(x)$ such that $D(x) \subseteq [lb(x), ub(x)]$. Then, one might split the search space by the cubes

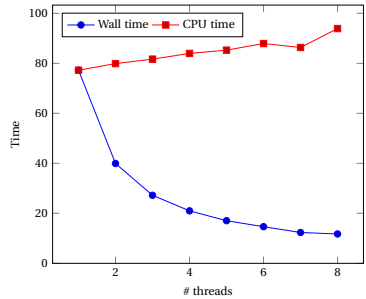


Figure 3.10. mqueens: scaling behavior of Gecode.

$x \leqslant \left\lfloor \frac{ub(x)+lb(x)}{2} \right\rfloor$ and $x > \left\lfloor \frac{ub(x)+lb(x)}{2} \right\rfloor$.

Figure 3.10 shows the scaling behaviour of GECODE, which follows this approach, when run on the benchmarks `queens` for up to 8 cores. As can be seen, the solver scales almost linearly. In [185, 186], the search space is split in a huge amount of disjoint parts, and every solver receives a list of cubes. Therefore, the solvers can start working independently from each other, and no more communication is required. This approach is tuned for optimisation problems in [159]. Here, the authors assume that the search strategy given with the CP problem is good in many cases, and very good solutions can be found in parts of the search space that are somewhat close to solutions that would be found by the respective search strategy, which often simulates the behaviour of greedy algorithms. The suggest to measure the discrepancy between a solver and the given search strategy by the number of branches in which the solver does not decide as provided by the strategy. This allows them to assign solver processes to different discrepancy, and run them in parallel without further communication.

The confidence in the search strategy provided is used in [64] to decide how to split the search space. On the one hand, the search for good results may be sped up significantly if a good search strategy is used, and all solver processes run on parts of the search space which are somewhat close to the path given be this branching behaviour. On the contrary, if the given strategy is bad, significantly speedups can be achieved by also sending solvers to different parts of the search space. In some cases, the authors report super-linear speedups with this approach.

Besides the domain of “classical" CP, some research has been conducted in parallelising solvers in similar domains.

Wintersteiger et al. presented a parallel version of their SMT solver Z3 in [214] which basically transfers the ideas of ManySAT [116]. This is, they allow for exchanging some short clauses between the solver threads running in parallel, and initialise variable activities randomly. The results are rather mixed, and only 4 threads are used. Furthermore, they used a set of quite easy benchmarks, which does not sufficiently show the usability of their approach on harder problems. Jordan et al. use a search space splitting approach without any knowledge exchange, and report significant improvements on up to 64 threads [130].

In the field of quantified boolean formulas (QBF), Balyo et al. used a portfolio approach [28], similar to the one they used in [29] and report super-linear speedups. Unfortunately, most of this improvement can be achieved with 2 nodes, each of which uses 4 processes with 4 solver threads, which is 32 threads altogether. The authors do not report results for lower numbers of threads. Using more cores yields a little more speedup, but the improvements are very limited.

Altogether, splitting the search space is the predominant approach for CP solvers, whereas some knowledge exchange is used more often in domains which build upon SAT solver techniques.

3.5 Parallel Constraint Programming with Learning

In [100], we built upon the parallel version of CHUFFED developed in [64] to test if, and how well, a constraint programming solver with learning can be parallelised. As CHUFFED is both kind of a SAT and CP solver, it is not clear how to parallelise it. The authors of the parallel CP solver presented in [185] write:

> Our approach relies on the assumption that the resolution time of disjoint subproblems is equivalent to the resolution time of the union of these subproblems. If this condition is not met, then the parallelization of the search of a solver (not necessarily a CP Solver) based on any decomposition method, like simple static decomposition, work stealing or embarrassingly parallel methods may be unfavorably impacted.

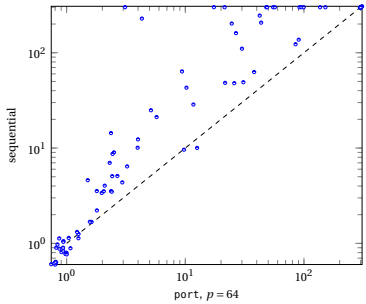


Figure 3.11. `suite`: comparison between sequential and `port`.

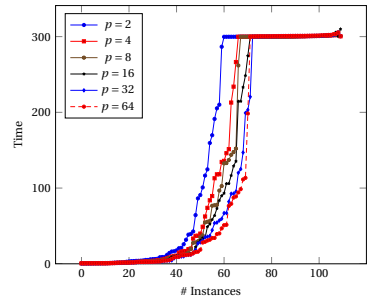


Figure 3.12. `suite`: scaling of `port` with number of cores.

In the case of learning solvers, this is clearly not the case, as clauses learned from conflicts in one part of the search space are used to avoid redundant work in subsequent search.

On the other hand, learning constraint programming solvers typically do not create large resolution-based proofs. Here, the learned clauses are rather used to avoid repeated search in a local part of the search space. In CHUFFED, for example, the maximum size of the learned clause database is fixed to a maximum of 100,000 clauses, contrary to SAT solvers where the maximum database size is increased over the time to yield a complete solver.

In the remainder of this section, we present different approaches of parallelising CHUFFED, and discuss results on a set of 110 benchmarks, denoted `suite`. These are optimisation benchmarks taken from the MiniZinc Challenges 2013 and 2014, respectively[4]. The benchmarks files can be found on github[5].

3.5.1 Portfolio-based Parallel LCG

The first approach we present here is a portfolio as it is commonly used in SAT solving. It is based on a master-slave architecture where communication is performed via MPI, which is used throughout this chapter. The VSIDS activities are initialised randomly and differently for each solver process, and some clause exchange between the solvers is allowed. Clauses learned by one solver

[4] `http://www.minizinc.org/challenge.html`

[5] `https://github.com/the-kiel/CP_benchmarks`

are sent to the master process, which dispatches them to the other solvers. Whenever a solver process finds a solution, this induces a new bound on the objective value, which is send to the other processes as unit clause. We will denote this configuration by `port`.

Figure 3.11 shows a scatter plot, which compares the running times on our benchmark suite between a sequential, and a portfolio solver on 64 cores, with a timeout of 300 seconds. On benchmarks with sequential running times of at most one second, only little speedup can be observed. Here, the overhead of starting 64 MPI processes does not pay off, as the sequential solver is fast enough.

On the contrary, for harder problems, the portfolio solver clearly outperforms the sequential version, and is able to solve 9 more problems to optimality, i. e., finding a solution and proving that no better solution exists. The scaling behaviour is given in Figure 3.12 as a cactus plot. The more processes are used, the faster the solver gets. The drawback of this presentation is that only some of the benchmarks give meaningful insights: The easiest 40 benchmarks can be solved quite fast by all solver configurations, whereas the hardest 40 benchmarks time out on all configurations. We therefore discuss the results in this section in a two-fold manner. Firstly, we measure speedups in terms of pure running times. Secondly, we ask how fast the parallel solver is when asked to provide a better solution than the sequential solver. We therefore conducted the following experiment: For every benchmark, the sequential solver is run with a timeout of 10,800 seconds. We both record the running time, and the objective value of the best solution found within this time. Next, we run the parallel solver, and abort it once a better solutions than the one from the sequential solver has been found. In case the sequential solver found the optimum solutions, the parallel solver thus had to prove optimality in order to show that no better solution exists.

Table 3.1 shows the result from this setup. The middle columns of the table show the geometric mean and median of the speedups on all benchmarks. On the whole benchmark set, the configuration using 64 cores achieves a sublinear average speedup of 9.1. However, this result already indicates that it is in fact possible to use CHUFFED in a portfolio configuration.

The right-hand side considers the speedups obtained on hard benchmark. Here, we considered a benchmark as hard if the sequential version of CHUFFED could not solve it to optimality within 300 seconds. In this way, we got 51 hard

Table 3.1. `suite`: speedups when searching for good solutions with a portfolio solver.

#CPUs	easy		all		hard	
	avg	median	avg	median	avg	median
4	0.9	1.2	3.2	2.0	13.4	4.3
8	1.2	1.8	4.7	3.7	23.5	9.0
16	1.2	1.6	6.1	4.5	39.4	14.6
32	1.3	1.7	7.7	7.4	62.1	38.0
64	1.3	1.7	9.1	10.1	84.6	42.0

problems, out of which sequential CHUFFED could solve 15 within 3 hours, and 36 timed out. On these hard problems, super-linear average speedups can be observed for all tested configurations, with a maximum of 84 when using 64 cores. Conversely, the left-hand side shows speedups for the easy benchmarks only. Here, the sequential solver performs quite well, and the overhead of starting MPI processes and collecting results afterwards prevents speedups.

In parallel SAT solving, clause exchange strategies [18, 97, 115] are crucial for the performance of portfolio based SAT solvers. Thus, we were interested in the impact of the amount of clauses exchanged between solvers. Figure 3.13 compares the result of a portfolio solver with clause exchange, and a solver which only communicates bounds on the objective value.

Except for two cases, clause sharing seems to have a positive impact. Unfortunately, the impact appears limited. Sharing clauses of size 2 clearly helps, but further experiments showed mixed results when using more communication.

We may conclude that the portfolio-based approach works surprisingly well, even though clause sharing does not work as smoothly as in the case of parallel SAT solving.

3.5.2 Splitting the Search Space

Next, we consider the more classical parallelisation approach in constraint programming, which is to explicitly split the search space into disjoint subspaces, and solve them independently. In CHUFFED, this is implemented by a work stealing approach in a master-slave-architecture. Consider a solver instance which has branched on literals d_1, d_2, d_3, as depicted in Figure 3.14.

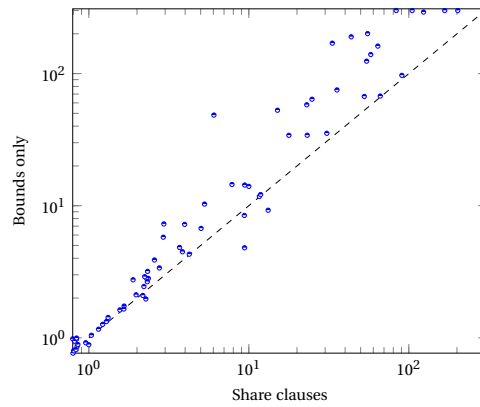


Figure 3.13. `suite`: impact of clause sharing on `port` with 64 cores.

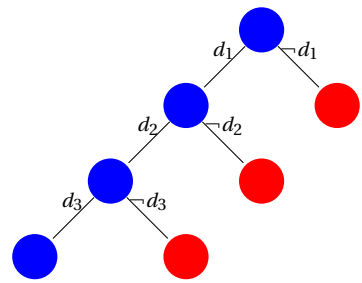


Figure 3.14. Work stealing in CHUFFED: Red nodes denote roots of the search tree on the search space induced by the newly created cubes, whereas the blue nodes restrict the search space of the solver from which work was stolen.

If the master sends a work stealing request to this solver, it will add the conjunction $d_1 \wedge d_2 \wedge d_3$ to the guiding path which describes its part of the search space, denoted by the blue node, and return the new cubes $\neg d_1$, $d_1 \wedge \neg d_2$ and $d_1 \wedge d_2 \wedge \neg d_3$, marked by red nodes. The choice of choosing the topmost 3 literals is originated in [64], where branching to the “left" side was seen as branching in the direction of a search strategy. In our experiments, this approach worked well, and changing it had no impact on the results.

Analogously to the last section, a comparison between the parallel solver

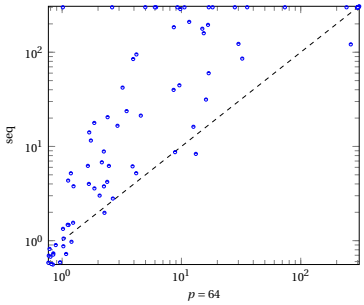


Figure 3.15. `suite`: comparison between sequential and SSS with 64 cores.

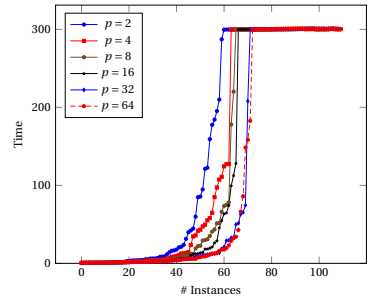


Figure 3.16. `suite`: scaling of SSS with number of cores.

Table 3.2. `suite`: speedups when searching for good solutions with a SSS solver.

#CPUs	all		hard	
	avg	median	avg	median
4	2.8	2.0	10.6	4.5
8	5.0	3.8	25.5	9.7
16	6.7	5.9	41.2	19.5
32	9.6	8.0	72.5	58.9
64	12.7	15	136.8	104.0

on 64 cores and the sequential version of CHUFFED is shown in Figure 3.15. Compared to the portfolio approach, the search space splitting solver, which we will refer to as SSS, yields stronger results, and solves 12 more benchmarks than the sequential solver. The scaling behaviour is shown in Figure3.16. Here, the difference between the setting on 32 and 64 cores seems quite small.

Table 3.2 shows the results obtained when trying to find better solutions than the sequential solver. Here, a maximum average speedup of 136 can be observed on 64 cores, and both average and median speedups are super-linear for the hard benchmarks in every configuration. Furthermore, the solver scales smoothly to 64 cores, and performance improvements can be observed whenever more computing resources are used. When considering the complete set of benchmarks, the results are sub-linear, but stronger than those achieved using the portfolio solver.

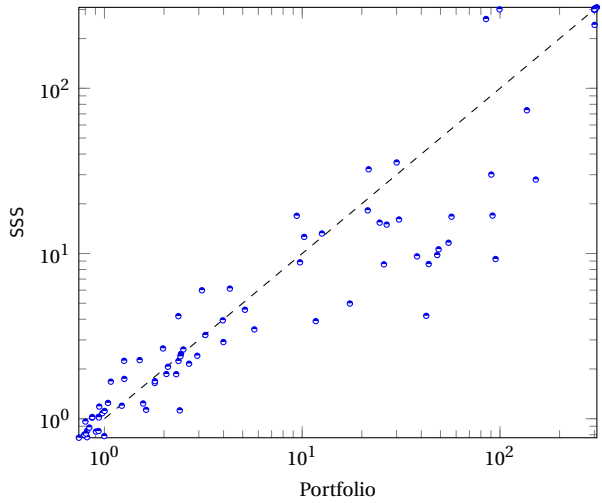


Figure 3.17. `suite`: comparison of `SSS` and `port`, 64 cores.

This is also reflected in the scatter plot in Figure 3.17, where the portfolio and search space splitting solver on 64 cores are compared. The work stealing solver performs better in average. On the other hand, the portfolio solver is faster on some benchmarks, and can solve 2 benchmarks within the time limit.

It is noteworthy that it appears highly beneficial for the solver performance to exchange bounds on the objective value. For experimental reasons, we ran the `SSS` configuration twice. In one case, we disabled any communication except for work stealing. In the other setting, we allowed the solver to communicate information about the best solution found so far. In both cases, 64 cores and a timeout of 10,800 seconds was used. The results are shown in a scatter plot in Figure 3.19: The solver which shares the bound information clearly outperforms the configuration without communication, as all solver processes can operate on a smaller subset of the search space by pruning it with these bounds. The geometric mean of speedups between these configurations is 2.3 for all benchmarks, and rises to 14 when considering only benchmarks that could not be solved within 300 seconds by the non-communicating version. These results emphasise the idea of [64] where the authors argue that it is important for the performance of parallel solvers to find good solutions quickly,

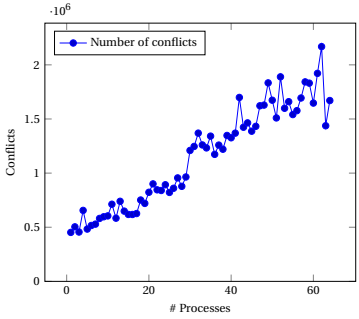


Figure 3.18. `queens`: number of conflicts in search space splitting parallel solving without clause sharing.

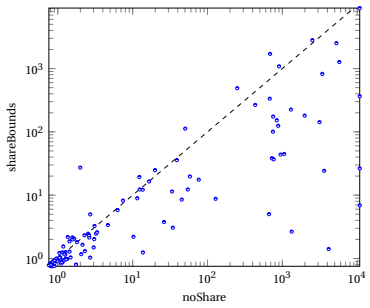


Figure 3.19. `suite`: impact of bounds sharing in `SSS` with 64 cores.

contrary to other approaches [186] which try to avoid any communication, where new bounds on the objective value are not given to the solvers while they are working on one cube.

One weakness of the `SSS` solver is shown in Figure 3.18, exemplified by the number of conflicts that occur when solving the benchmark `queens`, which was already mentioned in Section 3.2.5. In this example, the solver finds an optimum solution very fast, and most work is spent proving its optimality. However, the number of conflicts that the parallel solvers face increases with a rising number of processes used. This indicates that clause learning works less efficient if the search space is split into many small parts, as it gets less likely that learned clauses are helpful in other parts of the search space.

3.5.3 Objective Probing

The experiments presented in Section 3.2.5 indicate that for some optimisation problems, finding a good solutions is similarly hard as finding any solutions. Furthermore, when bounds on the objective value are used that are much lower than the optimum objective, the resulting subproblems can be solved quite fast in many cases. Moreover, finding good solutions early is important for parallel solvers, as it prunes large parts of the search space, and therefore redundant work can be avoided [64].

The results shown in Figures 3.21 and 3.20 substantiate this argumentation.

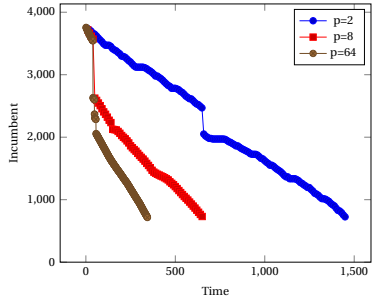


Figure 3.20. `cargo`: Development of incumbent bounds for portfolio parallel solving.

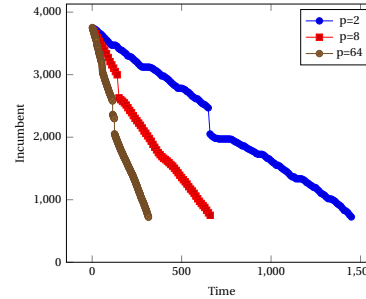


Figure 3.21. `cargo`: Development of incumbent bounds for search space splitting parallel solving.

Here, we monitored the run of both a `port` and `SSS` configuration on the benchmark `cargo`. The figures show how the best objective value improves over time. In both cases, this improvement is quite slow, and many sub-optimal solutions are found. The scaling behaviour is limited: The `SSS` configuration on 64 cores terminates after 315 seconds, and the `port` configuration after 344 seconds, which is a speedup of 4.6 and 4.2, respectively. As shown in Figure 3.5, even a sequential solver can find close-to-optimal solutions for this problem in less than 10 seconds.

Thus, one may try to push a parallel solver towards good solutions. In a sequential setting, a binary search on the objective value may be a bad choice, as the solver can get stuck on a subproblem slightly below (or for maximisation problems, slightly above) the optimum value, as these problems tend to be hard to solve. On the contrary, this is not as bad in a parallel setting. Here, one may seek a trade-off between avoiding work by pushing the solver towards good solutions, implying tighter bounds on the objective and therefore stronger pruning, and some redundant work spent with too tight bounds.

We therefore created subproblems based on the objective value. If lower and upper bounds lb and ub on the objective value are given, and n parallel solver are used for probing, we enumerate the solvers with indices $i \in \{0,\dots,n-1\}$, and compute a bound

$$bound(i) = lb + i\frac{ub - lb}{n}$$

for solver i, which is used as upper bound on the objective value in case of minimisation problems, and as lower bound otherwise. If one of the solvers find a solution which satisfies its bound, this is communicated to other solvers as common. Conversely, if it proves that there is no solution satisfying this bound, this implies a new lower bound on the objective value. In both cases, the solver can either compute a new bound on the objective, or continue working without artificial bounds. In our experiments, we used half of the available cores for probing on the objective value, and the other half was run in portfolio or `SSS` configuration, respectively. The solvers running objective probing guessed new bounds twice, and afterwards joined the other solver processes. This setting works well on a wide range of problems, however, it might be interesting to try and choose the number of probing solvers or the number of re-tries automatically.

Figure 3.22 shows the impact of this approach when solving the benchmark `cargo`. Here, the 64 cores were used, out of which 32 performed objective probing, and the others were configured as portfolio solvers. An optimum solution can be found in 3 seconds in this setting, which is a speedup of 114.7 compared to the normal portfolio setting on 64 cores, cf. Figure 3.21, and 483 compared to the sequential version of CHUFFED.

Figure 3.25 compares the plain portfolio configuration with the one using probing, denoted by `portP`. On some benchmarks, the performance is slightly decreased, whereas significant speedups can be observed on many others. Similarly, we observe super-linear speedups compared to the sequential solver, presented in Table 3.3. On 64 cores, the speedup on hard benchmarks is increased from 84.6 to 152, and the median from 42 to 133.6.

We implemented an analogous approach with search space splitting, called `SSSP`. Here, one half of the solvers guess objective values, and the other half runs a search space splitting approach with work stealing. Again, the probing solvers do at most 2 guesses, and join the other solvers afterwards. A scatter plot comparing the sequential version of CHUFFED with this approach is given in Figure 3.23, which clearly shows the positive impact of the parallelisation. The speed-ups for both all and hard benchmarks are shown in Table 3.3. Also

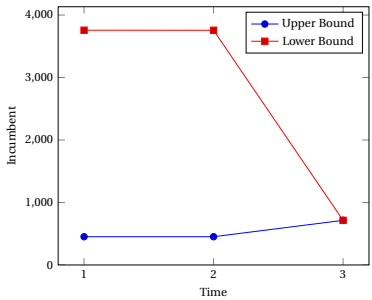


Figure 3.22. `cargo`: Lower and upper bounds on the objective in an objective probing portfolio solver.

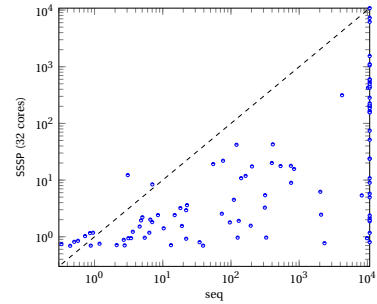


Figure 3.23. `suite`: speedup in finding good solutions with objective probing `SSSP` on 32 cores.

Table 3.3. `suite`: speedups when searching for good solutions.

#CPUs	SSSP				portP			
	all		hard		all		hard	
	avg	median	avg	median	avg	median	avg	median
4	3.7	3.2	15.5	7.4	4.3	2.4	18.2	7.4
8	6.2	4.0	41.8	20.2	6.1	4.2	33.9	8.4
16	9.6	7.6	78.9	34.6	9.2	7.0	68.4	26.9
32	12.7	13.3	121.3	58.5	11.4	8.9	107.5	77.4
64	15.6	13.8	193.8	107.0	13.6	14.6	152.0	133.6

here, the combination of objective probing and search space splitting pays off, the speed-up on 64 cores is increased from 136.8 to 193.8 when using 64 cores.

3.5.4 Further Diversification

We also compared the performance of the portfolio and the search space splitting configuration. Figures 3.26 and 3.27 show that the search space splitting configuration appears stronger in average. Conversely, there are some problems that can be solved by the portfolio configuration, and time out on the search space splitting solver. Thus, it appeared promising to build a portfolio of parallel solvers.

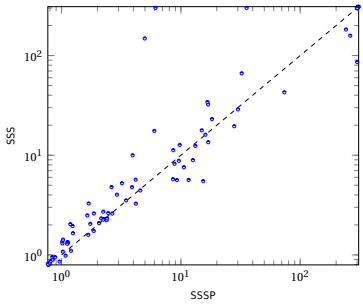


Figure 3.24. `suite`: comparison between `SSS` and `SSSP`.

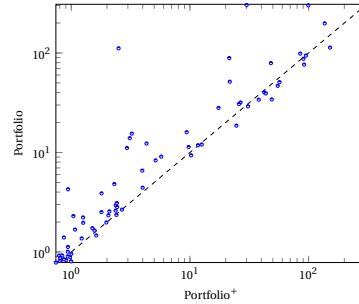


Figure 3.25. `suite`: comparison between `port` and `portP`.

Table 3.4. `suite`: speedups for `hybrid` when searching good solutions.

#CPUs	all		hard	
	avg	median	avg	median
4	4.3	3.3	18.4	6.3
8	6.3	4.7	38.2	19.9
16	9.6	6.4	79.9	43.0
32	11.7	8.8	116.0	62.4
64	15.7	16.0	196.0	140.0

We therefore split the solver processes in a similar way as before: One third of the processes ran objective probing, one third was run as portfolio solver, and one third ran search space splitting. After their second guess, the objective probing solvers joined the search space splitting solvers. When considering the speed-ups for computing good solutions, this configuration, denoted `hybrid`, is slightly faster than the `SSSP` configuration on 64 cores, as depicted in Table 3.4.

