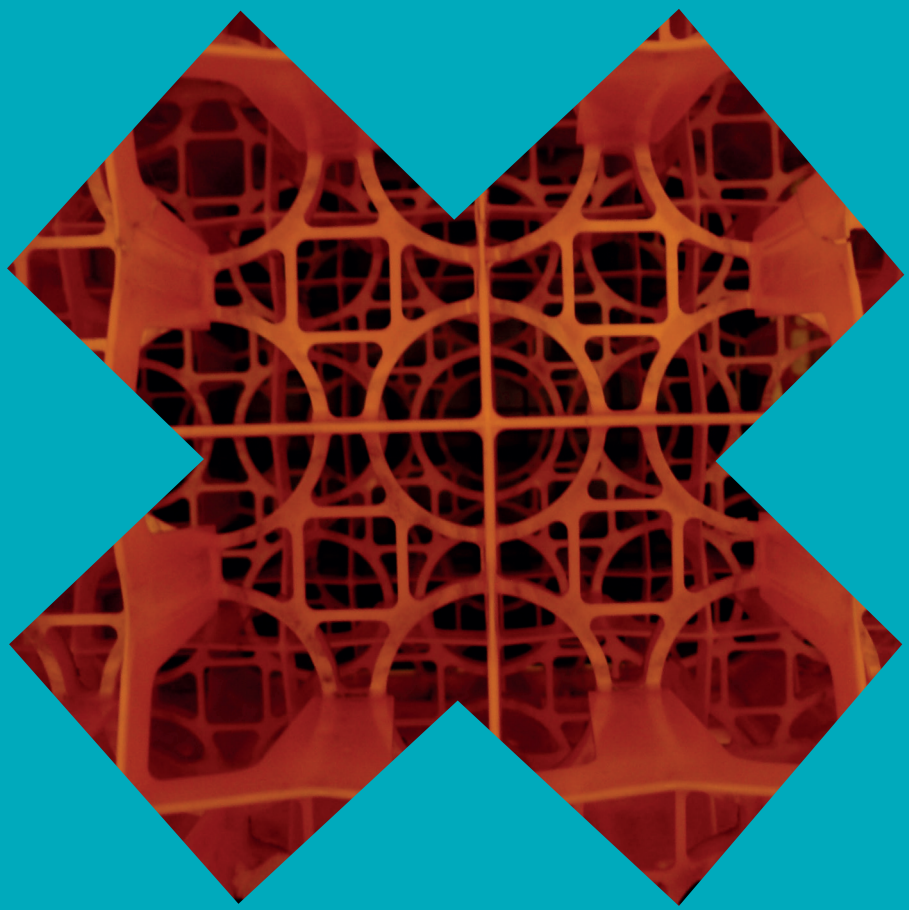
Department of Information and Computer Science

Grid Based Propositional Satisfiability Solving

Antti E. J. Hyvärinen



Aalto University

DOCTORAL DISSERTATIONS

Aalto University publication series
DOCTORAL DISSERTATIONS 118/2011

Grid Based Propositional Satisfiability Solving

Antti E. J. Hyvärinen

Doctoral dissertation for the degree of Doctor of Science in Technology to be presented with due permission of the School of Science for public examination and debate in Auditorium T2 at the Aalto University School of Science (Espoo, Finland) on the 28th of November 2011 at noon.

Aalto University
School of Science
Department of Information and Computer Science

Supervisor
Professor Ilkka Niemelä

Instructor
Doctor Tommi Junttila

Preliminary examiners
Professor Bernd Becker, Albert-Ludwigs-University Freiburg, Germany
Professor Lakhdar Saïs, Université Lille Nord de France, France

Opponent
Professor Toby Walsh, National ICT Australia and University of New South Wales, Australia

Aalto University publication series
DOCTORAL DISSERTATIONS 118/2011

ISBN 978-952-60-4368-5 (pdf)
ISBN 978-952-60-4367-8 (printed)
ISSN-L 1799-4934
ISSN 1799-4942 (pdf)
ISSN 1799-4934 (printed)

Aalto Print
Helsinki 2011

Finland

The dissertation can be read at http://lib.tkk.fi/Diss/

441 697
Printed matter

Aalto University

Abstract

Aalto University, P.O. Box 11000, FI-00076 Aalto www.aalto.fi

Author
Antti E. J. Hyvärinen

Name of the doctoral dissertation
Grid Based Propositional Satisfiability Solving

Publisher School of Science

Unit Department of Information and Computer Science

Series Aalto University publication series DOCTORAL DISSERTATIONS 118/2011

Field of research Theoretical Computer Science

Manuscript submitted 14 June 2011 **Manuscript revised** 1 September 2011

Date of the defence 28 November 2011 **Language** English

☐ **Monograph** ☒ **Article dissertation (summary + original articles)**

Abstract

This work studies how grid and cloud computing can be applied to efficiently solving propositional satisfiability problem (SAT) instances. Propositional logic provides a convenient language for expressing real-world originated problems such as AI planning, automated test pattern generation, bounded model checking and cryptanalysis. The interest in SAT solving has increased mainly due to improvements in the solving algorithms, which recently have increasingly focused on using parallelism offered by multi-CPU computers. Partly orthogonally to these improvements this work studies several novel approaches to parallel solving of SAT instances in a grid of widely distributed "virtual" computers instead of workstations or supercomputers.

Two types of parallel SAT solving approaches are analyzed and used as building blocks for more complex systems: using several solvers which compete to solve a given instance in parallel, and splitting the search space of the instance and solving the resulting partitions in parallel. The work presents several efficient partitioning functions, critical in successful splitting according to an analytical result, and presents novel solving systems that are less dependent on the partitioning function efficiency. Finally, the work studies combining clause learning, a key technique in modern SAT solvers, with the novel types of parallel solvers. Different heuristics are studied for filtering clauses learned in parallel, and the work proposes techniques which allow exchanging the clauses between different splits.

The practical significance of the results are studied using large, standard benchmark sets from SAT competitions. Some of the approaches are able to solve several instances that have either not been solved at all by any other solver, or which are significantly slower to solve with other solvers.

Keywords Constraint Based Search, Randomized Search, SAT, DPLL procedure, Clause Learning, Parallel Computing, Cloud Computing, Grid Computing

ISBN (printed) 978-952-60-4367-8 **ISBN (pdf)** 978-952-60-4368-5

ISSN-L 1799-4934 **ISSN (printed)** 1799-4934 **ISSN (pdf)** 1799-4942

Location of publisher Espoo **Location of printing** Helsinki **Year** 2011

Pages 207 **The dissertation can be read at** http://lib.tkk.fi/Diss/

A? Aalto-yliopisto

Tiivistelmä

Aalto-yliopisto, PL 11000, 00076 Aalto www.aalto.fi

Tekijä
Antti E. J. Hyvärinen

Väitöskirjan nimi
Pilvilaskentapohjainen lauselogiikan toteutuvuustarkastus

Julkaisija Perustieteiden Korkeakoulu

Yksikkö Tietojenkäsittelytieteen laitos

Sarja Aalto University publication series DOCTORAL DISSERTATIONS 118/2011

Tutkimusala Tietojenkäsittelyteoria

Käsikirjoituksen pvm 14.06.2011 **Korjatun käsikirjoituksen pvm** 01.09.2011

Väitöspäivä 28.11.2011 **Kieli** Englanti

☐ **Monografia** ☒ **Yhdistelmäväitöskirja (yhteenveto-osa + erillisartikkelit)**

Tiivistelmä

Tutkin väitöskirjassani lauselogiikan toteutuvuusongelmana kuvattujen, rakenteisten ongelmien ratkaisemista pilvilaskentaympäristössä. Lauselogiikka on luonteva kuvauskieli useisiin käytännön tarpeista nouseviin ongelmiin, kuten tekoälysuunnitteluun, automaattiseen testihahmojen luomiseen piirisuunnittelussa, rajoitettuun mallintarkastukseen ja kryptoanalyysiin. Lauselogiikan käytännön merkitys on kasvanut ratkaisualgoritmien kehittyessä, ja viimeaikoina erityisesti rinnakkaisuuden hyödyntäminen on noussut keskeiseksi tutkimuskysymykseksi. Uutena näkökulmana työssä tutkitaan useita lähestymistapoja toteutuvuustarkastamiseen, kun laskentaympäristö koostuu laajasti hajautetusta joukosta "virtuaalisia" tietokoneita työasemien tai supertietokoneiden sijaan.

Tutkittavat ratkaisumenetelmät pohjaavat kahteen peruskäsitteeseen: ratkaisijoiden kilpailuttamiseen rinnakkain ja hakuavaruuden jakamiseen. Analyysin mukaan hyvien osa-avaruuksien tuottaminen on keskeistä, jotta jälkimmäinen menetelmä kaikissa tapauksissa nopeuttaa ratkaisua. Työssä tutkitaan hyviä osa-avaruuksia tuottavia heuristiikkoja, sekä useita uudenlaisia tapoja osa-avaruuksien läpikäyntiin siten, että menetelmien tehokkuus ei riipu voimakkaasti heuristiikoista. Menetelmät yhdistetään lauselogiikan toteutuvuustarkastimissa keskeiseen klausuulioppimiseen tutkimalla erilaisia klausuulien suodatusmenetelmiä ja tehokkaita klausuulien siirtotapoja hakuavaruuksien välillä.

Työssä tutkitaan myös esitettyjen menetelmien käytännön merkitystä laajalti käytössä olevien, toteutuvuustarkastinkilpailujen testiongelmien avulla. Menetelmillä pystyttiin ratkaisemaan eräitä tunnettuja testiongelmia ensimmäistä kertaa, ja muita tällaisia ongelmia huomattavasti nopeammin kuin millään aiemmin tunnetulla menetelmällä.

Avainsanat Rajoitepohjainen haku, satunnaistettu haku, Lauselogiikan toteutuvuusongelma, DPLL-algoritmi, klausuulioppiminen, rinnakkaislaskenta, pilvilaskenta, grid-laskenta

ISBN (painettu) 978-952-60-4367-8 **ISBN (pdf)** 978-952-60-4368-5

ISSN-L 1799-4934 **ISSN (painettu)** 1799-4934 **ISSN (pdf)** 1799-4942

Julkaisupaikka Espoo **Painopaikka** Helsinki **Vuosi** 2011

Sivumäärä 207 **Luettavissa verkossa osoitteessa** http://lib.tkk.fi/Diss/

Contents

Preface

This thesis is the result of the research I have conducted in the Computational Logic Group led by Professor Ilkka Niemelä.

Professor Bernd Becker (Albert-Ludwigs-University Freiburg) and Professor Lakhdar Saïs (Université Lille Nord de France) have reviewed the thesis. I express my gratitude to them for the time invested in giving insightful and constructive analysis on the manuscript, and Prof. Becker for the suggestions that helped me further improve the presentation. Professor Ilkka Niemelä and Doctor Tommi Junttila, my thesis supervisor and instructor, respectively, have been invaluable in sharing their expertise, giving good advice at the numerous occasions I have needed it in varying subjects, be they scientific, practical, or related to something completely different. I am grateful to them for the opportunity of working around a fascinating subject with a clear goal.

Some of the experiments of this thesis were run on the resources provided by the NorduGrid Collaboration and the Finnish Material Science Grid M-grid. I would like to thank the people involved in these projects, in particular the Grid Computing Group at the Finnish CSC, and the people involved in the Advanced Resource Connector (ARC) related projects.

I have received funding for the research from the Department of Computer Science, Hecse Graduate school and the Academy of Finland. I am also grateful for the personal grants from the Jenny and Antti Wihuri Foundation, the Emil Aaltonen Foundation, the Finnish Foundation for Technology Promotion, and the Nokia Foundation.

I have enjoyed immensely the working environment in and around the Computational Logic Group. My colleagues have been helpful, supportive, and good company. I would in particular like to thank Jori Dubrovin who not only helped me in the analytics at a critical moment, but also made me swim while we were simultaneously going through the same hectic

period of preparing our thesis manuscripts.

The cover photograph is taken by Heikki Hiltunen in the men's room of a former Kallio based bar Oluthuone Poirotti in February 28, 2009 at 9.23 pm. The photo is an artist's view of a computing grid, and illustrates a composition of plastic beer carriers forming interesting geometric patterns. I think it captures something essential from SAT solving, where the trivial and mundane suddenly becomes strange and fascinating once the right perspective is found.

Finally I want to thank my friends, who have helped me stay in touch with music as well as life in general, both those here on my home grounds but also the friends I made in Cambridge, UK while working for Microsoft Research in Summer 2011. I have a growing and supportive family, an asset which I greatly respect. Most importantly, I am fortunate to be married to Hissu who has been with me in this project from the Day One.

Espoo, October 27, 2011

Antti Hyvärinen

List of Publications

This thesis consists of an overview and of the following publications which are referred to in the text by their Roman numerals.

I Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Strategies for Solving SAT in Grids by Randomized Search. In *Proceedings of the 9th International Conference on Artificial Intelligence and Symbolic Computation (AISC 2008)*, volume 5144 of Lecture Notes in Artificial Intelligence, pages 125–140, July/August 2008.

II Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Incorporating Clause Learning in Grid-Based Randomized SAT Solving. *Journal on Satisfiability, Boolean Modeling and Computation*, volume 6, number 4, pages 223–244, June 2009.

III Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning Search Spaces of a Randomized Search. *Fundamenta Informaticae*, volume 107, Number 2–3, pages 289–311, September 2011.

IV Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Partitioning SAT Instances for Distributed Solving. In *Proceedings of the 17th international conference of Logic for Programming, Artificial Intelligence, and Reasoning (LPAR-17)*, volume 6397 of *Lecture Notes in Computer Science*, pages 372–386, October 2010.

V Antti E. J. Hyvärinen, Tommi Junttila, and Ilkka Niemelä. Grid-Based SAT Solving with Iterative Partitioning and Clause Learning. In *Proceedings of the 17th International Conference on Principles and Practice*

of Constraint Programming (CP 2011), volume 6876 of *Lecture Notes in Computer Science*, pages 385–399, September 2011.

Author's Contribution

Publication I: "Strategies for Solving SAT in Grids by Randomized Search"

The author is responsible for the description of the computing model, the two schemes described in the work, and conducting and documenting the experiments. The approach for solving several SAT instances is due to Dr. Junttila, and the general idea of using parallel restart strategies in this context is due to Dr. Junttila and Prof. Niemelä.

Publication II: "Incorporating Clause Learning in Grid-Based Randomized SAT Solving"

The author has implemented the framework, the simplification, the different heuristics and the modifications required to obtain the clauses from a SAT solver. The length-based heuristics are folklore, while the frequency-based heuristic is, in this context, by the authors. The author has conducted and reported the experiments.

Publication III: "Partitioning Search Spaces of a Randomized Search"

The terms *simple distribution* and *plain partitioning* are by Niemelä and Junttila, whereas the author has coined the terms *safe and repeated partitioning*. Most of the proofs in the work are by the author and Junttila. The implementations, the experiments, and their documentations are by the author.

Publication IV: "Partitioning SAT Instances for Distributed Solving"

The *partition tree* concept is by the authors, and the proof for non-increasing run time is by the author. The two partitioning function implementations based on DPLL lookahead and scattering VSIDS are by Junttila as is the idea for speeding up lookahead computation. The author has implemented the scattering lookahead partitioning function and the general framework. The experiments on lookahead techniques were conducted and documented by Junttila, while the rest of the experiments were conducted by the author.

Publication V: "Grid-Based SAT Solving with Iterative Partitioning and Clause Learning"

The idea of transferring learned clauses between arbitrary nodes of a partition tree is by Junttila, and the description of the learning partition tree is by Junttila and the author. The partitioning functions, the modifications required to the solver to obtain the learned clauses, and the two clause tagging approaches are implemented by Junttila. The idea of the assumption-based clause tagging is by Junttila whereas the flag-based clause tagging is by the author in this context. The simplifications and the framework are by the author, as are the documenting and performing the experiments.

Contributions of the Publications

The work described in this thesis focuses on different approaches to using grid and cloud computing in efficient parallel solving of propositional satisfiability problem (SAT) instances.

Publication I

The work specifies the grid / cloud computing model considered in the thesis. The work explores using portfolios to obtain speed-up in a realistic grid environment. Two techniques for parallelizing restart strategies [Luby et al. 1993; Walsh 1999], essential also for efficient sequential SAT solvers, are introduced. The resulting parallel algorithms are first studied in a controlled model with varying parameters and then experimented with in a real grid environment. Finally the work proposes an efficient algorithm for distributed solving of several SAT instances.

Publication II

The work extends the portfolio approach presented in [PI] by introducing parallel conflict-driven clause learning to it. One of the challenges in the resulting *clause learning simple distributed SAT solving* (CL-SDSAT) approach is that it produces enormous amounts of learned clauses. To solve this problem, several clause filtering heuristics are studied in controlled experiments with a number of benchmark instances. A length-based heuristic is then used to study in controlled experiments the "cumulative" aspect of parallel learning, where new clauses are learned based on earlier clauses learned in parallel. The full framework is implemented using a real grid environment and succeeds in solving several SAT in-

stances which were not solved in a solver competition. An earlier version of the work was published in the proceedings of AIMSA 2008 [Hyvärinen et al. 2008].

Publication III

The work compares analytically the portfolio and the *search space splitting* approaches to obtaining speed-up in distributed SAT solving. Search space splitting is studied using a model for worst and best case *partitioning functions*. The analysis shows that the efficiency of the partitioning approach depends heavily on whether the instance is satisfiable with many solutions, "barely satisfiable" with few solutions, or unsatisfiable. The portfolio and search space splitting are combined to *safe* and *repeated partitioning* approaches. A run time distribution analysis shows that the safe approach is superior to the repeated approach. Finally the studied approaches are compared experimentally using an actual implementation of a partitioning function. In this setting repeated partitioning often performs better than the other approaches. An earlier version of the work was published in AI*IA 2009 [Hyvärinen et al. 2009].

Publication IV

The work develops further an iterative partitioning approach based on *partition trees*, first described in [Hyvärinen et al. 2006]. The partition tree approach uses a partitioning function to iteratively construct a tree of increasingly constrained *derived formulas*. The satisfiability of the original formula can be determined once a sufficient number of derived formulas have been solved. We prove that unlike in plain partitioning, the expected time required to determine the satisfiability of a formula can never increase if more parallelism is used in the partition tree approach. The work introduces two new partitioning functions based on *unit propagation lookahead*, and compares them against a previously introduced function using *scattering* [Hyvärinen et al. 2006]. A novel approach for speeding up the computation of the lookahead is described with a technique originating from conflict driven clause learning. The partition tree approach is implemented in a real grid environment and used to solve

several instances that were not solved by state-of-the-art SAT solvers.

Publication V

The work describes a framework which combines clause learning to the partition tree approach to allow the transferral of intermediary results from derived formulas that either were shown unsatisfiable or failed to be solved due to a resource exhaustion in the grid. The framework attaches learned clause databases to the nodes of the tree and specifies the simplification process which allows restricting the sizes of the databases. The work presents two techniques for transferring clauses learned in the nodes of the trees. The techniques are based on tagging learned clauses with the information on which constraints they depend on using either Boolean *flags* or more general *assumptions*. The flag based tagging is less expensive but restricts the clause transferral more compared to the assumption based tagging. The effect of the learned clauses in both tagging approaches are analyzed in a controlled computing environment and an implementation of the approach is used to solve challenging instances from a SAT competition. The results show that the flag based tagging provides speed-up in both controlled experiments and a real implementation, the resulting solver succeeding in solving many instances beyond the reach of modern SAT solvers.

1. Introduction

This work develops methods for solving instances of the *propositional satisfiability problem* (SAT), which concerns deciding whether a given propositional formula over a set of Boolean variables evaluates to true for some true/false assignment on the variables. As the platform for solving SAT instances, this work considers *computational grid* environments consisting of high performance computing clusters connected to the Internet.

In recent years, SAT has experienced an increase of interest from the industry and other organizations outside the academia as computationally challenging problems arise from fields such as planning [Kautz and Selman 1992], automated test pattern generation [Larrabee 1992; Biere and Kunz 2002], cryptanalysis [Mironov and Zhang 2006] and bounded model checking [Marques-Silva 2008].

The results of this thesis suggest that often, when the solving of such problems requires large amounts of computing power, the solution can be efficiently computed in distributed environments. In particular, the solving can be organized to independent computations, *jobs*, that are dynamically scheduled to a grid. The results have a practical impact, since currently grids and clouds seem to provide a particularly appealing computing paradigm supported by technological advances, environmental concerns, social well-being and economic aspects. Firstly, the worldwide investments on high quality, low energy communication infrastructure make high-volume data transfers inexpensive, and virtualization techniques by manufacturers of of-the-shelf computing nodes allows provisioning of operating environments which are suitable for a large variety of consumers. Wide scale distributed computing is also fault tolerant by nature. Secondly, the energy-hungry computing can be dynamically transferred to sites using renewable energy sources such as wind and solar power, or to locations where synergy can be obtained, for example, with heating.

Thirdly, in some cases data centers can be renovated to already existing buildings creating a more diverse ecosystem of entrepreneurship to rural areas previously relying strongly on heavy industries. Finally, the change is driven by the new business layer providing administration and hardware at a relatively low cost to the consumers of computing power.

1.1 The SAT Problem

The SAT problem is a representative of the class of NP-complete decision problems [Cook 1971], for which all known algorithms need in the worst case exponential number of steps with respect to the size of the problem instance. If one of the NP-complete problems has such a polynomial time solving procedure, then the same procedure could be used for every problem in NP. The proof of (non-)existence of such a procedure is one of the six currently open Millennium Prize Problems of the Clay Mathematics Institute. Many engineering problems can be expressed naturally as SAT instances, and can then be efficiently solved by *SAT solvers*, pieces of software sometimes capable of solving non-trivial formulas consisting of more than a million variables. Since the first implementations of SAT solvers [Davis and Putnam 1960; Davis et al. 1962], originally designed for first-order theorem proving, SAT solving has experienced tremendous enhancements in algorithm design, and recent solvers, such as [Moskewicz et al. 2001; Eén and Sörensson 2004], represent in many ways the state-of-the-art in solving instances of NP-complete problems. The paradigm of encoding a given problem of an application domain to, e.g., SAT is often called *declarative problem solving*. In this paradigm the programmer is relieved from the algorithm design, a task already performed in building the solver. Instead, the emphasis is on how to build the encoding. Figure 1.1 illustrates the process with a schematic diagram where the problem instance flows to the direction of the solid arrows and the results are indicated by the dashed arrows. The topic of this work is in studying different approaches for distributing the solver.

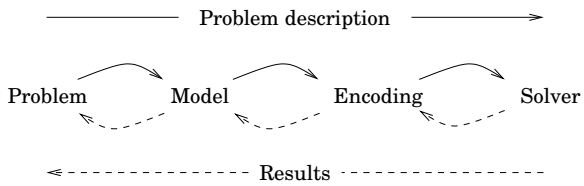


Figure 1.1. Declarative Problem Solving

1.2 Related Approaches

Possibly the most famous research project related to declarative problem solving is the Fifth Generation Project [Shapiro et al. 1993] which aimed at using massive amounts of parallel resources for efficient computing in artificial intelligence. The project built on ideas developed for PROLOG systems [Sterling and Shapiro 1987]. Compared to SAT, PROLOG systems have complex procedural semantics, which renders for example their parallelization more difficult [Ranjan et al. 1999]. The simplicity makes SAT problem description also more suitable for automating the transition from model to encoding in Fig. 1.1.

The constraint satisfaction problem (CSP) [Rossi et al. 2006] is often viewed as a generalization of SAT offering a wider spectrum of variable constraints and thereby a richer modeling language. Interaction between developing SAT and CSP solving techniques is intense. For example, once the *conflict driven clause learning* techniques [Marques-Silva and Sakallah 1999] originating from CSP [Dechter 1990] proved extremely successful in SAT, they have again received interest in CSP solver developers, and when implemented with care, can provide significant speed-up [Gent et al. 2010]. Often the constraints are expressible fairly compactly as SAT instances. Indeed, [Walsh 2000] and [Huang 2008] argue that SAT encodings of some constraints are more compact and efficient to solve. General differences of SAT and CSP are studied, for example, in [Bordeaux et al. 2006] and the efficiencies in [Mancini et al. 2008].

An alternative to programming using PROLOG systems is *Answer set programming* [Niemelä 1999] (ASP), a logic programming paradigm also closely related to CSP, which uses the stable model semantics [Gelfond and Lifschitz 1988] as its basis. The paradigm has an established track record in planning [Dimopoulos et al. 1997], product configuration [Soininen and Niemelä 1999], formal verification [Heljanko 1999], and even biology [Erdem and Türe 2008], among others. In part, the success of the paradigm is due to several highly optimized implementations [Simons

et al. 2002; Leone et al. 2006; Drescher et al. 2008]. Stable model semantics are closely related to SAT [Ben-Eliyahu and Dechter 1994; Lin and Zhao 2004; Janhunen 2006]. Recent experimental evaluations, such as [Gebser et al. 2007; Mancini et al. 2008], suggest that in some cases of practical relevance the machinery developed for SAT solvers is valuable in finding stable models of ASP programs.

Both SAT and ASP place rather strict limits on what application domains can be efficiently described. While in theory any polynomial-time algorithm can be encoded as a SAT instance using the construction of Cook, the straightforward process is hopelessly inefficient, for example, in case one has to represent integers or floating-point arithmetics. The relatively new approach of *satisfiability modulo theories* (SMT) [Ganzinger et al. 2004; Bozzano et al. 2005; Nieuwenhuis et al. 2006; Sebastiani 2007; de Moura and Bjørner 2008], combines the successful SAT solving with methods specifically designed for expressing domain-specific information. The SMT solvers work on encodings where the propositional part is augmented with a theory T which embeds the special features of the problem being modeled. The additional theory is expressed in a form where theory-specific algorithms can be used in addition to the powerful algorithms for propositional satisfiability. Assume, for example, that one has to model a scheduling problem with time represented as continuous values. This is often inefficient to express as a SAT problem. However, if the encoding is based on SMT, then the part of the domain considering integers can be encoded as a linear arithmetics theory T over reals while the propositional part is still efficiently solvable using algorithmic ideas that have proved useful in propositional theories. It is an interesting research question to what extent the results in this work are useful also in designing distributed SMT solvers.

SAT solvers can be roughly divided into two categories: *local search* solvers based on random walk or similar incomplete methods [Selman and Kautz 1993], and complete solvers usually based on *backtracking search*, such as those based on the Davis-Putnam-Logemann-Loveland (DPLL) [Davis et al. 1962; Davis and Putnam 1960] algorithm. Outside of this categorization lie methods based on *knowledge compilation*, such as *binary decision diagrams* [Bryant 1986], and the more recently introduced *decomposable negation normal form* [Darwiche 2001]. The focus of this work is on using the complete backtracking solvers in distributed environments. Unlike the local search methods, DPLL solvers are able

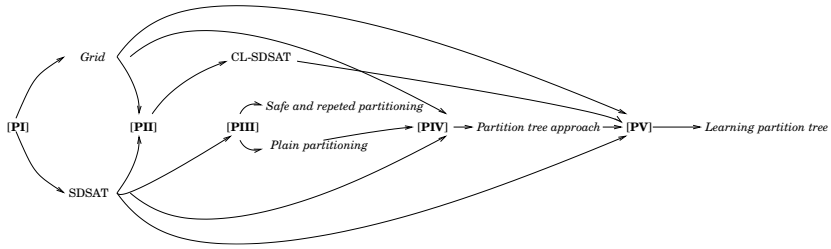


Figure 1.2. Organization of the results

to prove unsatisfiability, and are less prone to exponential memory consumption sometimes observed in methods based on knowledge compilation. Compared to the local search methods, the DPLL solvers also usually perform significantly better on industrial SAT instances.

1.3 Contributions

This work develops distributed SAT solving approaches for computing grids. The goal of the work is to solve extremely challenging instances using environments where simultaneous computations can only communicate with a single master process and have tight resource limits. The challenge is addressed by starting from rigorous study of intuitive solving approaches, and building increasingly complex approaches so that the new designs are driven by the previous results. The practical relevance of the results is established with experimental and analytical comparisons.

The work describes seven approaches for distributed SAT solving. While each can be used for solving as such, they can also be seen as a hierarchy of increasingly powerful and complex solving approaches. The approaches are

- simple distributed SAT solving (SDSAT),
- clause learning simple distributed SAT solving (CL-SDSAT),
- the plain partitioning approach,
- the safe partitioning approach,
- the repeated partitioning approach,
- the partition tree approach, and
- the learning partition tree approach.

Figure 1.2 illustrates the relations of the solving approaches and grid computing, and points also to the publications reporting them. The SDSAT approach is studied with the grid computing model in [PI] and experimen-

tally shown to be a solid way of solving challenging SAT instances. The SDSAT approach is extended to CL-SDSAT in [PII], which solves several highly challenging SAT instances beyond the reach of SDSAT. The plain, safe and repeated partitioning approaches are studied and compared against the SDSAT approach and each other in [PIII]. The idea of SDSAT is combined to plain partitioning in [PIV]. The resulting partition tree approach solves several instances that could not be solved by any other solver in the experiments. Finally, [PV] combines the ideas developed in CL-SDSAT with the partition tree approach, and the resulting learning partition tree is again shown to perform better than the partition tree approach.

2. Propositional Satisfiability and SAT Solvers

This chapter discusses the propositional satisfiability problem (SAT) and SAT solvers including *conflict driven clause learning* [Marques-Silva and Sakallah 1999]. A particular emphasis is on random run times and the closely related *restart strategies* [Luby et al. 1993; Walsh 1999]. Some of the results presented in this chapter are studied in [PI], but extended here with a large number of experiments and a new restart strategy [Biere 2008].

2.1 Propositional Satisfiability

Let $B = \{x_1, \ldots, x_n\}$ be a set of Boolean variables. The set of *literals* $\{x_i, \neg x_i \mid x_i \in B\}$ consists of variables x_i and negated variables $\neg x_i$, $1 \leq i \leq n$. A disjunction of literals is a *clause* and a conjunction of clauses is a *formula* in conjunctive normal form (CNF). Whenever convenient, the clauses are interpreted as sets of literals, and formulas as sets of clauses. The truth values of literals are determined by a subset of literals called a *truth assignment*. A truth assignment τ is *conflicting* if both $x, \neg x \in \tau$ for some variable x, and *complete* if all variables x_i appear in it. Non-conflicting assignments are *consistent* and non-complete *partial*. A literal l is true in a consistent assignment τ if $l \in \tau$ and false if $\neg l \in \tau$. As usual, $\neg\neg l$ is equivalent to l. Given a formula ϕ, the set $\mathrm{Lits}(\phi)$ consists of all literals $l, \neg l$ such that l appears in a clause of ϕ. Variables and literals that are not either true or false are unknown. A clause C is *satisfied* by τ if C contains a true literal and a CNF formula ϕ is satisfied by τ if all clauses in ϕ are satisfied. For example, let ϕ be the formula

$$\phi = (\neg e \vee b) \wedge (\neg d \vee a) \wedge (\neg a \vee c) \wedge (\neg c \vee a) \wedge (\neg a \vee \neg b \vee \neg d). \qquad (2.1)$$

Then ϕ is satisfied by $\tau = \{\neg a, b, \neg c, \neg d, \neg e\}$. The problem of determining whether a given formula has a satisfying truth assignment is called the *propositional satisfiability problem* (SAT).

A formula ϕ' is a *logical consequence* of ϕ, denoted $\phi \models \phi'$, if each satisfying truth assignment of ϕ also satisfies ϕ'. Two formulas ϕ and ϕ' are *equivalent*, denoted $\phi \equiv \phi'$, if they are satisfied by exactly the same truth assignments.

2.2 Conflict-Driven Clause Learning SAT Solvers

Most current complete SAT solvers, such as zCHAFF [Moskewicz et al. 2001], MINISAT [Eén and Sörensson 2004], LINGELING [Biere 2010] and CRYPTOMINISAT [Soos et al. 2009], extend the classical DPLL solvers [Davis and Putnam 1960; Davis et al. 1962] with *clause learning* techniques [Marques-Silva and Sakallah 1999; Zhang et al. 2001]. The underlying idea is to perform a backtracking search on partial truth assignments which are extended with heuristically chosen *decision literals*. If a clause of length k contains $k-1$ false literals, the remaining literal must necessarily be true in order for the whole formula to be true. Such literals "forcibly set" to true are called *implied literals*. The implied literals are obtained by computing the *unit propagation* closure, and the process potentially results in a conflicting truth assignment. If the truth assignment becomes conflicting, the algorithm identifies a "reason" for the conflict, represented as a clause, based on stored information on the propagation sequence. The algorithm uses such clauses for two purposes: to guide the backtracking and to prevent similar conflicts from arising in the subsequent steps of the algorithm. This is done by conjoining the clause with the formula. The resulting algorithm differs enough from DPLL to qualify for a new name, the *conflict-driven clause learning* (CDCL) algorithm.

Computing the unit propagation closure corresponds to repeatedly identifying clauses having all but one literal false and the remaining literal unknown, and extending the truth assignment with the unknown literal until no such clauses exist.

Definition 1 *Given a formula ϕ and a truth assignment τ, the* unit propagation closure $\mathrm{UP}(\phi, \tau)$ *is the smallest set $\tau' \supseteq \tau$ containing τ and the literals a_i such that there is a clause $(a_1 \vee \ldots \vee a_k) \in \phi$ containing a_i and $\{\neg a_1, \ldots, \neg a_{i-1}, \neg a_{i+1}, \ldots, \neg a_k\} \subseteq \tau'$.*

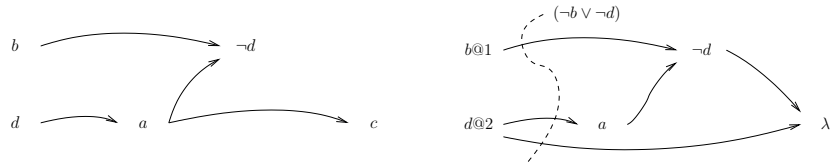


Figure 2.1. An implication graph for ϕ in Eq. (2.1) obtained by computing $\mathrm{UP}(\phi, bd)$ (*left*), and a corresponding conflict graph (*right*). The graph on the right also indicates the decision levels and a unique implication point (UIP) cut

Computing the unit propagation closure is useful in searching for a satisfying truth assignment using a backtracking search algorithm. Assuming there is a truth assignment satisfying ϕ and containing a set of literals τ, then the satisfying truth assignment must also contain literals in $\mathrm{UP}(\phi, \tau)$:

Proposition 1 *Let ϕ be a formula, $\tau = \{b_1, \ldots, b_m\}$ a partial truth assignment, and $\mathrm{UP}(\phi, \tau) = \{a_1, \ldots, a_n, b_1, \ldots, b_m\}$ the corresponding unit propagation closure. Then $\phi \wedge b_1 \wedge \ldots \wedge b_m \models a_1 \wedge \ldots \wedge a_n \wedge b_1 \wedge \ldots \wedge b_m$.*

An algorithm making decisions and computing the unit propagation closure gives an order for the obtained literals in a natural way. Therefore the truth assignment can be seen as an initially empty ordered sequence of literals $\tau = v_1 v_2 \ldots v_m$ where v_m is either a decision literal or an implied literal. In the latter case there is a clause $C \in \phi$ such that some literal l in C equals v_m, and all other literals $l' \in C \setminus \{l\}$ appear negated earlier in the sequence τ. In this case the clause C is said to imply l. If there are several such clauses, one of them is arbitrarily chosen as the implying clause.

The information on decision literals, and implied literals and clauses can be organized as a directed, acyclic *implication graph* where vertices are the literals of the truth assignment and the edges encode how the literals were assigned. More specifically, the graph has the edges from $\neg a_1, \ldots, \neg a_{i-1}, \neg a_{i+1}, \ldots, \neg a_k$ to a_i, if a_i was implied by the clause $(a_1 \vee \ldots \vee a_k)$. Decision literals, on the other hand, have no incoming edges. For example, consider the formula ϕ in Eq. (2.1) and one possible conflicting truth assignment $\mathrm{UP}(\phi, bd) = bdac\neg d$. In one of the possible propagation orders, the literals $a, c, \neg d$ are implied by the clauses $(\neg d \vee a), (\neg a \vee c), (\neg a \vee \neg b \vee \neg d)$, respectively, while both b and d are decisions. The resulting implication graph corresponding to this propagation order is the one on the left in Fig. 2.1.

The example illustrates that a truth assignment can become conflicting

during the unit propagation. As mentioned above, the CDCL algorithm will determine the reason for the conflict using the implying clauses. Once a truth assignment τ becomes conflicting, a *conflict graph* is constructed from the implication graph. The propagation is interrupted once the first conflicting literal pair $x, \neg x$ is obtained in τ. Then all literals having no directed path to either x or $\neg x$ are removed and a new literal $\lambda \notin \mathrm{Lits}(\phi)$ is connected with edges from x and $\neg x$. The right hand side graph of Fig. 2.1 illustrates the conflict graph obtained with the above process from the left hand side implication graph.

The reason clause for the conflict is learned by analyzing the conflict graph. The graph is partitioned to *conflict* and *reason sets*, where the former consists initially of λ, by a *conflict cut*. Intuitively, the conflict results from the assignments represented in the reason side. The analysis simply consists of changing the partitioning by moving one or more implied literals, connected with an edge to a literal in the conflict set, from the reason set to the conflict set.

A given conflict cut uniquely defines a *conflict clause* $C = \neg c_1 \vee \ldots \vee \neg c_n$ where each c_i is in the reason set and has an edge to a literal in the conflict set. The following two propositions state that a conflict clause C has the following two properties: (i) $\phi \models C$, and (ii) all literals of C appear negated in the related conflicting truth assignment τ. We will use these two properties in showing that the CDCL algorithm terminates and provides correct results. The proofs follow the ideas in [Zhang and Malik 2003].

Proposition 2 *All conflict clauses obtained from the formula ϕ by the conflict analysis described above are logical consequences of ϕ.*

Proof. The claim is trivially true for any clause if ϕ is unsatisfiable. Therefore we assume that ϕ is satisfied by a complete, consistent truth assignment τ. Then the claim states that each clause defined by the conflict cut contains a literal also in τ. The conflict analysis starts with a clause $x \vee \neg x$ and since τ is complete, either $x \in \tau$ or $\neg x \in \tau$. Assume now that the claim holds for an arbitrary conflict clause $C = (\neg c_1 \vee \ldots \vee \neg c_m)$ and $\neg c_i \in C$ is an implied literal moved to the conflict side. Let c_i be implied by the clause $A = (a_1 \vee \ldots \vee a_{i-1} \vee c_i \vee a_{i+1} \vee \ldots \vee a_m)$. By the assumptions both clauses A and C contain a literal from the satisfying truth assignment τ. The resulting conflict clause $C' = (A \cup C) \setminus \{c_i, \neg c_i\}$ must also contain a literal from τ, since the two literals c_i and $\neg c_i$ missing

from C' could not be the only literals of A and C in τ by the consistency of τ. This completes the induction proof. □

Since the conflict clauses C are logical consequences of ϕ, conjoining them with ϕ does not change the set of truth assignments satisfying ϕ, and we have the equality $\phi \equiv \phi \wedge C$.

The next result essentially shows that a conflict clause is "false under the conflicting truth assignment" that initiated the conflict analysis. This will be useful in showing when the conflict clause can be used for backtracking in the search.

Proposition 3 *Let τ be a conflicting truth assignment and C a conflict clause in the conflict graph obtained from τ. Then all literals of C appear negated in τ.*

Proof. We will prove this by induction on the conflict analysis. The initial conflict cut consists of the literals x and $\neg x$, and by construction they both appear negated in a conflicting τ. Assume then that the claim holds for a conflict clause C, and the conflict analysis moves the implied literal $\neg c_i \in C$ to the conflict side. By the assumptions, $\neg c_i$ is false and c_i is implied by a clause A such that $c_i \in A$. Therefore the other literals of A must appear negated in τ, and hence the literals of the new conflict clause $(C \cup A) \setminus \{c_i, \neg c_i\}$ all appear negated in τ. □

The CDCL algorithm conjoins one conflict clause into the formula ϕ every time the algorithm finds a conflict during propagation. Typically some computing is involved to produce as short a clause as possible [Sörensson and Biere 2009], and some implementations include more than one such clause [Zhang et al. 2001].

To guide the backtracking search, the CDCL algorithm keeps track of the implications and decisions using *decision levels*. As a literal is included to the truth assignment, it is labeled with a decision level equal to the number of decision literals in the truth assignment. For example in Fig. 2.1, the decision levels of b, d, a, c, and $\neg d$ are 1, 2, 2, 2, and 2, respectively. The decision level 0 is special in the sense that, by Prop. 1, all literals implied on level 0 are already proved to be logical consequences of the formula ϕ.

A conflict clause is *asserting*, if it contains a single literal in the highest decision level. Asserting conflict clauses can be used for guiding the backtracking: by Prop. 3 all literals of a conflict clause are false. The lit-

eral in the highest decision level is unique in a given asserting clause, and therefore removing all literals with decision levels higher than the second highest decision level in the asserting clause results in the asserting clause implying the unique literal. If the asserting clause contains only a single literal, only the literals on decision level 0 are preserved in the truth assignment. The cut corresponding to the only asserting clause $\neg b \vee \neg d$ is shown in Fig. 2.1.