3.6 Conclusion and Open Questions

In this chapter, we gave a brief introduction to constraint programming, and discussed the parallelisation of the clause learning CP solver CHUFFED.

Both a portfolio approach and the typical approach in constraint program-

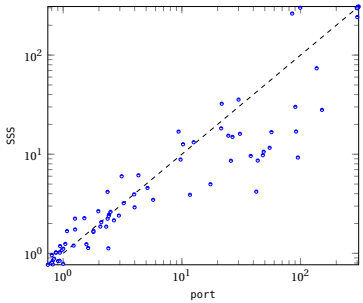


Figure 3.26. `suite`: comparison between `SSS` and `port`, 64 cores.

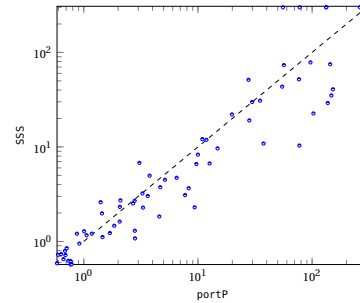


Figure 3.27. `suite`: comparison between `SSS` and `port`, 8 cores.

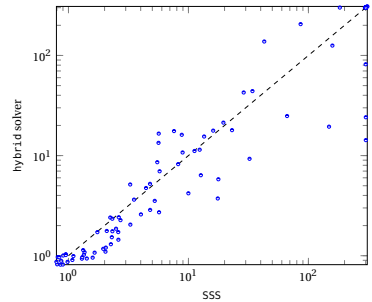


Figure 3.28. `suite`: comparison between `hybrid` and `SSS` solver, 64 cores.

ming, the splitting of the search space into disjoint subspaces, work well. Together with some optimistic probing on the objective value, we achieved significant speed-ups, especially in the search for good solutions. Here, all configurations we tried gave a super-linear speed-up.

However, there were many choices regarding that worked well in our experiments, but were done without a clear reason. This includes the amount of solvers used for objective probing, the number of re-guesses, and the splitting between portfolio and search space splitting solvers in the hybrid approach. It might be interesting to either find better choices here, or find a reason why these choices worked well.

We do not see as impressive results when parallelising the proof of optimality. Here, splitting the search space performs better than the portfolio approach, however, the speed-ups are sub-linear, even when the solver are allowed to exchange clauses. We consider this the main open problem: Parallelise learning solvers such that they scale on unsatisfiable formulas.

This work was conducted by the author during his stay in Melbourne, Australia, with Peter J. Stuckey. Furthermore, the author would like to thank Graeme Gange for many helpful discussions.

Chapter 4

Sorting Networks

"alea iacta est."

GAIUS IULIUS CAESAR

4.1 Introduction

Sorting is a fundamental and well-studied problem in Computer Science. Given a sequence $a = (a_1, \ldots, a_n)$, one seeks to order the elements in some permutation Π such that $\Pi(a)_i \leqslant \Pi(a)_{i+1}$ for $1 \leqslant i \leqslant n-1$ and some ordering $\leqslant$.

There are many algorithms which solve this problem. A rather simple sorting algorithm is Insertion Sort [80, p. 26]. The idea of this algorithm is to insert elements into a sorted prefix of the input sequence, cf. Algorithm 10. Here, a rather simple version of the algorithm is given which requires $\mathcal{O}(n^2)$ comparisons on every input sequence of length n, wheras smarter version require only a linear number of comparisons when run on sorted sequences.

More sophisticated sorting algorithms are often based on the principle of Divide&Conquer. A prominent example for this is Merge Sort [80, p. 30]. Here, the input is split in two subsequences of (almost) equal size, which are sorted. Afterwards, the results are merged to a single sorted sequence. It can easily be seen that the merging step can be implemented using a linear number of comparisons, and thus the overall running time is $\mathcal{O}(n\log(n))$. In fact, this running time is optimal for sorting algorithms which operate by comparing the elements of the input sequence [80].

At a first glance, the Divide&Conquer approach of Merge Sort seems to lend itself to a trivial implementation. The recursive calls to Merge Sort can be run in parallel, and the results are merged sequentially afterwards. An analysis of

Algorithm 10: Insertion Sort

```
   Data: Array a of length n
   Result: a is sorted
1  if n > 1 then
   |  /* Sort the first n − 1 elements                    */
2  |  sort(a[0...n-2]);
   /* Insert the last element at the right position      */
3  for i = n − 1 to 1 do
4  |  if a[i] < a[i-1] then
   |  |  /* swap                                         */
5  |  |  swap(a, i-1, i);
```

Algorithm 11: Sorting algorithm for 4 elements

```
   Data: Array a of length 4
   Result: a is sorted
1  compareAndSwap(a, 0, 1);
2  compareAndSwap(a, 2, 3);
3  compareAndSwap(a, 0, 2);
4  compareAndSwap(a, 1, 3);
5  compareAndSwap(a, 1, 2);
```

the running time reveals that the running time T can be analysed to be

$$T(1) = 1$$
$$T(n) = \max\{T\left(\left\lceil \frac{n}{2} \right\rceil\right), T\left(\left\lfloor \frac{n}{2} \right\rfloor\right)\} + \mathcal{O}(n),$$

which yields a linear running time. Here, the final call to Merge is the bottleneck, as it requires $\mathcal{O}(n)$ comparisons.

In this section we will consider one possible approach for parallel sorting algorithms, so-called Sorting Networks. Sorting Networks are data-oblivious sorting algorithms, as the sequence of comparisons they perform is independent of the input sequence. An example for such algorithms is given in Algorithm 11.

Here, a procedure compareAndSwap is used to compare two elements and swap them, if necessary. From this presentation it is not obvious that this

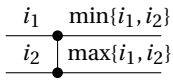


Figure 4.1. A comparator.

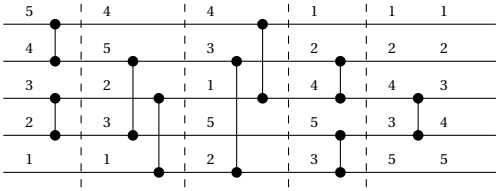


Figure 4.2. A sorting network on 5 channels, operating on the input $(5,4,3,2,1)$.

algorithm actually sorts all possible inputs in increasing order. In fact, checking this is a co-NP complete problem [173]. As we will discuss shortly, in order to ensure the correctness of a data oblivious sorting algorithm for inputs of size n, it is sufficient to check if all 2^n input vectors containing only the values 0 and 1 are sorted.

It is common to present Sorting Networks as Knuth diagrams [139], as shown in Figure 4.2. Here, the input values are connected to some channels on the left-hand side of the diagram. A vertical connection between these channels denotes a comparator, as depicted in Figure 4.1. Each comparator compares its input values, and writes them in sorted order to the channels on its right-hand side.

Here, the vector $(5,4,3,2,1)$ is given as input at the left-hand side of the network. The left-most, upper-most comparator compares its inputs, 4 and 5, and writes them in a sorted order on its output channels. In this way, the sequence becomes more and more sorted from the left to the right side of the network, and the outputs on the right-hand side of the network are sorted increasingly, when read from top to bottom.

Figure 4.3 shows an Insertion Sort, represented as sorting network. The first comparator on the left-hand side creates a sorted subsequence on the first two channels. The next two comparators insert the value from the third channel into this sorted sequence, and this process is repeated until every element has been inserted.

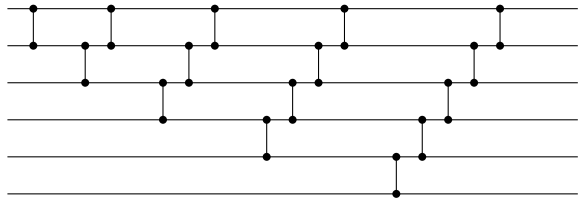


Figure 4.3. A Sorting Network performing an Insertion Sort.

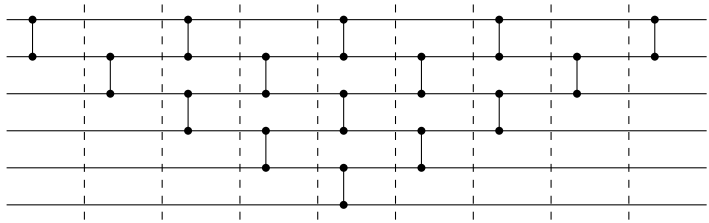


Figure 4.4. A Sorting Network performing an Insertion Sort in parallel steps.

As this implementation is data-oblivious, comparators have to be provided for every comparison that might occur in the software implementation, thus, this network consists of 15 comparators. In fact, this is sub-optimal, as there exist sorting networks for 6 input values which only require 12 comparators [139]. Each of the comparators in this network can be shifted to the left or right inside the interval between other comparators touching the same channels. Figure 4.4 shows the result of such shifts. Here, the comparators are divided into some layers, depicted by dashed vertical lines. All comparators in such a layer perform compare-and-swap operations on disjoint channels, thus they can be implemented to run in parallel. This network sorts 6 inputs in 9 parallel steps. The fact that the operations performed by a sorting network are known independently of the input data, and some of them can be performed in parallel, arose interest in using sorting networks for the implementation of parallel sorting algorithms in hardware.

Capannini et al. implemented a sorting network-based sorting algorithm on graphics cards [61], whereas Skliarova and Sklyarov used FPGAs [200]. These implementations are used e.g. for median filtering, and are promoted by FPGA

manufacturers [24].

In software implementations of Merge Sort or QuickSort, it is common to fall back to naïve algorithms like Insertion Sort on small inputs or as base case in recursive calls. Codish et al. replaced this base case by a software implementation of optimal sorting networks [72]. Here, each comparator can be represented by conditional move (CMOV) instructions, and therefore the algorithm can be implemented without any branch instructions.

In the application of SAT solving, Cardinality Networks, which are simplified versions of Sorting Networks, are used to encode cardinality constraints [11]. Given a constraint $\sum_{i=0}^{n} l_i \leqslant k$ for some literals l_i, the literals are used as inputs of a sorting network. The network itself can be encoded using $\mathcal{O}(n\log^2(k))$ clauses and variables, which outperforms e.g. sequential counters, which require $\mathcal{O}(nk)$ additional clauses and variables [198]. Furthermore, this encoding enforces arc consistency: If some k' literals are set to 1, unit propagation is sufficient to set the highest k' outputs to 1 as well. If $k' > k$, this leads to a conflict, and the solver can backtrack without performing further search in this part of the search space.

The remainder of this chapter is structured as follows. First, we will discuss the construction of efficient sorting networks in Section 4.1.1. Next, we will introduce some notation which is necessary to reason formally about sorting networks. Unfortunately, there is a flaw of precision in the literature on sorting networks, which is mostly based on different interpretations of notations. Therefore, we present correct and consistent versions of some theorems and their proofs in Section 4.2. Next, we turn to SAT encodings of sorting networks, which can be used both to find sorting networks of minimum depth, i. e., minimum number of parallel sorting steps, and prove lower bounds on the numbers of layers required to sort some n inputs. We show how the aforementioned theorems help to remove symmetries from the search space, and decompose the occurring problems into subproblems, which can be solved independently. We improve upon the SAT encodings used in former papers in Section 4.3, and show how a closer analysis of the symmetries on sorting networks can be used to set up an optimisation problem which effectively minimises the size of the SAT formulas generated to encode the existence of sorting networks with certain properties. This leads to new upper bounds, presented in Section 4.4, and the main result, a new lower bound on the depth of sorting networks for 17 elements.

Algorithm 12: Bubble Sort

```
   Data: Array A of length n
   Result: A is sorted
1  for i = n-1 to 1 do
2      for j = 0 to i do
3          if A[j] > A[j + 1] then
4              swap(A, j, j+1);
```

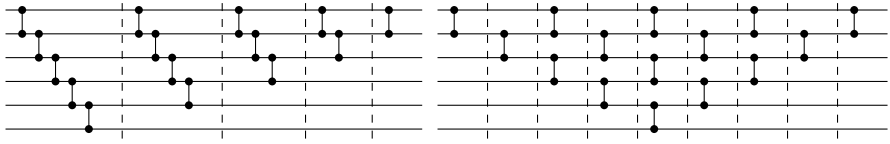


Figure 4.5. A Bubble Sort implemented as sorting network.

4.1.1 Construction of Sorting Networks

As seen in the introduction, the easiest way of constructing a sorting network is to re-use data oblivious sorting algorithms like Bubble Sort as depicted in Algorithm 12. Figure 4.5 shows the Bubble Sort for 6 elements represented as sorting network. Again, we can shift the comparators to gain layers of data-independent comparators.

Interestingly, this shift yields exactly the same sorting network as the one that was created from an Insertion Sort.

A drawback of this network is that only one layer is maximal in the sense that no further comparator can be added. This problem is overcome in the odd-even-transposition sort, shown in Figure 4.6.

These networks consists of n layers, each of them filled with comparators connecting channels $(2i, 2i+1)$ in odd, and $(2i+1, 2i+2)$ in even layers, where the layers are numbered from left to right, beginning with 1. Thus, the time required to sort the inputs is $\mathcal{O}(n)$.

It is well-known that there exist sorting algorithms like Merge Sort or Heap-Sort which require $\mathcal{O}(n\log(n)))$ comparisons. Thus, the question arose if it

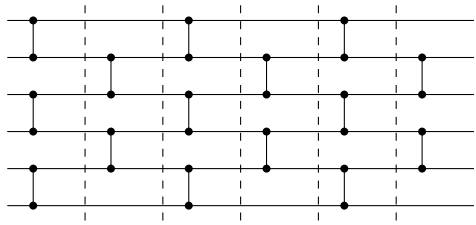


Figure 4.6. Odd-even-transposition sort.

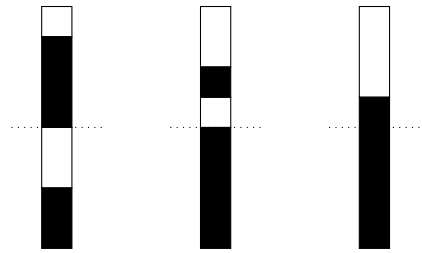
is possible to construct sorting networks which use the same magnitude of comparators.

In 1968, Batcher presented the first algorithms to construct sorting networks for which the number of comparators, denoted s_n, is bounded by $s_n \in \mathcal{O}(n\log(n)^2)$ and the number of layers, or depth, d_n, is $\mathcal{O}(\log(n)^2)$ [32]. In order to discuss these networks, the following theorem is helpful.

Theorem 4.1. *(0/1-principle [139]) If a network on n input lines sorts all 2^n sequences of* 0*s and* 1*s into nondecreasing order, it will sort any sequence of n numbers into nondecreasing order.*

Thus, it is sufficient to consider only binary input vectors to reason on the correctness of sorting networks. Batcher gave two algorithms for constructing sorting networks, Bitonic Merge Sort and Odd-Even- Merge Sort. Here we will present only the latter one. For simplicity, we assume that the input is a sequence of length $n = 2^k$ for some $k \in \mathbb{N}$.

Similar to the well-known Merge Sort algorithm, it applies a merge procedure on sorted sequences, gaining larger and larger sorted sequences until eventually the whole input is sorted. By the 0/1-principle, it is sufficient to consider binary input sequences. Figure 4.7(a) shows a binary input sequence for a merge: The upper, and lower halves are sorted. The sequence of 0s is drawn as a white box here, whereas 1s are represented by a black box. In a first step, Batcher's odd-even Merge Sort sorts the subsequences consisting of all entries with even and odd indices, respectively. The core insight behind this is that the number of ones in the odd and the even subsequences differ at most by 2. Thus, after sorting the odd and even subsequences, the result is either sorted already, or almost sorted, as depicted in Figure 4.7(b). In the latter case,



(a) First (b) Second (c) Third

Figure 4.7. The input of the merge procedure(4.7(a)), after sorting the odd and even subsequences (4.7(b)), and sorted result after final step(4.7(c))

Algorithm 13: Odd-Even- Merge Sort

Data: Array A of length 2^k
Result: A is sorted
1 **if** $k \geqslant 1$ **then**
2 OddEvenMergeSort($A[0 \ldots 2^{k-1}-1]$);
3 OddEvenMergeSort($A[2^{k-1} \ldots 2^k-1]$);
4 OddEvenMerge(A);

the data can be sorted by comparing adjacent elements, and sorting them, if necessary.

In fact, the odd and even subsequences do not have to be sorted, as both of them consist of two sorted subsequences. Therefore, it is sufficient to merge the subsequences recursively. The odd-even Merge Sort is shown in Algorithm 13, and the respective merging routine OddEvenMerge in Algorithm 14 [193].

The running time for the merging step can be seen by a recurrence relation. For an input array of size 2 exactly one comparison is performed. For some 2^k inputs, there are two recursive calls, which can be done in parallel. Afterwards, one further parallel comparison step is performed, which runs in time $\mathcal{O}(1)$. Therefore, we get

$$T_m(2) = 1$$

Algorithm 14: OddEvenMerge

Data: Array A of length 2^k, $A[0..2^{k-1}-1]$ and $A[2^{k-1}..2^k-1]$ are sorted.
Result: A is sorted

```
1 if k > 1 then
2     OddEvenMerge(A[0]A[2]...A[n−2]);
3     OddEvenMerge(A[1]A[3]...A[n−1]);
4     for i=0 to 2^(k−1) − 1 do
5         compare(A[2i+1], A[2i+2]);
6 else
7     compare(A[0], A[1]);
```

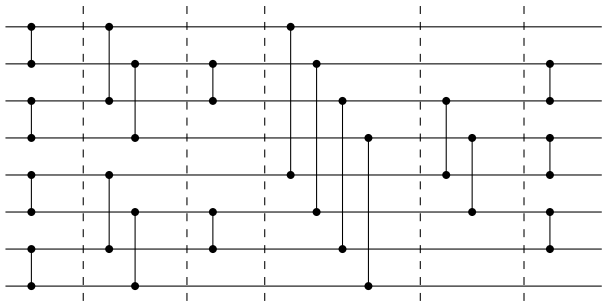


Figure 4.8. An odd-even Merge Sort network on 8 channels.

$$T_m(2^k) = 1 + T_m(2^{k-1})$$
$$T_m(2^k) \in \theta(k)$$

which is $\theta(\log(n))$ for n elements. Analogously, it can be seen that the time required by odd-even Merge Sort is $\theta(\log^2(n))$, and the number of comparators required is $\theta(n\log^2(n))$.

Figure 4.8 shows an odd-even Merge Sort for 8 elements as a sorting network. The first 3 layers sort the first and last 4 inputs independently of each other. Afterwards, the results are merged in another 3 layers.

These networks reduced the gap to sorting networks of size $\mathcal{O}(n\log(n))$ significantly. As no further progress was seen for some years, Donald E. Knuth

Table 4.1. Optimal size (s_n) and depth (d_n) of sorting networks on n inputs, for $n \leqslant 12$.

n	1	2	3	4	5	6	7	8	9	10	11	12
s_n	0	1	3	5	9	12	16	19	25	29	35 33	39 37
d_n	0	1	3	3	5	5	6	6	7	7	8	8

conjectured in [139] that $s_n \in \theta(n\log(n)^2)$ was the optimal asymptotic bound on sorting network sizes. This conjecture was refuted in 1983 by Ajtai, Komlós and Szemerédi who presented a construction of size $s_n \in \mathcal{O}(n\log(n))$ and depth $d_n \in \mathcal{O}(\log(n))$, thus closing the gap between upper and lower bound. This was a fundamental result as it showed that data oblivious sorting algorithms can be as fast as less restricted sorting algorithms.

However, their construction, called AKS network, is infamous for the large constants hidden in the big-O notation. Paterson gave a simplified construction and improved analysis in 1990, yielding an upper bound on the network size of $s_n \leqslant 6100n\log(n)$ [176]. For $n \geqslant 2^{27}$, this was further improved to $s_n \leqslant 1830\log(n) - 58657$, cf. [67]. These big constants make Batcher's Merge Sort, and other asymptotically suboptimal approaches like Parberry's Pairwise Sorter [175], superior to the AKS network for all practically relevant input sizes.

4.1.2 Bounds on Depth and Size of Sorting Networks

Further research has focussed on determining the optimal size and depth of small sorting networks. This is not only of theoretical interest, as minimally small sorting networks can used as base case e. g., in the odd-even Merge Sort. Table 4.1 shows the optimal size and depth of sorting networks on $1 \leqslant n \leqslant 12$ channels. The optimum depth values can be found in [137], and the lower bound on 9 and 10 channels was proven by Ian Parberry in 1989 [172]. The last optimum values on the size of sorting networks for 9 and 10 inputs were found by Codish et al. in 2014 [71].

For more than 10 inputs, there remains a gap between lower and upper bounds on the size of sorting networks. The most recent results here are due to Valsalam and Miikkulainen [211], who improved the upper bounds on 17 to 20 channels. These new sorting networks were found by a evolutionary search

Table 4.2. Lower bounds on the depth of sorting networks, derived from lower bounds on the number of comparators.

n	37	55	79	119	165	245
$\hat{d}_n$	10	11	12	13	14	15

Table 4.3. Best known values and bounds on optimal size (s_n) and depth (d_n) of sorting networks on n inputs, for $13 \leqslant n \leqslant 24$. The contributions of the publications discussed in this chapter are shown in boldface.

n	13	14	15	16	17	18	19	20	21	22	23	24
s_n	45	51	56	60	71	78	86	92	103	108	118	123
	41	45	49	53	58	63	68	73	78	83	88	93
d_n	9	9	9	9	**10**	11	**11**	**11**	12	12	**12**	**12**
						10	**10**	**10**	**10**	**10**	**10**	**10**

which especially considered symmetries in sorting networks.

Lower bounds on the number of comparators can be used to derive lower bounds on the depth of a sorting network. On n channels, each layer may contain at most $\lfloor \frac{n}{2} \rfloor$ comparators. If $\hat{s}_n$ and $\hat{d}_n$ denote lower bounds on the number of comparators and layers, respectively, this yields

$$\hat{d}_n \geqslant \left\lceil \frac{\hat{s}_n}{\lfloor \frac{n}{2} \rfloor} \right\rceil$$

Using the optimality result that sorting 10 inputs requires 29 comparators [71], and the following inequality by Voorhis [212]

$$\hat{s}_n \geqslant \hat{s}_{n-1} + \lceil \log_2(n) \rceil,$$

this yields lower bounds on the number of layers, some values are shown in Table 4.2

Sorting networks for up to 16 channels with optimum depth can be found in "The Art of Computer Programming" from 1973 [139], however, their optimality was only proven by Bundala and Závodný in 2014 [56]. They used a SAT-based approach which took advantage of properties of the first two layers of sorting networks, this will be discussed in detail in the next sections.

An improved sorting network for 18 channels was found by Al-Haj Baddar in 2009 [22]. She used a tool, SortNet, to assist her in hand-crafting this network.

Further improved sorting networks for 17, 19 and 20 channels were found by Ehlers and Müller in 2014 [93]. The improved upper bounds on 23 and 24 inputs were found by applying techniques from [71] to create a good prefix, and completing it to a complete sorting network using a SAT solver [90]. The optimality for the case of 17 channels could be proven by a SAT-based approach [94].

All new bounds that this thesis is involved with are marked in bold in Table 4.3.

4.2 Properties of Sorting Networks

In this section we will discuss properties of sorting networks. These will be helpful to break symmetries in the search space in the following sections. None of the theorems in this section were proven by the author of this thesis, however, there is a flaw in precision in the literature. Thus, we fix some notation, and then give consistent and correct proofs for the theorems presented here.

4.2.1 Notation and Definitions

Let us first fix some notation. Given a sequence $s = (s_1, \ldots, s_n)$, we will denote its length by $|s|$. Its i-th element is s_i, for $1 \leqslant i \leqslant |s|$. Given two indices i and j, the subsequence starting at index i and ending at index j is $s[i, j]$ for $1 \leqslant i \leqslant j \leqslant |s|$. A prefix of a sequence $s = (s_1, \ldots, s_n)$ is a sequence $s[1, k]$ for $k \leqslant n$, and a suffix is a sequence $s[k, n]$ for $k \geqslant 1$. Given two sequences a and b, their concatenation is defined as $ab = (a_1, \ldots, a_{|a|}, b_1, \ldots, b_{|b|})$. The k-wise concatenation of a is denoted by a^k.

We will denote permutations which swap the elements at indices i and j, leaving all other elements at the same positions, by (i, j).

We will denote comparators connecting channels i and j as a tuple (i, j). This comparator will sort its smaller input value to the channel i, and the larger value to channel j. The length of a comparator (i, j) is $|i - j|$. In the introduction, we considered only comparators which sort their inputs such that the numerically smaller input is sorted to the upper, and the larger input is sorted to the lower output.