Proposition 4 *The CDCL algorithm which learns asserting clauses always terminates and outputs* Unsat *if and only the formula ϕ is unsatisfiable.*

Proof. Each time a conflict is found on decision level higher than 0, there is at least one asserting cut for the related conflict graph, that is, one containing the decision literal in the highest decision level. The truth assignments generated after propagation by the CDCL algorithm can be therefore seen as a sequence $\tau_1\tau_2\ldots\tau_t$ where τ_t either contains a pair of literals $x, \neg x \in \tau_t$ on the decision level 0 and nothing on other levels if the instance is unsatisfiable, or is complete and consistent if the instance is satisfiable. To see this we associate an increasing sequence with each truth assignment $\mathrm{UP}(\phi, \tau)$ constructed with propagation by the CDCL algorithm, show that there is a limit on the length of the sequence and conclude that the CDCL algorithm must therefore terminate. The sequence consists of the number of literals on each decision level on the truth assignment. Let τ_i be a truth assignment with k_i decision literals. Then the corresponding *population list* $D_{\tau_i} = d_0^{\tau_i} d_1^{\tau_i} \ldots d_{k_i}^{\tau_i}$ gives the number $d_m^{\tau_i}$ of literals on decision level m in τ_i.

The ordering $\prec$ is defined as $\tau_i \prec \tau_j$ if and only if there is a decision level m such that $d_m^{\tau_i} < d_m^{\tau_j}$ and $d_0^{\tau_i} = d_0^{\tau_j}, \ldots, d_{m-1}^{\tau_i} = d_{m-1}^{\tau_j}$. Suppose now that the CDCL algorithm finds a conflict in $\mathrm{UP}(\phi, \tau_i)$ and learns an asserting clause C. The next step of the algorithm is to compute $\mathrm{UP}(\phi \wedge C, \tau_i')$, where τ_i' is obtained from τ_i by removing the literals having decision levels higher than the second highest decision level on the literals of C. Since C is implying in τ_i', also $\mathrm{UP}(\phi, \tau_i) \prec \mathrm{UP}(\phi \wedge C, \tau_i')$. The algorithm thus produces an increasing sequence of truth assignments. The largest element in this sequence corresponds to the conflicting truth assignment containing all literals of ϕ on the decision level 0. Hence the algorithm must terminate, either by finding a complete consistent truth assignment or by finding an inconsistent truth assignment where the conflict is on the

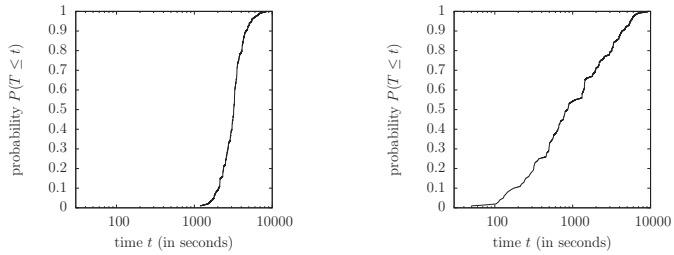


Figure 2.2. Run time distributions for formulas *total-10-13-u* and *mizh-md5-48-5*, both from the industrial category of the SAT 2007 solver competition

decision level 0.

The latter case occurs if and only if the instance is unsatisfiable since $\phi \models C$ for asserting clauses C by Prop. 2 and $\phi \models \bigwedge \mathrm{UP}(\phi, \emptyset)$. □

Interestingly, Prop. 4 only requires that the conflict analysis can be performed, and hence the conflict graph can be constructed. Therefore the algorithm is free to forget previously learned asserting clauses C which are not required to construct the conflict graph. This is essential in most formulas to avoid memory exhaustion.

2.2.1 Randomness in Solver Run Times

Many branch-and-bound backtracking algorithms exhibit high variance in run times when small alternations are introduced to the search process. As this phenomenon is by no means specific to SAT solving, there is a significant amount of related work in many different contexts [Speckenmeyer et al. 1988; Li and Wah 1990; Prestwich and Mudambi 1995; Grama and Kumar 1999]. Intuitively the idea is that an "unlucky" choice in the heuristic search can lead to a part of the branch-and-bound tree which is particularly difficult to solve. Sometimes such areas could have been omitted, had the search been performed in a slightly different order.

Although the CDCL search described above differs somewhat from a classical tree-based search, experiments show that also run times of these solvers exhibit similar behavior. The range of run times depends on the instance being solved and for example MINISAT [Eén and Sörensson 2004] run times vary from two-fold to several orders of magnitude. Two *cumulative run time distributions* for benchmark instances from the SAT 2007 competition, showing the probability that an instance is solved in time less than a give t, are shown in Fig. 2.2.

Some SAT instances, when solved with a CDCL algorithm, obey a *heavy-tailed run time distribution* [Gomes et al. 2000]. Such distributions have a significant probability of producing "outlier" samples, that is, run times which are far from median run times. The distributions behave in practice as if they would have an infinite standard deviation or even an infinite mean. Since propositional formulas have a finite search space and the CDCL algorithm is complete, the statistics are of course finite. However, since the search space is in the worst case exponential in the size of the formula, the statistics can in practice be considered as infinite for some formulas [Gomes et al. 1998].

The small variations to the CDCL search result, for example, from the learned clause generation and the process used for selecting decision literals. The process for selecting decision literals relies on some of the numerous heuristics described in the literature (e.g., [Jeroslow and Wang 1990; Hooker and Vinay 1995; Li and Anbulagan 1997a; Marques-Silva 1999; Moskewicz et al. 2001; Lagoudakis and Littman 2001; Herbstritt and Becker 2003; Irgens and Havens 2004; Heule and van Maaren 2006]). Most heuristics employ randomization to break ties, and often implement a form of deliberate increase in the random behavior either by introducing a *heuristic equivalence parameter* [Gomes and Selman 2001] or by simply mixing the *random heuristic* (a heuristic which selects a literal pseudo-randomly from the set of all unknown literals) together with a more context-dependent heuristic. Hence it is natural to express the run time of a solver as a random variable and a related probability distribution.

Let T be the random variable describing the time required to solve a given formula ϕ with a CDCL solver S randomized using, for example, some of the above mentioned approaches. The *cumulative run time distribution* $q_T(t)$ gives the probability that $T \leq t$. We will use the cumulative distribution to express the expected time required to solve the instance ϕ with the solver S. By definition, this is

$$\mathbb{E}T = \int_0^\infty t q'_T(t) \mathrm{d}t, \tag{2.2}$$

where $q'_T(t)$ is the derivative of the cumulative distribution $q_T(t)$.

2.2.2 Restarts and Randomization

For formulas having distributions with "high dynamics", such as the heavy-tailed distribution, it is useful to interrupt the search procedure after

some time and start the search again from the beginning. The motivation is that if the solver has made an unlucky choice and ended in a difficult subtree, it is better to reject some of the partial results obtained so far and hope to find an easier subtree. Indeed, it can be shown that restarts can eliminate heavy-tailed distributions [Gomes et al. 2000].

An algorithm can perform restarts according to a schedule called *restart strategy* $\textsc{Strat} = (t_1 t_2 \ldots)$, which is a sequence of time values called restart limits. Applying $\textsc{Strat}$ to the solver S solving a formula ϕ results in S first running for t_1 steps. If ϕ is not solved in these steps, S will restart by clearing the truth assignment and starting anew, this time running for t_2 steps. The process is continued until ϕ is finally solved. As a result of applying the restart strategy, the time required to solve ϕ will change in general. Although the CDCL solvers store learned clauses when restarting, we will analyze here the "hard restarts", where all learned clauses, including the clauses of length 1, are removed. The restriction will be lifted in later experiments (see Sect. 4.4), but will be used here to build a better understanding of the observed phenomena. Let the time required to solve the instance ϕ with the solver S be again described by the random variable T. The random variable $T_{\textsc{Strat}}$ describes the time required to solve ϕ with S and the restart strategy $\textsc{Strat}$. If the cumulative distribution associated with T is known, it is possible to derive a closed form representation of the expected run time for some restart strategies. For example, an important special case is the *fixed restart strategy* $\textsc{Fixed} = (a a \ldots)$. The expected run time using this strategy is

$$\mathbb{E}T_{\textsc{Fixed}} = \sum_{i=0}^{\infty} \left(\int_0^a (t + ia)(1 - q(a))^i q'(t) \mathrm{d}t \right), \tag{2.3}$$

where $(1 - q(a))^i q'(t)\mathrm{d}t$ is the differential probability that the formula is solved after i restarts at time $(t + ia)$. By regrouping and noting that a and i do not depend on t, we obtain

$$\mathbb{E}T_{\textsc{Fixed}} = \int_0^a q'(t)\mathrm{d}t \sum_{i=0}^{\infty} \left(ia(1 - q(a))^i \right) + \int_0^a t q'(t)\mathrm{d}t \sum_{i=0}^{\infty} \left((1 - q(a))^i \right). \tag{2.4}$$

The first sum converges to $\sum_{i=0}^{\infty} ia(1 - q(a))^i = a(1 - q(a))/q(a)^2$, the second sum converges to $\sum_{i=0}^{\infty}(1 - q(a))^i = 1/q(a)$, and the first integral equals $\int_0^a q'(t)\mathrm{d}t = q(a)$. Finally, the second integral can be written as $\int_0^a t q'(t)\mathrm{d}t = aq(a) - \int_0^a q(t)\mathrm{d}t$ by integration by parts. Finally, we have the following.

$$\mathbb{E}T_{\textsc{Fixed}} = \frac{a - \int_0^a q(t)\mathrm{d}t}{q(a)} \tag{2.5}$$

Intuitively, the expected run time when using the fixed restart strategy is low if it is likely that the instance is solved within time a.

It can be shown that for any distribution $q(t)$ there is a value a^* such that the fixed restart strategy $\text{OPT} = (a^* a^* \ldots)$ results in lowest expected value among all restart strategies [Luby et al. 1993]. However, the value a^* depends on the distribution which is in general unknown. Furthermore, introducing the hard restarts considered here can break Prop. 4 if improperly implemented; if the shortest proof for the instance ϕ takes more than a steps, solving with the fixed restart strategy cannot succeed. To preserve the termination property of the CDCL algorithm, there must be no upper bound on the restart limits.

Two widely used unbounded restart strategies are an exponential strategy $\text{STRAT}^{\text{E}} = (e_1 e_2 \ldots)$ where $e_i = \alpha 2^{\beta(i-1)}$ for some $\alpha \geq 1$ and $\beta > 1$, and a universal strategy $\text{STRAT}^{\text{U}} = (u_1 u_2 \ldots)$, where

$$u_i = \begin{cases} \alpha 2^{k-1}, & \text{if } i = 2^k - 1 \text{ for some } k \in \mathbb{N} \\ u_{i-2^{k-1}+1}, & \text{if } 2^{k-1} \leq i < 2^k - 1, \end{cases} \tag{2.6}$$

and $\alpha \geq 1$. In [Luby et al. 1993] the authors show that the expected run time of a solver using the universal restart strategy is within a logarithmic factor from the run time obtained with the optimal strategy OPT. Finally, the PICOSAT restart strategy described in [Biere 2008] combines the exponential growth with the large number of short restarts of the universal strategy. The resulting strategy can be expressed as the *nested exponential* strategy $\text{STRAT}^{\text{NE}} = (p_1 p_2 \ldots)$, where

$$p_i = \begin{cases} \alpha, & \text{if } i = 1 \\ \alpha \beta^{i-b_{i-1}-1}, & \text{if } b_{i-1} + 1 \leq i \leq b_i, \end{cases} \tag{2.7}$$

where $b_1 = 0$, $b_i = b_{i-1} + i$, $\alpha \geq 1$, and $\beta > 1$. Intuitively, this strategy consists of exponentially growing sequences of length 1, 2, 3, ..., each having as prefix the previous sequence.

2.3 Experiments on Hard Restarts

In the following, we study the question whether "hard restarts" can be used to speed up sequential solving of SAT instances. The tables 2.1 and 2.2 report the results of applying hard restart strategies to solving instances from the SAT Competition 2007 (SAT-Comp 2007). The set of instances was selected by running MINISAT 1.14 once on all the instances

that were solved in the competition from the *industrial* and *crafted* categories and selecting those that had a run time exceeding 1000 seconds. The qualifying instances were then solved without timeout one hundred times each to obtain an experimental run time distribution. Finally the expected run times (column Exp), optimal restart strategy (FIXED$^{a^*}$) and the universal, exponential and nested exponential strategies (STRAT$^{\mathrm{U}}$, STRAT$^{\mathrm{E}}$ and STRAT$^{\mathrm{NE}}$, respectfully) were computed from the obtained distribution. The restart strategies limit run times in seconds, $\alpha = 15$ for the strategies STRAT$^{\mathrm{U}}$, STRAT$^{\mathrm{E}}$ and STRAT$^{\mathrm{NE}}$ and $\beta = 1.2$ for STRAT$^{\mathrm{E}}$ and STRAT$^{\mathrm{NE}}$. The column labeled a^* reports the optimal restart limit for each instance. The symbol ∞ is used in case the optimal restart limit equals the maximum run time of the experimental distribution.

Based on the results in Table 2.1, the unsatisfiable instances are usually best solved by placing no limits on the run times. The hard restart strategies seem, with the exception of FIXED$^{a^*}$, to slow down the solving significantly. The situation changes dramatically when satisfiable instances are considered in Table 2.2. In particular, the optimal restart limit is in most cases different from the maximum run time in the experimental distribution, and the speed-up obtained by using the optimal restart strategy is more significant. Based on the results, in these cases also the restart strategies perform better, in many cases providing a clear speed-up compared to the approach without restart strategies.

Table 2.1. Sequential run times for some unsatisfiable instances from SAT-Comp 2007

Instance	Exp	Fixed$^{a^*}$	a^*	Strat^{U}	Strat^{E}	Strat^{NE}
999999000001nc	2065.12	2065.12	∞	25178.01	4175.60	181029.86
AProVE07-03	1196.58	1196.58	∞	14331.09	2807.77	128570.77
AProVE07-08	1839.21	1839.21	∞	20677.32	3634.56	161594.01
AProVE07-09	4015.97	4015.97	∞	52052.98	8542.41	305390.51
AProVE07-16	1563.48	1563.48	∞	18112.02	3196.98	149989.89
AProVE07-27	4183.33	4183.33	∞	58138.89	8410.43	447728.47
QG7-dead-dnd001	1321.93	1321.93	∞	12477.87	2785.58	89235.24
QG7-dead-dnd002	1701.78	1701.78	∞	17761.29	3367.85	154848.66
QG7-gensys-icl100	3406.20	3406.20	∞	42239.22	7412.73	360003.58
QG7-gensys-ukn003	1593.94	1593.94	∞	16221.69	3279.30	179429.58
QG7a-gensys-icl001	7259.33	7259.33	∞	106637.18	16272.85	804435.60
clqcolor-10-07-09	1900.03	1900.03	∞	24637.56	3844.71	181642.03
connm-ue-csp-sat-n800-d0.02-s925928766	1557.55	1557.55	∞	17418.72	3275.33	135539.39
SGI_30_50_30_20_1-dir	976.24	976.24	∞	10570.94	2052.09	105048.62
SGI_30_50_30_20_3-dir	1432.93	1432.93	∞	14847.32	3128.40	162195.24
hwb-n26-01-S1957858365	709.11	709.11	∞	6531.42	1427.38	78762.33
lksat-n1000-m6860-k4-l4-s1935114289	1238.28	1238.28	∞	13933.90	2801.63	130583.36
cube-11-h13-unsat	1745.61	1745.61	∞	17642.63	3591.27	128136.33
dated-10-11-u	9889.17	9889.17	∞	148006.50	20927.15	860980.84
dated-10-13-u	4116.58	4116.58	∞	56671.52	8297.39	448975.93
dated-5-15-u	1551.26	1551.26	∞	16203.59	2997.25	113575.78
dated-5-17-u	3088.02	3088.02	∞	33116.81	5826.29	205497.77
emptyroom-4-h21-unsat	5205.60	5205.60	∞	69127.84	11127.73	498802.02
eq.atree.braun.11.unsat	3096.28	3096.28	∞	34776.15	6935.44	333100.26
hwb-n26-03-S540351185	1212.50	1212.50	∞	14393.06	2857.24	138739.13
hwb-n28-01-S136611085	1557.14	1557.14	∞	15604.97	3208.09	177451.43
hwb-n28-02-S818962541	4654.40	4654.40	∞	71133.53	8926.73	555514.00
linvrinv5	2828.63	2828.63	∞	33829.46	6630.71	329030.57
manol-pipe-f9b	10617.75	10560.99	32949.69	101754.70	17256.17	404736.19
manol-pipe-f9n	11026.38	10702.17	26206.90	119435.21	16154.78	504421.51
manol-pipe-g10nid	1222.47	1222.47	∞	10912.65	2339.16	78107.98
mod2c-3cage-unsat-10-2	3020.26	3020.26	∞	35066.27	6687.95	331113.88
mod2c-3cage-unsat-10-3	2580.28	2580.28	∞	32081.18	5877.59	274846.06
phnf-size10-exclusive-luckySeven	891.82	891.82	∞	8748.36	1841.55	72177.90
pmg-12-UNSAT	4268.80	4268.80	∞	65529.71	8234.97	493123.07
pyhala-braun-unsat-40-4-02	2641.35	2641.35	∞	32593.35	5960.27	285365.95
s101-100	2528.70	2528.70	∞	31678.80	5842.86	274817.87
s97-100	2001.74	2001.74	∞	25504.97	4141.72	224352.32
sortnet-6-ipc5-h11-unsat	4886.02	4886.02	∞	71664.77	10466.43	517394.31
total-10-13-u	3278.80	3278.80	∞	38248.02	6623.49	239536.07
unsat-set-b-fclqcolor-10-07-09	2027.35	2027.35	∞	25946.60	4201.85	191066.65
uts-l06-ipc5-h33-unknown	1114.95	1114.95	∞	10799.92	2298.63	69221.58
Total time	129013	128632	–	$1.5 * 10^6$	252685	$1.12 * 10^7$

Table 2.2. Sequential run times for some satisfiable instances from SAT-Comp 2007

Instance	Exp	Fixed$^{a^*}$	a^*	Strat^{U}	Strat^{E}	Strat^{NE}
cube-11-h14-sat	4831.89	4831.89	∞	62717.69	10604.28	430459.73
dated-10-13-s	2276.95	717.79	29.05	940.75	928.13	948.18
dated-10-17-s	7197.18	1172.84	8.06	2174.17	4110.83	2071.90
emptyroom-4-h22-sat	11475.60	11475.60	∞	72819.84	21242.36	119033.39
mizh-md5-48-5	1659.23	1236.54	899.27	3525.81	1710.71	9899.53
mizh-sha0-35-3	287.51	223.31	98.08	373.70	225.30	1238.03
mizh-sha0-36-2	2951.31	901.17	36.32	1798.98	2529.31	2569.08
mod2-rand3bip-sat-250-3	1180.48	1180.48	∞	2849.09	1722.08	5775.16
mod2-rand3bip-sat-280-1	2381.83	941.61	9.18	1218.03	1721.33	1573.72
sortnet-7-ipc5-h16-sat	21449.26	16003.16	156.55	34718.75	20463.02	46256.75
vmpc_28	623.22	13.16	0.14	120.25	406.85	113.08
Total time	56314.5	38697.6	–	183257	65664.2	619939

3. Grid Computing

Grid computing allows a user to execute computational tasks in parallel using a large pool of computing resources provided by several computing clusters through a uniform interface. Grid and the related cloud computing have recently gained interest because of various economical and environmental reasons. This section describes the grid computing environment that will be used in the algorithms discussed later in the work.

3.1 The Computing Model

A computing task to be executed in a grid is called a *job*. The grid interface allows submission of jobs, monitoring their status while they are running, and retrieving their results once they finish. Apart from being monitored, the jobs may not communicate with each other or the user while running. The user may request a certain amount of resources, such as CPU time and memory, for each job from the grid. The request has to be agreed upon with the grid system when the job is submitted. Once submitted, the execution of the job must not exceed the limits. In most of the applications discussed in this work there is no simple pool of tasks that need to be executed, but instead a master process uses the previously obtained results in addition to its own computing to construct the tasks on-the-fly. The process of constructing the tasks and executing them is called a *work flow*. The work flow starts when the first task is constructed and ends when a time limit for the work flow is reached or a solution can be determined.

Figure 3.1 depicts how the jobs are executed in the grid. The number of computing elements available to a user in this model is fixed to N. Each job executes, shown in darker shade in the figure, until it finishes nor-

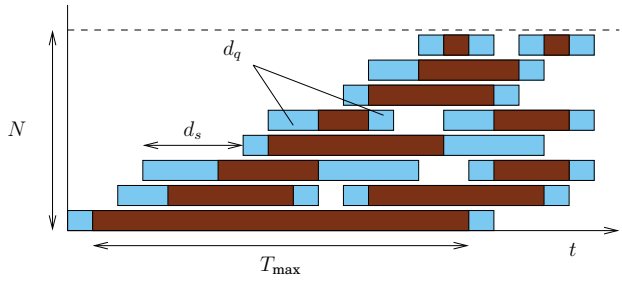


Figure 3.1. Schematic visualization of grid computing

mally or exhausts the requested resources, represented by the time limit $T_{\max}$ in the figure. In addition the jobs suffer two kinds of delays modeled as random variables: the submission delay d_s and the queuing delay d_q. The queuing delay is the time the job spends in the batch queue system of a cluster without using the CPU. The submission delay is the time difference between starting two job submissions, and includes the time required to query the clusters, transfer the job files, and the occasional time required to query the statuses of the jobs running in the grid.

3.2 Job Management

In a typical scenario considered in this work a user initiates a work flow to determine the satisfiability of a formula and would like to receive the result as soon as possible. For software management reasons it is useful to separate the task of constructing the jobs from shepherding them in the grid, since typically the challenges in the two are totally different. The latter includes ensuring that jobs eventually get to run in the grid in a reasonable time, taking corrective actions if this seems not to happen and keeping a "blacklist" of clusters where such actions are required.

The experiments in this work use a *job manager* to simplify bookkeeping of such events. The job manager acts as a layer between the grid interface and the user. The system is described in [Pitkanen et al. 2008] where it is also used in a similar role for a medical image processing application.

Jobs may fail to execute in the grid due to several reasons. The most common reason in our experiments is the failure to reach the CPU in a reasonable time. In most of the experiments considered in this work the wall clock time for the work flow is limited. Therefore excessively long queuing times are unacceptable and are considered failures. In a widely

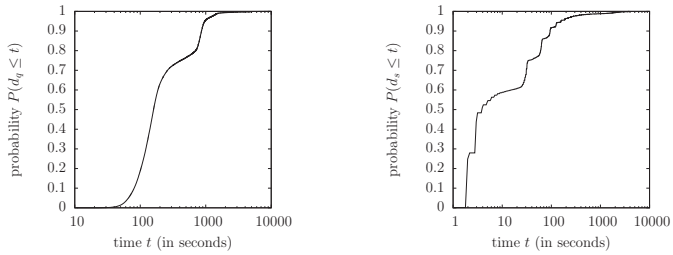


Figure 3.2. Cumulative queuing delay distribution obtained from nearly 200,000 jobs run between August 18, 2010 and February 18, 2011 (left), and cumulative submission delay distribution obtained from 21,000 jobs run between February 15, 2011 and March 2, 2011 (right)

distributed grid there are other reasons for failed jobs, such as sudden, unexpected service breaks or even hardware failures.

The job manager keeps track of clusters which produce failures. These clusters are blacklisted for a period of time to allow the problem causing the failure to be solved. In case a job fails in a cluster, the job manager also takes care of its resubmission. The number of resubmissions is limited to avoid getting stuck at submitting a faulty job.

The jobs are monitored approximately every half a minute by the job manager. As the software in the clusters also checks periodically whether jobs have finished, it is common that several jobs finish simultaneously. Because of the submission delay, refilling the grid requires more time than what would be needed if the jobs would finish one-by-one. Instead of submitting a new job immediately after an old job finishes, several jobs need to be submitted sequentially. The job manager lessens this effect by selecting, in work flows where this is possible, the job run times uniformly at random from a time range.

Figure 3.2 illustrates the statistics for the queuing and submission delays measured for different work flows in the m-grid computing environment, running in Finnish universities and computing centers. The left figure shows the experimental distribution of the queuing delay d_q in a work flow related to the *iterative partitioning approach* (see Ch. 5) where each job has a maximum run time range of 60 – 90 minutes. The statistics were collected during a half year period and consist of nearly 200,000 jobs. The majority of the jobs, roughly 60%, waste 80 to 260 seconds in the queue. Once the job has spent 600 seconds queuing in the cluster without getting CPU time, the job is resubmitted in another cluster. The second increase in the probability between 720 and 900 seconds reflects these

jobs that were resubmitted after a queuing time-out. Finally there is a small number of jobs that failed near the end of their first submission after running the maximum of 90 minutes, and were then solved at another cluster. The statistics do not include jobs that did not finish successfully after the second submission.

A similar distribution for the submission delay is given in the right of Fig. 3.2. The statistics are collected from a work flow related to the *CL-SDSAT approach* (see Sect. 4.4). In this work flow the time required to construct the jobs is usually less than one second, only in 5% of the cases more than 5 seconds and never longer than 22 seconds. Therefore the submission delays are much lower than the queuing delays: usually the submission can be done in less than four seconds. The plateaus in the distribution result from the periodic polling of the job states, and long submission times related to filling the grid are somewhat rare.

4. Parallel Solving based on Algorithm Portfolios

The inherent randomness in SAT solver run times can be utilized in obtaining a natural parallelization approach. In such approaches the goal is to run in parallel several solvers with different search parameters, such as restart and learning strategies, decision heuristics, or completely different algorithms such as local search, on the same formula and obtain the solution from the first solver determining the satisfiability. This *algorithm portfolio approach* [Rice 1976; Huberman et al. 1997; Gomes and Selman 2001] has been extensively studied [Janakiram et al. 1988; Janakiram et al. 1988; Luby and Ertel 1994; Petrik and Zilberstein 2006; Inoue et al. 2006; Gebser et al. 2011], and has recently proved surprisingly efficient in solving structured formulas [Hamadi et al. 2009a; Hamadi et al. 2009b; Guo et al. 2010; Biere 2010].

For simplicity, this work follows an approach where a SAT solver uses a small amount of randomness in its decision heuristic to obtain an effect similar to the more complex portfolios. The first part of this chapter studies the *simple distributed SAT solving* (SDSAT) approach, where the solvers only communicate the success or failure in determining satisfiability to the master process. The second part studies an extension of the SDSAT approach called *clause learning simple distributed SAT solving* (CL-SDSAT), where the solvers may also return learned clauses in case they fail to determine satisfiability.

The results on the SDSAT approach are based on [PI]. The approach is studied using different "hard" restart strategies in a parallel setting. This chapter complements [PI] by giving a significant amount of new experimental data and studying the *nested exponential* strategy. On the other hand, unlike in [PI], the alternate distribution schedules are not discussed here, and the grid delays are assumed to be zero.

The results on the CL-SDSAT approach are based on [PII]. The dis-

cussion focuses on the algorithmic framework for CL-SDSAT. The framework is used for describing the effect of different heuristics for sharing the learned clauses, as well as some scalability results. The results in [PII] are extended with new experiments using benchmark instances from SAT-Comp 2009.

Both the SDSAT and the CL-SDSAT approaches result potentially in good speed-up. The experiments show that the SDSAT approach is inherently limited in a grid environment with fixed length jobs for SAT formulas, whereas the CL-SDSAT approach can improve the solving capabilities of its underlying solver, and enable the solving of formulas not solvable sequentially in reasonable time limits.

4.1 Simple Distributed SAT Solving

The Simple Distributed SAT Solving (SDSAT) approach is based on running several randomized SAT solvers in a distributed or parallel computing environment on a given formula, and obtaining the result from the first solver that finishes. The idea of studying random behavior when parallelizing backtracking search is not new. For example, [Janakiram et al. 1988; Janakiram et al. 1988] study the effect in randomized, parallel branch-and-bound algorithms. Restarts in parallel settings are studied in [Luby and Ertel 1994] on Las Vegas algorithms that are similar to the randomized SAT solvers. More general view is taken by [Huberman et al. 1997] studying a setting where parallel solvers have different search parameters, and recently learning good portfolios has been studied, for example, in [Petrik and Zilberstein 2006].

Speed-up can be obtained by the portfolio SAT solving approach, where a given formula is solved simultaneously by several different solvers. There are different ways of combining solvers so that the speed-up would be as good as possible for a wide range of benchmarks (see, e.g., [Inoue et al. 2006; Hamadi et al. 2009a; Hamadi et al. 2009b; Biere 2010]). One effective approach is to simply introduce a small amount of randomness in the heuristic while keeping the search strategy of the solver otherwise intact. This provides an interesting setting for obtaining speed-up as it requires virtually no modifications to the underlying solver. The results in, e.g., [Wintersteiger et al. 2009] also suggest that it compares favorably to many other portfolio based approaches. In this case we are given a

randomized solver and a formula such that the probability that the solver solves the instance within time t is $q_T(t)$. Assume now we are given given n simultaneously running solvers for solving the formula. As the formula is solved if at least one of the solvers solves the formula within time t, the probability of solving within time t becomes

$$q_{T^n}(t) = 1 - (1 - q_T(t))^n. \tag{4.1}$$

Depending on the distribution $q_T(t)$, the expected run time $\mathbb{E}T^n$ of the simple distribution approach can be be significantly lower than the expected run time $\mathbb{E}T$ of a single solver. This chapter studies this effect in distributions obtained from a wide range of formulas.

4.2 Parallel Restart Strategies

The properties of the SDSAT approach is studied in the grid computing environment discussed in Ch. 3 using a parallel adaptation of the restart strategies discussed in Sect. 2.2.2. This will be done by applying a sequential restart strategy STRAT to a work flow consisting of a sequence of jobs $j_1 j_2 \ldots$. A *finite restart strategy* is a sequence of restart limits $t_1 t_2 \ldots t_k$. Let the maximum run time of a job in the grid environment be $T_{\max}$. If the sum of the restart limits $\sum_{i=1}^{k} t_i \leq T_{\max}$, then a finite restart strategy can be executed in a job of a work flow. We will use the following construction for obtaining a finite restart strategy from a restart strategy $\text{STRAT} = t_1 t_2 \ldots$

$$\text{finite}(\text{STRAT}) = \begin{cases} T_{\max}, & \text{if } t_1 > T_{\max}, \text{ and} \\ t_1 t_2 \ldots t_k, & k \text{ maximizes } \sum_{i=1}^{k} t_i \leq T_{\max} \text{ otherwise.} \end{cases} \tag{4.2}$$

Given a restart strategy STRAT, let $\text{STRAT}_1, \text{STRAT}_2, \ldots$ be a recursively defined sequence of restart strategies so that $\text{STRAT}_1 = \text{STRAT}$ and the restart strategy STRAT_i is obtained by removing the restart limits given by $\text{finite}(\text{STRAT}_{i-1})$ from STRAT_{i-1}. A *parallel restart strategy* is the result of mapping a given restart strategy to finite restart strategies that are executed in jobs of a work flow. Two different mappings are used in [PI] to obtain parallel restart strategies. In the *faithful* parallelization scheme the mapping is done so that the job j_1 uses the restart strategy $\text{finite}(\text{STRAT}_1)$, the job j_2 uses $\text{finite}(\text{STRAT}_2)$ and so forth. The *straightforward* parallelization scheme assumes a grid environment with n computing resources. The scheme, introduced in [Luby and Ertel 1994], gives

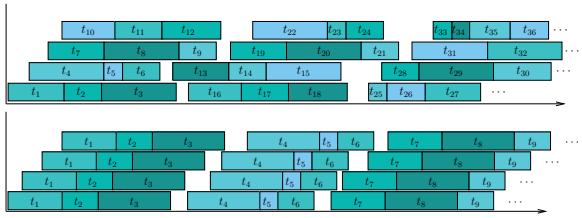


Figure 4.1. The faithful scheme used in the experiments (*top*) and the straightforward scheme (*bottom*)

the jobs $j_1 \dots j_N$ the finite restart strategy finite($\textsc{Strat}_1$), while the jobs $j_{N+1} \dots j_{2N}$ receive the restart strategy finite($\textsc{Strat}_2$) and so forth. Figure 4.1 illustrates the schemes. The experiments in [PI] suggest that the faithful scheme performs better than the straightforward scheme in most cases.

As discussed in Sect. 2.2.2 and [Luby et al. 1993], a restart strategy $\textsc{Opt} = (a^* a^* \dots)$ is optimal for a given run time distribution. It is relatively straightforward to come up with a distribution where a more elaborate restart strategy provides a better speed-up in the parallel case. The following example, adapted from [Luby and Ertel 1994], illustrates the phenomenon.

Example 1 *Given* $0 < p < 1$*, consider the distribution*

$$q(t) = \begin{cases} 0 & \text{if } t < 1, \\ p & \text{if } 1 \le t < 10, \text{ and} \\ 1 & \text{if } t \ge 10, \end{cases} \tag{4.3}$$

where the probability of solving the instance at time 1 is p*, and solving the instance at time 10 is* $(1-p)$*. Exactly two fixed strategies can be useful in solving this distribution,* $\textsc{Fixed}^1 = (1, 1, \dots)$ *and* $\textsc{Fixed}^{10} = (10, 10, \dots)$*. In an environment with* $n = 2$ *parallel computing elements, the expected run times for the two strategies turn out to be*

$$\mathbb{E}T^2_{\textsc{Fixed}^1} = (1-p)\sum_{i=0}^{\infty}(1-p)^{2i}(i+1) = \frac{1}{1-p^2}, \tag{4.4}$$

and

$$\mathbb{E}T^2_{\textsc{Fixed}^{10}} = 2p(1-p) + p^2 + 10(1-(2p(1-p)+p^2)) = 9p^2 - 18p + 10. \tag{4.5}$$

However, the restart strategy $S = (10, 1, 1, \dots)$*, having expected run time*

$$\mathbb{E}T^2_S = p\sum_{i=0}^{8}(1-p)^i(i+1) + 10(1-p\sum_{i=0}^{8}(1-p)^i), \tag{4.6}$$

results in lower run time than either one of the two fixed restart strategies for small values of p. For example when $p = 0.05$, we have the expected run times

$$\begin{aligned} \mathbb{E}T^2_{\text{FIXED}^1} &\approx 10.256, \\ \mathbb{E}T^2_{\text{FIXED}^{10}} &\approx 9.123, \textit{ and} \\ \mathbb{E}T^2_S &\approx 8.025. \end{aligned} \tag{4.7}$$

4.3 Experiments on Parallel Restart Strategies

This section studies the effect of using different restart strategies in the grid computing environment discussed in Ch. 3. The distributions used in the experiments are the same as those used in Sect. 2.3. The computing environment is assumed to be zero-delayed in the following experiments and the case with non-zero delays is further analyzed in [PI]. The distributions are computed using an Intel Xeon 5130 dual-core dual-CPU computers with 16 GB memory running MINISAT 1.14 so that a single computer is reserved for each solving. The resulting distributions are then used for simulating the SDSAT approach with the parallel restart strategies. The results are reported in tables 4.1 and 4.3 for 16 cores, and 4.2 and 4.4 for 64 cores. The table lists the following restart strategies in the columns:

- FIXED$^{T_{\max}}$ denotes the run time with the parallel restart strategy based on the fixed restart strategy FIXED$^{T_{\max}} = T_{\max} T_{\max} \ldots$.

- FIXED$^{a^*}$ shows the results with a restart strategy obtained by minimizing Eq. (2.5) over a substituting $q(t) = q_{T^n}(t)$ from Eq. (4.1). Determining a^* requires in practice solving the formula, and is therefore in practice never available. This restart strategy can be seen as an idealization.

- STRAT$^{\text{U}}$ denotes the universal restart strategy (see Eq. (2.6)),

- STRAT$^{\text{E}}$ denotes the exponential restart strategy,

- STRAT$^{\text{NE}}$ denote the nested exponential restart strategy (see Eq. (2.7))

- Exp denotes the values obtained directly from the expected solving time

when running n solvers in parallel with the SDSAT approach until one of them finds a solution.

- Min shows the minimum sampled run time. This value gives an estimate of the minimum time required to solve the instance with any restart strategy or number of cores.

The results show that many unsatisfiable formulas in this benchmark set are best solved with the exponential strategy. Surprisingly, the results are typically better with the exponential strategy than with the strategy FIXED^{a^*}, especially with 64 cores. The run time with the strategy FIXED^{a^*} is consistently equal to the expected run time. From this we may conclude that only the more elaborate restart strategies can provide better speed-up than the straightforward approach of running the solver until a solution is found. The speed-up obtained with the exponential restart strategy is greater with 16 cores than with 64 cores. Typically the run times are close to the minimum run time, indicating that not much speed-up can be obtained with increasing the number of CPUs.

Based on the results one could argue that by using the exponential restart strategy STRAT^{E} one would gain a small speed-up compared to using the fixed restart strategy $\text{FIXED}^{T_{\max}}$. Figure 4.2 shows scatter plots for the two restart strategies. Each point in the figures corresponds to an instance of the *application* category of the Satisfiability Competition 2009 (SAT-Comp 2009) run with 16 cores (left) and 64 cores (right). The vertical cordinate of each point is the wall clock run time for the restart strategy $\text{FIXED}^{T_{\max}}$ and the horizontal for STRAT^{E}. A point above the diagonal means that the strategy STRAT^{E} has a lower run time. The results are again reported without delays. Based on the figures no significant gain or loss is obtained by using the more complex restart strategy.

It is not clear how well these results generalize to multicore solving, as the experiments in this section do not consider any kind of memory bus congestion issues often experienced with SAT solving in particular. For example, [Martins et al. 2010] reports a 15% decrease in efficiency when running four threads in a quad-core CPU.