Definition 4.2 (Standard Comparator)**.** A standard comparator is a tuple (i, j) such that $i < j$.

In the sorting networks presented so far, all comparators were standard comparators.

Definition 4.3 (Standard Comparator Network)**.** A standard comparator network S on n channels is a sequence of standard comparators $c_k = (i_k, j_k)$ such that $0 \leqslant i_k < j_k < n$ for each comparator.

This definition does not actually require the network to sort its inputs - the empty sequence is also a standard comparator network. To describe that a comparator network sorts its inputs, we next consider the definition of outputs of a comparator network.

Definition 4.4 (Outputs of a Comparator Network)**.** Let $\mathcal{I}$ denote the inputs of a comparator network C on n channels. By the 0/1-principle, it is sufficient to consider $\mathcal{I} \subseteq \{0,1\}^n$. Let outputs denote a function which maps a comparator network and a set of input vectors to the set of output vectors. This function can be defined inductively by computing the impact of the first comparator on C on the inputs, and applying the suffix of C to the result. If C is the empty sequence, nothing is sorted, and therefore the outputs equal the inputs.

$$\text{outputs}(\varepsilon, \mathcal{I}) = \mathcal{I}$$

Otherwise, let (i, j) denote the first comparator of C, i. e., $C = (i, j)C'$. Without loss of generality we may assume that $i < j$. We obtain $\mathcal{I}'$ by applying the comparator (i, j) on all elements of $\mathcal{I}$, and then apply C' on $\mathcal{I}'$. Formally, if we apply a comparator (i, j) on some $a \in \{0,1\}^n$, the result is

$$app((i,j),a) = \begin{cases} a, a_i \leqslant a_j \\ a[0 \ldots i-1]a_j a[i+1 \ldots j-1]a_i a[j+1 \ldots n-1], \text{else} \end{cases}$$

With this, we may give the outputs of a comparator network as

$$\text{outputs}((i,j)C', \mathcal{I}) = \text{outputs}(C', \{x | x = app((i,j), x'), x' \in \mathcal{I}\})$$

Definition 4.5 (Standard Sorting Network)**.** A standard sorting network is a standard comparator network which sorts all possible input sequences non-decreasingly.

There exist also sorting networks which a non-standard, e.g. Batcher's Bitonic MergeSort [32], as shown in Figure 4.9. These networks sort the first half of the input sequence non-decreasingly, and the second half non-increasingly. Afterwards, the resulting bitonic sequence is sorted.

Here, comparators are drawn as arcs, and the higher input value is written to the channel the arc points to. We will denote such comparators as Max-Min-Comparators.

Definition 4.6 (Max-Min-Comparator)**.** A max-min-comparator is a tuple (i, j) such that $i > j$.

Networks that contain max-min-comparators are called non-standard networks.

Definition 4.7 (Non-standard Comparator Network)**.** Non-standard comparator networks are networks created of min-max-comparators and max-min-comparators.

The outputs of such a non-standard-network can be defined analogiously to the outputs of a standard comparator network by adjusting the definition of "app".

Definition 4.8 (Non-standard Sorting Network)**.** A non-standard sorting network is a non-standard comparator network that sorts all its inputs.

Next, we consider networks which sort up to a permutation. As will be shown later on, each such network can be transformed into a standard sorting network.

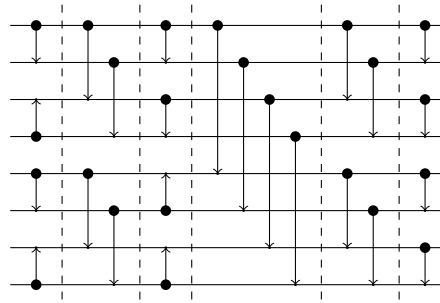


Figure 4.9. A Bitonic MergeSort on 8 channels.

4.2.2 Permutations of Sorting Networks

Definition 4.9 (Generalised Sorting Network)**.** A non-standard comparator network C is called generalised sorting network if, for some permutation Π, all its outputs are sorted after the application of Π.

Given a standard sorting network, a generalised sorting network can be obtained by permuting its channels.

Definition 4.10 (Permutations of Sorting Networks)**.** Let $C = (c_1, \ldots, c_s)$ denote a comparator network, and $\Pi : \{1, \ldots, n\} \mapsto \{1, \ldots, n\}$ denote a permutation. We define the permutation of C as $\Pi(C) = (\Pi(c_1), \ldots, \Pi(c_s))$, and $\Pi(c_k) = (\Pi(i_k), \Pi(j_k))$.

Lemma 4.11 (Permutation of Generalised Sorting Networks)**.** *Let C denote a generalised sorting network on n channels and d layers. Then, for any permutation $\Pi : \{1, \ldots, n\} \mapsto \{1, \ldots, n\}$, the permuation of C under Π, $\Pi(C)$, is also a generalised sorting network.*

Proof. ([172]) As C is a generalised sorting network, there exists a permutation Π_C such that the outputs of C are sorted after applying Π_C on them. Therefore, applying $\Pi_C \circ \Pi^{-1}$ on the outputs of $\Pi(C)$ yields sorted outputs. □

Lemma 4.12. *Every generalised sorting network which consists only of standard comparators is a standard sorting network.*

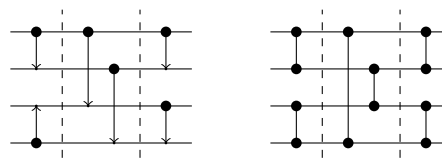
Proof. ([174]) By the definition of a generalised sorting network C, there exists a fixed permutation Π_C such that the outputs of C, permuted by Π_C, are sorted. Consider the input $(1, 2, \ldots, n)$. As C is made from standard comparators, none of them will swap its input, thus, the output is $(1, 2, \ldots, n)$, and therefore $\Pi_C = id$. □

Knuth mentioned in [137, Exercise 5.3.4.16] that non-standard sorting networks are no more powerful than standard sorting networks. He furthermore describes an algorithm, called untangle, which can turn every sorting network into a standard sorting network. This algorithm is shown in Algorithm 15. In its original version, the algorithm is applied to non-standard sorting networks. Here, we consider a more general version which applies to generalised sorting networks.

Algorithm 15: untangle

Data: Generalised sorting network $C = ((i_1, j_1), \ldots, (i_n, j_n))$ with permutation Π_C
Result: A standard sorting network

```
1  Π := Π_C;
2  for k = 1 to n do
3      if i_k > j_k then
           /* Swap channels i_k and j_k by untangling comparators
              with higher indices                                  */
4          for l = k to n do
5              if i_l = i_k then
6                  i_l := j_k;
7              else if i_l = j_k then
8                  i_l := i_k;
9              if j_l = i_k then
10                 j_l := j_k;
11             else if j_k = j_l then
12                 j_l :=i_k;
13         Π := (i_k, j_k) ∘ Π;
14     return C;
```



(a) Bitonic Merge Sort on 4 channels (b) Untangled version

Figure 4.10. A bitonic Merge Sort and the result of untangling it.

As an example, Figure 4.10 shows a bitonic Merge Sort on 4 channels, which is a non-standard sorting network, and a standard sorting network obtained by untangling.

Next, we prove the correctness of the untangle-algorithm.

Lemma 4.13. *After the k-th iteration of the outer loop, the first k comparators are min-max-comparators.*

[174]. If $i_k > j_k$, the channels are swapped. In later iterations, this comparator is not touched anymore, the claim follows by induction. □

Lemma 4.14. *After the k-th iteration of the outer loop, C is a generalised sorting network, and the respective permutation is Π.*

Proof. We show that if the claim holds before the k-th iteration, it also holds after [174]. Before the first iteration, C is a generalised sorting network with permutation $\Pi = \Pi_C$. If, in iteration k, $i_k < j_k$, nothing is done; the claim holds by induction. Otherwise, the channels i_k and j_k are swapped after the $(k-1)$-th comparator. Thus, also the outputs on channels i_k and j_k are swapped. As, by inductive hypothesis, C was a generalised sorting network before this iteration, applying $\Pi \circ (i_k, j_k)$ on the outputs yields sorted outputs. □

Lemma 4.15. *The result of the untangle algorithm, denoted C, is a standard sorting network.*

Proof. According to Lemma 4.13, C consists only of standard comparators, and by Lemma 4.14, it is a generalised sorting network. Thus, it is a standard sorting network by Lemma 4.12. □

We will use these theorems to permute the channels of sorting networks, and make the results standard sorting networks again by untangling. This is feasible due to the next Corollary.

Corollary 4.16. *Let C denote a standard sorting network. If the channels of C are permuted by some permutation Π, then untangling yields a standard sorting network.*

4.2.3 Prefixes of Sorting Networks

Next, we consider prefixes of sorting networks. As we will see, if the first layer of a sorting network is not maximum, i. e., another comparator can be added to it, this yields another standard sorting network. Thus, we may break symmetries in the search space by enforcing maximum first layers. The next lemma is based on the observation that a comparator network which sorts some set $\mathcal{I}$ of inputs also sorts all subsets of $\mathcal{I}$.

Lemma 4.17 ([172])**.** *Let $C = AB$ denote a standard sorting network. If* outputs$(A') \subseteq$ outputs(A) *for some prefix A', then $C' = A'B$ is also a sorting network.*

Proof. Assume there was an input x that is sorted by C, but not by C'. This is, there must be an output of A' that cannot be sorted by B. With outputs$(A') \subseteq$ outputs(A), this is a contradiction. □

Corollary 4.18 ([172])**.** *Consider a standard sorting network $S = L_1 L_2 \dots L_d$ on n channels and d layers. If there are two channels $i < j$ which are not connected to any comparator in the first layer, the comparator (i, j) can be added to the first layer.*

Proof. Adding a comparator to the first layer does not create any new outputs for this layer. By Lemma 4.17, S remains a standard sorting network. □

The next lemma allows for even stronger symmetry breaks by considering also permutations on the prefixes.

Lemma 4.19 ([56])**.** *Let $S = AC$ be a standard sorting network, and B denote a prefix such that* outputs$(B) \subseteq \Pi($outputs$(A))$ *for some permutation Π. Then there exists a standard sorting network BC', and C' uses as many comparators and layers as C.*

Proof. Permuting B by Π^{-1}, we get outputs$(B) = \Pi($outputs$(\Pi^{-1}(B)))$. Thus, $\Pi^{-1}(B)C$ is a non-standard sorting network by Lemma 4.17, which can be untangled to a standard sorting network by Lemma 4.15. □

The next lemma was mentioned in [56] as a hint by Donald Knuth. The proof for this lemma can be found in [55].

Lemma 4.20 (Reflections)**.** *Consider a standard sorting network $C = ((i_1, j_1), \dots, (i_k, j_k))$ on n channels. C can be reflected to $C^R = ((n - j_1 + 1, n - i_i + 1), \dots, (n - j_k + 1, n - i_k + 1))$, where C^R also is a standard sorting network.*

The above definitions were used in [56] to speed up the search for depth-optimal sorting networks. We decompose the search space by considering only some prefixes, and checking if these can be extended to a full sorting network. The next definition was introduced in [71] in the context of minimum-sized sorting networks, but can be adapted to this case.

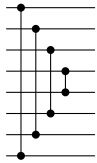


Figure 4.11. First layer as used by Bundala&Závodný ("BZ-style").

Definition 4.21 (Complete set of filters)**.** Let $\mathcal{F}^k(n,d)$ denote a set of k-layer-prefixes of sorting networks on n channels. $\mathcal{F}^k(n,d)$ is a complete set of filters if, if there exists a sorting network for n channels and d layers, there also exists one starting with a prefix from $\mathcal{F}^k(n,d)$.

If n and d are clear from the context, we may just write $\mathcal{F}^k$. Obviously, the set of all k-layer-prefixes on n channels is a complete filter. Finding a size-minimal set of filters is as hard as determining whether a sorting network on n channels and d layers exists: If this is not the case, the empty set is a complete set of filters. Otherwise, the prefix of any such sorting network can be chosen as complete set of filters. Therefore, one seeks to compute small, but not necessarily minimum sets of filters.

Corollary 4.22. *For every maximal layer L, $\{L\}$ is a complete set of filters on one layer.*

Parberry used this lemma, and considered only first layers of the form $L^1_P = (2i, 2i+1)$[172]. This massively prunes the search space, and significantly reduces the size of the SAT formulas we will present in the next section, as only outputs of this first layer have to be considered in subsequent search.

Bundala and Závodný extended this approach to the first two layers of sorting networks. They considered a set $\mathcal{F}^2_{BZ}(n)$ where all prefixes started with the first layer $L^1_{BZ} = ((0,n-1),(1,n-2),\ldots)$, cf. Figure 4.11. Furthermore, they shrank the set such that no dominated prefixes remained, and no prefixes which could be obtained from each other by permutations and reflections.

Formally, let $F_2(n)$ denote the set of all 2-layer prefixes on n channels. If, for two prefixes $A, B \in F_2(n)$, the outputs of A are a strict subset of the (possibly permuted) outputs of B, then B can be removed by Lemma 4.19.

$$\text{outputs}(A) \subsetneq \Pi(\text{outputs}(B)) \Rightarrow B \notin \mathcal{F}^2_{BZ}(n)$$

Furthermore, if A can be obtained from B by permuting the channels and untangling, only one of these prefixes has to be considered, this is

$$\text{outputs}(A) = \Pi(\text{outputs}(B)) \Rightarrow (A \notin \mathcal{F}^2_{BZ}(n) \vee B \notin \mathcal{F}^2_{BZ}(n))$$

Lastly, reflections were removed.

$$A = B^R \Rightarrow (A \notin \mathcal{F}^2_{BZ}(n) \vee B \notin \mathcal{F}^2_{BZ}(n))$$

To generate this set, they enumerated all possible 2-layer-prefixes beginning with a fixed, maximal first layer, and reduced this set by pairwise comparing the outputs. This approach is computationally expensive, and was used only for generating prefixes on 13 channels. In [74], Codish et al. presented an efficient algorithm to generate 2-layer-prefixes of depth-optimal sorting networks. This algorithm can easily generate the set $\mathcal{F}^2_{BZ}$ for 20 and more channels.

4.2.4 Suffixes of Sorting Networks

Another symmetry break was introduced by Codish et al. in [73]. The most important observation here is that it is sufficient to restrict the last two layers of sorting networks to short comparators, as long comparators in these layers would be redundant, i. e., they would never swap their inputs. Furthermore, it is sufficient to consider only few different structures here.

4.3 SAT-based Search for Improved Bounds

Although already Parberry used a SAT solver in [172], most SAT encodings currently used for the modeling of sorting networks are extensions of the encoding presented by Morgenstern and Schneider [160]. Here, we present the encoding from [70], which fixed some flaws in the presentation by Bundala and Závodný. Afterwards, we discuss an improved version of this encoding, which we firstly presented in [94].

We aim to encode the proposition "There exists a sorting network C_n^d on n channels, which uses at most d layers". A sorting network is a comparator network that sorts all its inputs. Therefore, we reformulate the proposition to "There exists a comparator network C_n^d on n channels, which uses at most d layers, and sorts all of its inputs".

In this encoding, we use boolean variables $g_{i,j}^k$ to denote that the channels i and j are connected by a comparator in layer k. By definition, each channel must be used at most once in each layer, which can be encoded as

$$\begin{aligned}
\text{once}_i^k(C_n^d) &= \bigwedge_{1 \leqslant i \neq j \neq \ell \leqslant n} \left(\neg g^k_{\min(i,j),\max(i,j)} \vee \neg g^k_{\min(i,\ell),\max(i,\ell)} \right) \\
\text{valid}(C_n^d) &= \bigwedge_{1 \leqslant k \leqslant d, 1 \leqslant i \leqslant n} \text{once}_i^k(C_n^d) \\
\text{used}_i^k(C_n^d) &= \bigvee_{j<i} g_{j,i}^k \vee \bigvee_{i<j} g_{i,j}^k
\end{aligned}$$

Therefore, a comparator network is valid if "once" holds for every channel and every layer. A channel i is used in layer k if one comparator (i,j) or (j,i) is connected to it. To encode that a comparator network is actually a sorting network, we create a formula

$$\phi_n^d = \text{valid}(C_n^d) \wedge \bigwedge_{x \in \mathbb{B}^n} \text{sorts}(C_n^d, x)$$

which describes that C_n^d sorts all its inputs. To encode that a single input x is sorted, we introduce vectors $v^0, \ldots, v^k \in \mathbb{B}^n$. The vector v^0 encodes the input, and v^i describes the output of i-th layer. Furthermore, let y denote a sorted permutation of x. Then, we connect v^0 with the input, v^d with the sorted output, and the other values via an update constraint.

$$\begin{aligned}
\text{sorts}(C_n^d, x) = &\bigwedge_{1 \leqslant i \leqslant n} v_i^0 \leftrightarrow x_i \\
&\wedge \bigwedge_{1 \leqslant k \leqslant d, 1 \leqslant i \leqslant n} \text{update}_i^k(C_n^d, v^{k-1}, v_i^k) \\
&\wedge \bigwedge_{1 \leqslant i \leqslant n} (v_i^d \leftrightarrow y_i)
\end{aligned}$$

This update constraint describes how the comparators update the values within the comparator network. If a channel is not connected to a comparator,

it just transports its value from left to right. Otherwise, the value is updated.

$$\begin{aligned}\text{update}_i^k(C_n^d, v, w) = &\left(\neg\,\text{used}_i^k(C_n^d) \rightarrow (w \leftrightarrow v_i)\right) \wedge \\ &\bigwedge_{1 \leqslant j < i} \left(g_{j,i}^k \rightarrow \left(w \leftrightarrow (v_j \vee v_i)\right)\right) \wedge \\ &\bigwedge_{i < j \leqslant n} \left(g_{i,j}^k \rightarrow \left(w \leftrightarrow (v_j \wedge v_i)\right)\right)\end{aligned}$$

In many cases, we will fix a prefix P of a comparator network. Then, only the outputs of this prefix have to be sorted by the suffix of the network. Therefore, it suffices to consider only outputs of P rather than all possible inputs in the SAT encoding. Then, these constraints can be reduced to

$$\phi_n^d(P) = \text{valid}(C_n^{d-|P|}) \wedge \bigwedge_{x \in \text{outputs}(P)} \text{sorts}(C_n^{d-|P|}, x)$$

Furthermore, another optimisation was suggested by Donald Knuth, and used in [56]. Consider an input of the form $0x$ where the input of the first channel is set to 0. This is, no comparator will change the value on the first channel, and this can be hard-coded in the SAT formula. This equivalently holds for inputs of the form $0^l x 1^t$.

Corollary 4.23. *Consider an input $0^l x 1^t$. The output of each possible prefix of a standard comparator network working on this input has the form $0^l x' 1^t$.*

We will refer to the prefix 0^l as leading zeros, and the suffix 1^t as tailing ones. The number of unsorted elements in the infix of the input, x, is also called windows size.

Alternatively, such situations can be detected by Failed Literal Branching [151].

This encoding was used in [56] to prove a lower bound of 9 layers for sorting networks on $n \geqslant 13$ channels. As sorting networks for 13 to 16 channels were known which actually used 9 layers, this settled their optimality.

Provided longer prefixes, it was also strong enough to compute faster sorting networks for 17 and 20 channels [93]. We will discuss these new upper bounds in the next section.

Improved SAT Encoding

In order to prove the new lower bound for sorting networks on more than 16 channels, we improved this encoding. The following example from [70] will show the basic idea.

Example 4.24. Consider a sorting network on $n = 6$ channels, an input sequence $x^0 = (0,1,0,1,0,1)$, and the output of the first layer, denoted by $x^1 = (x_1^1, x_2^1, x_3^1, x_4^1, x_5^1, x_6^1)$. Figure 4.12(a) illustrates this setting, where a "?" on an input value indicates that we do not know whether a comparator will be placed somewhere on the corresponding channel. This situation can be seen as a node in the search tree of a SAT solver, where no variable determining whether one of the comparators is used has been assigned yet. By Corollary 4.23, $x_1^1 = 0$ and $x_6^1 = 1$, so $x^1 = (0, x_2^1, x_3^1, x_4^1, x_5^1, 1)$, as indicated in the second layer of Figure 4.12(a).

Now consider the value of x_4^1. Clearly, the only first level comparator that will change the value on the fourth channel is $(4,5)$, any other comparator will leave this value unchanged.

Therefore, adding any other comparator on channel 5 determines that $x_4^1 = 1$ and one could specifically add propagation clauses of the form $g_{i,5}^1 \to x_4^1$ for $1 \leqslant i < 4$. Figure 4.12(b) illustrates the situation where comparator $(3,5)$ is placed in layer 1. Channels 3 and 5 are now in use, hence the "?" is removed from the corresponding input values. The values of x_3^1 and x_5^1 are determined by the comparator. Moreover, as argued above, the value of x_4^1 is set to 1. Figure 4.12(c) illustrates the situation if a second comparator, $(1,4)$, is added to layer 1. The value $x_2^1 = 1$ is determined by an argument similar to the one that determined $x_4^1 = 1$. □

In order to encode the reasons for values on channels to remain unchanged, for every layer k and every pair of channels (i, j) we introduce propositional variables $\text{oneDown}_{i,j}^k$ and $\text{oneUp}_{i,j}^k$, which indicate whether there is a comparator $g_{\ell,j}^k$ for some $i \leqslant \ell < j$ or $g_{i,\ell}^k$ for some $i < \ell \leqslant j$, respectively.

$$\text{oneDown}_{i,j}^k \leftrightarrow \bigvee_{i<\ell\leqslant j} g_{i,\ell}^k \qquad \text{noneDown}_{i,j}^k \leftrightarrow \neg\,\text{oneDown}_{i,j}^k$$

$$\text{oneUp}_{i,j}^k \leftrightarrow \bigvee_{i\leqslant\ell<j} g_{\ell,j}^k \qquad \text{noneUp}_{i,j}^k \leftrightarrow \neg\,\text{oneUp}_{i,j}^k$$

To make use of these new propositional variables, given an input

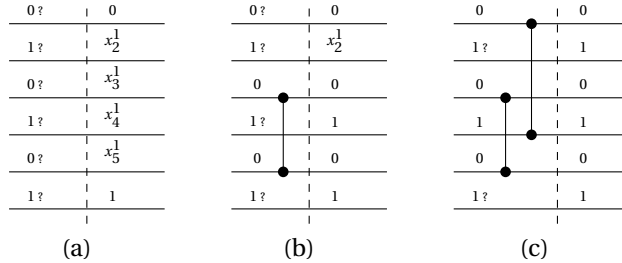


Figure 4.12. Propagations for the first layer of a sorting network on 6 channels determined from the input sequence 010101 (see Example 4.24).

$\vec{x} = (0,0,\ldots,0,x_t,x_{t+1},\ldots,x_{t+r-1},1,1,\ldots,1)$, for all $t \leqslant i \leqslant t+r-1$ and at each layer k, we add the following constraints to the definition of sorts.

$$\bigwedge_{1\leqslant k\leqslant d} v_i^{k-1} \wedge \text{noneDown}_{i,t+r-1}^k \rightarrow v_i^k$$
$$\bigwedge_{1\leqslant k\leqslant d} \neg v_i^{k-1} \wedge \text{noneUp}_{t,i}^k \rightarrow \neg v_i^k$$

These new definitions hence generalise the reason why the value on one channel remains unchanged. Furthermore, they allow to remove some clauses from the original encoding.

Example 4.25. Consider one of the new constraints. It says that, if the value on some channel i equals 1, and there is no comparator in this pointing downwards within the respective window of this input, then the value on this channel will be unchanged by this layer.

$$(v_i^{k-1} \wedge \text{noneDown}_{i,t+r-1}^k) \rightarrow v_i^k$$
$$(\neg v_i^{k-1} \vee \neg \text{noneDown}_{i,t+r-1}^k \vee v_i^k) \qquad \text{(as clause)}$$

If the original encoding contains a constraint

$$(v_i^{k-1} \wedge g_{j,i}^k) \rightarrow v_i^k$$
$$(\neg v_i^{k-1} \vee \neg g_{j,i}^k \vee v_i^k) \qquad \text{(as clause)}$$

this clause becomes redundant in the new encoding, and can be removed. This

can be seen by the following resolution steps.

$$\frac{(\neg g^k_{j,i} \vee \neg \text{oneDown}^k_{i,t+r-1}) \qquad (\text{oneDown}^k_{i,t+r-1} \vee \text{noneDown}^k_{i,t+r-1})}{(\neg g^k_{j,i} \vee \text{noneDown}^k_{i,t+r-1})}$$

Another resolution step yields

$$\frac{(\neg v^{k-1}_i \vee \neg \text{noneDown}^k_{i,t+r-1} \vee v^k_i) \qquad (\neg g^k_{j,i} \vee \text{noneDown}^k_{i,t+r-1})}{(\neg v^{k-1}_i \vee \neg g^k_{j,i} \vee v^k_i)}$$

Thus, the clause from the original encoding became redundant, and may be removed. This holds symmetrically for the predicates “noneUp". □

These new predicates come with some advantages. Firstly, they allow for more propagations, as depicted in Example 4.24, which increases the consistency of nodes in the search tree. Secondly, the size of the number of clauses in the resulting SAT formulas can be reduced significantly, as we will detail in the next section in Table 4.6. Thirdly, these predicates generalise on the reason why the value on one channel remains unchanged. As discussed in Section 2.2.5, such generalisations allow for smaller proofs. As we will discuss when summarising the impact of all techniques, this effect can be seen when analysing the statistics of solver runs on these formulas.