Table 4.1. Parallel restart strategies on unsatisfiable instances with 16 cores

Name	FIXED$^{T_{\max}}$	FIXED$^{\alpha^*}$	STRAT^{U}	STRAT^{E}	STRAT^{NE}	Exp	Min
999999000001nc	1218.79	1218.79	1645.78	**1201.90**	12050.84	1218.79	1071.51
AProVE07-03	**954.01**	**954.01**	1052.54	955.87	8668.53	954.01	923.89
AProVE07-08	**1074.13**	**1074.13**	1430.46	1082.87	10072.37	1074.13	774.47
AProVE07-09	2075.72	2075.22	4800.72	**2028.56**	20966.15	2075.22	1552.37
AProVE07-16	**1048.83**	**1048.83**	1250.86	1056.70	9877.37	1048.83	879.383
AProVE07-27	2989.44	2952.72	7816.30	**2929.53**	28988.91	2952.72	2681.72
clqcolor-10-07-09	**1292.80**	**1292.80**	1530.99	1308.50	12269.79	1292.80	1198.31
connm-ue-csp-sat-n800-d0.02-s925928766	**926.77**	**926.77**	1185.90	931.91	8650.57	926.77	847.79
cube-11-h13-unsat	**919.08**	**919.08**	1341.03	944.41	8931.40	919.08	703.41
dated-10-13-u	3060.44	**3022.28**	8529.79	3051.15	30307.07	3022.28	2634.31
dated-5-15-u	773.02	773.02	1071.38	**763.94**	7389.70	773.02	583.42
dated-5-17-u	**1430.65**	**1430.65**	2474.74	1461.95	14331.09	1430.65	1076.53
emptyroom-4-h21-unsat	3980.55	**3333.06**	15431.82	3346.65	33762.49	3333.06	2826.16
eq.atree.braun.11.unsat	2256.08	2256.08	3476.13	**2240.11**	21622.92	2256.08	1900.74
hwb-n26-01-S1957858365	**584.48**	**584.48**	662.95	586.72	5288.25	584.48	541.18
hwb-n26-03-S540351185	**1004.99**	**1004.99**	1112.83	1018.90	9150.24	1004.99	915.99
hwb-n28-01-S136611085	**1271.54**	**1271.54**	1379.97	1276.40	11843.20	1271.54	1222.56
hwb-n28-02-S818962541	20390.45	3839.98	120253.79	3824.65	37383.67	3839.98	3596.46
linvrinv5	**2346.70**	**2346.70**	2890.58	2365.55	22070.89	2346.70	2168.45
lksat-n1000-m6860-k4-l4-s1935114289	943.01	943.01	1079.81	**939.84**	8837.96	943.01	892.25
manol-pipe-f9b	3284.95	**2897.48**	13832.88	2911.97	26169.26	2897.48	1505.45
manol-pipe-f9n	5031.82	**3297.12**	21136.59	3362.23	31256.28	3297.12	2561.90
manol-pipe-g10nid	**564.88**	**564.88**	773.35	567.43	5334.36	564.88	452.97
mod2c-3cage-unsat-10-2	**2316.56**	**2316.56**	3200.24	2322.92	22186.57	2316.56	2100.62
mod2c-3cage-unsat-10-3	**1901.87**	**1901.87**	2643.14	1940.09	18202.98	1901.87	1806.69
phnf-size10-exclusive-luckySeven	**508.77**	**508.77**	693.87	522.80	4595.64	508.77	439.27
pmg-12-UNSAT	4202.59	**3408.44**	14307.67	3411.77	32730.85	3408.44	3201.25
pyhala-braun-unsat-40-4-02	**1942.92**	**1942.92**	2556.73	1952.30	18710.17	1942.92	1741.8
QG7-dead-dnd001	622.41	622.41	921.81	**606.25**	5758.97	622.41	438.34
QG7-dead-dnd002	**1076.71**	**1076.71**	1403.67	1101.27	10097.25	1076.71	877.98
QG7-gensys-icl100	**2513.79**	**2513.79**	4353.16	2519.19	24436.45	2513.79	2323.00
QG7-gensys-ukn003	**1287.10**	**1287.10**	1426.69	1297.45	12058.16	1287.10	1207.62
s101-100	1870.68	1870.68	2385.08	**1866.75**	18510.20	1870.68	1646.75
s97-100	**1593.43**	**1593.43**	1858.12	1596.60	15218.74	1593.43	1425.26
SGI_30_50_30_20_1-dir	**772.58**	**772.58**	851.78	781.07	6957.91	772.58	720.65
SGI_30_50_30_20_3-dir	**1180.57**	**1180.57**	1303.48	1191.67	10775.70	1180.57	1076.47
sortnet-6-ipc5-h11-unsat	5247.16	**3428.12**	21159.76	3500.82	33392.97	3428.12	3081.37
total-10-13-u	**1701.67**	**1701.67**	3474.52	1742.68	16132.87	1701.67	1189.01
unsat-set-b-fclqcolor-10-07-09	1339.28	1339.28	1649.27	**1338.32**	12873.22	1339.28	1011.96
uts-l06-ipc5-h33-unknown	**488.64**	**488.64**	720.14	498.83	4699.11	488.64	394.69
Total time	89989.9	**67981.2**	162196	68348.5	652561	67981.2	58194.0

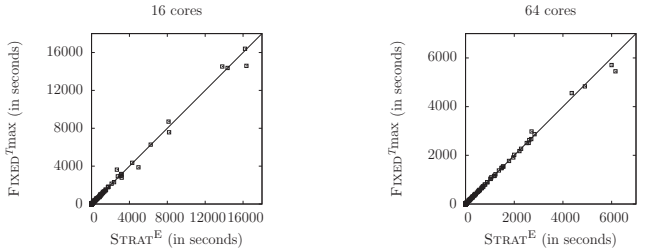


Figure 4.2. Scatter plots for the exponential strategy STRAT^{E} (on the horizontal axis) and the fixed strategy FIXED$^{T_{\max}}$ (on the vertical axis) for 16 (*left*) and 64 (*right*) cores. Each point on the plot corresponds to one instance from the *application* category of SAT-Comp 2009

Table 4.2. Parallel restart strategies on unsatisfiable instances with 64 cores

Name	FIXED$^{T_{\max}}$	FIXED$^{\alpha^*}$	STRAT^{U}	STRAT^{E}	STRAT^{NE}	Exp	Min
999999000001nc	1097.30	1097.30	1217.53	**1091.74**	3635.46	1097.30	1071.51
AProVE07-03	**929.81**	**929.81**	954.39	930.75	2693.13	929.81	923.89
AProVE07-08	**868.46**	**868.46**	1070.43	873.75	3071.37	868.46	774.47
AProVE07-09	1685.02	1685.02	2187.87	**1669.81**	5996.16	1685.02	1552.37
AProVE07-16	919.55	919.55	1036.74	**917.59**	3051.46	919.55	879.383
AProVE07-27	**2724.18**	**2724.18**	3187.54	2727.98	8889.88	2724.18	2681.72
clqcolor-10-07-09	**1223.94**	**1223.94**	1294.52	1222.36	3738.15	1223.94	1198.31
connm-ue-csp-sat-n800-d0.02-s925928766	**859.52**	**859.52**	933.19	861.69	2776.68	859.52	847.79
cube-11-h13-unsat	**770.53**	**770.53**	917.82	774.79	2665.47	770.53	703.41
dated-10-13-u	**2715.90**	**2715.90**	3294.69	2725.08	9305.16	2715.90	2634.31
dated-5-15-u	621.97	621.97	752.72	**612.49**	2217.66	621.97	583.42
dated-5-17-u	1173.10	1173.10	1442.47	**1161.43**	4110.50	1173.10	1076.53
emptyroom-4-h21-unsat	2981.42	2978.67	4977.42	**2965.05**	9997.88	2978.67	2826.16
eq.atree.braun.11.unsat	**2002.87**	**2002.87**	2316.82	2004.98	6692.66	2002.87	1900.74
hwb-n26-01-S1957858365	550.54	550.54	583.36	**548.33**	1654.99	550.54	541.18
hwb-n26-03-S540351185	936.17	936.17	1005.70	**933.93**	2893.18	936.17	915.99
hwb-n28-01-S136611085	1229.79	1229.79	1270.72	**1228.67**	3700.24	1229.79	1222.56
hwb-n28-02-S818962541	6661.30	3623.84	25475.02	3635.29	11334.12	3623.84	3596.46
linvrinv5	2207.00	2207.00	2397.34	**2201.85**	6860.62	2207.00	2168.45
lksat-n1000-m6860-k4-l4-s1935114289	897.91	897.91	940.98	**897.50**	2760.81	897.91	892.25
manol-pipe-f9b	2039.18	2038.45	3798.45	**1986.08**	7189.72	2038.45	1505.45
manol-pipe-f9n	2727.88	2672.87	7051.39	**2623.65**	9737.61	2672.87	2561.90
manol-pipe-g10nid	477.56	477.56	537.69	**476.39**	1577.65	477.56	452.97
mod2c-3cage-unsat-10-2	**2133.11**	**2133.11**	2403.48	2145.82	6850.56	2133.11	2100.62
mod2c-3cage-unsat-10-3	1817.37	1817.37	1914.53	**1816.27**	5612.06	1817.37	1806.69
phnf-size10-exclusive-luckySeven	456.49	456.49	499.86	**454.87**	1440.11	456.49	439.27
pmg-12-UNSAT	3243.91	3239.33	5598.29	**3224.99**	10253.94	3239.33	3201.25
pyhala-braun-unsat-40-4-02	**1779.49**	**1779.49**	1993.90	1780.19	5806.67	1779.49	1741.8
QG7-dead-dnd001	489.39	489.39	617.80	**485.33**	1680.44	489.39	438.34
QG7-dead-dnd002	**908.27**	**908.27**	1082.50	915.85	3113.83	908.27	877.98
QG7-gensys-icl100	2371.10	2371.10	2598.43	**2366.25**	7452.06	2371.10	2323.00
QG7-gensys-ukn003	**1222.60**	**1222.60**	1271.92	1223.60	3769.13	1222.60	1207.62
s101-100	**1714.32**	**1714.32**	1897.07	1716.77	5683.92	1714.32	1646.75
s97-100	1480.41	1480.41	1595.93	**1472.03**	4647.91	1480.41	1425.26
SGI_30_50_30_20_1-dir	733.02	733.02	774.79	**732.54**	2197.40	733.02	720.65
SGI_30_50_30_20_3-dir	**1115.71**	**1115.71**	1175.65	1119.01	3411.16	1115.71	1076.47
sortnet-6-ipc5-h11-unsat	3175.41	3120.33	6918.18	3102.86	10402.33	3120.33	3081.37
total-10-13-u	1344.92	1344.92	1706.18	**1326.45**	4913.85	1344.92	1189.01
unsat-set-b-fclqcolor-10-07-09	**1133.14**	**1133.14**	1345.40	1158.07	3895.12	1133.14	1011.96
uts-l06-ipc5-h33-unknown	414.50	414.50	477.63	**409.41**	1365.39	414.50	394.69
Total time	63834.1	60678.4	72989.8	**60521.5**	199046	60678.4	58194.0

Table 4.3. Parallel restart strategies on satisfiable instances with 16 cores

Name	FIXED$^{T_{\max}}$	FIXED$^{\alpha^*}$	STRAT^{U}	STRAT^{E}	STRAT^{NE}	Exp	Min
cube-11-h14-sat	2926.15	2891.47	7905.75	**2877.41**	28288.35	2891.47	2628.68
dated-10-13-s	64.18	48.55	61.19	70.02	**59.60**	64.18	10.09
dated-10-17-s	116.60	75.21	92.52	131.00	**94.27**	116.60	8.06
emptyroom-4-h22-sat	2829.24	**2236.62**	5503.84	2735.48	7297.10	2417.49	393.28
mizh-md5-48-5	133.88	133.88	239.64	**133.12**	610.57	133.88	49.76
mizh-sha0-35-3	**30.62**	**30.62**	30.68	30.80	96.92	30.62	23.43
mizh-sha0-36-2	87.21	**68.92**	94.95	77.33	155.29	87.21	25.65
mod2-rand3bip-sat-250-3	116.29	105.73	179.73	**102.73**	361.95	116.29	40.16
mod2-rand3bip-sat-280-1	84.61	**63.11**	76.03	100.67	87.40	84.61	9.18
sortnet-7-ipc5-h16-sat	2062.73	**1071.07**	2740.82	1759.80	3082.93	1965.84	156.55
vmpc_28	7.30	**0.86**	7.84	9.17	5.10	7.30	0.14
Total time	8458.81	**6726.04**	16933	7924.8	39777.5	7799.2	3344.98

Table 4.4. Parallel restart strategies on satisfiable instances with 64 cores

Name	FIXED$^{T_{\max}}$	FIXED$^{a^*}$	STRAT^{U}	STRAT^{E}	STRAT^{NE}	Exp	Min
cube-11-h14-sat	**2682.79**	**2682.79**	3168.93	2697.59	8746.08	2682.79	2628.68
dated-10-13-s	16.45	**16.00**	16.90	16.79	17.77	16.45	10.09
dated-10-17-s	32.61	**20.84**	27.45	35.32	28.82	32.61	8.06
emptyroom-4-h22-sat	709.36	666.96	1559.06	**657.61**	2120.44	708.95	393.28
mizh-md5-48-5	73.64	73.64	88.59	**73.48**	176.45	73.64	49.76
mizh-sha0-35-3	24.10	24.10	24.20	**23.93**	29.95	24.10	23.43
mizh-sha0-36-2	30.12	30.12	33.01	**28.70**	56.13	30.12	25.65
mod2-rand3bip-sat-250-3	47.37	47.37	57.13	**45.76**	104.98	47.37	40.16
mod2-rand3bip-sat-280-1	21.77	**19.56**	22.82	22.35	22.58	21.77	9.18
sortnet-7-ipc5-h16-sat	404.61	**319.82**	523.72	464.86	857.29	404.78	156.55
vmpc_28	0.55	**0.26**	0.63	0.59	0.55	0.55	0.14
Total time	4043.37	**3901.46**	5522.44	4066.98	12161	4043.13	3344.98

4.4 Clause Learning with Simple Distributed SAT Solving

Based on the experiments in the previous section it seems that when solved with a randomized modern clause learning SAT solver, many formulas have a relatively high minimum solving time. A grid computing environment which places hard limits on solving times cannot therefore be used for solving some formulas with a given SAT solver and the techniques based solely on randomization and restarts. On the other hand, if a formula can be solved in the environment, the parallelism potentially results in substantial speed-up.

This section studies an improvement over the SDSAT approach where the clauses learned in an earlier job which reached a run time or memory limit are used in successor jobs. The learned clauses are transferred to a *clause database* stored in the master process, where they are filtered using a *parallel clause learning heuristic*, and then submitted with the formula on the subseqeuent jobs. It turns out that this *CL-SDSAT* approach improves significantly the underlying solver. In practice it is possible to solve some formulas which could not be solved in reasonable time or memory limits without this technique. However, in some cases the new learned clauses can slow down the solving by increasing the overhead related to memory access of the solver.

Sharing of the learned clauses plays a central role in the discussion. The size of a clause set $||S||$ is the total number of literals in S, that is, $||S|| = \sum_{C \in S} |C|$. The unit clauses are handled specially in the process: they are always stored in the clause database, and do not contribute to the size of the database.

Figure 4.3 shows a version of the CL-SDSAT algorithm and the related concepts. The clause database, initialized on line 1, is denoted by

$ClauseDB$, and is annotated with an index j to facilitate the representation of the results. The set U contains the unit clauses that are proved true in all satisfying truth assignments of the input formula ϕ, if any exist. The shorthand notation $\mathrm{UP}(\phi) = \mathrm{UP}(\phi, \emptyset)$ denotes computing the unit propagation closure of ϕ on empty truth assignment.

The first part of the loop in lines 5–6 consists of submitting the formula, all unit clauses and a heuristically selected subset of $ClauseDB$ of size at most $SubmSize$ to the grid so that the n computing resources are filled. The next phase is to receive the results in lines 8–14. The $Receive(i)$ receives from the resource i a tuple consisting of the result of the computing, which can be Sat, Unsat or Indet, and a set L of learned clauses. If the formula is found either satisfiable or unsatisfiable, the algorithm is terminated. Otherwise the set of unit clauses is updated using the learned clauses on line 13 and the clause database updated on line 14, again using a heuristic function $Merge$ and limiting the maximum size of the database to $MaxDBSize$.

The function $Merge$ has a central role in discussing clause sharing both here and later in Sect. 6.2. Firstly, the function acts as a heuristic for selecting learned clauses, and secondly, it simplifies the learned clauses using the set of literals U obtained by unit propagation. Two operations are involved in the simplification:

(i) removing satisfied clauses (clauses C such that $C \cap U \neq \emptyset$), and

(ii) removing the false literals $\neg l$ from clauses so that given a clause C, the simplified clause becomes $C' = \{l \in C \mid \neg l \notin U\}$.

4.5 Experiments on the Algorithmic Framework

It is interesting to contemplate on the different types of heuristics that can be implemented both for $Choose$ and $Merge$. This section studies the following four possibilities discussed also in [PII]:

- $Choose_{123}$ only considers clauses of length 1, 2, or 3. If the size of the resulting database is greater than the limit, the shorter clauses are preferred. This type of approach is used in many portfolio based solvers. For example, [Biere 2010] only transfers clauses of length 1 to other solvers, and [Hamadi et al. 2009b] only clauses that have at most eight literals.

Input: ϕ, a propositional formula;
n, number of cores;
$MaxDBSize$, the maximum size for the database;
$SubmSize$, the maximum submit size

1 $ClauseDB^0 := \emptyset$
2 $U := \mathrm{UP}(\phi)$
3 $j := 0$
4 **while** True:
5 **for** $i := 1$ to n:
6 $Submit(\phi \cup U \cup Choose(ClauseDB^j, SubmSize))$
7 $ClauseDB^{j+1} := ClauseDB^j$
8 **for** $i := 1$ to n:
9 $(result, L) := Receive(i)$
10 **if** *result* is in $\{\mathrm{Sat}, \mathrm{Unsat}\}$:
11 **return** *result*
12 **else** :
13 $U := \mathrm{UP}(\phi \cup U \cup ClauseDB^{j+1} \cup L)$
14 $ClauseDB^{j+1} := Merge(U, ClauseDB^{j+1}, L, MaxDBSize)$
15 $j := j + 1$

Figure 4.3. The CL-SDSAT Algorithmic Framework

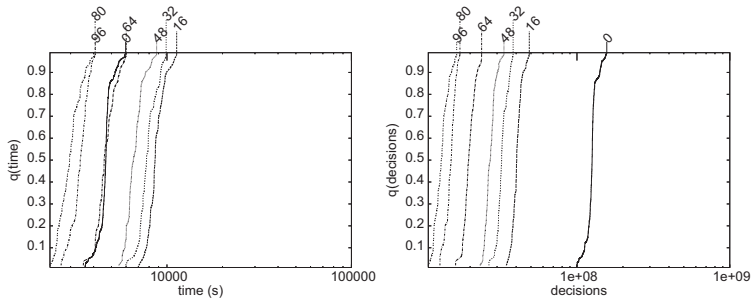


Figure 4.4. Run time distribution for $n = 0, 16, 32, 48, 64, 80,$ and 96 parallel cores for instance *hwb-n28-02-S818962541* (left). The expected run time with the learned clauses is higher than the expected run time without the learned clauses for $n = 16$, $n = 32$ and $n = 48$. However, the decision distribution (right) does not show a similar slow-down.

- $Choose_{\text{len}}$ returns the shortest clauses. This approach is more general than $Choose_{123}$, as it always returns clauses even if the argument set contains only clauses longer than some limit.

- $Choose_{\text{freq}}$ returns the most common learned clauses. As the parallel search is allowed to overlap, it is not unlikely that the same clause is learned many times in different jobs.

- $Choose_{\text{rand}}$ returns a randomly selected set of clauses.

We can now move to the first analysis of the algorithmic framework described in Fig. 4.3. The idea is to see how the database of clauses $ClauseDB$ affects the expected run time and number of decisions needed to solve a formula. In particular, we are studying the effect of $ClauseDB^1$ while $n = 8$. Table 4.5, adapted from [PII], illustrates the effect of the heuristic when $SubmSize = 100000$ literals and $MaxDBSize$ is unlimited. The database is constructed by running the $n = 8$ solvers for a time corresponding to 25% of the previously measured minimum run times. Once the resulting learned clauses are merged and simplified in line 14, the formula $\phi \cup U \cup Choose(ClauseDB^1, SubmSize)$, constructed in line 6, is solved using a randomized solver (MINISAT v1.14) 50 times to obtain a reliable estimate of the run time distribution.

Based on the results, the fixed clause length heuristic $Choose_{123}$ is the best performing heuristic, while the heuristic $Choose_{\text{len}}$ gives almost no reduction in run time, loosing to the random heuristic $Choose_{\text{rand}}$. We still note that the heuristic $Choose_{\text{len}}$ performs very well when measuring the number of decisions.

Table 4.5. Expected run times for a selection of benchmarks from the SAT 2007 competition

Name	Base	$Choose_{\text{len}}$	$Choose_{\text{freq}}$	$Choose_{123}$	$Choose_{\text{rand}}$
AProVE07-09	4 016	**1 994**	2 616	2 264	3 393
	8 461 866	**4 388 463**	4 716 035	5 532 927	7 451 391
eq.atree.-	3 096	2 967	2 152	**1 439**	2 481
braun.11.unsat	22 311 255	**7 831 761**	13 263 105	9 034 391	14 404 941
SGI_30_50_30_20_3-	1 432	**70**	485	211	343
dir	1 240 001	**165 721**	541 943	357 978	467 458
cube-11-h14-sat	4 832	**4 483**	4 939	4 888	5 294
	1 273 485	**967 851**	1 096 322	1 238 110	1 313 385
dated-10-11-u	9 889	2 037	**1 977**	2 187	5 240
	1 639 566	1 058 664	**998 103**	1 146 487	2 246 003
emptyroom-4-	5 205	**1 498**	1 631	1 704	1 954
h21-unsat	1 885 355	**688 156**	813 027	853 642	1 052 777
unsat-set-b-	2 027	**1 153**	1 388	1 196	1 864
fclqcolor-10-07-09	41 172 989	**13 696 945**	29 946 390	25 945 033	26 103 961
hwb-n28-02-	4 654	14 128	5 001	**4 454**	10 211
S818962541	125 472 477	**68 950 042**	123 220 119	97 041 128	82 550 196
linvrinv5	2 828	7 837	2 620	**2 518**	4 030
	40 917 769	**25 824 068**	37 369 017	36 283 860	32 008 217
manol-pipe-f9b	10 620	13 336	9 196	**7 120**	10 814
	4 954 967	5 308 314	4 328 594	**3 401 500**	5 101 791
mod2c-3cage-	3 020	3 827	2 659	**2 496**	4 392
unsat-10-2	271 766 780	**62 714 188**	221 568 484	195 269 018	87 430 751
pmg-12-UNSAT	4 268	9 372	4 189	**2 955**	7 876
	84 245 813	**40 690 352**	69 882 275	48 750 743	56 061 825
pyhala-braun-	2 641	887	1 086	**782**	1 348
unsat-40-4-02	2 775 304	**1 001 999**	1 855 329	1 436 653	2 245 269
QG7-gensys-	1 594	760	1 196	**513**	1 506
ukn003	6 799 632	**2 081 121**	5 256 338	2 737 811	5 436 088
s101-100	2 528	5 047	2 502	**2 428**	4 907
	170 749 796	47 196 913	167 440 762	166 645 578	**46 054 481**
sortnet-6-ipc5-	4 886	1 521	2 893	**1 507**	4 694
h11-unsat	2 743 833	**900 265**	1 842 295	980 166	2 607 584
total-10-13-u	3 279	1 296	**1 109**	1 695	1 722
	1 178 947	690 406	**682 302**	998 194	997 008
Sum	73 383	72 213	47 639	**40 357**	72 069
	789 589 835	**284 378 607**	684 820 440	597 653 219	373 533 126

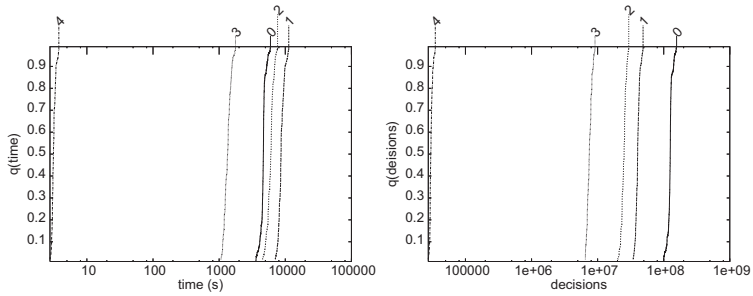


Figure 4.5. Effect of conjoining the clause database $ClauseDB^j$ with an instance for $j = 0, 1, 2, 3$ and 4. Increasing the "depth" of the clause database makes this instance faster to solve once $j \geq 3$.

The results in Table 4.5 show that it is not uncommon that conjoining learned clauses with a formula increases the expected run times. Figure 4.4 shows an example, using the $Choose_{\text{len}}$ heuristic for selecting the clauses. The figure shows the effect of increasing the number of parallel cores n in the CL-SDSAT algorithmic framework. The instance is on the average slower to solve when $n < 64$, although the number of decisions needed to solve the instance decreases monotonously. A likely explanation for this is that the large amount of learned clauses results in a higher memory footprint slowing down the solving more than what is gained from the decrease in decisions. For practical reasons the size of the learned clause database $MaxDBSize$ was limited to ten million literals in these experiments, while $SubmSize = 100000$.

It is also interesting to study the effect of increasing the "depth" of the learned clause database, that is, the effect of increasing j in $ClauseDB^j$. Higher values of j signify that the clauses learned in parallel can be used to derive more clauses. Figure 4.5 shows the effect for $n = 16$ for the formula studied also in Fig. 4.4. Clearly this type of cumulative learning performs significantly better than increasing only the number of simultaneously running solvers. The phenomenon is confirmed for several other formulas in [PII], and will play a key role in the development of cumulative learning for the iterative partitioning approach in Ch. 6.

4.6 The CL-SDSAT Implementation

As a part of this work we have implemented the CL-SDSAT algorithm for the grid computing environment discussed in Ch. 3. The implementation of the CL-SDSAT algorithmic framework limits the database size

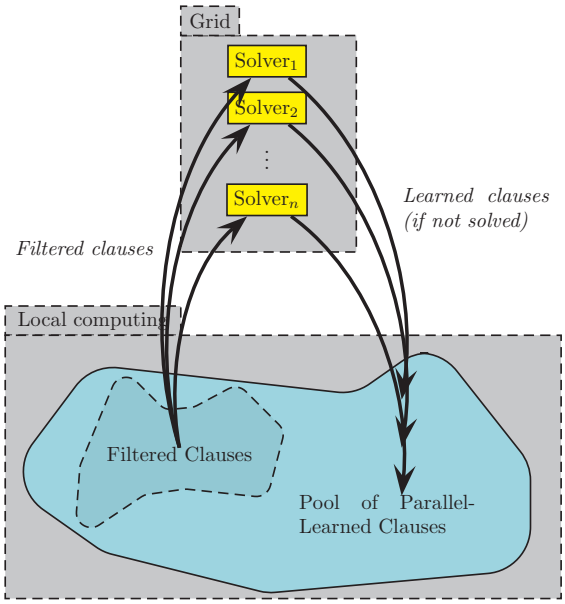


Figure 4.6. The CL-SDSAT Process

MaxDBSize to one million literals, using the $Choose_{\text{len}}$ heuristic to choose the clauses once the limit is reached. The outgoing clauses are selected using the same heuristic, and the size of the set is limited to 100 000 literals. Unlike in the algorithmic framework of Fig. 4.3, the implementation does not wait for all the parallel running solvers to time out before submitting the new solvers to the freed resources. This decision was taken since the delays in the grid environment vary (see Fig. 3.2), and some solvers might therefore finish much earlier than the less lucky solvers. As a result, the related work flow could be described as the process in Fig. 4.6, where a clause database is maintained in the master process using heuristics and jobs are constructed using the database available at the time a resource becomes available.

The CL-SDSAT algorithm is based on MINISAT 2.2.0. The solver is used in two ways; firstly as the clause-producing solver in the workers, and secondly in the master process for handling the simplification, and removing subsumed and duplicate learned clauses. Figure 4.7 compares the run time of MINISAT 2.2.0 based CL-SDSAT approach to MINISAT 2.2.0 for the application category instances of SAT-Comp 2009. The times reported for CL-SDSAT include the grid delays, job run time was limited to approximately one hour, memory usage was limited to 2 gigabytes and at most 64 cores were used from the grid simultaneously. The work flow run time was limited to 6 hours. The times reported for MINISAT 2.2.0 are com-

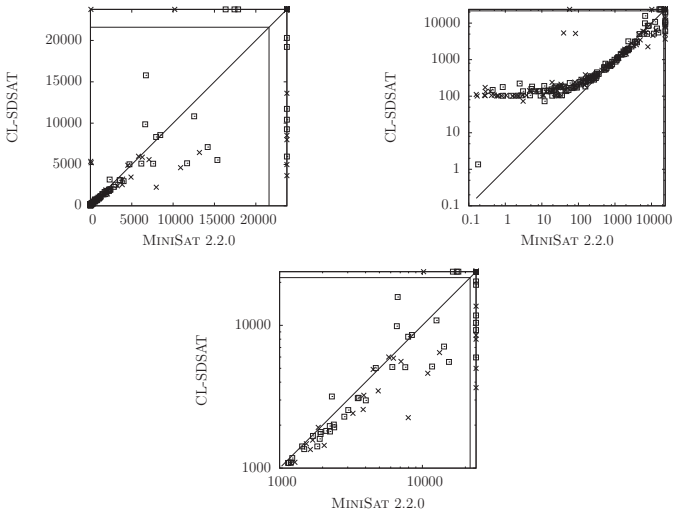


Figure 4.7. Comparison of the CL-SDSAT approach against MiniSat 2.2.0. The boxes (□) represent unsatisfiable and the crosses (×) satisfiable instances. The right figure shows the data in logarithmic scale, whereas the left figure uses linear scale. The bottom figure is a zoom into the right figure.

puted in a 12 core AMD Opteron 2435 system. The full node was reserved for each run to prevent other processes causing memory bus congestion. Time was limited to 6 hours and memory usage to 24 gigabytes.

Despite the high latency the CL-SDSAT approach seems to perform well in solving instances and manages to solve several instances that were not solved with MiniSat 2.2.0, as illustrated by the time outs on the borders of the graphs. The slowdown in easy instances is particularly visible in the right graph using logarithmic scale. As the difficulty of the instances increases, also the gain from the CL-SDSAT algorithm becomes clearly visible. Based on the results it would be particularly interesting to study the effect of adding to the clause database the frequently occurring clauses, and using more information, such as the *Literals Blocks Distance* defined in [Audemard and Simon 2009], for assessing the quality of the clauses. However, this is left to further work at this point.

5. Parallel Solving Based on Partitioning

Parallel SAT solving has in the past relied largely on methods where the search space is "forcibly" partitioned into non-overlapping searches. This approach, used for example in [Speckenmeyer 1989; Böhm and Speckenmeyer 1996; Zhang et al. 1996], is natural when using traditional DPLL-style solvers where the search is organized as a tree.

The SDSAT and CL-SDSAT approaches described in the preceding chapter do not force the solvers to perform different searches on the formula, but instead rely on randomization in the heuristic to provide speed-up. The idea in these portfolio approaches is that it is unlikely that two randomized solvers would be searching the solution in a similar fashion.

The differences between the partitioning and portfolio approaches have been actively studied [Bonacina 1999; Bonacina 2000; Grama and Kumar 1999; Bordeaux et al. 2009]. The most recent results in particular from SAT competitions suggest that the portfolio based approach performs better in practice [Hamadi et al. 2009b; Hamadi et al. 2009a; Biere 2010]. Recently also approaches which combine elements from both partitioning and portfolios have received some interest [Bonacina 2001; Segre et al. 2002; Hyvärinen et al. 2006; Dequen et al. 2009; Ohmura and Ueda 2009; Gebser et al. 2011].

This chapter discusses several approaches to avoiding search overlap with stronger means than just relying on probability. The approaches are based on inserting additional constraints to a formula resulting in two or more *derived formulas*. The constraints, represented either as conjunctions of clauses or as partial truth assignments, are constructed so that solving sufficient number of the derived formulas allows determining the satisfiability of the original formula. The chapter presents several approaches to organizing the partitioning based search, and analyzes the effects of the approaches to the expected time required to determine the

satisfiability of a formula.

The chapter combines this topic, covered in [PIII] and [PIV], under a single discussion. While the scope of [PIII] in particular is in both satisfiable and unsatisfiable formulas, the main emphasis here is to develop a uniform notation for the unsatisfiable formulas. Also other topics, covered in more detail in the publications, are considered more lightly here. In particular, the discussion on the *lookahead partitioning function* in Sect. 5.5 is covered in much more detail in [PIV], and the experimental results for the studied partitioning approaches are given in [PIII].

5.1 Plain Partitioning

The basic idea in the *plain partitioning approach* is quite simple: a propositional formula ϕ is divided to n derived formulas $\phi_1, \ldots, \phi_n$ that are solved in parallel with a SAT solver S called the *underlying* solver of the approach. The derived formulas are obtained with a *partitioning function* and satisfy the following conditions:

(1) $\phi \equiv \phi_1 \vee \ldots \vee \phi_n$, and

(2) $\phi_i \wedge \phi_j$ is unsatisfiable if $i \neq j$.

If all the derived formulas are unsatisfiable, then ϕ is also unsatisfiable, whereas if at least one of the derived formulas is satisfiable, also ϕ is satisfiable. Of particular interest in this section is how much faster a given formula can be solved with the plain partitioning approach compared to solving the formula directly with the solver S.

As the idea in plain partitioning is quite fundamental, it is natural that many parallel SAT solvers, such as [Zhang et al. 1996; Blochinger et al. 2003; Schubert et al. 2009; Schulz and Blochinger 2010], use a similar idea as their basis. The solving approaches used in these differ from plain partitioning, for example, by the use of load balancing, where new derived formulas are constructed from formulas being solved as the satisfiability of previous formulas is determined. As a result, the number of derived formulas n is not fixed in these parallel SAT solvers.

Despite such differences, an analysis of the plain partitioning approach gives insight also to practical parallel solving. The main result in this section is that the plain partitioning approach is “risky” in the following sense. Assume that for any cumulative probability distribution $q(t)$ there exists a formula ϕ_q such that the probability of solving ϕ_q with S in time

less than or equal to t is $q(t)$. If the partitioning function is from a certain natural class described in Def. 2, and n is fixed and sufficiently large, there is always an unsatisfiable formula so that the expected run time of the plain partitioning approach will be higher than the expected run time of the underlying solver S. The result is a generalization of a result in [PIII] stating that if the derived formulas are exactly as difficult as the original formula, the expected run time of the plain partitioning is never lower than that of the underlying solver.

The approach is analyzed in a spirit similar to the analysis of the portfolio style SDSAT approach in Ch. 4. In particular, we will assume that given a formula, the time required to determine its satisfiability with a solver S is a random variable T with cumulative distribution $q_T(t)$. To simplify the discussion, we will assume for now that given a number $n \geq 2$, the partitioning function produces n derived instances which are all solved in parallel using n CPUs or cores.

We will first introduce a model describing how a partitioning function affects the run time distributions of the derived formulas. We assume that the solver S performs with the same probability a given search that takes time t_ϕ in the formula ϕ but, due to the partitioning constraints, a shorter time t_{ϕ_i} in the derived formulas ϕ_i. The efficiency $\varepsilon(n) = t_\phi / t_{\phi_i}$ of the partitioning function is assumed to depend only on the number n of derived formulas. This reasoning results in a model where, given a formula with the run time distribution $q_T(t)$ on a solver S, the n derived formulas all have the distributions $q_T(\varepsilon(n)t)$.

The efficiency model that will be used in the proof is $\varepsilon(n) = n^\alpha$, where $0 \leq \alpha \leq 1$ is a constant depending on the partitioning function. This model can be motivated in two ways. Firstly, the efficiency satisfies the following natural properties:

(1) $1 \leq \varepsilon(n) \leq n$,

(2) $\varepsilon(n) \leq \varepsilon(n+1)$, and

(3) $(\varepsilon(n))^p = \varepsilon(n^p)$ for all $p \in \mathbb{N}$

The first condition states that the partitioning function should not make a particular search of S superlinearly faster or slow the search down. The second condition requires that the efficiency does not decrease as more derived formulas are created. The last condition states that if a partitioning function $P(\phi, n)$ is used to produce n^p derived formulas recursively, the resulting efficiency must equal the efficiency of $P(\phi, n^p)$ where the derived formulas are all generated at once.

Secondly, the model $\varepsilon(n) = n^\alpha$ can be derived from the following constructive application of partitioning. Assume there is a procedure for splitting the search space of an arbitrary formula ϕ following the run time distribution $q_T(t)$ to a fixed number $n_0 \geq 2$ of derived formulas $\phi_1, \ldots, \phi_{n_0}$. Assume further that the derived formulas ϕ_i have run time distributions $q_T(\beta t)$ where $1 \leq \beta \leq n_0$. Applying this procedure first to ϕ and then recursively to the derived formulas i times in total results in $n = n_0^i$ derived formulas with run time distribution $q_T(\beta^i t)$. Hence the recursive application of the procedure results in a partitioning function $P(\phi, n)$ defined for values $n = n_0^i$ with efficiency β^i. Since $i = \log_{n_0} n$, we have

$$\beta^i = \beta^{\log_{n_0} n} = e^{\frac{\ln n}{\ln n_0} \ln \beta} = (e^{\ln n})^{\frac{\ln \beta}{\ln n_0}} = n^{\frac{\ln \beta}{\ln n_0}} = n^\alpha,$$

where $\alpha = \ln \beta / \ln n_0$.

Alternative expressions for the efficiency include a linear model $\varepsilon'(n) = \max(\beta n, 1)$, where $0 \leq \beta \leq 1$ is a constant. However, the condition (3) does not hold for $\varepsilon'(n)$. For example setting $\beta = 0.9$, $n = 2$ and $p = 2$ results in $(\varepsilon'(2))^2 = 3.24$, while $\varepsilon'(4) = 3.6$.

We are now ready to define the partitioning function more precisely.

Definition 2 *Given a formula ϕ with run time distribution $q_T(t)$ on solver S and a partitioning factor $n \geq 2$, a* partitioning function $P : (\phi, n) \mapsto (\Pi_1, \ldots, \Pi_n)$ *is a function mapping the formula ϕ to n* partitioning constraints $\Pi_1, \ldots, \Pi_n$. *The partitioning constraints can be used to produce n* derived formulas $\phi_i = \phi \wedge \Pi_i, 1 \leq i \leq n$. *The derived formulas then satisfy the following two properties:*

(i) $\phi \equiv \phi_1 \vee \ldots \vee \phi_n$, *and*

(ii) $\phi_i \wedge \phi_j$ *is unsatisfiable for all* $i \neq j$.

The run time distribution of each of the derived formulas on solver S is described by the probability distribution $q_T(\varepsilon(n)t)$, where

$$\varepsilon(n) = n^\alpha, 0 \leq \alpha \leq 1 \tag{5.1}$$

describes the efficiency *of the partitioning function.*

We will denote by $\mathbb{E}T^n_{\text{plain-part}(\alpha)}$ the expected time required to determine the satisfiability of ϕ with the plain partitioning approach using a partitioning function with efficiency $\varepsilon(n) = n^\alpha$. A partitioning function is called

void if $\alpha = 0$ and hence $\varepsilon(n) = 1$. In this case all the derived instances are as difficult to solve as the original formula. A partitioning function is called *ideal* if $\alpha = 1$, that is, $\varepsilon(n) = n$.

Once the partitioning function is defined, we are now ready to show the first part of our main result stating that for non-ideal partitioning functions there are distributions where solving with plain partitioning is slower than solving with the underlying solver.

Proposition 5 *Let $P(\phi, n)$ be a partitioning function as in Def. 2, where $0 \leq \alpha < 1$, and S a SAT solver. Then for every n and every α there exists a distribution $q_n(t)$ such that if the solving of an unsatisfiable instance follows $q_n(t)$ on S, then the expected run time $\mathbb{E}T$ of S is lower than the expected run time $\mathbb{E}T^n_{\textit{plain-part}(\alpha)}$ of the plain partitioning approach.*

Proof. The family of distributions $q_n(t)$ we will use in the proof is

$$q_n(t) = \begin{cases} 0 & \text{if } t < t_1, \\ 1 - \frac{1}{n} & \text{if } t_1 \leq t < t_2, and \\ 1 & \text{if } t \geq t_2, \end{cases} \tag{5.2}$$

where $t_1 < t_2$. Thus the probabilities that the formula is solved by S exactly in time t_1 is $1 - 1/n$ and in time t_2 is $1/n$. The expected run time for a formula following the distribution $q_n(t)$ on S is

$$\mathbb{E}T = (1 - \frac{1}{n})t_1 + \frac{1}{n}t_2. \tag{5.3}$$

The expected run time of the plain partitioning approach using the partition function $\varepsilon(n) = n^\alpha$ can be derived by noting that all derived formulas need to be solved before the result can be determined. This means that either all solvers are "lucky", and determine the unsatisfiability in time t_1/n^α, or at least one of the solvers runs for time t_2/n^α, which will then become the run time of the approach. This results in

$$\mathbb{E}T^n_{\text{plain-part}(\alpha)} = \left(1 - \frac{1}{n}\right)^n \frac{t_1}{n^\alpha} + \left(1 - (1 - \frac{1}{n})^n\right) \frac{t_2}{n^\alpha}. \tag{5.4}$$

We claim that for every α, there are values for n, t_1 and t_2 such that $\mathbb{E}T < \mathbb{E}T^n_{\text{plain-part}(\alpha)}$. Dividing both sides of the resulting inequality by t_2 and setting $k = t_1/t_2$ results in

$$(1 - \frac{1}{n})k + \frac{1}{n} < \frac{(1 - \frac{1}{n})^n}{n^\alpha}k + \frac{1 - (1 - \frac{1}{n})^n}{n^\alpha},$$

which can be reordered to

$$k\left((1 - \frac{1}{n}) - \frac{(1 - \frac{1}{n})^n}{n^\alpha}\right) < \frac{1 - (1 - \frac{1}{n})^n}{n^\alpha} - \frac{1}{n}.$$

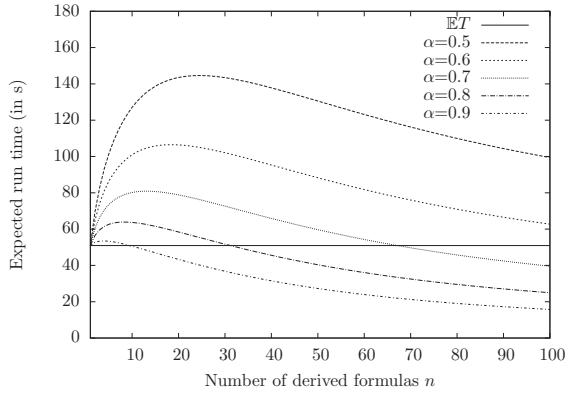


Figure 5.1. The scalability of the plain partitioning approach for the distribution $q_{20}(t)$ in Eq. (5.2) where $t_1 = 1$ and $t_2 = 1000$.

We note that $(1 - \frac{1}{n}) > (1 - \frac{1}{n})^n / n^\alpha$ when $n \geq 2$, and therefore the left side of the inequality is positive and can be made arbitrarily small by setting k small. It remains to show that the right side of the inequality is positive for sufficiently large n, i.e.,

$$\frac{n - (1 - \frac{1}{n}^n)n - n^\alpha}{n^{\alpha+1}} > 0.$$

Since $n^{\alpha+1}$ is always positive, we may simplify this and factor n from the nominator, resulting in

$$1 - (1 - \frac{1}{n})^n - n^{\alpha-1} > 0. \tag{5.5}$$

Noting that $\lim_{n\to\infty}(1 - \frac{1}{n})^n = \frac{1}{e} \approx 0.3$, and that $\lim_{n\to\infty} 1 - n^{\alpha-1} = 1$ if $\alpha < 1$, we get the desired result, that is, for sufficiently large n, there are values t_1 and t_2 such that $t_1 < t_2$ and $\mathbb{E}T < \mathbb{E}T^n_{\text{plain-part}(\alpha)}$. □

The following example illustrates the performance of the plain partitioning approach for distributions of type Eq. (5.2).