Next, we will consider another optimisation technique to reduce the size of the SAT encoding, and then discuss the impact of both techniques.

4.3.1 Prefix-Optimisation

As discussed in Section 4.2, the channels of standard sorting networks may be permuted, and untangling the result yields another standard sorting network. In this section, we will see that different permutations of a prefix are not equally handy for SAT solvers, a fact that we can use to compute permutations that yield small SAT formulas, and thus help reducing the running time of the SAT solver. Let us first consider an example.

Example 4.26. Consider the three prefixes on 6 channels and 2 layers shown in Figure 4.13. All of them are equivalent in the sense that they are untangled permutations of each other. The leftmost prefix, P_1, starts with the first layer L^1_P used by Parberry. Next to it, the set of non-sorted outputs of P_1 is given.

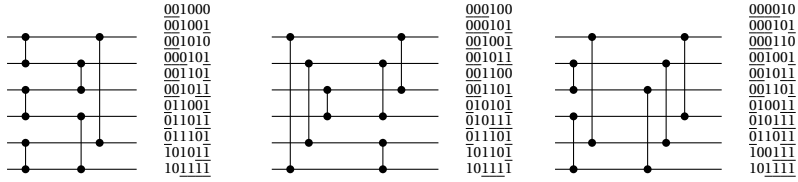


Figure 4.13. Three permutations of the same 2-layer-prefix on 6 channels. C_1 (left) has L_P^1 as first layer, C_2 (middle) has L_{BZ}^1 as first layer, and C_3 (right) is optimised to minimise total window size in the outputs from the prefix. Each prefix is accompanied by the set of its non-sorted outputs.

Encoding all 11 outputs with all their channels would create 66 channels to consider in the SAT encoding. Leading zeros, and tailing ones are underlined — these do not appear in the SAT encoding. Thus, we may hard-wire 31 channels, and keep only 35 channels in the SAT encoding on which the respective values are variable.

The prefix in the middle, P_2, starts with a first layer L_{BZ}^1 as used by Bundala and Závodný [56]. Here, one more channel can be hard-wired, and only 34 channels have to be encoded.

The last prefix was permuted such that the number of leading zeros and tailing ones was maximised. In this case, only 28 channels have to be considered.

We will first consider the impact of this observation on the outputs of sets of prefixes, and the resulting SAT formula. Afterwards, we show a simple, yet effective approach to optimise prefixes.

Table 4.4 shows the number of channels to consider when using L_P^1 and L_{BZ}^1 as prefix, respectively. For all values of n, the first layer L_{BZ}^1 requires less channels than L_P^1. The intuition here is that L_{BZ}^1 contains more long comparators, which have a stronger impact on the windows size. For example, every input $1x0$ will be turned into $0x'1$ by L_{BZ}^1. Alternatively, the comparators L_P^1 connect channels of the form $(2i, 2i+1)$, thus the windows size of an input can be reduced at most by 2 by this prefix.

When considering prefixes on 2 layers, we consider the set of filters beginning with L_P^1 and L_{BZ}^1, together with permutations of those which were created to minimise the number of channels to consider, cf. Table 4.5. Also here, the

Table 4.4. Number of channels to consider in the encoding after the first layer.

n	5	6	7	8	9	10	11	12	13	14	15	16	17
L_P^1	44	84	233	408	1,016	1,704	4,013	6,564	14,948	24,060	53,585	85,296	186,992
L_{BZ}^1	36	72	196	358	876	1,524	3,532	5,962	13,380	22,128	48,628	79,246	171,612

Table 4.5. Average number of channels (over the complete set of filters) to consider in the encoding after the second layer.

n	5	6	7	8	9	10	11	12	13	14	15	16	17
$\mathcal{F}_P$	20	43	110	196	456	786	1,651	2,715	5,534	9,094	17,808	28,581	55,314
$\mathcal{F}_{BZ}$	18	40	97	178	402	714	1,480	2,483	5,014	8,406	16,332	26,633	51,221
$\mathcal{F}_{opt}$	17	35	89	156	362	619	1,328	2,168	4,503	7,371	14,711	23,496	46,331

prefixes beginning with L_{BZ}^1 are superior to those beginning with L_P^1, and both yield larger encodings than the optimised prefixes. The numbers given here refer to the prefixes generated by the algorithm of Codish et al. [74] and are just examples for sets of prefixes beginning with L_P^1 and L_{BZ}^1. For example, applying the permutation $(0,2)(1,3)$ on prefixes beginning with L_P^1 yields another prefix with the same first layer, but possibly a different second layer, and thus different properties.

The impact of the combination of the new SAT encoding and different prefix styles can be seen in Table 4.6. Here, we compare formula sizes which encode a sorting network on 17 channels and 9 layers, with 2000 distinct inputs.

Table 4.6. Sizes of the SAT formula depending on choice of prefix and encoding.

Prefix type	Encoding	# variables	#clauses
$\mathcal{F}_P$	old	115,815	4,861,186
$\mathcal{F}_{BZ}$	old	104,769	4,260,513
$\mathcal{F}_{opt}$	old	89,057	3,438,352
$\mathcal{F}_P$	improved	117,446	2,803,674
$\mathcal{F}_{BZ}$	improved	106,393	2,270,755
$\mathcal{F}_{opt}$	improved	90,513	1,598,509

Choosing an optimised prefix reduces the number of both clauses and variables significantly. The improved encoding adds a few variables, but decreases the number of clauses. In the combination of new encoding and optimised prefix, the number of clauses is reduced by a factor of 3. Furthermore, the outputs of the permuted prefixes come with smaller window sizes. As comparators which are longer than the maximum window size of all outputs will never swap the values on the channels they are connected to, they could in theory be removed from the encoding, yielding a equisatisfiable formula. We experimentally found that SAT solvers detect this fact, the VSIDS activities of variables denoting the use of a long comparator are extremely low.

We will now turn to the techniques used to compute the improved prefixes. In the subsequent sections, the computation of new upper and lower bounds will be discussed, together with the impact of these improvements in the respective settings.

In order to compute a permutation which minimises the sum of windows sizes of the outputs of a prefix, one might try all possible permutations, and inspect the outputs of the respective permutation of the prefix. For n channels, the running time of this naïve approach is $\Omega(n!2^n)$, which is intractable even for small values of n.

Given a prefix P on n channels and a permutation, one may evaluate the sum of window sizes of its outputs by first permuting and untangling the prefix. Then, iterate over all 2^n input sequences, to compute the set of outputs of the permuted prefix, and count the windows sizes. This approach was chosen in the first implementation [94]. However, if several permutations of the same prefix have to be evaluated, it is smarter to compute the set of outputs once. When untangling the prefix with respect to some permutation, the untangle algorithm (Algorithm 15) can be used to compute the permutation which the prefix permutation induces on the set of outputs. A linear-time implementation of this algorithm is given in the Appendix A.2. As the set of outputs of a prefix is significantly smaller than the set of inputs, this improves the running time.

Let us now consider the search for a decent, i. e., not necessarily optimal, permutation to apply to a given prefix. In [94], we chose a simple evolutionary algorithm to search for good permutations. An initial population is created by choosing some permutations uniformly randomly by a Fisher-Yates-Shuffle [103, 136]. Afterwards, in some iterations offspring are created by randomly swapping two channels, cf. Algorithm 16. The fitness function is

Algorithm 16: Evolutionary Algorithm for the Permutation of Prefixes

Data: Prefix P on n channels.
Result: Permuted version of P

```
1  Pop={P};
   /* Create an initial population                                  */
2  for i=0 to 32 do
3      Choose permutation Π : [n−1] ↦ [n−1] ;
4      Create P′ by permuting P with Π and untangling;
5      Pop := Pop ∪ {P′};
6  nextGen := Pop ;
7  for i=0 to 32 do
8      for p ∈ Pop do
9          Choose permutation Π : [n−1] ↦ [n−1] ;
10         Create P′ by permuting p with Π and untangling;
11         nextGen := nextGen ∪ {p, P′};
       /* Choose prefixes with minimum sum of window sizes          */
12     Pop := best 32 prefixes from nextGen;
13 return Best prefix from Pop;
```

computed as the sum of windows sizes of the inputs. After each iteration, half of the population is removed.

We experimentally found that a population size of 32 and 32 iterations yielded good results.

However, even better results can be obtained by a gradient descent approach, cf. Algorithm 17. Again, we start with a uniformly randomly permuted version of P, and compute the pair of channels which, if swapped, yield the steepest descent in terms of the sum of windows sizes. Here, a function “eval" is used to count the window sizes of the outputs of the permuted prefix. This is iterated until a local minimum has been found.

Experimentally, we found that this approach requires very few attempts to find a good solution. The source code of the implementation is given in Appendix A.2. The running times to improve the complete set of filters is shown in Table 4.7.

Compared to the running time of the SAT-solver we used, these values are

Algorithm 17: Gradient Descent Algorithm

```
   Data: Prefix P on n channels.
   Result: Permuted version of P
 1 Choose permutation Π : [n − 1] ↦→ [n − 1];
 2 P := Π(P);
 3 Π := id;
 4 bestVal := eval(P, id);
 5 repeat
 6 |   Π := id;
 7 |   for 0 ⩽ i < j < n do
 8 |   |   obj := eval(P, (i, j));
 9 |   |   if obj < bestVal then
10 |   |   |   bestVal := obj;
11 |   |   |   Π := (i, j);
12 |   P := Π(P);
13 until Π = id;
14 return P;
```

Table 4.7. Running times for improving the sets of prefixes.

n	7	8	9	10	11	12	13	14	15	16	17
time(s)	0.01	0.02	0.07	0.12	0.68	1.45	10.7	15.8	109	166	1,066

neglectable. When proving the lower bound for $n = 17$ inputs, the overall CPU time used was $27 \cdot 10^6$ seconds, compared to 1,066 seconds for finding good permutations of the prefixes.

4.3.2 Preprocessing

There exists a plethora of preprocessing techniques for SAT solving [88, 121–123, 152, 154], many of which did not prove helpful in this case. Interestingly, we found that Failed Literal Branching [151] was helpful, as many literals “failed" and therefore could be fixed at the root of the search tree. We implemented Failed Literal Branching in our extension of MiniSAT [89][1]. The specific impact

[1] The source code is available at `https://github.com/the-kiel/JCSS`.

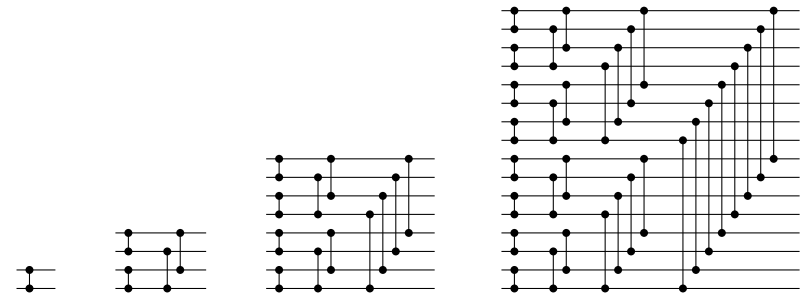


Figure 4.14. Generating partially ordered sets for $n \in \{2, 4, 8, 16\}$ inputs.

will be discussed in the next sections.

4.4 Improved Upper Bounds

Several ways have been considered to find new sorting networks of minimum size and depth. Genetic algorithms appear to be stronger when searching sorting networks with a small number of comparators [211]. The algorithmic constructions like the odd-even Merge Sort discussed in the introduction run in polynomial time, but yield suboptimal networks. Al-Baddar et al. used a tool, SortNet, which supports the process of hand-crafting sorting networks: Given some prefix, SortNet computes properties of the set of outputs [21, 23]. With these insights, new sorting networks for 18 and 22 channels were found.

Morgenstern et al. used a SAT-based approach to construct sorting networks. However, they were only able to reproduce sorting networks of optimum depth for up to 10 channels. In this section, we consider a combination of these techniques. We create a prefix for a sorting network, and use a SAT-solver to try and extend it to a full sorting network.

A good starting point for good prefixes are so-called Green Filters [139], as shown in Figure 4.14. Their construction is fairly simple. For $n = 2$ inputs, connect the two channels with a comparator. For $n = 2^k$ for some $k > 1$, construct Green Filters on the upper and lower $\frac{n}{2}$ channels, and connect these filters in the next layer with comparators of length $\frac{n}{2}$. For 2 channels, the Green Filter

Table 4.8. Number of Outputs of Green Filters.

n	2	4	8	16
#outputs	3	6	20	168

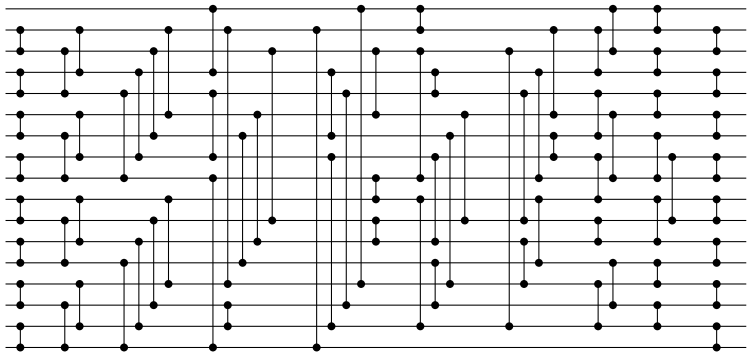


Figure 4.15. Sorting network for 17 channels of depth 10, starting with a Green Filter on the first 3 layers.

is a sorting network itself. On larger numbers of channels, Green Filters are similar to the prefixes of Batcher's odd-even Merge Sort, cf. Figure 4.8. The output of such a filter is a partially ordered set [23]. If the input vector contains at least one 1, then the output channel with highest index will be set to 1, and analogously for 0s. This can be seen by an inductive argument: The Green Filter on 2 channels is a sorting network, and for $n > 2$ the highest and lowest channels are connected in the last layer.

Furthermore, Green Filters significantly reduce the number of distinct output vectors, as shown in Table 4.8. We conjecture that there are no better prefixes in terms of number of outputs.

Conjecture 4.27. *Let $n = 2^k$. There is no prefix on k layers which yields a lower number of distinct output vectors than a Green Filter.*

A Green Filter on 2^k channels and $k' < k$ layers consists of the first k' layers of a Green Filter. We used a Green Filter on 3 layers for our first result in the field of sorting networks, a new upper bound on the number of layers required for sorting 17 inputs. The best known sorting network before had 11 layers, and

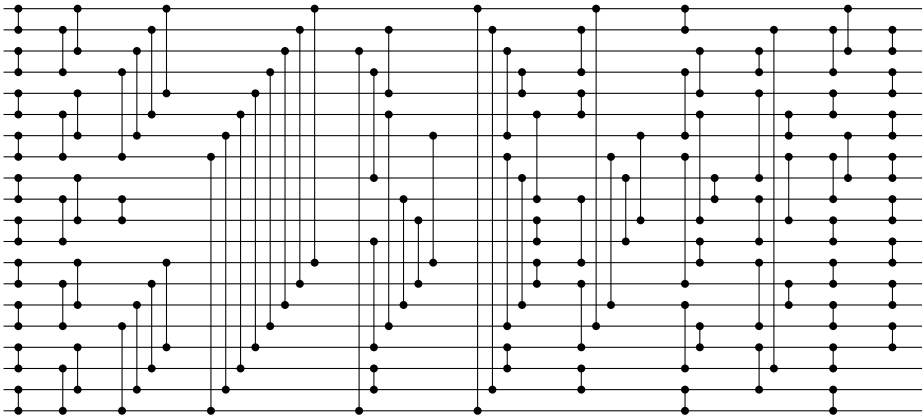


Figure 4.16. A sorting network for 20 channels of depth 11. The first 4 layers consist of a Green Filter on 4 layers, and a complete sorting network on the 4 innermost channels.

could be generated from the network on 18 channels and 11 layers presented in [23] by removing one channel.

We used a Green Filter on 3 channels as prefix, leaving one channel untouched. This prefix had 800 distinct output vectors, which is manageable for state of the art SAT solvers [93] and the encoding described above. The resulting network is shown in Figure 4.15. It was found without any of the symmetry breaks by Codish et al., optimised prefixes or the new SAT encoding.

Similarly, a new sorting network for 20 channels was found. In this case, we used a Green Filter on 16 channels, and a complete sorting network on the 4 remaining channels. The Green Filter has 168 distinct outputs, and the 4-channel sorting network 5 outputs. Therefore, 840 distinct vectors have to be sorted by the suffix of the sorting network. The result, shown in Figure 4.16, uses 11 layers, thus improving the old upper bound by one layer. By monotonicity, this also gives an improved upper bound for 19 inputs.

These results gave improved bounds, however, the running times were quite high, as none of the above mentioned improvements was used. Table 4.9 shows the running times when using different combinations of improved techniques [70]. In the general picture, all techniques improve the running times,

Table 4.9. Impact of the different optimisations in the time required to find the new sorting networks on 17 and 20 channels [70].

Symmetry Break (last layers)	Improved Encoding	Opt. Prefix	Time/s $n = 17$	Time/s $n = 20$
yes	yes	yes	17	46
yes	yes	no	78	560
yes	no	yes	37	13,199
yes	no	no	265	1,255
no	yes	yes	64	97
no	yes	no	838	412
no	no	yes	3,723	36,159
no	no	no	9,995	45,596

and their combination yields a speed up of 558 and 991 when computing the sorting network on 17 and 20 channels, respectively. Yet, the running times in these cases are unstable as SAT solvers might be lucky to get the “right" random seed, and find a solution quickly even on a bad encoding. Consider for example the second and sixth row in Table 4.9. For $n = 20$ channels, the SAT solver was slower when using the symmetry break on the last layers.

We submitted some SAT formulas on sorting networks to the SAT Competition 2016 [95][2]. In these, we encoded the search for different sorting networks, using different combinations of the optimisation described in the previous section[3]. Not all submitted formulas were used, however, in some cases formulas were used which described the same setting up to one of the above techniques. This allows to see the impact on 28 different SAT solvers. A summary of all results is given in the Appendix A.3.

Table 4.10 shows the impact of the different optimisations on satisfiable formulas which encode the existence of a sorting network on 16 channels and 9 layers. Here, the first column indicates whether the symmetry break by Codish, Cruz-Filipe and Schneider-Kamp was used. The next columns indicate an

[2] cf. `http://baldur.iti.kit.edu/sat-competition-2016/`

[3] The formulas are publicly available at `https://github.com/the-kiel/SAT_benchmarks`.

Table 4.10. Results on satisfiable instances from the SAT Competition 2016. In each formula, a sorting network on 16 channels and 9 layers was sought.

CCS	prefix	enc	pre	solved	time (all)	time (solved)
no	no	no	yes	1	4971	4218
no	no	yes	yes	7	4494	2910
no	yes	no	yes	28	1051	911
no	yes	yes	yes	28	767	615
yes	no	no	yes	4	4705	2878
yes	yes	no	yes	28	458	296

optimised prefix, the improved propositional encoding, and if a preprocessor was used on the formula beforehand, which was the case in all formulas used in the competition. Here, the optimised prefix has the largest impact. When used, 28 out of 29 participating solvers were able find a sorting network in all combinations with other techniques — the 29th solver timed out on all benchmarks. All other techniques still have some impact. Using the symmetry break on the suffixes increases the number of terminating solvers from 1 to 4, the improved encoding yields 7 successful runs.

However, the success in finding new sorting networks still depends on the choice of a good prefix.

Our last result on upper bounds is a sorting network on 24 channels, and 12 layers, which improves upon the best previously known sorting network by one layer [90]. Alternatively, the number of inputs sortable by 12 layers is increased by two. When considering only binary input sequences, this network must sort all 2^{24} inputs. If the first layer is fixed to L^1_P, which is a Green Filter on one layer, the number of inputs to consider is reduced to $3^{12} = 531{,}441$, which is not tractable for the SAT based approach. Similarly, a Green Filter on two layers would result in $6^6 = 46{,}556$ inputs, and $20^3 = 8{,}000$ for three layers, cf. Table 4.8. Both is intractable for current encodings and SAT solvers.

We therefore designed a prefix on 12 channels and 5 layers which yields a small number of distinct outputs, and created a prefix for 24 channels by using two copies of the smaller prefix.

In [71], Codish et al. computed the set of all prefixes of sorting networks up to permutation, subsumption and reflections. Their algorithm can be adapted

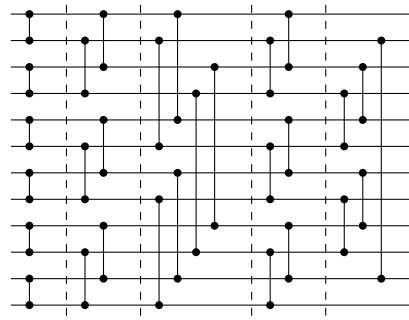


Figure 4.17. Prefix of a sorting network on 12 channels, and 5 layers.

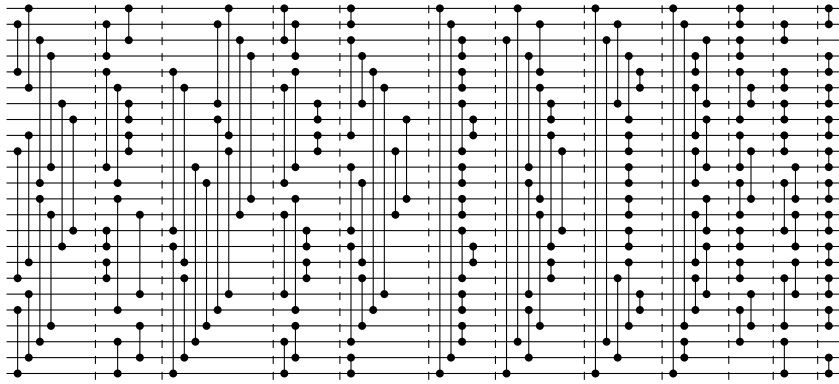


Figure 4.18. New Sorting Network on 24 channels. The prefix was permuted to gain an easier SAT formula.

to also work with layered prefixes. As this does not scale to 12 channels, we used a greedy approach as follows. The first layer can be chosen as L_p^1, as any maximal layer is fine here. Next, we generated the set of all 2-layer-prefixes beginning with L_p^1 up to permutations, subsumptions and reflections. Out of this set, we chose the 32 prefixes with minimum number of distinct outputs, ties were broken randomly. Iterating this process gave the prefix on 5 layers shown in Figure 4.17. This prefix has 34 different output vectors, duplicating it yields a prefix for 24 channels with $34^2 = 1{,}156$ outputs. As the prefix comes

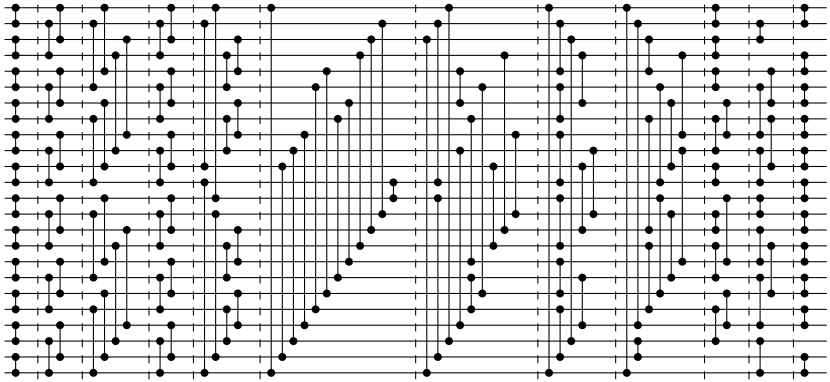


Figure 4.19. Permutation of the sorting network from Figure 4.18. Here, the optimisation of the prefix was undone, so the structure of the prefix becomes visible again.

with two unused channels in its last layer, we connected these channels when creating the final prefix. In order to reduce the running time of the SAT solver, we permuted the prefix to reduce the size of the SAT formula to solve. The resulting sorting network is shown in Figure 4.18.