Example 2 *Assume there is a formula following the distribution $q_{20}(t)$ such that $t_1 = 1$ and $t_2 = 1000$, and a partition function $\varepsilon(n) = n^{0.7}$ for this formula. The expected run time of the solver S, given by Eq. (5.3), is $\mathbb{E}T \approx 50.95$, while the expected run time of the plain partitioning algorithm, from Eq. (5.4), is $\mathbb{E}T^{20}_{\textit{plain-part}(0.7)} \approx 78.84$. The scalability of the expected run time $\mathbb{E}T^n_{\textit{plain-part}(\alpha)}$ of the plain partitioning approach is shown for the distribution $q_{20}(t)$ for different values of α in Fig. 5.1.*

Note that the proof does not hold if the partitioning function is ideal, since the left hand side of the inequality (5.5) is negative if $\alpha = 1$. The

requirement that the partitioning function is ideal will turn out to be sufficient to guarantee that the expected run time of the plain partitioning approach is never higher than the expected run time of S, that is, $\mathbb{E}T \geq \mathbb{E}T^n_{\text{plain-part}(1)}$ for all n and T. To see this, we will first derive an expression for $\mathbb{E}T^n_{\text{plain-part}(\alpha)}$ for an arbitrary distribution $q_T(t)$ and an arbitrary partitioning function.

Let $q_T(t)$ be a run time distribution of an unsatisfiable formula ϕ with a randomized SAT solver S, and $t_{\max}$ the maximum time required to solve ϕ with S (hence $q_T(t) = 1$ if $t \geq t_{\max}$ and $q_T(t) < 1$ otherwise). The n partitions have run time distributions $q_T(\varepsilon(n)t)$ and since they all need to be shown unsatisfiable, the run time distribution of the plain partitioning approach is $q_T(\varepsilon(n)t)^n$. Hence by Eq. (2.2) the expected run time of the plain partitioning approach is given by

$$\mathbb{E}T^n_{\text{plain-part}(\alpha)} = \int_0^{t_{\max}/\varepsilon(n)} t \frac{\mathrm{d}}{\mathrm{d}t} q_T(\varepsilon(n)t)^n \mathrm{d}t,$$

where $\frac{\mathrm{d}}{\mathrm{d}t} q_T(\varepsilon(n)t)^n = n\varepsilon(n) q_T(\varepsilon(n)t)^{n-1} q'_T(\varepsilon(n)t)$ is the derivative of the distribution function. Substituting $\varepsilon(n)t = \tau$ above, the expected run time can be written

$$\begin{aligned} \mathbb{E}T^n_{\text{plain-part}(\alpha)} = & \int_0^{t_{\max}} \tfrac{\tau}{\varepsilon(n)} n\varepsilon(n) q_T(\tau)^{n-1} q'_T(\tau) \tfrac{\mathrm{d}\tau}{\varepsilon(n)} \\ & = \int_0^{t_{\max}} \tfrac{n}{\varepsilon(n)} \tau q_T(\tau)^{n-1} q'_T(\tau) \mathrm{d}\tau. \end{aligned} \tag{5.6}$$

We can now state the following proposition that increasing the number of derived instances in ideal plain partitioning does not result in increased expected run time.

Proposition 6 *Let $n \geq 1$, $\varepsilon(n) = n^1 = n$ be the efficiency of an ideal partitioning function, and $q_T(t)$ be the run time distribution of an unsatisfiable formula with a randomized solver. Then $\mathbb{E}T^n_{\textit{plain-part}(1)} \geq \mathbb{E}T^{n+1}_{\textit{plain-part}(1)}$.*

Proof. Substituting $\varepsilon(n) = n$ in Eq. (5.6) results in $\mathbb{E}T^n_{\text{plain-part}(1)} = \int_0^{t_{\max}} \tau q_T(\tau)^{n-1} q'_T(\tau) \mathrm{d}\tau$. Since $q_T(\tau) \leq 1$ when $0 \leq \tau \leq t_{\max}$, we immediately have the desired result $\mathbb{E}T^n_{\text{plain-part}(1)} \geq \mathbb{E}T^{n+1}_{\text{plain-part}(1)}$. □

Finally from the propositions 5 and 6 we get the main result concerning unsatisfiable instances.

Proposition 7 *The expected run time of the plain partitioning approach, $\mathbb{E}T^n_{\textit{plain-part}(\alpha)}$, is guaranteed not to be higher than the expected run time $\mathbb{E}T$ of the underlying solver S if and only if the partitioning function is ideal, that is, $\alpha = 1$.*

It is a strong requirement that the efficiency of a partitioning function must be ideal in order to never increase the time required to solve a formula, and it would be tempting to draw the conclusion that this requirement is never met. The practical implications of the above negative result are not as dramatic. Unsatisfiable formulas rarely have such pathological distributions partly because the solvers employ restart strategies known to eliminate this type of behavior [Gomes et al. 2000]. Furthermore, it is not impossible for the partitioning function to provide even superlinear speedup if, for example, the partitioning constraints are somehow related to the possible *back door set* [Williams et al. 2003] of the formula. Nevertheless it is interesting to contemplate on what role this phenomenon has in practice when solving formulas with approaches using partitioning functions.

5.2 Guiding Paths

A widely used technique for implementing plain partitioning is based on *guiding paths* [Zhang et al. 1996], where the search space of a SAT solver is split on demand by copying a modification of the solver's decision stack to other solvers. The guiding paths can be constructed either so that idle solvers "steal" work from other solvers, or so that busy solvers actively push their work to idle solvers. The two ways of constructing the guiding paths are analyzed, for example, in [Blumofe and Leiserson 1994]. Guiding paths are used in a wide range of solvers, including [Böhm and Speckenmeyer 1996; Zhang et al. 1996; Okushi 1999; Blochinger et al. 2003; Jurkowiak et al. 2005; Feldman et al. 2005; Balduccini et al. 2005; Gressmann et al. 2005; Chrabakh and Wolski 2006; Pontelli et al. 2007; Le and Pontelli 2007; Michel et al. 2007; Gil et al. 2009; Schubert et al. 2009; Chu et al. 2009; Martins et al. 2010; Schulz and Blochinger 2010].

The intuition in guiding paths based techniques is that a SAT solver may split its search space on demand by copying a modification of its decision stack to other solvers. In the following the modifications on the decision stack is always done to the decision literal in the lowest decision level where modification has not yet been done. Compared to the definition of guiding paths in [Zhang et al. 1996], this definition is simpler but covers most of the current implementations.

More technically, let c be an initially zero integer, ϕ a propositional for-

mula being solved by a CDCL SAT solver S, and $l_1 \dots l_n$ the current decision literals of S, ordered by the decision level. A *guiding path* consists of the $c \geq 0$ first decision literals $l_1 \dots l_c$ of S. A solver is only allowed to change the guiding path by including more literals to it. The guiding path alters the behavior of the CDCL solver in three ways:

(i) If the solver backtracks to a decision level d lower than c, the solver redoes the decisions $l_d \dots l_c$ to avoid deviating from the guiding path.

(ii) If a conflict is detected during the redoing of the decisions, the solver terminates its search, indicating that $\phi \wedge l_1 \wedge \dots \wedge l_c$ is unsatisfiable.

(iii) A solver S may at any point, when it is on a decision level $d > c$, split its search space by replacing its guiding path with $l_1 \dots l_{c+1}$ and delegating a new guiding path $l_1 \dots l_c \neg l_{c+1}$ to a solver S'.

A guiding path based solving approach terminates if one of the solvers finds a satisfying solution, or all solvers have proved their guiding paths unsatisfiable. No two solvers can find the same satisfying truth assignments, since by (i) and (ii) a solver always searches its guiding path, and by (iii) the guiding paths of any two solvers differ at least by one literal.

The guiding path approach provides a convenient way of performing load balancing. Whenever there are free computing resources, any of the running solvers that are on a decision level higher than c may simply delegate a new guiding path by (iii) to a free resource.

If the formula to be solved is unsatisfiable, then each new delegation increases the number of instances that need to be shown unsatisfiable. The delegation operation in the guiding path approach can be seen as an application of a partitioning function $P(\phi \wedge l_1 \wedge \dots \wedge l_c, 2) = (\neg l_{c+1}, l_{c+1})$ resulting in the derived formulas $\phi_1 = \phi \wedge l_1 \wedge \dots \wedge l_c \wedge \neg l_{c+1}$, and $\phi_2 = \phi \wedge l_1 \wedge \dots \wedge l_c \wedge l_{c+1}$.

If the partitioning function is void, then the run time of the guiding path approach approaches the maximum run time of S as the number of delegations increases. In this case the guiding path approach cannot provide speed-up compared to S. It is an interesting question for further work under what conditions a result similar to Prop. 7 holds for the guiding path approaches with a non-void partitioning function. Many modern guiding path based parallel solvers also incorporate clause learning between the solvers [Blochinger et al. 2003; Schubert et al. 2009; Schulz and Blochinger 2010]. This will be later discussed in Ch. 6.

5.3 Iterative Partitioning with Partition Trees

The result of Prop. 7 showing that the plain partitioning approach is "vulnerable" to certain distributions of unsatisfiable formulas raises the question whether there are other solving approaches that use a partitioning function but are immune to the increased expected run times in all unsatisfiable cases. Given an unsatisfiable formula, the challenge in plain partitioning is that the number of formulas needed to show unsatisfiable increases as more derived formulas are produced.

A trivial solution is to attempt solving both the formula ϕ and the derived formulas using $n + 1$ CPUs or cores. This solution corresponds to solving the formula with the plain partitioning approach and the underlying solver S in parallel, and guarantees that the expected run time of the approach would be at most as high as the expected run time of S. However, by Prop. 7, it is possible that the run time of the plain partitioning approach increases as more resources are used, and this would affect adversely also the behavior of the proposed solution.

The *iterative partitioning approach*, presented originally in [Hyvärinen et al. 2006], is based on a hierarchical partitioning of formulas to increasingly constrained derived formulas which are organized as a tree. The satisfiability of the original formula can then be determined by solving a sufficient number of the derived formulas independently with S. The intuition behind the approach is that the possible increase of the expected run time by Prop. 7 is avoided since every time a formula is partitioned, also its solving is attempted directly with a solver S.

This section gives a formalization and an analysis of the iterative partitioning approach using the concept of a *partition tree* defined as follows.

Definition 3 *A* partition tree $\mathcal{T}_\phi$ *of a formula* ϕ *is a finite* n*-ary tree rooted at* ν_0*. The nodes* ν_i *are associated with constraints: the constraints of the root consist of the formula* ϕ *and the constraints of the other nodes are obtained using a partitioning function on their parents. More precisely,*

1. $Constr(\nu_0) := \phi$*,*

and given a node ν_i*, its children* $\nu_{i,1}, \ldots, \nu_{i,n}$*, and a rooted path* $\nu_0, \ldots, \nu_i$ *in the partition tree, the partitioning constraints of the child nodes are*

2. $Constr(\nu_{i,k}) := \Pi_k$ *where* $\Pi_k \in P(Constr(\nu_0) \wedge \ldots \wedge Constr(\nu_i), n)$.

Finally, each node ν_i *represents the derived formula*

3. $\phi_{\nu_i} := Constr(\nu_0) \wedge \ldots \wedge Constr(\nu_i)$.

In the iterative partitioning approach a partition tree $\mathcal{T}_\phi$ is constructed in breadth first order and the solving of each derived formula ϕ_{ν_i} is attempted in parallel with a solver S until the satisfiability of ϕ can be determined. The satisfiability of a node ν_i can be determined either by solving ϕ_{ν_i} with S, or determining the satisfiability of all the child nodes $\nu_{i,1}, \ldots \nu_{i,n}$.

The iterative partitioning approach guarantees that its expected run time does not increase as more CPUs are introduced, even if the partitioning function is void. We will show this for partition trees $\mathcal{T}_\phi^k$, where all rooted paths to the leaves are of length k. As is conventional, we say that the height of $\mathcal{T}_\phi^k$ is k.

Proposition 8 *Let ϕ be an unsatisfiable formula, $\mathcal{T}_\phi^k$ and $\mathcal{T}_\phi^m$ be two partition trees of height k and m, respectively, constructed with a void partition function, and $k < m$. Then the expected run time of the partition tree approach when using $\mathcal{T}_\phi^m$ is less than or equal to the expected run time of the partition tree approach when using $\mathcal{T}_\phi^k$.*

Proof. We show by induction on the height of the partition tree that the probability that ϕ is solved within time t cannot decrease, from which the claim follows. Let $q(t)$ be the probability that ϕ is solved sequentially within time t, $q'(t)$ be its derivative at t, and let $q_i(t)$ denote the probability that ϕ is solved within time t using a partition tree $\mathcal{T}_\phi^i$ of height i. Then the probability $q_0(t) = q(t)$. The probability that the formula is solved within time t with the partition tree approach using a tree of height one is $q_1(t) = \int_0^t (q'(\tau) + (1 - q(\tau))nq'(\tau)q(\tau)^{n-1})\mathrm{d}\tau$, that is, the integral of the sum of probability $q'(\tau)\mathrm{d}\tau$ that the formula is solved in the root of the tree at time τ, and the probability that the formula has not been solved in the root, has been solved by all children but one by time τ, and is solved at time τ in the last child. A direct calculation shows that $q_1(t) \geq q_0(t)$. Assume now that $q_k(t) \geq q_{k-1}(t)$ for all $t \geq 0$. As previously, $q_{k+1}(t) = \int_0^t (q'(\tau) + (1 - q(\tau))nq_k'(\tau)q_k(\tau)^{n-1})\mathrm{d}\tau = q(t) + q_k(t)^n - \int_0^t q(\tau)nq_k'(\tau)q_k(\tau)^{n-1}\mathrm{d}\tau$. Integration by parts on the negative term results in $q_{k+1}(t) = q(t) + q_k(t)^n - q_k(t)^n q(t) + \int_0^t q_k(\tau)^n q'(\tau)\mathrm{d}\tau = q(t) + (1 - q(t))q_k(t)^n + \int_0^t q_k(\tau)^n q'(\tau)\mathrm{d}\tau$. By the induction hypothesis $q_{k+1}(t) \geq q(t) + (1 - q(t))q_{k-1}(t)^n + \int_0^t q_{k-1}(\tau)^n q'(\tau)\mathrm{d}\tau = q_k(t)$ □

In practice the construction of the tree is not atomic, but the nodes of the tree can be expanded at different times in the breadth first order. As the

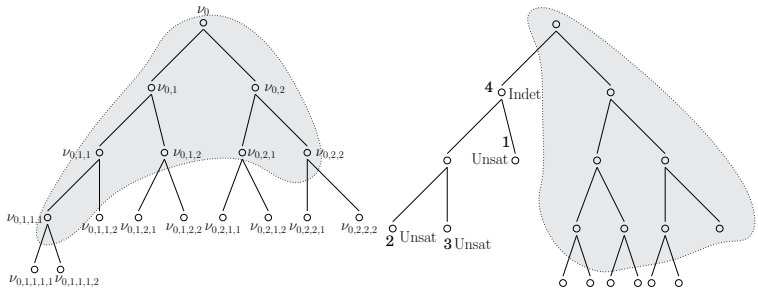


Figure 5.2. Illustration of the partition tree approach. The shaded area represents jobs running simultaneously, the numbers indicate the order in which the jobs terminate and the solid lines represent the edges of the tree

construction of the tree is not immediate, the tree expansion can use information obtained from earlier solving attempts. The straightforward information, used in the approach and presented in [Hyvärinen et al. 2006], is not to expand a subtree rooted at a formula shown unsatisfiable.

The following example illustrates the use of the iterative partitioning and the related partition tree.

Example 3 *Figure 5.2 illustrates how the partition tree approach runs in an environment with* $m = 8$ *parallel resources. The left tree shows the initial setup, and the right tree shows how the solving has proceeded after one of the SAT solvers terminates in a memory out and three of the solvers return unsatisfiable for their respective formulas. In both trees the shaded area indicates the set of formulas currently being solved. The formulas shown unsatisfiable are labeled with* Unsat *and the formula having exceeded its resource limit is labeled with* Indet *on the right-hand-side tree. There is no need to solve* $\nu_{0,1,1}$ *once* $\nu_{0,1,1,1}$ *and* $\nu_{0,1,1,2}$ *are shown unsatisfiable.*

5.4 Safe and Repeated Partitioning

Another approach to avoiding the increase of expected run time in solving unsatisfiable instances is to combine the plain partitioning approach with the approaches based on randomization. This way the inherent randomness in run times of SAT solvers and the reduction in search space provided by the partitioning function can be used simultaneously to obtain speed-up. This work discusses two such composite approaches, presented in [PIII]:

- The *safe partitioning approach* uses the partitioning function to derive formulas each of which are solved with the SDSAT approach; and

- the *repeated partitioning approach* produces several sets of derived formulas with a partitioning function, and solves these sets in parallel using one solver per derived instance.

The use of safe partitioning approach has been suggested in [Ohmura and Ueda 2009; Gebser et al. 2011], whereas the repeated partitioning approach is closely related to *hard restarts* in guiding path based approaches [Gebser et al. 2011]. This work analyzes a setting where n^2 resources are used so that in safe partitioning the partitioning function results in n partitions which are solved using n solvers each. In repeated partitioning the partitioning function is repeated n times for the same formula, resulting again in n^2 formulas.

The safe partitioning approach consists of applying a partitioning function $P(\phi, n) = (\Pi_1, \ldots, \Pi_n)$, and solving each derived formula $\phi \wedge \Pi_i$, $1 \leq i \leq n$, with the SDSAT approach using n solvers. It suffices then to show each derived instance unsatisfiable with one solver. Intuitively the approach provides speed-up since derived formulas should be easier to solve than the original formula, and, assuming the solving times of the derived formulas obey a non-trivial random distribution, the SDSAT approach results in lower run times for the derived formulas. The repeated partitioning approach, on the other hand, consists of applying a family of partitioning functions $P^j(\phi, n) = (\Pi_1^j, \ldots, \Pi_n^j)$, $1 \leq j \leq n$, and solving each derived formula $\phi \wedge \Pi_i^j$, $1 \leq i \leq n$, $1 \leq j \leq n$ with a solver S. To show a formula unsatisfiable it suffices to show unsatisfiable any set of derived formulas $\phi \wedge \Pi_1^k, \ldots, \phi \wedge \Pi_n^k$ for a fixed k. The approach is expected to result in speed-up as the derived formulas are easier to solve than the original formula, but also because it is possible that one of the partitioning functions P^j could work better than some other partitioning function. The analysis will ignore the latter point, but it is worth pointing out the experimental results in [PIII] suggest this as significant in providing speed-up in practice.

Based on the definition we can immediately give the run time distributions of the two composite approaches using the equations (4.1) and (5.6) for simple distribution and plain partitioning. The cumulative run time distribution for safe partitioning of unsatisfiable formulas $q_{T_{\text{safe-part}}}(t)$ is

given by substituting $q_T(t)$ in (4.1) by (5.6), yielding

$$q_{T_{\text{safe-part}}}(t) = (1 - (1 - q(\varepsilon(n)t))^n)^n, \tag{5.7}$$

and the repeated partitioning by substituting $q_T(t)$ in (5.6) by (4.1), resulting in

$$q_{T_{\text{rep-part}}}(t) = 1 - (1 - q(\varepsilon(n)t)^n)^n. \tag{5.8}$$

Based on equations (5.7) and (5.8) it is proved in [PIII] that the expected run time of the repeated partitioning is always at least the expected run time of the safe partitioning, independent of the partitioning function or number of CPUs n.

Proposition 9 *Let $q_T(t)$ be the run time distribution of an unsatisfiable formula. Then $\mathbb{E}T_{\textit{safe-part}} \leq \mathbb{E}T_{\textit{rep-part}}$.*

The publication [PIII] also gives an example distribution with which the expected run time of the repeated partitioning approach is higher than that of the underlying solver.