The prefix optimisation step, i. e., permuting and untangling the channels, makes it hard to understand the structure of this sorting network.

Thus, Figure 4.19 shows the sorting network gained by reconstructing the original prefix, and untangling the suffix accordingly. This network contains several redundant comparators, some of which were added by the symmetry break on the last layers.

A nicer drawing is shown in Figure 4.20, where redundant comparators were removed. The resulting sorting network has 125 comparators. Thus, this sorting network improves the upper bound on the number of layers, but there exist sorting networks on 24 channels using only 123 comparators [211], so the new sorting network is sub-optimal in that sense.

In this section, we presented improved upper bounds for sorting networks on 17, 19, 20, 23 and 24 channels. Accordingly, these improved networks can be used as base cases for recursive constructions as used in the odd-even Merge Sort and yield better networks for larger numbers of inputs. We next turn to our main result, a lower bound on the number of layers required to sort 17 inputs,

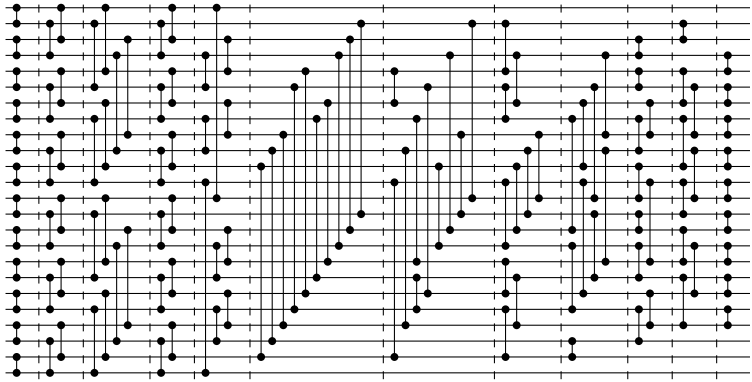


Figure 4.20. The Sorting Network from Figure 4.19 without redundant comparators. Thus, this network has only 125 comparators.

Table 4.11. Results on unsatisfiable instances from the SAT Competition 2016. In each formula, a sorting network on 13 channels and 8 layers was sought.

CCS	prefix	enc	pre	solved	time (all)	time (solved)
no	no	no	yes	28	1408	1408
yes	yes	no	no	28	71	71
yes	yes	no	yes	28	47	47
yes	yes	yes	yes	28	59	59
yes	no	yes	no	28	419	419

which settles the optimality of the sorting network on 17 channels presented in this section.

4.5 Improved Lower Bounds

For $n \leqslant 16$ channels, the gap between lower and upper bounds on the depth of sorting networks was closed by Parberry [172] and Bundala and Závodný [56]. By monotonicity, this also gave a lower bound of 9 layers for sorting networks on 17 channels. As shown in the last section, the upper bound for 17 channels is 10 layers, therefore, we sought to either prove the optimality of this sorting

network, or find a better one.

Parberry proved lower bounds for 9 and 10 channels by showing that L_P^1 cannot be extended to a sorting network on 6 channels, therefore, the networks given in [139] were optimal. As discussed in Section 4.2, this is sufficient as every sorting network for 9 channels and 6 layers could be transformed into one having L_P^1 as first layer.

Bundala and Závodný extended this approach to the first two layers. They generated complete sets of filters, sets of prefixes on 2 layers such that every sorting network could be transformed into one having a prefix from this set. Thus, it was sufficient to check whether one of these prefixes could be extended to a full sorting network.

Following this approach, we generated the complete set of filters for 17 channels using the algorithm by Codish et al. [74]. In a first attempt, we used the encoding shown in Section 4.3 with a few optimisations. Comparators in the last layer were restricted to length 1, and comparators in the second-to-last layer to length 3, referring to a preliminary version of [73]. Furthermore, we encoded inputs with only one unsorted 0 or 1 in the window of unsorted inputs as a reachability problem. This attempt was of limited success. We were able to prove that some of the prefixes could not be extended to a full sorting network, but for harder cases the SAT solver did not terminate within 50 days, so we aborted the experiment.

After developing and implementing the ideas of improving the SAT encoding, and permuting the prefixes, we ran the experiments again. The improvements clearly reduced the running times, we were able to prove that no prefix from $\mathcal{F}_{BZ}^2(17)$ can be extended to a sorting network on 9 layers. The overall running time was $27.63 \cdot 10^6$ seconds, with a maximum of 97,112 seconds. Thus, we may state the main result of this section.

Theorem 4.28. *There is no sorting network on* 17 *channels which uses less than* 10 *layers.*

In the remainder of this section we present a more detailed analysis of the impact of the different improvements, and a technique which helped us to gain a deeper understanding of proofs of the non-existence of sorting networks.

We implemented a refinement-based approach for generating sorting networks, similar to techniques used in software verification [69]. The loop, drawn in Figure 4.21, starts with a SAT formula ϕ which encodes the existence of a

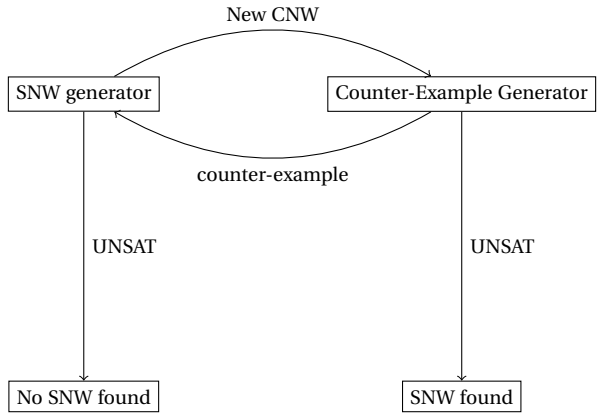


Figure 4.21. CEGAR loop for finding sorting networks

comparator network, without considering any constraints on the relationship between inputs and outputs. If such a network is found, it is handed to a second SAT solver which tries to generate a counter-example to the assumption that this was an actual sorting network, which is an input that cannot be sorted by this comparator network. If this is successful, the constraint that this particular input must be sorted is added to ϕ, and a new comparator network is sought. Otherwise, if no counter-example exists, the comparator network actually is a sorting network, and the loop terminates. On the contrary, if no more comparator network can be generated after adding some constraints, this implies that no sorting network exists. This loop was helpful to gain new insights, especially in finding good numbers of inputs to encode into a formula when seeking to prove the non-existence of sorting networks.

We will now turn to the impact of these ideas. Table 4.12 shows running times for the proof that no sorting network on 16 channels and 8 layers exists. For this proof, all 211 prefixes from the sets of filters $\mathcal{F}_P$, $\mathcal{F}_{BZ}$ and $\mathcal{F}_{opt}$ had to be checked. The overall time for this is decreased by a factor of 4 when using optimised prefixes rather than ones beginning with L_P^1, and the maximum time for a single prefix by a factor of 9.

However, it is complicated to draw a clear conclusion of the impact on single techniques. Firstly, they show a different impact on satisfiable cases,

Prefix	Overall time (s)	Maximum time (s)
$\mathcal{F}_P$	22,241	326
$\mathcal{F}_{BZ}$	10,927	150
$\mathcal{F}_{opt}$	5,492	36

Table 4.12. Impact of permuting the prefix when proving that no sorting network for 16 channels with at most 8 layers exists.

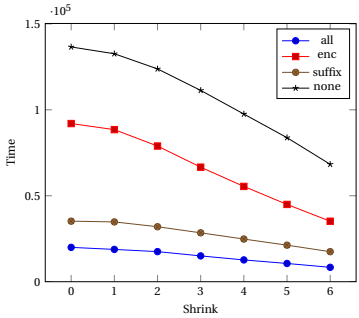


Figure 4.22. Results for optimised prefixes on 16 channels.

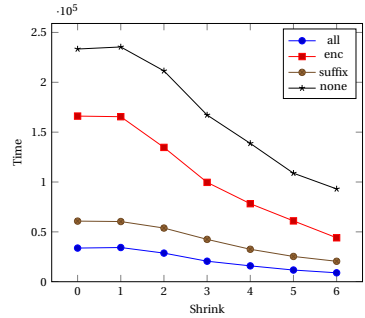


Figure 4.23. Results for BZ-prefixes on 16 channels.

i. e., cases in which a sorting network is sought and found, and unsatisfiable cases which show the non-existence of sorting networks of a particular depth. Secondly, they depend on each other. For example, using the constraints on the suffix from [73] allows for fixing more variables by Failed Literal Branching. Figures 4.22 and 4.23 show the running times for proving the non-existence of a sorting network on 16 channels and 8 layers using optimised, and BZ-style prefixes. The x-axis shows how much inputs are shrunk, i. e., a shrink of k denotes that only inputs of window sizes at most $16-k$ were used in the encoding.

The tests were run using both the improved encoding and the symmetry breaks on the last layers, only one of them, or none of them. If all inputs are used, the formulas become quite large, which increases the running time significantly. Here, the optimised prefixes are clearly beneficial compared to the prefixes used by Bundala and Závodný. When decreasing the window size

Table 4.13. The impact of the new SAT encoding: The new variables allow for shorter proofs, as can be seen by the number of conflicts. Furthermore, the solving times are reduced significantly.

prefix	encoding	conflicts	time
0	standard	19127761	70700
0	improved	9832380	24476
51	standard	29268392	118128
51	improved	17811760	50699

Table 4.14. The impact of the new SAT encoding when prefixes in BZ-style are used.

prefix	encoding	conflicts	time
0	standard	22179733	76698
0	improved	10686754	23818
51	standard	> 60018753	> 428666
51	improved	> 59001902	> 428324

if the inputs used, this gap becomes significantly smaller. Furthermore, the symmetry break on the last layers by Codish et al. has a significantly stronger impact here than the improved encoding.

We ran similar experiments for the case of 17 channels. Table 4.13 shows the running times and number of conflicts when proving that two prefixes from $\mathcal{F}^2_{BZ}(17)$ cannot be extended to a full sorting network on 9 layers. The prefix with index 0 was one of the easiest in our experiments, whereas the one with index 51 was one of the hardest cases.

Here, the optimised version of the prefixes were used. In both cases, using the improved encoding reduces both the number of conflicts and the running time. The reduced number of conflicts can be seen as a validation of the idea of generalising the reasons why the value on a channel remained unchanged, the SAT solver can benefit from this. The reduced running time is both due to the reduced number of conflicts, and the smaller encoding. Next, we re-ran these experiments using prefixes in BZ-style, the results are given in Table 4.14. On the first prefix, the solver achieves similar results as when using the optimised version of it, the running time is less than 10% higher here when using the

standard encoding, and even slightly lower with the improved encoding. In the second case, the result is different; We aborted this experiment after several days. Even using the improved encoding, the solver was not able to prove unsatisfiability in this time. Thus, using the optimised version of the prefix gave a speed-up of at least 8 in this case.

4.6 Conclusion and Open Questions

In this chapter, we discussed the concept of sorting networks, and some of their properties. The main subject here was one property, the number of parallel sorting steps required to sort an input. We discussed a SAT encoding which describes sorting networks of bounded depth, and showed how this can be improved. Furthermore, we used the notion of symmetries of sorting networks in a new way, such that we could use an optimisation algorithm which computes permuted versions of prefixes of sorting networks which in turn significantly reduce the size of SAT encodings, and the running time required by a SAT solver on these formulas. This technique has an especially strong impact when searching for new sorting networks rather than seeking to prove their non-existence.

We presented new sorting networks, which improve the upper bounds on the number of layers required to sort 17, 19, 20, 23 and 24 inputs. These networks can be used as base case either in software implementations of sorting algorithms, or when constructing sorting networks recursively e. g., using an odd-even Merge Sort.

Furthermore, we were able to prove the non-existence of sorting networks for 17 inputs and 9 layers, which settles the optimality of our sorting network for 17 inputs, thereby answering the main open question from [73]. Donald E. Knuth sent us an email congratulating to this result, and shows the sorting network in [138].

By monotonicity, this result also increases the lower bound on the number of layers for up to 36 inputs.

Future work

Settling the optimality of a 10-layer sorting network for 17 inputs immediately gives rise to the question about optimal bounds for 18 inputs. We conjecture

that either finding a network on 10 channels or proving non-existence is within reach when using some rather evolutionary than revolutionary improvements.

- ▻ The symmetry break on the suffix of a sorting network from [73] could be extended to the last 3 layers. This would both remove further symmetries from the search space, and allow for fixing more variables in the SAT encoding. We furthermore conjecture that is is sufficient to consider last layers without "gaps", i. e., where unconnected channels only exist at the top and bottom of the last layer.

- ▻ When using prefixes on two layers, some symmetry breaks could be applied on the third layer of a sorting network. Consider the Green Filter on 12 channels and 2 layers in Figure 4.24, which consists of 3 segments, each of which is a Green Filter on 4 channels. Connecting the top-most channel to a channel from the second segment is symmetric to connecting it to the respective channel from the third segment, thus, one of these comparators could be forbidden.

- ▻ We only introduced predicates in the SAT encoding which describe why the value on a channel is not changed by a particular layer. It might be possible to analogously generalise on the reasons why values are changed, yielding stronger explanations, i. e., learned clauses.

- ▻ When trying to find improved upper bounds, a set of prefixes on three or four layers might be generated using a limited discrepancy search (LDS) [119]. Consider a greedy algorithm which, given a prefix on some layers L, generates a prefix on one more layer by iteratively adding comparators (i, j) to the prefix such that the number of outputs $|\text{outputs}(L(i, j))|$ is minimised. The core idea behind LDS is that if greedy algorithms fail, they often fail because of only a few bad decisions. Thus, considering solutions which are similar to the one generated by the greedy algorithm might yield good results. This can be seen as a search limited to some local neighbourhood of the solution given by a greedy algorithm. Before using a SAT solver to try and extend these prefixes to full sorting networks, symmetric solutions should be removed as presented in [71].

A drawback of the SAT encodings used so far is that they do not consider symmetries of prefixes, which we would consider a major achievement. Here,

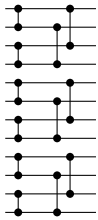


Figure 4.24. Green Filter on 12 channels and 2 layers.

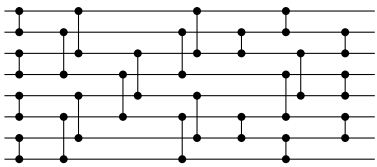


Figure 4.25. A Sorting Network on 8 channels with short comparators.

the term of symmetries relates to the relation between sets of outputs of different prefixes. It is not clear if these can be encoded into SAT efficiently.

When implementing sorting networks on FPGAs, long comparators make it hard to find a layout on the chip which achieves high clock rates [164]. Here, it would be beneficial to find sorting networks which are close to optimal, but use only short comparators, like the one shown in Figure 4.25. There has been some research on sorting networks in mesh topologies [144], and in other, rather artifical topologies [30]. It is clear that sorting networks for n channels with depth $\mathcal{O}(\log(n))$ require comparators of length $\Omega(\frac{n}{\log(n)})$, as they must be able to transport values from the top-most to the bottom-most channel, and vice versa. However, we are not aware of results which relate the maximum length of comparators to non-trivial bounds on size and depth of sorting networks.

Chapter 5

Further Publications

"What else?"

GEORGE CLOONEY

Besides the research presented in this thesis, I was involved in further activities. In this chapter, I will give a brief overview on them.

- In the field of software verification, we sought to find data races. These are unsynchronised memory accesses of parallely executed threads to the same memory location where at least one access is writing [166]. Our work focussed on software used in embedded systems, mostly in the automotive domain. We developed a tool, Gropius, which computed a coarse over-approximation of the set of possible data races in a program, which was presented in [99] and [98]. Being only a prototype, it was able to analyse real-world software of about 300,000 lines of code in a few seconds, and found bugs there. Gropius is now actively maintained and extended by Philipp Sieweck.

- A totally different field is the design and analysis of algorithms operating on sequences. I was involved in research regarding a version of the pattern matching problem. In k-abelian pattern matching, two words are considered equivalent if the subsequences or factors of length k occur in both words in the same multiplicity. Some algorithms for this problem, together with experimental results were presented in [91] and [92]. My contributition here was mostly the implementation of the developed algorithms and the experimental section.

- Graph Drawing is an interesting field of research with numerous practical applications. We considered CP-based and Integer Programming-based

approaches for computing optimal graph layouts with respect to different criteria. In CP-based approaches, we used the parallel version of CHUFFED presented in Chapter 3, which showed an impressive performance. Some results comparing a new heuristic with optimum results obtained by integer programming were presented in [189], an extended version was submitted to the Journal of Graph Algorithms and Applications and is currently under review.

Appendix A

Appendix

A.1 List of Benchmarks Used in the Paper “Communication in Massively-Parallel SAT Solving"

grieu-vmpc-31.cnf.gz
post-cbmc-zfcp-2.8-u2.cnf.gz
velev-npe-1.0-9dlx-b71.cnf.gz
velev-vliw-uns-4.0-9.cnf.gz
gss-19-s100.cnf.gz
gss-20-s100.cnf.gz
md5_47_4.cnf.gz
md5_48_1.cnf.gz
md5_48_3.cnf.gz
dated-10-13-u.cnf.gz
dated-5-13-u.cnf.gz
ACG-15-10p0.cnf.gz
UTI-20-10p0.cnf.gz
minxorminand128.cnf.gz
AProVE07-27.cnf.gz
zfcp-2.8-u2-nh.cnf.gz
maxor064.cnf.gz
gss-18-s100.cnf.gz
gus-md5-08.cnf.gz
ACG-20-5p0.cnf.gz
slp-synthesis-aes-top27.cnf.gz
aes_32_3_keyfind_2.cnf.gz
blocks-blocks-36-0.130-NOTKNOWN.cnf.gz
blocks-blocks-36-0.150-NOTKNOWN.cnf.gz
grid-strips-grid-y-3.045-NOTKNOWN.cnf.gz
grid-strips-grid-y-3.055-NOTKNOWN.cnf.gz
grid-strips-grid-y-3.065-SAT.cnf.gz
transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.020-NOTKNOWN.cnf.gz
transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.030-NOTKNOWN.cnf.gz
transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.040-NOTKNOWN.cnf.gz
transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.050-NOTKNOWN.cnf.gz
transport-transport-city-sequential-35nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.020-NOTKNOWN.cnf.gz
transport-transport-two-cities-sequential-15nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.020-NOTKNOWN.cnf.gz
transport-transport-two-cities-sequential-15nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.030-NOTKNOWN.cnf.gz
md5_47_1.cnf.gz
md5_47_2.cnf.gz
md5_47_3.cnf.gz
md5_48_2.cnf.gz
md5_48_4.cnf.gz
md5_48_5.cnf.gz
10pipe_k.cnf.gz
7pipe_k.cnf.gz
8pipe_k.cnf.gz
9pipe_k.cnf.gz
10pipe_q0_k.cnf.gz
11pipe_q0_k.cnf.gz
8pipe_q0_k.cnf.gz

A. Appendix

partial-10-11-s.cnf.gz
partial-10-19-s.cnf.gz
partial-5-17-s.cnf.gz
partial-5-19-s.cnf.gz
dated-10-11-u.cnf.gz
partial-5-13-s.cnf.gz
partial-5-15-s.cnf.gz
total-10-15-s.cnf.gz
itox_vc1033.cnf.gz
itox_vc1130.cnf.gz
AProVE07-02.cnf.gz
AProVE07-11.cnf.gz
vmpc_29.cnf.gz
vmpc_30.cnf.gz
vmpc_33.cnf.gz
vmpc_34.cnf.gz
9vliw_m_9stages_iq3_C1_bug1.cnf.gz
9vliw_m_9stages_iq3_C1_bug10.cnf.gz
9vliw_m_9stages_iq3_C1_bug4.cnf.gz
9vliw_m_9stages_iq3_C1_bug7.cnf.gz
9vliw_m_9stages_iq3_C1_bug8.cnf.gz
9vliw_m_9stages_iq3_C1_bug9.cnf.gz
9dlx_vliw_at_b_iq4.cnf.gz
AProVE09-06.cnf.gz
countbitssrl032.cnf.gz
minxor128.cnf.gz
minxorminand064.cnf.gz
gss-17-s100.cnf.gz
gss-23-s100.cnf.gz
gss-24-s100.cnf.gz
ACG-15-10p1.cnf.gz
ACG-20-5p1.cnf.gz
UR-15-10p0.cnf.gz
UR-15-10p1.cnf.gz
UR-20-5p0.cnf.gz
UR-20-5p1.cnf.gz
UTI-15-10p1.cnf.gz
UTI-20-5p1.cnf.gz
ndhf_xits_21_SAT.cnf.gz
rbcl_xits_14_SAT.cnf.gz
velev-pipe-sat-1.0-b7.cnf.gz
vmpc_32.renamed-as.sat05-1919.cnf.gz
partial-10-17-s.cnf.gz
AProVE07-03.cnf.gz
9vliw_m_9stages_iq3_C1_bug6.cnf.gz
q_query_3_L70_coli.sat.cnf.gz
maxor128.cnf.gz
maxxor032.cnf.gz
maxxor064.cnf.gz
gss-21-s100.cnf.gz
gss-22-s100.cnf.gz
UCG-15-10p1.cnf.gz
UTI-20-10p1.cnf.gz
ndhf_xits_19_UNKNOWN.cnf.gz
slp-synthesis-aes-bottom13.cnf.gz
slp-synthesis-aes-bottom14.cnf.gz
slp-synthesis-aes-top26.cnf.gz
aaai10-planning-ipc5-pathways-13-step17.cnf.gz
aaai10-planning-ipc5-pathways-17-step20.cnf.gz
aaai10-planning-ipc5-pathways-17-step21.cnf.gz
aaai10-planning-ipc5-pipesworld-12-step15.cnf.gz
aaai10-planning-ipc5-TPP-21-step11.cnf.gz
aaai10-planning-ipc5-TPP-30-step11.cnf.gz
hwmcc10-timeframe-expansion-k45-pdtpmsgoodbakery-tseitin.cnf.gz
hwmcc10-timeframe-expansion-k45-pdtvissoap1-tseitin.cnf.gz
hwmcc10-timeframe-expansion-k50-pdtpmsns2-tseitin.cnf.gz
smtlib-qfbv-aigs-ext_con_032_008_0256-tseitin.cnf.gz
smtlib-qfbv-aigs-lfsr_004_127_112-tseitin.cnf.gz
smtlib-qfbv-aigs-servers_slapd_a_vc149789-tseitin.cnf.gz
aes_32_3_keyfind_1.cnf.gz
E02F20.cnf.gz

E02F22.cnf.gz
E04F19.cnf.gz
openstacks-sequencedstrips-nonadl-nonnegated-os-sequencedstrips-p30_3.085-SAT.cnf.gz
transport-transport-city-sequential-25nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.060-SAT.cnf.gz
transport-transport-city-sequential-35nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.030-NOTKNOWN.cnf.gz
transport-transport-city-sequential-35nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.040-NOTKNOWN.cnf.gz
transport-transport-three-cities-sequential-14nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.030-NOTKNOWN.cnf.gz
transport-transport-two-cities-sequential-15nodes-1000size-3degree-100mindistance-3trucks-10packages-2008seed.040-SAT.cnf.gz
velev-pipe-sat-1.0-b9.cnf.gz
esawn_uw3.debugged.cnf.gz
aes_16_10_keyfind_3.cnf.gz
aes_24_4_keyfind_2.cnf.gz
aes_24_4_keyfind_4.cnf.gz
dated-5-19-u.cnf.gz
9dlx_vliw_at_b_iq9.cnf.gz
minandmaxor128.cnf.gz
gss-25-s100.cnf.gz
gus-md5-11.cnf.gz
UR-20-10p1.cnf.gz
rpoc_xits_15_SAT.cnf.gz
vmpc_35.renamed-as.sat05-1921.cnf.gz
vmpc_36.renamed-as.sat05-1922.cnf.gz
slp-synthesis-aes-top25.cnf.gz
aes_64_1_keyfind_1.cnf.gz
E00N23.cnf.gz
transport-transport-three-cities-sequential-14nodes-1000size-4degree-100mindistance-4trucks-14packages-2008seed.020-NOTKNOWN.cnf.gz
SAT_dat.k80.cnf.gz
SAT_dat.k65.cnf.gz
11pipe_11_ooo.cnf.gz
6s20.cnf.gz
6s20.cnf.gz
arcfour_initialPermutation_6_15.cnf.gz
ctl_4201_555_unsat.cnf.gz
ctl_4201_555_unsat_pre.cnf.gz
ctl_4291_567_1_unsat.cnf.gz
k_unsat.cnf.gz
p01_lb_05.cnf.gz
pb_200_03_lb_01.cnf.gz
pb_200_03_lb_02.cnf.gz
pb_400_10_lb_00.cnf.gz
bivium-39-200-0s0-0xdcfb6ab71951500b8e460045bd45afee15c87e08b0072eb174-43.cnf.gz
bivium-40-200-0s0-0xd447c33176b6b675fd5f8dc3a5deda46569dc34eedf37da020-6.cnf.gz
hitag2-10-60-0-0xbc15b17d0353413-10.cnf.gz
hitag2-8-60-0-0xfba1a41b5dfd7f7-52.cnf.gz
hitag2-7-60-0-0xe97b5f1bee04d70-47.cnf.gz

A.2 Algorithms for the Optimisation of Prefixes

Header file for different sorting network related operations

```
1 #include <string.h>
2 #include <iostream>
3 #include <vector>
4 #include <set>
5 #include <map>
6 #include <cstdio>
7 #include <cstdlib>
8 #include <cassert>
9 #include <algorithm>
```

```
10
11 using namespace std;
12
13 /*
14  * Simple structure for a comparator
15  */
16 struct comparator{
17     int layer;
18     int from;
19     int to;
20     comparator(int _a, int _b, int _c) : layer(_a), from(_b), to(_c) {} ;
21     bool operator<(const comparator & other){
22         if(layer != other.layer)
23             return layer < other.layer;
24         if(from != other.from)
25             return from < other.from;
26         return to < other.to;
27     }
28 };
29
30 /*
31  * Parse all layers from a given file.
32  */
33 void parseAllLayers(vector<vector<comparator> > & out,
34                     char * fileName, int diff){
35     FILE * fp = fopen(fileName, "r");
36     int BUF_SIZE = 1<<16;
37     char buffer[BUF_SIZE];
38     int rowRead = 0;
39     while(fp && !feof(fp)){
40         int scan = fscanf(fp, "%s", buffer);
41         if(scan >= 0){
42             char * buff = strtok(buffer, ";");
43             vector<comparator> tmp;
44             while(buff){
45                 int a, b, c;
46                 a = atoi(buff);
47                 buff = strtok(NULL, ";");
48                 b = atoi(buff);
49                 buff = strtok(NULL, ";");
50                 c = atoi(buff);
51                 buff = strtok(NULL, ";");
52                 tmp.push_back(comparator(a, b-diff, c-diff));
53             }
54             out.push_back(tmp);
```

```
55          }
56          rowRead++;
57      }
58  }
59  /*
60   * Determine whether a particular bit in "val" is set
61   */
62  bool isSet(int val, int bit){
63      return val & (1<<bit);
64  }
65  /*
66   * Swap the bits of "val" at position a and b
67   */
68  int swap(int val, int a, int b){
69      assert(a < b);
70      if(val & (1<<a))
71          if(!(val & (1<<b)))
72              val ^= ((1<<a) | (1<<b));
73      return val;
74  }
75  /*
76   * Compute the window size of an input "val" for a sorting network on "n" inputs.
77   */
78  int getWindowSize(int val, int n){
79      int tailingOnes = 0;
80      int leadingZeros = 0;
81      while(leadingZeros < n && !isSet(val, leadingZeros))
82          leadingZeros++;
83      while(tailingOnes < n && isSet(val, n-tailingOnes-1))
84          tailingOnes++;
85      return n - (tailingOnes+leadingZeros);
86  }
87  /*
88   * Compute the outputs of the prefix of a sorting network with respect to
89   * the possible outputs of comparators from this prefix before index "index"
90   */
91  void getOutputs(int index, vector<comparator> prefix, int val, set<int> & outputs){
92      if(prefix[index].layer > 0){
93          for(int i = index ; i < prefix.size() ; i++){
94              val = swap(val, prefix[i].from, prefix[i].to);
95          }
96          outputs.insert(val);
97      }
98      else{
99          /* both outputs zero */
```

```
100         getOutputs(index+1, prefix, val, outputs);
101         /* zero, one */
102         getOutputs(index+1, prefix, val | (1<<prefix[index].to), outputs);
103         /* both outputs 1 */
104         getOutputs(index+1,
105                    prefix,
106                    val | ((1<<prefix[index].to)|(1<<prefix[index].from)),
107                    outputs);
108     }
109 }
110 /*
111  * Faster version of the computation of outputs: For each comparator
112  * in the first layer, consider only its three possible outputs.
113  */
114 void getOutputs(vector<comparator> & prefix, set<int> & outs, int n){
115     // Assume here that the first layer is maximal,
116     // i.e. there is at most one unused channel!
117     int unusedChannel = -1;
118     vector<bool> used(n, false);
119     for(int i = 0 ; i < prefix.size() ; i++){
120         if(prefix[i].layer > 0)
121             break;
122         used[prefix[i].from] = true;
123         used[prefix[i].to] = true;
124     }
125     for(int i = 0 ; i < n ; i++){
126         if(used[i] == false){
127             assert(unusedChannel < 0);
128             unusedChannel = i;
129         }
130     }
131     getOutputs(0, prefix, 0, outs);
132     if(unusedChannel >= 0)
133         getOutputs(0, prefix, 1<<unusedChannel, outs);
134 }
135
136 /*
137  * Compute the outputs of the prefix given as "comps"
138  */
139 void getAllOutputs(vector<comparator> & comps, int n, vector<int> & out){
140     set<int> outs;
141     getOutputs(comps, outs, n);
142     out.insert(out.end(), outs.begin(), outs.end());
143 }
144 /*
```

```
145  * Linear-time algorithm to untangle (the prefix of) a generalised
146  * sorting network. The idea is to store the permutation rather
147  * than permuting the whole suffix whenever
148  * one comparator is turned upside-down.
149  */
150 void untanglePrefix(vector<comparator> & prefix,
151                     vector<int> & permutation,
152                     vector<comparator> & permuted_prefix){
153     assert(permuted_prefix.size() == 0);
154     vector<int> pi(permutation.begin(), permutation.end());
155     for(int i = 0 ; i < prefix.size() ; i++){
156         int from = prefix[i].from;
157         int to = prefix[i].to;
158         assert(from != to);
159         assert(pi[from] != pi[to]);
160         if(pi[from] > pi[to]){
161             int tmp = pi[from];
162             pi[from] = pi[to];
163             pi[to] = tmp;
164         }
165         permuted_prefix.push_back(comparator(prefix[i].layer, pi[from], pi[to]));
166     }
167 }
```

```
1  #include <string.h>
2  #include <iostream>
3  #include <vector>
4  #include <set>
5  #include <map>
6  #include <cstdio>
7  #include <cstdlib>
8  #include <cassert>
9  #include <algorithm>
10
11 #include "SNW.h"
12 using namespace std;
13
14 /*
15  * Using the fisher-yates-shuffle to generate a random permutation.
16  */
17 void fisherYates(int n, vector<int> & out){
18     out.clear();
19     for(int i = 0 ; i < n ; i++)
20         out.push_back(i);
21     while(n > 0){
```

```
22          int next = rand() % n;
23          n--;
24          int tmp = out[next];
25          out[next] = out[n];
26          out[n] = tmp;
27      }
28  }
29
30
31  int numInputs2Count = 20000;
32  /*
33   * Count the sum of window sizes for some inputs, preferably choose
34   * inputs of small window size
35   */
36  int countWindowSizes(const vector<int> & in, int n){
37      vector<int> counts(n+1, 0);
38      for(vector<int>::const_iterator it = in.begin() ; it != in.end() ; it++){
39          counts[getWindowSize(*it, n)]++;
40      }
41      int rVal = 0;
42      int countedAlready = 0;
43      for(int i = 0 ; i < counts.size() ; i++){
44          if(countedAlready + counts[i] > numInputs2Count){
45              rVal += (numInputs2Count - countedAlready) * i;
46              break;
47          }
48          else{
49              rVal += i * counts[i];
50              countedAlready += counts[i];
51          }
52      }
53      return rVal;
54  }
55
56  /*
57   * Perform "untangling".
58   * This does not actually touch the prefix - it just computes
59   * the permutation which is performed with respect to the outputs
60   */
61  void getPermutation(vector<comparator> & prefix,
62                      vector<int> & permutation_in,
63                      vector<int> & permutation_out){
64      assert(permutation_out.size() == 0);
65      permutation_out.insert(permutation_out.end(),
66                             permutation_in.begin(),
```

```
67                              permutation_in.end());
68      for(int i = 0 ; i < prefix.size() ; i++){
69          // is this one reversed? If so, change permutation!
70          int from = prefix[i].from;
71          int to = prefix[i].to;
72          if(permutation_out[from] > permutation_out[to]){
73              int tmp = permutation_out[from];
74              permutation_out[from] = permutation_out[to];
75              permutation_out[to] = tmp;
76          }
77      }
78  }
79
80  /*
81    Permute the output according to "permutation"
82   */
83  int permuteOutput(int val, vector<int> & permutation, int n){
84      int index = 0;
85      int rVal = 0;
86      while(val != 0){
87          if(val & 1){
88              rVal |= (1 << permutation[index]);
89          }
90          val >>= 1;
91          index++;
92      }
93      return rVal;
94  }
95  /*
96   * Permute all outputs with respect the some given permutation
97   */
98  void getPermutedOutputs(vector<int> & permutation,
99                          vector<int> & in,
100                         vector<int> & out,
101                         int n){
102     for(vector<int>::iterator it = in.begin() ; it != in.end() ; it++){
103         out.push_back(permuteOutput(*it, permutation, n));
104     }
105 }
106
107 int gradient_descent(vector<comparator> & p,
108                      int n,
109                      vector<int> & initial_permutation,
110                      int (*f) (const vector<int>&, int),
111                      bool randomised){
```

```
112     bool local_minimum = false;
113     vector<int> current_solution(initial_permutation.begin(),
114                                  initial_permutation.end());
115     vector<int> outputs;
116     getAllOutputs(p, n, outputs);
117     vector<int> initialOutputs;
118     getPermutedOutputs(current_solution, outputs, initialOutputs, n);
119     int initialRating = f(initialOutputs, n);
120     int best_rating = initialRating;
121     int pi_new_1;
122     int pi_new_2;
123     while(!local_minimum){
124         local_minimum = true;
125         // Assume a local minimum has been found, unless we find a
126         // better solution in this iteration.
127         // Swap 2 channels, and check if this improves "f".
128         // If so, take the pair which yields the steepest descent
129         for(int i = 0 ; i < n ; i++){
130             for(int j = i+1 ; j < n ; j++){
131                 vector<int> new_permutation(current_solution.begin(),
132                                             current_solution.end());
133                 int tmp = new_permutation[i];
134                 new_permutation[i] = new_permutation[j];
135                 new_permutation[j] = tmp;
136                 vector<int> untangled;
137                 getPermutation(p, new_permutation, untangled);
138                 vector<int> permutedOutputs;
139                 getPermutedOutputs(new_permutation, outputs, permutedOutputs, n);
140                 int new_rating = f(permutedOutputs, n);
141                 if(new_rating < best_rating){
142                     best_rating = new_rating;
143                     local_minimum = false;
144                     pi_new_1 = i;
145                     pi_new_2 = j;
146                 }
147             }
148         }
149         if(!local_minimum){
150             int tmp = current_solution[pi_new_1];
151             current_solution[pi_new_1] = current_solution[pi_new_2];
152             current_solution[pi_new_2] = tmp;
153         }
154     }
155     return best_rating;
156 }
```

```
157
158 void optimisePrefix(vector<comparator> & p,
159                      int n,
160                      int (*f) (const vector<int>&, int)){
161     // Create outputs of this prefix
162     vector<int> outs;
163     getAllOutputs(p, n, outs);
164     cout << "Initial ranking: " << f(outs, n) << endl;
165     int best_rating = 1<<30;
166     for(int i = 0 ; i < 3 ; i++){
167         vector<int> perm;
168         fisherYates(n, perm);
169         int new_found = gradient_descent(p, n, perm, f, false);
170         best_rating = min(best_rating, new_found);
171     }
172    cout << "Best rating: " << best_rating << endl;
173 }
174 /*
175  * Parameters:
176  *  1) the input file containing the prefixes to optimise
177  *  2) the number of input bits
178  *  3) (optional) If the numbering of the channels does not
179  *              begin with "0", this is the offset
180  */
181 int main(int argc, char ** argv)
182 {
183     int n = atoi(argv[2]);
184     int diff = argc >= 4 ? atoi(argv[3]) : 0;
185     vector<vector<comparator> > allLayers;
186     parseAllLayers(allLayers, argv[1], diff);
187     for(vector<vector<comparator> >::iterator it = allLayers.begin() ;
188             it != allLayers.end() ;
189             it++){
190         vector<comparator> & prefix = *it;
191         optimisePrefix(prefix, n, countWindowSizes);
192     }
193     return 0;
194 }
```

Table A.1. Comparison of results from the SAT Competition 2016.

Channels	Layers	CCS	prefix	enc	pre	solved	time (all)	time (solved)
13	8	no	no	no	yes	28	1532	1408
13	8	yes	no	yes	no	28	576	418
13	8	yes	yes	no	no	28	240	70
13	8	yes	yes	no	yes	28	217	47
13	8	yes	yes	yes	yes	28	229	59
13	9	no	no	no	yes	28	213	42
13	9	no	no	yes	yes	28	181	9
13	9	no	yes	no	yes	28	178	6
13	9	no	yes	yes	yes	28	175	3
13	9	yes	no	no	yes	28	200	28
13	9	yes	no	yes	yes	28	181	9
16	8	no	no	no	no	16	3606	2475
16	8	no	no	no	yes	27	2751	2584
16	8	no	yes	no	yes	28	463	301
16	9	no	no	no	yes	1	4971	4218
16	9	no	no	yes	yes	7	4494	2910
16	9	no	yes	no	yes	28	1051	911
16	9	no	yes	yes	yes	28	767	615
16	9	yes	no	no	yes	4	4705	2878
16	9	yes	yes	no	yes	28	458	296
17	9	yes	yes	yes	yes	0	4998	—

A.3 Comparison of Results from SAT Competition 2016

This table summarises results from the SAT competition 2016. Here, CCS denotes whether the symmetry break on suffixes by Codish, Cruz-Filipe and Schneider-Kamp was used. The use of an optimised prefix is indicated by “prefix", and “enc" denotes the newly introduced predicates. A preprocessor was used if “pre" is yes.

Bibliography

[1] Aug. 2016. URL: www.minizinc.org.

[2] *26th IEEE int. conf. tools with artificial intelligence, ICTAI 2014, limassol, cyprus, nov. 10-12, 2014*. IEEE Computer Society, 2014.

[3] Ignasi Abío, Graeme Gange, Valentin Mayer-Eichberger, and Peter J. Stuckey. "On CNF encodings of decision diagrams". In: *Integration of AI and OR Techniques in Constraint Programming - 13th Int. Conf., CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proc.* Ed. by Claude-Guy Quimper. Vol. 9676. LNCS. Springer, 2016, pp. 1–17.

[4] Ignasi Abío, Valentin Mayer-Eichberger, and Peter J. Stuckey. "Encoding linear constraints with implication chains to CNF". In: *Principles and Practice of Constraint Programming - 21st Int. Conf., CP 2015, Cork, Ireland, Aug. 31 - Sept. 4, 2015, Proc.* Ed. by Gilles Pesant. Vol. 9255. LNCS. Springer, 2015, pp. 3–11.

[5] Ignasi Abio, Robert Nieuwenhuis, Albert Oliveras, Enric Rodriguez-Carbonell, and Peter J. Stuckey. "To encode or to propagate? the best choice for each constraint in sat". In: *Principles and Practice of Constraint Programming: 19th Int. Conf., CP 2013, Uppsala, Sweden, Sept. 16-20, 2013. Proc.* Ed. by Christian Schulte. Berlin, Heidelberg: Springer Berlin Heidelberg, 2013, pp. 97–106. ISBN: 978-3-642-40627-0.

[6] Michael Alekhnovich, Jan Johannsen, Toniann Pitassi, and Alasdair Urquhart. "An exponential separation between regular and general resolution". In: *Theory of Computing* 3.1 (2007), pp. 81–102.

[7] Roberto Amadini, Maurizio Gabbrielli, and Jacopo Mauro. "Portfolio approaches for constraint optimization problems". In: *Learning and Intelligent Optimization - 8th Int. Conf., Lion 8, Gainesville, FL, USA, Feb. 16-21, 2014. Revised Selected Papers*. Ed. by Panos M. Pardalos, Mauricio G. C. Resende, Chrysafis Vogiatzis, and Jose L. Walteros. Vol. 8426. LNCS. Springer, 2014, pp. 21–35.

[8] Roberto Amadini and Peter J. Stuckey. “Sequential time splitting and bounds communication for a portfolio of optimization solvers”. In: *Principles and Practice of Constraint Programming - 20th Int. Conf., CP 2014, Lyon, France, Sept. 8-12, 2014. Proc.* Ed. by Barry O’Sullivan. Vol. 8656. LNCS. Springer, 2014, pp. 108–124.

[9] Carlos Ansótegui, Jesús Giráldez-Cru, and Jordi Levy. “The community structure of SAT formulas”. In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th Int. Conf., Trento, Italy, June 17-20, 2012. Proc.* Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. LNCS. Springer, 2012, pp. 410–423. ISBN: 978-3-642-31611-1.

[10] Carlos Ansótegui, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. “Using community structure to detect relevant learnt clauses”. In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th Int. Conf., Austin, TX, USA, Sept. 24-27, 2015, Proc.* Ed. by Marijn Heule and Sean Weaver. Vol. 9340. LNCS. Springer, 2015, pp. 238–254.

[11] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. “Cardinality networks: a theoretical and empirical study”. In: *Constraints* 16.2 (2011), pp. 195–221.

[12] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, Jean-Marie Lagniez, and Cédric Piette. “Revisiting clause exchange in parallel SAT solving”. In: *Theory and Applications of Satisfiability Testing - SAT 2012 - 15th Int. Conf., Trento, Italy, June 17-20, 2012. Proc.* Ed. by Alessandro Cimatti and Roberto Sebastiani. Vol. 7317. LNCS. Springer, 2012, pp. 200–213. ISBN: 978-3-642-31611-1.

[13] Gilles Audemard, Benoît Hoessen, Saïd Jabbour, and Cédric Piette. “Dolius: A distributed parallel SAT solving framework”. In: *POS-14. 5th Pragmatics of SAT workshop, workshop SAT 2014 Conf., part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria.* Ed. by Daniel Le Berre. Vol. 27. EPiC Series in Computing. EasyChair, 2014, pp. 1–11.

[14] Gilles Audemard, George Katsirelos, and Laurent Simon. “A restriction of extended resolution for clause learning SAT solvers”. In: *Proc. 24th AAAI Conf. on Artificial Intelligence, AAAI 2010, Atlanta, Georgia, USA, July 11-15, 2010.* Ed. by Maria Fox and David Poole. AAAI Press, 2010.

[15] Gilles Audemard, Jean-Marie Lagniez, Bertrand Mazure, and Lakhdar Sais. “On freezing and reactivating learnt clauses”. In: *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th Int. Conf., SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proc.* Ed. by Karem A. Sakallah and Laurent Simon. Vol. 6695. LNCS. Springer, 2011, pp. 188–200. ISBN: 978-3-642-21580-3.

[16] Gilles Audemard, Jean-Marie Lagniez, Nicolas Szczepanski, and Sébastien Tabary. “An adaptive parallel SAT solver”. In: *Principles and Practice of Constraint Programming - 22nd Int. Conf., CP 2016, Toulouse, France, Sept. 5-9, 2016, Proc.* Ed. by Michel Rueher. Vol. 9892. LNCS. Springer, 2016, pp. 30–48. ISBN: 978-3-319-44952-4.

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

[18] Gilles Audemard and Laurent Simon. “Lazy clause exchange policy for parallel SAT solvers”. In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th Int. Conf., Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proc.* Ed. by Carsten Sinz and Uwe Egly. Vol. 8561. LNCS. Springer, 2014, pp. 197–205.

[19] Gilles Audemard and Laurent Simon. “Predicting learnt clauses quality in modern SAT solvers”. In: *IJCAI 2009, Proc. 21st Int. Joint Conf. on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009.* Ed. by Craig Boutilier. 2009, pp. 399–404.

[20] Gilles Audemard and Laurent Simon. “Refining restarts strategies for SAT and UNSAT”. In: *Principles and Practice of Constraint Programming - 18th Int. Conf., CP 2012, Québec City, QC, Canada, Oct. 8-12, 2012. Proc.* Ed. by Michela Milano. Vol. 7514. LNCS. Springer, 2012, pp. 118–126.

[21] Sherenaz W. Al-Haj Baddar. “Finding better sorting networks”. PhD thesis. Kent State University, 2009.

[22] Sherenaz W. Al-Haj Baddar and Kenneth E. Batcher. “An 11-step sorting network for 18 elements”. In: *Parallel Processing Letters* 19.1 (2009), pp. 97–103.

[23] Sherenaz W. Al-Haj Baddar and Kenneth E. Batcher. *Designing sorting networks: a new paradigm*. Springer, 2011.

[24] Daniele Bagni. “Median filter and sorting network for video processing with vivado hls”. In: *Xcell Journal* 86 (2014), pp. 20–28.

[25] Adrian Balint, Marijn JH Heule, Anton Belov, and Matti Järvisalo. “The application and the hard combinatorial benchmarks in sat competition 2013”. In: *Proc. of SAT Competition* (2013), p. 99.

[26] Thomas Ball, Vladimir Levin, and Sriram K. Rajamani. “A decade of software model checking with SLAM”. In: *Commun. ACM* 54.7 (2011), pp. 68–76.

[27] T Balyo, M J H Heule, and M J Järvisalo, eds. *Proc. of sat competition 2016 : solver and benchmark descriptions*. Department of Computer Science Series of Publications B , vol. B-2016-1 , University of Helsinki , Helsinki, 2016.

[28] Tomas Balyo and Florian Lonsing. “Hordeqbf: A modular and massively parallel QBF solver”. In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th Int. Conf., Bordeaux, France, July 5-8, 2016, Proc.* Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. LNCS. Springer, 2016, pp. 531–538.

[29] Tomas Balyo, Peter Sanders, and Carsten Sinz. “Hordesat: A massively parallel portfolio SAT solver”. In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th Int. Conf., Austin, TX, USA, Sept. 24-27, 2015, Proc.* Ed. by Marijn Heule and Sean Weaver. Vol. 9340. LNCS. Springer, 2015, pp. 156–172.

[30] Indranil Banerjee and Dana S. Richards. “Routing and sorting via matchings on graphs”. In: *CoRR* abs/1604.04978 (2016).

[31] Clark Barrett, Christopher L. Conway, Morgan Deters, Liana Hadarean, Dejan Jovanovic, Tim King, Andrew Reynolds, and Cesare Tinelli. “CVC4”. In: *Computer Aided Verification - 23rd Int. Int. Conf., CAV 2011, Snowbird, UT, USA, July 14-20, 2011. Proc.* Ed. by Ganesh Gopalakrishnan and Shaz Qadeer. Vol. 6806. LNCS. Springer, 2011, pp. 171–177.

[32] Kenneth E. Batcher. “Sorting networks and their applications”. In: *American Federation of Information Processing Societies: AFIPS Conf. Proc.: 1968 Spring Joint Computer Conf., Atlantic City, NJ, USA, 30 Apr. - 2 May 1968*. Vol. 32. AFIPS Conf. Proc. Thomson Book Company, Washington D.C., 1968, pp. 307–314.

[33] Paul Beame, Russell Impagliazzo, Toniann Pitassi, and Nathan Segerlind. “Memoization and DPLL: formula caching proof systems”. In: *18th Annu. IEEE Conf. on Computational Complexity (Complexity 2003), 7-10 July 2003, Aarhus, Denmark*. IEEE Computer Society, 2003, p. 248.

[34] Paul Beame, Henry A. Kautz, and Ashish Sabharwal. “Understanding the power of clause learning”. In: *IJCAI-03, Proc. 18th Int. Joint Conf. on Artificial Intelligence, Acapulco, Mexico, Aug. 9-15, 2003*. Ed. by Georg Gottlob and Toby Walsh. Morgan Kaufmann, 2003, pp. 1194–1201.

[35] Ralph Becket. *Specification of flatzinc*. G12. 2014.

[36] E. Bensana, Michel Lemaître, and Gérard Verfaillie. “Earth observation satellite management”. In: *Constraints* 4.3 (1999), pp. 293–299.

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

[38] Eli Ben-Sasson and Avi Wigderson. “Short proofs are narrow - resolution made simple”. In: *J. ACM* 48.2 (2001), pp. 149–169.

[39] Jeremias Berg and Matti Järvisalo. “Sat-based approaches to treewidth computation: an evaluation”. In: *26th IEEE Int. Conf. on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, Nov. 10-12, 2014*. IEEE Computer Society, 2014, pp. 328–335.

[40] Daniel Le Berre, ed. *POS-14. 5th pragmatics of SAT workshop, workshop SAT 2014 conf., part of floc 2014 during the vienna summer of logic, july 13, 2014, vienna, austria*. Vol. 27. EPiC Series in Computing. EasyChair, 2014.

[41] Christian Bessiere, ed. *Principles and practice of constraint programming - CP 2007, 13th int. conf., CP 2007, providence, ri, usa, sept. 23-27, 2007, proc*. Vol. 4741. LNCS. Springer, 2007.

[42] Dirk Beyer, Thomas A. Henzinger, Ranjit Jhala, and Rupak Majumdar. “The software model checker blast”. In: *STTT* 9.5-6 (2007), pp. 505–525.

[43] Armin Biere. “Adaptive restart strategies for conflict driven SAT solvers”. In: *Theory and Applications of Satisfiability Testing - SAT 2008, 11th Int. Conf., SAT 2008, Guangzhou, China, May 12-15, 2008. Proc.* Ed. by Hans Kleine Büning and Xishun Zhao. Vol. 4996. LNCS. Springer, 2008, pp. 28–33.

[44] Armin Biere. “Lingeling, plingeling and treengeling entering the sat competition 2013”. In: *Proc. of SAT Competition* (2013), pp. 51–52.

[45] Armin Biere. “Picosat essentials”. In: *JSAT* 4.2-4 (2008), pp. 75–97.

[46] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, Ofer Strichman, and Yunshan Zhu. “Bounded model checking”. In: *Advances in Computers* 58 (2003), pp. 117–148.

[47] Armin Biere, Alessandro Cimatti, Edmund M. Clarke, and Yunshan Zhu. “Symbolic model checking without bdds”. In: *Tools and Algorithms for Construction and Analysis of Systems, 5th Int. Conf., TACAS ’99, Held as Part European Joint Conf.s on the Theory and Practice of Software, ETAPS’99, Amsterdam, The Netherlands, Mar. 22-28, 1999, Proc.* Ed. by Rance Cleaveland. Vol. 1579. LNCS. Springer, 1999, pp. 193–207.

[48] Miquel Bofill, Robert Nieuwenhuis, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. “The barcelogic SMT solver”. In: *Computer Aided Verification, 20th Int. Int. Conf., CAV 2008, Princeton, NJ, USA, July 7-14, 2008, Proc.* Ed. by Aarti Gupta and Sharad Malik. Vol. 5123. LNCS. Springer, 2008, pp. 294–298.

[49] Max Böhm and Ewald Speckenmeyer. “A fast parallel sat-solver - efficient workload balancing”. In: *Ann. Math. Artif. Intell.* 17.3-4 (1996), pp. 381–400.

[50] Maria Luisa Bonet, Juan Luis Esteban, Nicola Galesi, and Jan Johannsen. “On the relative complexity of resolution refinements and cutting planes proof systems”. In: *SIAM J. Comput.* 30.5 (2000), pp. 1462–1484.

[51] George Boole. “The calculus of logic”. In: *The Cambridge and Dublin Mathematical Journal* 3 (1848). URL: http://www.maths.tcd.ie/pub/HistMath/People/Boole/CalcLogic/.

[52] George Boole. *The mathematical analysis of logic*. 1847.

[53] Craig Boutilier, ed. *IJCAI 2009, proc. 21st int. joint conf. on artificial intelligence, pasadena, california, usa, july 11-17, 2009*. 2009.

[54] Aaron R. Bradley. “Sat-based model checking without unrolling”. In: *Verification, Model Checking, and Abstract Interpretation - 12th Int. Conf., VMCAI 2011, Austin, TX, USA, Jan. 23-25, 2011. Proc.* Ed. by Ranjit Jhala and David A. Schmidt. Vol. 6538. LNCS. Springer, 2011, pp. 70–87.

[55] Daniel Bundala, Michael Codish, Luís Cruz-Filipe, Peter Schneider-Kamp, and Jakub Závodný. “Optimal-depth sorting networks”. In: *J. Comput. Syst. Sci.* 84 (2017), pp. 185–204.

[56] Daniel Bundala and Jakub Závodný. “Optimal sorting networks”. In: *Language and Automata Theory and Applications - 8th Int. Conf., LATA 2014, Madrid, Spain, Mar. 10-14, 2014. Proc.* Vol. 8370. LNCS. Springer, 2014, pp. 236–247.

[57] Hans Kleine Büning and Xishun Zhao, eds. *Theory and applications of satisfiability testing - SAT 2008, 11th int. conf., SAT 2008, guangzhou, china, may 12-15, 2008. proc.* Vol. 4996. LNCS. Springer, 2008.

[58] Joshua Buresh-Oppenheim and Toniann Pitassi. “The complexity of resolution refinements”. In: *J. Symb. Log.* 72.4 (2007), pp. 1336–1352.

[59] Samuel R Buss. *Handbook of proof theory*. Vol. 137. Elsevier, 1998.

[60] Cristian Cadar, Daniel Dunbar, and Dawson R. Engler. “KLEE: unassisted and automatic generation of high-coverage tests for complex systems programs”. In: *8th USENIX Symp. on Operating Systems Design and Implementation, OSDI 2008, Dec. 8-10, 2008, San Diego, California, USA, Proc.* Ed. by Richard Draves and Robbert van Renesse. USENIX Association, 2008, pp. 209–224.

[61] Gabriele Capannini, Fabrizio Silvestri, and Ranieri Baraglia. “Sorting on gpus for large scale datasets: A thorough comparison”. In: *Inf. Process. Manage.* 48.5 (2012), pp. 903–917.

[62] Geoffrey Chu. “Improving combinatorial optimization”. PhD thesis. University of Melbourne, 2011.

[63] Geoffrey Chu, Aaron Harwood, and Peter J. Stuckey. “Cache conscious data structures for boolean satisfiability solvers”. In: *JSAT* 6.1-3 (2009), pp. 99–120.

[64] Geoffrey Chu, Christian Schulte, and Peter J. Stuckey. “Confidence-based work stealing in parallel constraint programming”. In: *Principles and Practice of Constraint Programming - CP 2009, 15th Int. Conf., CP 2009, Lisbon, Portugal, Sept. 20-24, 2009, Proc.* Ed. by Ian P. Gent. Vol. 5732. LNCS. Springer, 2009, pp. 226–241.

[65] Geoffrey Chu and Peter J. Stuckey. “Structure based extended resolution for constraint programming”. In: *CoRR* abs/1306.4418 (2013). URL: http://arxiv.org/abs/1306.4418.

[66] Geoffrey Chu, Peter J Stuckey, and Aaron Harwood. “Pminisat: a parallelization of minisat 2.0”. In: *SAT race* (2008).

[67] Vasek Chvátal. *Lecture notes on the new aks sorting network*. Lecture Notes. URL: http://users.encs.concordia.ca/~chvatal/aks.pdf.

[68] Alessandro Cimatti and Roberto Sebastiani, eds. *Theory and applications of satisfiability testing - SAT 2012 - 15th int. conf., trento, italy, june 17-20, 2012. proc.* Vol. 7317. LNCS. Springer, 2012. ISBN: 978-3-642-31611-1.

[69] Edmund M. Clarke, Orna Grumberg, Somesh Jha, Yuan Lu, and Helmut Veith. “Counterexample-guided abstraction refinement”. In: *Computer Aided Verification, 12th Int. Conf., CAV 2000, Chicago, IL, USA, July 15-19, 2000, Proc.* Ed. by E. Allen Emerson and A. Prasad Sistla. Vol. 1855. LNCS. Springer, 2000, pp. 154–169. ISBN: 3-540-67770-4.

[70] Michael Codish, Luis Cruz-Filipe, Thorsten Ehlers, Mike Müller, and Peter Schneider-Kamp. “Sorting networks: to the end and back again”. In: *Journal of Computer and System Sciences* (2016).

[71] Michael Codish, Luís Cruz-Filipe, Michael Frank, and Peter Schneider-Kamp. “Twenty-five comparators is optimal when sorting nine inputs (and twenty-nine for ten)”. In: *26th IEEE Int. Conf. on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, Nov. 10-12, 2014*. IEEE Computer Society, 2014, pp. 186–193.

[72] Michael Codish, Luís Cruz-Filipe, Markus Nebel, and Peter Schneider-Kamp. “Applying sorting networks to synthesize optimized sorting libraries”. In: *Logic-Based Program Synthesis and Transformation - 25th Int. Symp., LOPSTR 2015, Siena, Italy, July 13-15, 2015. Revised Selected*

Papers. Ed. by Moreno Falaschi. Vol. 9527. LNCS. Springer, 2015, pp. 127–142.

[73] Michael Codish, Luís Cruz-Filipe, and Peter Schneider-Kamp. "Sorting networks: the end game". In: *Language and Automata Theory and Applications - 9th Int. Conf., LATA 2015, Nice, France, Mar. 2-6, 2015, Proc.* Ed. by Adrian Horia Dediu, Enrico Formenti, Carlos Martín-Vide, and Bianca Truthe. Vol. 8977. LNCS. Springer, 2015, pp. 664–675.

[74] Michael Codish, Luís Cruz-Filipe, and Peter Schneider-Kamp. "The quest for optimal sorting networks: efficient generation of two-layer prefixes". In: *16th Int. Symp. on Symbolic and Numeric Algorithms for Scientific Computing, SYNASC 2014, Timisoara, Romania, Sept. 22-25, 2014*. Ed. by Franz Winkler, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, Stephen M. Watt, and Daniela Zaharie. IEEE, 2014, pp. 359–366.

[75] Michael Codish, Michael Frank, Avraham Itzhakov, and Alice Miller. "Computing the ramsey number r(4, 3, 3) using abstraction and symmetry breaking". In: *Constraints* 21.3 (2016), pp. 375–393.

[76] Stephen A. Cook. "A short proof of the pigeon hole principle using extended resolution". In: *SIGACT News* 8.4 (Oct. 1976), pp. 28–32. ISSN: 0163-5700.

[77] Stephen A. Cook. "The complexity of theorem-proving procedures". In: *Proc. 3rd Annu. ACM Symp. on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*. Ed. by Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman. ACM, 1971, pp. 151–158.

[78] Stephen A. Cook and Robert A. Reckhow. "On the lengths of proofs in the propositional calculus (preliminary version)". In: *Proc. 6th Annu. ACM Symp. on Theory of Computing, Apr. 30 - May 2, 1974, Seattle, Washington, USA*. Ed. by Robert L. Constable, Robert W. Ritchie, Jack W. Carlyle, and Michael A. Harrison. ACM, 1974, pp. 135–148.

[79] Stephen A. Cook and Robert A. Reckhow. "The relative efficiency of propositional proof systems". In: *J. Symb. Log.* 44.1 (1979), pp. 36–50.

[80] Thomas H. Cormen, Charles E. Leiserson, Ronald L. Rivest, and Clifford Stein. *Introduction to algorithms, 3rd edition*. 3rd. The MIT Press, 2009.

[81] Nadia Creignou and Daniel Le Berre, eds. *Theory and applications of satisfiability testing - SAT 2016 - 19th int. conf., bordeaux, france, july 5-8, 2016, proc.* Vol. 9710. LNCS. Springer, 2016.

[82] Adnan Darwiche. "Decomposable negation normal form". In: *J. ACM* 48.4 (2001), pp. 608–647.

[83] Martin Davis, George Logemann, and Donald Loveland. "A machine program for theorem-proving". In: *Commun. ACM* 5.7 (July 1962), pp. 394–397. ISSN: 0001-0782.

[84] Martin Davis and Hilary Putnam. "A computing procedure for quantification theory". In: *J. ACM* 7.3 (July 1960), pp. 201–215. ISSN: 0004-5411.

[85] Augustus De Morgan. *Formal logic, or, the calculus of inference, necessary and probable*. London: Taylor and Walton, 1847. URL: https://archive.org/details/formallogicorthe00demouoft, %20https://archive.org/details/formallogicorca00morggoog, %20https://archive.org/details/formallogicorca01morggoog, %20https://archive.org/details/bub_gb_HscAAAAAMAAJ.

[86] Rina Dechter. "Enhancement schemes for constraint processing: backjumping, learning, and cutset decomposition". In: *Artif. Intell.* 41.3 (1990), pp. 273–312.

[87] Nicholas Downing, Thibaut Feydy, and Peter J. Stuckey. "Explaining alldifferent". In: *35th Australasian Computer Science Conf., ACSC 2012, Melbourne, Australia, Jan. 2012*. Ed. by Mark Reynolds and Bruce H. Thomas. Vol. 122. CRPIT. Australian Computer Society, 2012, pp. 115–124.

[88] Niklas Eén and Armin Biere. "Effective preprocessing in SAT through variable and clause elimination". In: *Theory and Applications of Satisfiability Testing, 8th Int. Conf., SAT 2005, St. Andrews, UK, June 19-23, 2005, Proc.* Ed. by Fahiem Bacchus and Toby Walsh. Vol. 3569. LNCS. Springer, 2005, pp. 61–75.

[89] Niklas Eén and Niklas Sörensson. "An extensible sat-solver". In: *Theory and Applications of Satisfiability Testing, 6th Int. Conf., SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003 Selected Revised Papers*. Ed. by Enrico Giunchiglia and Armando Tacchella. Vol. 2919. LNCS. Springer, 2003, pp. 502–518.

[90] Thorsten Ehlers. “Merging almost sorted sequences yields a 24-sorter”. In: *Inf. Process. Lett.* 118 (2017), pp. 17–20.

[91] Thorsten Ehlers, Florin Manea, Robert Mercas, and Dirk Nowotka. “K-abelian pattern matching”. In: *Developments in Language Theory - 18th Int. Conf., DLT 2014, Ekaterinburg, Russia, Aug. 26-29, 2014. Proc.* Ed. by Arseny M. Shur and Mikhail V. Volkov. Vol. 8633. LNCS. Springer, 2014, pp. 178–190.

[92] Thorsten Ehlers, Florin Manea, Robert Mercas, and Dirk Nowotka. “K-abelian pattern matching”. In: *J. Discrete Algorithms* 34 (2015), pp. 37–48.

[93] Thorsten Ehlers and Mike Müller. “Faster sorting networks for 17, 19 and 20 inputs”. In: *CoRR* abs/1410.2736 (2014).

[94] Thorsten Ehlers and Mike Müller. “New bounds on optimal sorting networks”. In: *Evolving Computability - 11th Conf. on Computability in Europe, CiE 2015, Bucharest, Romania, June 29 - July 3, 2015. Proc.* Ed. by Arnold Beckmann, Victor Mitrana, and Mariya Ivanova Soskova. Vol. 9136. LNCS. Springer, 2015, pp. 167–176.

[95] Thorsten Ehlers and Dirk Nowotka. “Sat-encodings of sorting networks”. In: *SAT COMPETITION 2016* (), p. 72.

[96] Thorsten Ehlers and Dirk Nowotka. “Sequential and parallel glucose hacks”. In: *SAT COMPETITION 2016* (), p. 39.

[97] Thorsten Ehlers, Dirk Nowotka, and Philipp Sieweck. “Communication in massively-parallel SAT solving”. In: *26th IEEE Int. Conf. Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, Nov. 10-12, 2014*. IEEE Computer Society, 2014, pp. 709–716.

[98] Thorsten Ehlers, Dirk Nowotka, and Philipp Sieweck. “Finding race conditions in real-time code by using formal software verification”. In: *FORMS/FORMAT 2014: 10th Symp. on Formal Methods for Automation and Safety in Railway and Automotive Systems*. Ed. by Eckehard Schnieder and Geza Tarnai. Institute for Traffic Safety and Automation Engineering, TU Braunschweig, 2014, pp. 38–47.

[99] Thorsten Ehlers, Dirk Nowotka, Philipp Sieweck, and Johannes Traub. “Formal software verification for the migration of embedded code from single- to multicore systems”. In: *Software Engineering 2014, Fachtagung des GI-Fachbereichs Softwaretechnik, 25. Februar - 28. Februar 2014, Kiel, Deutschland*. Ed. by Wilhelm Hasselbring and Nils Christian Ehmke. Vol. 227. LNI. GI, 2014, pp. 137–142.

[100] Thorsten Ehlers and Peter J. Stuckey. “Parallelizing constraint programming with learning”. In: *Integration of AI and OR Techniques in Constraint Programming - 13th Int. Conf., CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proc.* Ed. by Claude-Guy Quimper. Vol. 9676. LNCS. Springer, 2016, pp. 142–158.

[101] Thibaut Feydy, Andreas Schutt, and Peter Stuckey. “Semantic learning for lazy clause generation”. In: 2013.

[102] Thibaut Feydy and Peter J. Stuckey. “Lazy clause generation reengineered”. In: *Principles and Practice of Constraint Programming - CP 2009, 15th Int. Conf., CP 2009, Lisbon, Portugal, Sept. 20-24, 2009, Proc.* Ed. by Ian P. Gent. Vol. 5732. LNCS. Springer, 2009, pp. 352–366.

[103] R. A. Fisher and F. Yates. *Statistical Tables for Biological, Agricultural and Medical Research*. 1938.

[104] Kilian Gebhardt and Norbert Manthey. “Parallel variable elimination on CNF formulas”. In: *KI 2013: Advances in Artificial Intelligence - 36th Annu. German Conf. on AI, Koblenz, Germany, Sept. 16-20, 2013. Proc.* Ed. by Ingo J. Timm and Matthias Thimm. Vol. 8077. LNCS. Springer, 2013, pp. 61–73. ISBN: 978-3-642-40941-7.

[105] Gecode Team. *Gecode: generic constraint development environment*. Available from `http://www.gecode.org`. 2006.

[106] Ian P. Gent, ed. *Principles and practice of constraint programming - CP 2009, 15th int. conf., CP 2009, lisbon, portugal, sept. 20-24, 2009, proc.* Vol. 5732. LNCS. Springer, 2009.

[107] Jesús Giráldez-Cru and Jordi Levy. “Generating SAT instances with community structure”. In: *Artif. Intell.* 238 (2016), pp. 119–134.

[108] Andreas Goerdt. “Davis-putnam resolution versus unrestricted resolution”. In: *Ann. Math. Artif. Intell.* 6.1-3 (1992), pp. 169–184.

[109] Andreas Goerdt. “Regular resolution versus unrestricted resolution”. In: *SIAM J. Comput.* 22.4 (1993), pp. 661–683.

[110] Canan Gunes, Willem Jan van Hoeve, and Sridhar R. Tayur. “Vehicle routing for food rescue programs: A comparison of different approaches”. In: *Integration of AI and OR Techniques in Constraint Programming for Combinatorial Optimization Problems, 7th Int. Conf., CPAIOR 2010, Bologna, Italy, June 14-18, 2010. Proc.* Ed. by Andrea Lodi, Michela Milano, and Paolo Toth. Vol. 6140. LNCS. Springer, 2010, pp. 176–180.

[111] Long Guo, Saïd Jabbour, Jerry Lonlac, and Lakhdar Sais. “Diversification by clauses deletion strategies in portfolio parallel SAT solving”. In: *26th IEEE Int. Conf. on Tools with Artificial Intelligence, ICTAI 2014, Limassol, Cyprus, Nov. 10-12, 2014*. IEEE Computer Society, 2014, pp. 701–708.

[112] Shai Haim and Marijn Heule. “Towards ultra rapid restarts”. In: *CoRR* abs/1402.4413 (2014).

[113] Armin Haken. “The intractability of resolution”. In: *Theor. Comput. Sci.* 39 (1985), pp. 297–308.

[114] Philip Hall. “On representation of subsets”. In: *Quart. J. Math. (Oxford)* (1935), pp. 26–30.

[115] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. “Control-based clause sharing in parallel SAT solving”. In: *IJCAI 2009, Proc. 21st Int. Joint Conf. on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*. Ed. by Craig Boutilier. 2009, pp. 499–504.

[116] Youssef Hamadi, Saïd Jabbour, and Lakhdar Sais. “Manysat: a parallel SAT solver”. In: *JSAT* 6.4 (2009), pp. 245–262.

[117] Youssef Hamadi and Christoph M. Wintersteiger. “Seven challenges in parallel SAT solving”. In: *AI Magazine* 34.2 (2013), pp. 99–106.

[118] Daniel Harabor and Peter J. Stuckey. “Rail capacity modelling with constraint programming”. In: *Integration of AI and OR Techniques in Constraint Programming - 13th Int. Conf., CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proc.* Ed. by Claude-Guy Quimper. Vol. 9676. LNCS. Springer, 2016, pp. 170–186.

[119] William D. Harvey and Matthew L. Ginsberg. “Limited discrepancy search”. In: *Proc. 14th Int. Joint Conf. on Artificial Intelligence, IJCAI 95, Montréal Québec, Canada, Aug. 20-25 1995, 2 Volumes*. Morgan Kaufmann, 1995, pp. 607–615.

[120] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. “Solving and verifying the boolean pythagorean triples problem via cube-and-conquer”. In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th Int. Conf., Bordeaux, France, July 5-8, 2016, Proc.* Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. LNCS. Springer, 2016, pp. 228–245.

[121] Marijn Heule and Armin Biere. “Blocked clause decomposition”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 19th Int. Conf., LPAR-19, Stellenbosch, South Africa, Dec. 14-19, 2013. Proc.* Ed. by Kenneth L. McMillan, Aart Middeldorp, and Andrei Voronkov. Vol. 8312. LNCS. Springer, 2013, pp. 423–438. ISBN: 978-3-642-45220-8.

[122] Marijn Heule, Matti Järvisalo, and Armin Biere. “Clause elimination procedures for CNF formulas”. In: *Logic for Programming, Artificial Intelligence, and Reasoning - 17th Int. Conf., LPAR-17, Yogyakarta, Indonesia, Oct. 10-15, 2010. Proc.* Ed. by Christian G. Fermüller and Andrei Voronkov. Vol. 6397. LNCS. Springer, 2010, pp. 357–371. ISBN: 978-3-642-16241-1.

[123] Marijn Heule, Matti Järvisalo, and Armin Biere. “Efficient CNF simplification based on binary implication graphs”. In: *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th Int. Conf., SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proc.* Ed. by Karem A. Sakallah and Laurent Simon. Vol. 6695. LNCS. Springer, 2011, pp. 201–215. ISBN: 978-3-642-21580-3.

[124] Marijn Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. “Cube and conquer: guiding CDCL SAT solvers by lookaheads”. In: *Hardware and Software: Verification and Testing - 7th Int. Haifa Verification Conf., HVC 2011, Haifa, Israel, Dec. 6-8, 2011, Revised Selected Papers*. Ed. by Kerstin Eder, João Lourenço, and Onn Shehory. Vol. 7261. LNCS. Springer, 2011, pp. 50–65. ISBN: 978-3-642-34187-8.

[125] Marijn Heule and Sean Weaver, eds. *Theory and applications of satisfiability testing - SAT 2015 - 18th int. conf., austin, tx, usa, sept. 24-27, 2015, proc.* Vol. 9340. LNCS. Springer, 2015.

[126] Saïd Jabbour, Jerry Lonlac, Lakhdar Sais, and Yakoub Salhi. "Revisiting the learned clauses database reduction strategies". In: *CoRR* abs/1402.1956 (2014). URL: http://arxiv.org/abs/1402.1956.

[127] Matti Järvisalo and Allen Van Gelder, eds. *Theory and applications of satisfiability testing - SAT 2013 - 16th int. conf., helsinki, finland, july 8-12, 2013. proc.* Vol. 7962. LNCS. Springer, 2013.

[128] Matti Järvisalo, Marijn Heule, and Armin Biere. "Inprocessing rules". In: *Automated Reasoning - 6th Int. Joint Conf., IJCAR 2012, Manchester, UK, June 26-29, 2012. Proc.* Ed. by Bernhard Gramlich, Dale Miller, and Uli Sattler. Vol. 7364. LNCS. Springer, 2012, pp. 355–370. ISBN: 978-3-642-31364-6.

[129] Robert G. Jeroslow and Jinchang Wang. "Solving propositional satisfiability problems". In: *Ann. Math. Artif. Intell.* 1 (1990), pp. 167–187.

[130] Charles Jordan, Lukasz Kaiser, Florian Lonsing, and Martina Seidl. "Mpidepqbf: towards parallel QBF solving without knowledge sharing". In: *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th Int. Conf., Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proc.* Ed. by Carsten Sinz and Uwe Egly. Vol. 8561. LNCS. Springer, 2014, pp. 430–437.

[131] Richard M. Karp. "Reducibility among combinatorial problems". In: *Proc. of a Symp. on the Complexity of Computer Computations, held Mar. 20-22, 1972, at the IBM Thomas J. Watson Research Center, Yorktown Heights, New York.* Ed. by Raymond E. Miller and James W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 85–103.

[132] George Katsirelos and Fahiem Bacchus. "Generalized nogoods in csps". In: *Proc., The 20th Nat. Conf. on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conf., July 9-13, 2005, Pittsburgh, Pennsylvania, USA.* Ed. by Manuela M. Veloso and Subbarao Kambhampati. AAAI Press / The MIT Press, 2005, pp. 390–396.

[133] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. “Resolution and parallelizability: barriers to the efficient parallelization of SAT solvers”. In: *Proc. 27th AAAI Conf. on Artificial Intelligence, July 14-18, 2013, Bellevue, Washington, USA*. Ed. by Marie desJardins and Michael L. Littman. AAAI Press, 2013.

[134] George Katsirelos and Laurent Simon. “Eigenvector centrality in industrial SAT instances”. In: *Principles and Practice of Constraint Programming - 18th Int. Conf., CP 2012, Québec City, QC, Canada, Oct. 8-12, 2012. Proc.* Ed. by Michela Milano. Vol. 7514. LNCS. Springer, 2012, pp. 348–356.

[135] Joris Kinable, Willem-Jan van Hoeve, and Stephen F. Smith. “Optimization models for a real-world snow plow routing problem”. In: *Integration of AI and OR Techniques in Constraint Programming - 13th Int. Conf., CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proc.* Ed. by Claude-Guy Quimper. Vol. 9676. LNCS. Springer, 2016, pp. 229–245.

[136] Donald E. Knuth. *The art of computer programming, volume 2 (3rd ed.): seminumerical algorithms*. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1997.

[137] Donald E. Knuth. *The art of computer programming, volume 3: (2nd ed.) sorting and searching*. Redwood City, CA, USA: Addison Wesley Longman Publishing Co., Inc., 1998.

[138] Donald E. Knuth. *The art of computer programming, volume 4, fascicle 6: satisfiability*. 1st. Addison-Wesley Professional, 2015. ISBN: 0134397606, 9780134397603.

[139] Donald E. Knuth. *The art of computer programming, volume III: sorting and searching*. Addison-Wesley, 1973.

[140] Stephan Kottler and Michael Kaufmann. “Sartagnan-a parallel portfolio sat solver with lockless physical clause sharing”. In: *Pragmatics of SAT* (2011).

[141] Daniel Kroening and Ofer Strichman. *Decision procedures - an algorithmic point of view*. Texts in Theoretical Computer Science. An EATCS Series. Springer, 2008.

[142] Oliver Kullmann. “On a generalization of extended resolution”. In: *Discrete Applied Mathematics* 96-97 (1999), pp. 149–176.

[143] Oliver Kullmann, ed. *Theory and applications of satisfiability testing - SAT 2009, 12th int. conf., SAT 2009, swansea, uk, june 30 - july 3, 2009. proc.* Vol. 5584. LNCS. Springer, 2009.

[144] Manfred Kunde. "Optimal sorting on multi-dimensionally mesh-connected computers". In: *STACS 87, 4th Annu. Symp. on Theoretical Aspects of Computer Science, Passau, Germany, Feb. 19-21, 1987, Proc.* Ed. by Franz-Josef Brandenburg, Guy Vidal-Naquet, and Martin Wirsing. Vol. 247. LNCS. Springer, 1987, pp. 408–419. ISBN: 3-540-17219-X.

[145] Davide Lanti and Norbert Manthey. "Sharing information in parallel search with search space partitioning". In: *Learning and Intelligent Optimization - 7th Int. Conf., LION 7, Catania, Italy, Jan. 7-11, 2013, Revised Selected Papers*. Ed. by Giuseppe Nicosia and Panos M. Pardalos. Vol. 7997. LNCS. Springer, 2013, pp. 52–58. ISBN: 978-3-642-44972-7.

[146] Daniel Larraz, Albert Oliveras, Enric Rodríguez-Carbonell, and Albert Rubio. "Proving termination of imperative programs using max-smt". In: *Formal Methods in Computer-Aided Design, FMCAD 2013, Portland, OR, USA, Oct. 20-23, 2013*. IEEE, 2013, pp. 218–225.

[147] Vladimir I. Levenshtein. *Binary codes capable of correcting deletions, insertions, and reversals*. Tech. rep. 8. 1966, pp. 707–710.

[148] Matthew D. T. Lewis, Tobias Schubert, and Bernd Becker. "Multithreaded SAT solving". In: *Proc. 12th Conf. on Asia South Pacific Design Automation, ASP-DAC 2007, Yokohama, Japan, Jan. 23-26, 2007*. IEEE Computer Society, 2007, pp. 926–931. ISBN: 1-4244-0629-3.

[149] Jia Hui Liang, Vijay Ganesh, Ed Zulkoski, Atulan Zaman, and Krzysztof Czarnecki. "Understanding VSIDS branching heuristics in conflict-driven clause-learning SAT solvers". In: *Hardware and Software: Verification and Testing - 11th Int. Haifa Verification Conf., HVC 2015, Haifa, Israel, Nov. 17-19, 2015, Proc.* Ed. by Nir Piterman. Vol. 9434. LNCS. Springer, 2015, pp. 225–241. ISBN: 978-3-319-26286-4.

[150] Neha Lodha, Sebastian Ordyniak, and Stefan Szeider. "A SAT approach to branchwidth". In: *Theory and Applications of Satisfiability Testing - SAT 2016 - 19th Int. Conf., Bordeaux, France, July 5-8, 2016, Proc.* Ed. by Nadia Creignou and Daniel Le Berre. Vol. 9710. LNCS. Springer, 2016, pp. 179–195.

[151] Inês Lynce and João P. Marques Silva. “Probing-based preprocessing techniques for propositional satisfiability”. In: *15th IEEE Int. Conf. on Tools with Artificial Intelligence (ICTAI 2003), 3-5 Nov. 2003, Sacramento, California, USA*. IEEE Computer Society, 2003, p. 105.

[152] Norbert Manthey. “Coprocessor - a standalone SAT preprocessor”. In: *Applications of Declarative Programming and Knowledge Management - 19th Int. Conf., INAP 2011, and 25th Workshop on Logic Programming, WLP 2011, Vienna, Austria, Sept. 28-30, 2011, Revised Selected Papers*. Ed. by Hans Tompits, Salvador Abreu, Johannes Oetsch, Jörg Pührer, Dietmar Seipel, Masanobu Umeda, and Armin Wolf. Vol. 7773. LNCS. Springer, 2011, pp. 297–304. ISBN: 978-3-642-41523-4.

[153] Norbert Manthey. “Towards next generation sequential and parallel SAT solvers”. PhD thesis. Dresden University of Technology, 2015. URL: http://nbn-resolving.de/urn:nbn:de:bsz:14-qucosa-158672.

[154] Norbert Manthey, Marijn Heule, and Armin Biere. “Automated reencoding of boolean formulas”. In: *Hardware and Software: Verification and Testing - 8th Int. Haifa Verification Int. Conf., HVC 2012, Haifa, Israel, Nov. 6-8, 2012. Revised Selected Papers*. Ed. by Armin Biere, Amir Nahir, and Tanja E. J. Vos. Vol. 7857. LNCS. Springer, 2012, pp. 102–117.

[155] Norbert Manthey, Tobias Philipp, and Christoph Wernhard. “Soundness of inprocessing in clause sharing SAT solvers”. In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th Int. Conf., Helsinki, Finland, July 8-12, 2013. Proc.* Ed. by Matti Järvisalo and Allen Van Gelder. Vol. 7962. LNCS. Springer, 2013, pp. 22–39.

[156] Norbert Manthey and Ari Saptawijaya. “Towards improving the resource usage of SAT-solvers”. In: *Pragmatics of SAT (POS)*. July 2010.

[157] Michela Milano, ed. *Principles and practice of constraint programming - 18th int. conf., CP 2012, québec city, qc, canada, oct. 8-12, 2012. proc.* Vol. 7514. LNCS. Springer, 2012.

[158] Ilya Mironov and Lintao Zhang. “Applications of SAT solvers to cryptanalysis of hash functions”. In: *Theory and Applications of Satisfiability Testing - SAT 2006, 9th Int. Conf., Seattle, WA, USA, Aug. 12-15, 2006, Proc.* Ed. by Armin Biere and Carla P. Gomes. Vol. 4121. LNCS. Springer, 2006, pp. 102–115.

[159] Thierry Moisan, Claude-Guy Quimper, and Jonathan Gaudreault. “Parallel depth-bounded discrepancy search”. In: *Integration of AI and OR Techniques in Constraint Programming - 11th Int. Conf., CPAIOR 2014, Cork, Ireland, May 19-23, 2014. Proc.* Ed. by Helmut Simonis. Vol. 8451. LNCS. Springer, 2014, pp. 377–393.

[160] Andreas Morgenstern and Klaus Schneider. “Synthesis of parallel sorting networks using SAT solvers”. In: *Methoden und Beschreibungssprachen zur Modellierung und Verifikation von Schaltungen und Systemen (MBMV), Oldenburg, Germany, Feb. 21-23, 2011*. Ed. by Frank Oppenheimer. OFFIS-Institut für Informatik, 2011, pp. 71–80.

[161] Thomas Moscibroda and Roger Wattenhofer. “Coloring unstructured radio networks”. In: *Distributed Computing* 21.4 (2008), pp. 271–284.

[162] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. “Chaff: engineering an efficient SAT solver”. In: *Proc. 38th Design Automation Conf., DAC 2001, Las Vegas, NV, USA, June 18-22, 2001*. ACM, 2001, pp. 530–535.

[163] Leonardo Mendonça de Moura and Nikolaj Bjørner. “Z3: an efficient SMT solver”. In: *Tools and Algorithms for the Construction and Analysis of Systems, 14th Int. Int. Conf., TACAS 2008, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2008, Budapest, Hungary, Mar. 29-Apr. 6, 2008. Proc.* Ed. by C. R. Ramakrishnan and Jakob Rehof. Vol. 4963. LNCS. Springer, 2008, pp. 337–340.

[164] René Müller, Jens Teubner, and Gustavo Alonso. “Sorting networks on fpgas”. In: *VLDB J.* 21.1 (2012), pp. 1–23.

[165] Nicholas Nethercote, Peter J. Stuckey, Ralph Becket, Sebastian Brand, Gregory J. Duck, and Guido Tack. “Minizinc: towards a standard CP modelling language”. In: *Principles and Practice of Constraint Programming - CP 2007, 13th Int. Conf., CP 2007, Providence, RI, USA, Sept. 23-27, 2007, Proc.* Ed. by Christian Bessiere. Vol. 4741. LNCS. Springer, 2007, pp. 529–543.

[166] Robert H. B. Netzer and Barton P. Miller. “What are race conditions? some issues and formalizations”. In: *LOPLAS* 1.1 (1992), pp. 74–88.

[167] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. “Solving SAT and SAT modulo theories: from an abstract davis–putnam–logemann–loveland procedure to dpll(T)”. In: *J. ACM* 53.6 (2006), pp. 937–977.

[168] Vegard Nossum. “SAT-based preimage attacks on SHA-1”. MA thesis. Oslo, Norway: University of Osla, 2012.

[169] Chanseok Oh. “Between SAT and UNSAT: the fundamental difference in CDCL SAT”. In: *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th Int. Conf., Austin, TX, USA, Sept. 24-27, 2015, Proc.* Ed. by Marijn Heule and Sean Weaver. Vol. 9340. LNCS. Springer, 2015, pp. 307–323.

[170] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. “Propagation = lazy clause generation”. In: *Principles and Practice of Constraint Programming - CP 2007, 13th Int. Conf., CP 2007, Providence, RI, USA, Sept. 23-27, 2007, Proc.* Ed. by Christian Bessiere. Vol. 4741. LNCS. Springer, 2007, pp. 544–558.

[171] Barry O’Sullivan, ed. *Principles and practice of constraint programming - 20th int. conf., CP 2014, lyon, france, sept. 8-12, 2014. proc.* Vol. 8656. LNCS. Springer, 2014.

[172] Ian Parberry. “A computer assisted optimal depth lower bound for sorting networks with nine inputs”. In: *Proc. 1989 ACM/IEEE Conf. on Supercomputing*. Supercomputing ’89. Reno, Nevada, USA: ACM, 1989, pp. 152–161.

[173] Ian Parberry. “On the computational complexity of optimal sorting network verification”. In: *PARLE ’91: Parallel Architectures and Languages Europe, Volume I: Parallel Architectures and Algorithms, Eindhoven, The Netherlands, June 10-13, 1991, Proc.* Ed. by Emile H. L. Aarts, Jan van Leeuwen, and Martin Rem. Vol. 505. LNCS. Springer, 1991, pp. 252–269.

[174] Ian Parberry. *Parallel complexity theory*. 1987.

[175] Ian Parberry. “The pairwise sorting network”. In: *Parallel Processing Letters* 2 (1992), pp. 205–211.

[176] Mike Paterson. “Improved sorting networks with o(log N) depth”. In: *Algorithmica* 5.1 (1990), pp. 65–92.

[177] Gilles Pesant. “Balancing nursing workload by constraint programming”. In: *Integration of AI and OR Techniques in Constraint Programming - 13th Int. Conf., CPAIOR 2016, Banff, AB, Canada, May 29 - June 1, 2016, Proc.* Ed. by Claude-Guy Quimper. Vol. 9676. LNCS. Springer, 2016, pp. 294–302.

[178] Cédric Piette, Youssef Hamadi, and Lakhdar Sais. “Vivifying propositional clausal formulae”. In: *ECAI 2008 - 18th European Conf. on Artificial Intelligence, Patras, Greece, July 21-25, 2008, Proc.* Ed. by Malik Ghallab, Constantine D. Spyropoulos, Nikos Fakotakis, and Nikolaos M. Avouris. Vol. 178. Frontiers in Artificial Intelligence and Applications. IOS Press, 2008, pp. 525–529. ISBN: 978-1-58603-891-5.

[179] Slawomir Pilarski and Gracia Hu. “Speeding up SAT for EDA”. In: *2002 Design, Automation and Test in Europe Conf. and Expo. (DATE 2002), 4-8 Mar. 2002, Paris, France*. IEEE Computer Society, 2002, p. 1081.

[180] Knot Pipatsrisawat and Adnan Darwiche. “A lightweight component caching scheme for satisfiability solvers”. In: *Theory and Applications of Satisfiability Testing - SAT 2007, 10th Int. Conf., Lisbon, Portugal, May 28-31, 2007, Proc.* Ed. by João Marques-Silva and Karem A. Sakallah. Vol. 4501. LNCS. Springer, 2007, pp. 294–299. ISBN: 978-3-540-72787-3.

[181] Knot Pipatsrisawat and Adnan Darwiche. “On the power of clause-learning SAT solvers as resolution engines”. In: *Artif. Intell.* 175.2 (2011), pp. 512–525.

[182] Claude-Guy Quimper, ed. *Integration of AI and OR techniques in constraint programming - 13th int. conf., CPAIOR 2016, banff, ab, canada, may 29 - june 1, 2016, proc.* Vol. 9676. LNCS. Springer, 2016.

[183] Antonio Ramos, Peter van der Tak, and Marijn Heule. “Between restarts and backjumps”. In: *Theory and Applications of Satisfiability Testing - SAT 2011 - 14th Int. Conf., SAT 2011, Ann Arbor, MI, USA, June 19-22, 2011. Proc.* Ed. by Karem A. Sakallah and Laurent Simon. Vol. 6695. LNCS. Springer, 2011, pp. 216–229. ISBN: 978-3-642-21580-3.

[184] V.Nageshwara Rao and Vipin Kumar. “Superlinear speedup in parallel state-space search”. English. In: *Foundations of Software Technology and Theoretical Computer Science*. Ed. by KesavV. Nori and Sanjeev

Kumar. Vol. 338. LNCS. Springer Berlin Heidelberg, 1988, pp. 161–174. ISBN: 978-3-540-50517-4.

[185] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. "Embarrassingly parallel search". In: *Principles and Practice of Constraint Programming - 19th Int. Conf., CP 2013, Uppsala, Sweden, Sept. 16-20, 2013. Proc.* Ed. by Christian Schulte. Vol. 8124. LNCS. Springer, 2013, pp. 596–610.

[186] Jean-Charles Régin, Mohamed Rezgui, and Arnaud Malapert. "Improvement of the embarrassingly parallel search for data centers". In: *Principles and Practice of Constraint Programming - 20th Int. Conf., CP 2014, Lyon, France, Sept. 8-12, 2014. Proc.* Ed. by Barry O'Sullivan. Vol. 8656. LNCS. Springer, 2014, pp. 622–635.

[187] John Alan Robinson. "A machine-oriented logic based on the resolution principle". In: *J. ACM* 12.1 (1965), pp. 23–41.

[188] Francesca Rossi, Peter van Beek, and Toby Walsh. *Handbook of constraint programming (foundations of artificial intelligence)*. New York, NY, USA: Elsevier Science Inc., 2006.

[189] Ulf Rüegg, Thorsten Ehlers, Miro Spönemann, and Reinhard von Hanxleden. "A generalization of the directed graph layering problem". In: *Graph Drawing and Network Visualization - 24th Int. Symp., GD 2016, Athens, Greece, Sept. 19-21, 2016, Revised Selected Papers*. Ed. by Yifan Hu and Martin Nöllenburg. Vol. 9801. LNCS. Springer, 2016, pp. 196–208. ISBN: 978-3-319-50105-5.

[190] Vadim Ryvchin and Ofer Strichman. "Local restarts". In: *Theory and Applications of Satisfiability Testing - SAT 2008, 11th Int. Conf., SAT 2008, Guangzhou, China, May 12-15, 2008. Proc.* Ed. by Hans Kleine Büning and Xishun Zhao. Vol. 4996. LNCS. Springer, 2008, pp. 271–276.

[191] Karem A. Sakallah and Laurent Simon, eds. *Theory and applications of satisfiability testing - SAT 2011 - 14th int. conf., SAT 2011, ann arbor, mi, usa, june 19-22, 2011. proc.* Vol. 6695. LNCS. Springer, 2011. ISBN: 978-3-642-21580-3.

[192] T. Schrijvers, G. Tack, P. Wuille, H. Samulowitz, and P.J. Stuckey. "Search combinators". In: *Constraints* 18.2 (2013), pp. 269–305.

[193] Robert Sedgewick and Michael Schidlowsky. *Algorithms in java, third edition, parts 1-4: fundamentals, data structures, sorting, searching*. 3rd. Boston, MA, USA: Addison-Wesley Longman Publishing Co., Inc., 1998. ISBN: 0201361205.

[194] Dennis Sen. "SAT-Solven auf verteilten Systemen". MA thesis. Kiel, Germany: Kiel University, 2014.

[195] Claude. E. Shannon. "The synthesis of two-terminal switching circuits". In: *Bell System Technical Journal* 28.1 (1949), pp. 59–98. ISSN: 1538-7305.

[196] João P. Marques Silva and Karem A. Sakallah. "GRASP: A search algorithm for propositional satisfiability". In: *IEEE Trans. Computers* 48.5 (1999), pp. 506–521.

[197] Laurent Simon. "Post mortem analysis of SAT solver proofs". In: *POS-14. 5th Pragmatics of SAT workshop, workshop SAT 2014 Conf., part of FLoC 2014 during the Vienna Summer of Logic, July 13, 2014, Vienna, Austria*. Ed. by Daniel Le Berre. Vol. 27. EPiC Series in Computing. EasyChair, 2014, pp. 26–40.

[198] Carsten Sinz. "Towards an optimal CNF encoding of boolean cardinality constraints". In: *Principles and Practice of Constraint Programming - CP 2005, 11th Int. Conf., CP 2005, Sitges, Spain, Oct. 1-5, 2005, Proc.* Ed. by Peter van Beek. Vol. 3709. LNCS. Springer, 2005, pp. 827–831.

[199] Carsten Sinz and Uwe Egly, eds. *Theory and applications of satisfiability testing - SAT 2014 - 17th int. conf., held as part of the vienna summer of logic, VSL 2014, vienna, austria, july 14-17, 2014. proc.* Vol. 8561. LNCS. Springer, 2014.

[200] Valery Sklyarov and Iouliia Skliarova. "High-performance implementation of regular and easily scalable sorting networks on an FPGA". In: *Microprocessors and Microsystems - Embedded Hardware Design* 38.5 (2014), pp. 470–484.

[201] Mate Soos, Karsten Nohl, and Claude Castelluccia. "Extending SAT solvers to cryptographic problems". In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th Int. Conf., SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proc.* Ed. by Oliver Kullmann. Vol. 5584. LNCS. Springer, 2009, pp. 244–257.

[202] Niklas Sörensson and Armin Biere. “Minimizing learned clauses”. In: *Theory and Applications of Satisfiability Testing - SAT 2009, 12th Int. Conf., SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proc.* Ed. by Oliver Kullmann. Vol. 5584. LNCS. Springer, 2009, pp. 237–243.

[203] Peter J. Stuckey. “There are no CNF problems”. In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th Int. Conf., Helsinki, Finland, July 8-12, 2013. Proc.* Ed. by Matti Järvisalo and Allen Van Gelder. Vol. 7962. LNCS. Springer, 2013, pp. 19–21.

[204] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. “Niver: non increasing variable elimination resolution for preprocessing SAT instances”. In: *SAT 2004 - The 7th Int. Int. Conf. on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proc.* 2004.

[205] G. S. Tseitin. “Automation of reasoning: 2: classical papers on computational logic 1967–1970”. In: ed. by Jörg H. Siekmann and Graham Wrightson. Springer Berlin Heidelberg, 1983. Chap. On the Complexity of Derivation in Propositional Calculus, pp. 466–483.

[206] Olga Tveretina. “A conditional superpolynomial lower bound for extended resolution”. In: *Language and Automata Theory and Applications - 7th Int. Int. Conf., LATA 2013, Bilbao, Spain, Apr. 2-5, 2013. Proc.* Ed. by Adrian Horia Dediu, Carlos Martín-Vide, and Bianca Truthe. Vol. 7810. LNCS. Springer, 2013, pp. 559–569.

[207] Diego de Uña, Graeme Gange, Peter Schachte, and Peter J. Stuckey. “Steiner tree problems with side constraints using constraint programming”. In: *Proc. 30th AAAI Conf. on Artificial Intelligence, Feb. 12-17, 2016, Phoenix, Arizona, USA.* Ed. by Dale Schuurmans and Michael P. Wellman. AAAI Press, 2016, pp. 3383–3389.

[208] Alasdair Urquhart. “Regular and general resolution: an improved separation”. In: *Theory and Applications of Satisfiability Testing - SAT 2008, 11th Int. Conf., SAT 2008, Guangzhou, China, May 12-15, 2008. Proc.* Ed. by Hans Kleine Büning and Xishun Zhao. Vol. 4996. LNCS. Springer, 2008, pp. 277–290.

[209] Alasdair Urquhart. “The complexity of propositional proofs”. In: *Bulletin of the EATCS* 64 (1998).

[210] Alasdair Urquhart. “The depth of resolution proofs”. In: *Studia Logica* 99.1-3 (2011), pp. 349–364.

[211] Vinod K. Valsalam and Risto Miikkulainen. “Using symmetry and evolutionary search to minimize sorting networks”. In: *Journal of Machine Learning Research* 14.1 (2013), pp. 303–331.

[212] D.C. van Voorhis. “Toward a lower bound for sorting networks”. In: *Complexity of Computer Computations*. Ed. by R.E. Miller and J.W. Thatcher. The IBM Research Symposia Series. Plenum Press, New York, 1972, pp. 119–129.

[213] Siert Wieringa and Keijo Heljanko. “Concurrent clause strengthening”. In: *Theory and Applications of Satisfiability Testing - SAT 2013 - 16th Int. Conf., Helsinki, Finland, July 8-12, 2013. Proc.* Ed. by Matti Järvisalo and Allen Van Gelder. Vol. 7962. LNCS. Springer, 2013, pp. 116–132.

[214] Christoph M. Wintersteiger, Youssef Hamadi, and Leonardo Mendonça de Moura. “A concurrent portfolio approach to SMT solving”. In: *Computer Aided Verification, 21st Int. Conf., CAV 2009, Grenoble, France, June 26 - July 2, 2009. Proc.* Ed. by Ahmed Bouajjani and Oded Maler. Vol. 5643. LNCS. Springer, 2009, pp. 715–720.

[215] Hantao Zhang, Maria Paola Bonacina, and Jieh Hsiang. “PSATO: a distributed propositional prover and its application to quasigroup problems”. In: *J. Symb. Comput.* 21.4 (1996), pp. 543–560.

[216] Lintao Zhang, Conor F. Madigan, Matthew W. Moskewicz, and Sharad Malik. “Efficient conflict driven learning in boolean satisfiability solver”. In: *Proc. 2001 IEEE/ACM Int. Conf. on Computer-Aided Design, ICCAD 2001, San Jose, CA, USA, Nov. 4-8, 2001*. Ed. by Rolf Ernst. IEEE Computer Society, 2001, pp. 279–285. ISBN: 0-7803-7249-2.