If the formula to be solved is satisfiable, one can show the following proposition (see again [PIII] for the proof):

Proposition 10 $\mathbb{E}T_{\textit{safe-part}} = \mathbb{E}T_{\textit{rep-part}}$ *for satisfiable instances.*

Interestingly, the experimental results in [PIII] indicate that in practice the repeated partitioning approach is faster than the safe partitioning approach. This seems to result from the randomness in the used partitioning function not accounted for in the model. The construction of partitioning functions is discussed in detail in the next section.

5.5 Constructing Partitions

As seen from the preceding analytical discussion, the good quality of the partitioning function is critical in obtaining speed-up, and, in case of plain, safe and repeated partitioning based approaches, avoiding increase in expected run time. The partitioning functions considered here work by introducing constraints, represented as clauses, to a formula. The work introduces two types of partitioning functions, the *DPLL-based partitioning* producing only unit clauses, and the *scattering based partitioning*, which produces also longer clauses. Heuristics for constructing the constraints are used for increasing the likelihood of obtaining partitions

which result in low run time. All implementations of the partitioning functions are built on a CDCL SAT solver. In addition to the discussion in this section, [PIV] presents also an approach to combining lookahead and scattering, and performs an experimental comparison. Some experiments on two of the partitioning functions are also given later in Ch. 6.

The first partitioning function discussed here uses the unit propagation lookahead (see, e.g., [Heule and van Maaren 2009]), used in many non-learning CDCL SAT solvers, such as SATZ [Li and Anbulagan 1997b], and MARCH_DL [Heule and van Maaren 2006]. The goal is to use as decision literals the literals that result in highest number of unit propagations.

Computing the full lookahead for a formula ϕ is worst-case quadratic in the number of variables in ϕ. Therefore typical lookahead solvers only study a subset of promising literals of ϕ and use several optimizations in the computation. One such optimization based on the conflict graph of a CDCL solver is studied more closely in [PIV]. The *lookahead DPLL* partitioning function, used in some of the experiments in this work, implements the conflict graph optimization along with some other standard optimizations to produce evenly sized derived formulas. Given a formula ϕ, promising literals l are studied by computing the number of literals in the unit propagation closure $\mathrm{UP}(\phi, l)$ and $\mathrm{UP}(\phi, \neg l)$. As the number of literals in $\mathrm{UP}(\phi, l)$ might differ dramatically compared to $\mathrm{UP}(\phi, \neg l)$, the implementation scores literals based on the minimum of these two numbers. Once a heuristically good literal has been selected, the corresponding two derived formulas $\phi \wedge \mathrm{UP}(\phi, l)$ and $\phi \wedge \mathrm{UP}(\phi, \neg l)$ are recursively handled in a similar way. The binary tree up to the depth n constructed this way can be interpreted as consisting of 2^n derived formulas covering all potential satisfying truth assignments of ϕ, and the idea in DPLL based partitioning is to return exactly these formulas as the derived formulas.

It is interesting to study partitioning functions producing more general constraints. The derived formulas in DPLL based partitioning are of the form $\phi \wedge l_1 \wedge \ldots \wedge l_n$, but there is no need to limit partitioning functions to producing only constraints of unit clauses. The scattering based partitioning produces both unit and longer clauses as the constraints. The idea is to first run a CDCL solver for a fixed time to tune the heuristic of the solver. If the satisfiability of the formula is not determined in this time, the solver restarts, and starts to produce derived formulas. The first derived formula is produced by making the decisions $l^1_1 \ldots l^1_{d_1}$, and outputting the formula $\phi \wedge l^1_1 \wedge \ldots \wedge l^1_{d_1}$ as in DPLL based partitioning. Then, instead

of selecting the next branch of the search tree, the negation of the literals is inserted as a clause to ϕ. The solver restarts again, makes new decisions $l_1^2 \dots l_{d_2}^2$, and outputs the formula $\phi \wedge (\neg l_1^1 \vee \dots \vee \neg l_{d_1}^n) \wedge l_1^2 \wedge \dots \wedge l_{d_2}^2$. The process is continued until a sufficient number of derived formulas are produced. The idea leads to a partitioning function producing the derived formula ϕ_i such that

$$\phi_i = \begin{cases} \phi \ \wedge(l_1^1) \wedge \dots \wedge (l_{d_1}^1) & \text{if } i = 1, \\ \phi \ \wedge(\neg l_1^1 \vee \dots \vee \neg l_{d_1}^1) \wedge & \\ \quad \wedge \dots \wedge (\neg l_1^{i-1} \vee \dots \vee \neg l_{d_{i-1}}^{i-1}) \wedge & \\ \quad (l_1^i) \wedge \dots \wedge (l_{d_i}^i) & \text{if } 1 < i < n, \\ \phi \ \wedge(\neg l_1^1 \vee \dots \vee \neg l_{d_1}^1) \wedge \dots \wedge & \\ \quad \wedge(\neg l_1^{n-1} \vee \dots \vee \neg l_{d_{n-1}}^{n-1}) & \text{if } i = n. \end{cases} \tag{5.9}$$

Essentially the derived formulas consist of the original formula ϕ, a conjunction of unit clauses $(l_1) \wedge \dots \wedge (l_d)$ and clauses representing negations of the previously selected unit clauses. In order for the derived formulas to be of roughly equal size, the number of new unit clauses, denoted by d_i, should not in general be the same in all derived formulas. The selection of the number d_i is motivated so that the expected run time of each derived formula should be t/n, where t is the expected run time of the original formula and n is the total number of derived instances produced by the partitioning function. Hence the goal fraction r_i of the run time for the derived formula ϕ_i can be obtained from the equality

$$\frac{t}{n} = (t - (i-1)\frac{t}{n})r_i,$$

where $(i-1)\frac{t}{n}$ is the run time already contributed to the derived formulas $\phi_1, \dots, \phi_{i-1}$. Solving the above for r_i results in

$$r_i = \frac{1}{n-i+1} \tag{5.10}$$

The approach followed in this work is to assume that conjoining a literal with a formula halves the expected run time of the formula, and therefore the number d_i is chosen to be the integer minimizing the difference

$$\Delta = |r_i - 2^{-d_i}|. \tag{5.11}$$

Example 4 *Let ϕ be a propositional formula and P a partitioning function producing 3 partitions. From Eq. (5.10), the first fraction of the search space should be $r_1 = 1/3$. The value $d_1 = 2$ minimizes Δ in Eq. (5.11), the first derived formula becomes, by Eq. (5.9), $\phi_1 = \phi \wedge (l_1^1) \wedge (l_2^1)$. Similarly,*

$r_2 = 1/2$ *and the value* $d_2 = 1$ *minimizes* Δ*, the second derived formula becomes* $\phi_2 = \phi \wedge (\neg l_1^1 \vee \neg l_2^1) \wedge (l_1^2)$*. The final derived formula becomes then* $\phi_3 = \phi \wedge (\neg l_1^1 \vee \neg l_2^1) \wedge (\neg l_1^2)$.

In the experiments of this work, the vsids heuristic is used to select the decision literals. A similar approach is used, for example, in [Hyvärinen et al. 2006; Dequen et al. 2009; Martins et al. 2010].

The approach for choosing values for d_i using the model in Eq. (5.10) is not the only possibility. The following example illustrates how the scattering approach can "simulate" a DPLL-based partitioning.

Example 5 *Let* ϕ *be a propositional formula. Our target will be to build a partitioning function producing 4 derived formulas. Let the first derived formula be* $\phi_1 = \phi \wedge (l_1) \wedge (l_2)$*. Setting* $d_2 = 1$ *we may choose* $\phi_2 = \phi \wedge (\neg l_1 \vee \neg l_2) \wedge (l_1)$ *as the second derived formula. Since* $\mathrm{UP}((\neg l_1 \vee \neg l_2) \wedge (l_1)) = \{l_1, \neg l_2\}$*, the solving of* ϕ_2 *will proceed exactly as if the second derived formula would have been* $\phi_2 = \phi \wedge (l_1) \wedge (\neg l_2)$*, corresponding to the DPLL-based partitioning. Similarly it is possible to choose* $d_3 = 1$ *in Eq. (5.9) and* $\phi_3 = \phi \wedge (\neg l_1 \vee \neg l_2) \wedge (\neg l_1) \wedge (l_3)$ *resulting in the search corresponding to the DPLL-based partitioning derived formula* $\phi \wedge \neg l_1 \wedge l_3$*, and finally* $\phi_4 = \phi \wedge (\neg l_1 \vee \neg l_2) \wedge (\neg l_1) \wedge (\neg l_3)$.

The approach presented in the above example generalizes to producing also higher number of derived formulas. Let $S_n = (d_1, \ldots, d_n)$ denote the sequence producing n derived instances as in Ex. 5. Let $S_i = (d_1, \ldots, d_i)$ and $T_j = (e_1, \ldots, e_j)$ be two such sequences. We denote by $S_n + 1$ the sequence $(d_1 + 1, \ldots, d_n + 1)$ and by $(S_i) \cdot (T_j)$ the concatenation of the two sequences $(d_1, \ldots, d_i, e_1, \ldots, e_j)$. The scattering based partitioning function can "simulate" the DPLL based partitioning function producing $n = 2^k, k \geq 0$ derived instances by using a fixed variable ordering and the sequence S_n defined recursively as $S_1 = S_{k^0} = (0)$ and $S_{2^k} = (S_{2^{k-1}} + 1) \cdot (S_{2^{k-1}})$.

6. Learning and Partitioning

Clause learning has been one of the major breakthroughs in increasing SAT solver performance in structured combinatorial problems. The topic of this chapter is to combine clause learning with partitioning based solving approaches, and in particular with the iterative partitioning using partition trees, discussed in Ch. 5. By the construction of the partition tree, the clauses learned in one branch of the tree are not necessarily logical consequences in other branches. One of the main challenges tackled in this chapter is to efficiently compute how a learned clause depends on the branch so that the clause can be used in other branches. Two approaches to tracking the dependency are studied independently and by integrating them to the iterative partitioning approach.

The results obtained in this chapter are in line with those from the CL-SDSAT framework in Ch. 4, suggesting that combining parallel, cumulative learning with partitioning helps in solving especially the more difficult instances. The publications [PIV] and [PV] describe a high number of experiments on iterative partitioning both with and without cumulative learning. This chapter complements the discussion by giving examples on the tracking approaches, studying the effect of learned clauses, and comparing the iterative partitioning approach against several other SAT solving approaches.

6.1 Learned Clause Tagging

As discussed in Ch. 2, new clauses are learned by a CDCL SAT solver each time unit propagation results in an inconsistent truth assignment. If a solver is solving the formula $\phi \wedge \Pi$ for some partitioning constraint Π, then obtaining clauses which are logical consequences of a formula

ϕ requires in general modifications to the solver. The challenge in this section is to track whether clauses used in the analysis of a conflict depend on the partitioning constraints. As the clause learning techniques play a key role in modern CDCL solvers, the tracking should not slow down the solver excessively.

The solution taken in many guiding path based, learning, parallel SAT solvers (see, e.g., [Schulz and Blochinger 2010; Schubert et al. 2009]) is to encode partitions in the decision literals using guiding paths, as explained in 5.2. With such approaches the problem of tracking constraint dependency is handled by the underlying SAT solver implicitly. In many applications, such as bounded model checking (see, for example, [Wieringa et al. 2009; Eén et al. 2010; Ábrahám et al. 2011]) and the partition trees discussed here, this cannot be done done due to the more general nature of the constraints. There are two ways in which the partitioning constraints affect clauses learned by a solver in the more general setting discussed in this work.

(i) A partitioning constraint can directly enable learning clauses which are not necessarily logical consequences of the formula.

(ii) A partitioning constraint may result in a learned clause simplified so that it is no longer a logical consequence of the formula.

The following example illustrates both cases.

Example 6 *Consider the formula* $\phi = (y_1 \vee \neg x_1) \wedge (y_2 \vee \neg x_2) \wedge (y_3 \vee \neg x_3) \wedge (x_1 \vee x_2 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee x_2 \vee \neg x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_2 \vee x_3 \vee \neg x_4) \wedge (x_1 \vee \neg x_3 \vee x_4) \wedge (\neg x_1 \vee z_1 \vee z_2) \wedge (\neg x_1 \vee z_1 \vee \neg z_2)$ *and the partitioning constraint* $\Pi = (x_1 \vee x_3 \vee x_4)$. *Assume the formula* $\phi \wedge \Pi$ *being solved by a CDCL solver, and let* $\neg y_1 \neg y_2 \neg y_3$ *be the current decision literals of the solver. This results after propagation in a conflict shown in Fig. 6.1 (a); the related analysis results first in the asserting conflict clause* $(x_1 \vee x_2 \vee x_3)$. *Backtracking and propagation result in another conflict shown in Fig. 6.1 (b), producing another asserting conflict clause* $(x_1 \vee x_2)$. *Neither clause is a logical consequence of* ϕ *and both fall into the case (i) above. The solver backtracks and makes decisions* $\neg y_1 \neg y_3$. *This results again in two asserting conflict clauses,* $(x_1 \vee x_3)$ *in Fig. 6.1 (c) and* (x_1) *in Fig. 6.1 (d), again not logical consequences of* ϕ, *being examples of the case (i).*

The example continues in the bottom of Fig. 6.1, where $\neg z_1$ *is assumed, and conflict and the subsequent analysis results in the clause* $(\neg x_1 \vee z_1)$. *This clause is a logical consequence of* ϕ. *However, since* x_1 *is in the de-*

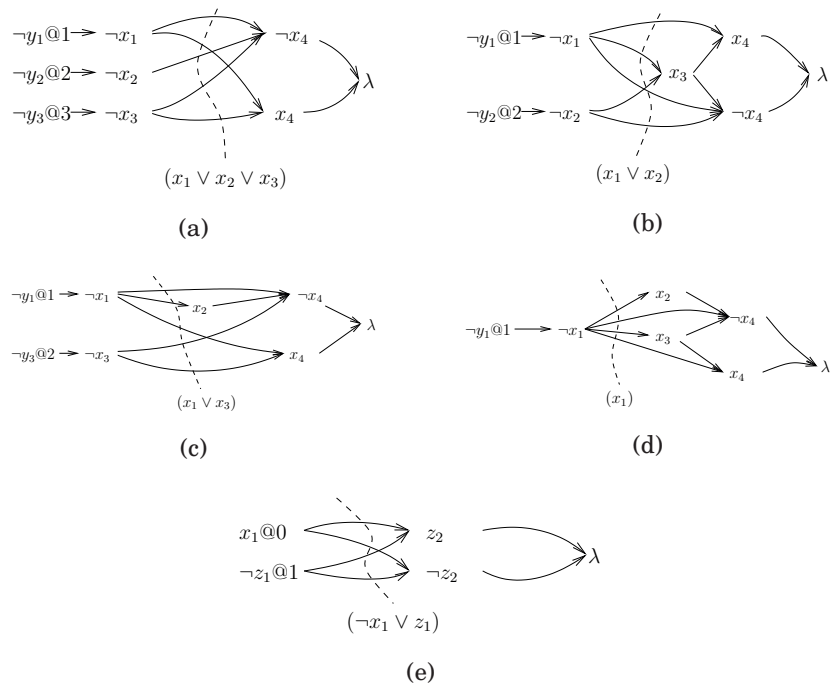


Figure 6.1. A learning and partitioning example. The top two figures (a) and (b) illustrate a conflict analysis resulting in two asserting conflict clauses that depend on a partitioning constraint $(x_1 \vee x_3 \vee x_4)$. The middle two figures (c) and (d) show another conflict analysis resulting in partitioning constraint dependent unit clause. The bottom figure shows an analysis resulting in a conflict clause $(\neg x_1 \vee z_1)$ which does not depend on the partitioning constraint, but can be simplified to (z_1) by using the unit clause (x_1) which depends on the partitioning constraint. The simplification is not shown in the figure.

cision level 0, it is in general useful to simplify the clause to (z_1)*, which again is not a logical consequence of* ϕ*. This is an example of the case (ii) above.*

As suggested by the example above, the goal here is to develop methods for tracking which partitioning constraints were used in a conflict analysis resulting in a learned clause. The following definition comes to use for this purpose.

Definition 4 *A* Constraint Aware Clause Producing (CACP) solver *takes as input a formula* ϕ *and set of partitioning constraints* $\Pi_1, \ldots, \Pi_k$ *and reports either a satisfying truth assignment for* $\phi \wedge \Pi_1 \wedge \ldots \wedge \Pi_k$ *or sets of learned clauses* $Lrnt(\phi)$ *and* $Lrnt(\Pi_j)$ *for* $1 \leq j \leq k$ *such that* $\phi \models Lrnt(\phi)$ *and* $\phi \wedge \Pi_1 \wedge \ldots \wedge \Pi_j \models Lrnt(\Pi_j)$.

The first approach to enabling such logging is called *assumption tagging*, and has previously been used in incremental SAT solving [Eén and

Sörensson 2003] and minimum unsatisfiable core extraction [Asín et al. 2010]. The idea is to disjoin to each partitioning constraint a new *assumption literal* that does not appear in the formula. The partitioning constraints will be "enabled" by setting the assumption literals false as the first decisions of the CDCL solver. If such a clause tagged with an assumption literal is used in conflict analysis, the resulting conflict clause will inherit the assumption literal. Once a clause tagged with an assumption literal gets involved in a conflict analysis, the assumption literal cannot disappear from the resulting conflict clauses as assumption literals only appear in one polarity in the formula.

A CDCL solver can be modified to a CACP solver by using assumption tagging as follows. Let ϕ be a formula, $\Pi_1 \dots \Pi_k$ partitioning constraints and $a_1 \dots a_k$ literals not appearing in ϕ. A CDCL solver S takes as input the formula $\phi \wedge A$, where

$$A = \bigwedge_{j=1}^{k} (a_j \vee \mathit{Constr}(\nu_j)).$$

The constraints are enabled by forcing the first decisions of the solver to $\neg a_1 \dots \neg a_k$. When a clause C containing an assumption literal is learned by the CDCL solver, it is, without the assumption literals, added either to the set $\mathit{Lrnt}(\Pi_j)$ where $j = \max\{j \mid a_j \in C\}$, or to the set $\mathit{Lrnt}(\phi)$ if $\{a_1, \dots, a_k\} \cap C = \emptyset$. If a conflict is found during the forced decisions $\neg a_1 \dots \neg a_j$ for some $j \leq k$, the set $\mathit{Lrnt}(\Pi_j)$ will only contain the empty clause $\perp$ indicating that the formula ϕ conjoined with a subset of the constraints $\Pi_1, \dots, \Pi_j$ is unsatisfiable. The following example illustrates the assumption based CACP solver on the formula in Ex. 6.

Example 7 *Let the formula and partitioning constraint in Ex. 6 be solved with a CACP solver using assumption tagging. Then the partitioning constraint is* $\Pi_1 = (a_1 \vee x_1 \vee x_3 \vee x_4)$*, and the learned clauses are* $(a_1 \vee x_1 \vee x_2 \vee x_3)$*,* $(a_1 \vee x_1 \vee x_2)$*,* $(a_1 \vee x_1 \vee x_3)$*,* $(a_1 \vee x_1)$ *and* $(a_1 \vee \neg x_1 \vee z_1)$ *respectively. These clauses are included to the set* $\mathit{Lrnt}(\Pi_1)$*. Finally, the last conflict analysis results in a learned clause* $(\neg x_1 \vee z_1)$ *which is correctly lacking the assumption literal as the clause is a logical consequence of* ϕ*. The clause will be included to the set* $\mathit{Lrnt}(\phi)$*.*

The efficiency of the tagging approach is critical in order for it to provide speed-up. The following experiment is used to study the overhead caused by the assumption tagging approach. Some formulas from SAT-Comp 2009, listed in the upper half of Table 6.1 were solved using MIN-

Table 6.1. Experiments on flag and assumption based tagging approaches

Instances used in the overhead measurement
dated-5-13-u, *dated-5-19-u*, *eq.atree.braun.12.unsat*, *gss-24-s100*, *mod4block_3vars_7gates*, *rbcl_xits_08_UNSAT*, *total-10-17-u*, *vmpc_34*
Instances used in the effect measurement
AProVE07-01, *AProVE07-25*, *countbitsarray02_32*, *dated-5-13-u*, *dated-5-19-u*, *eq.atree.braun.12.unsat*, *eq.atree.braun.13.unsat*, *gss-22-s100*, *gss-24-s100*, *gss-26-s100*, *gus-md5-11*, *gus-md5-14*, *mod4block_3vars_7gates*, *rbcl_xits_08_UNSAT*, *rpoc_xits_09_UNSAT*, *sgen1-unsat-109-100*, *simon-s02b-k2f-gr-rcs-w8*, *total-10-17-u*, *vmpc_34*

iSAT 2.2.0 with partitioning constraints encoded directly (see Ex. 6) and with the assumption tagging. The partitioning constraints were obtained with the iterative partitioning approach (see Sect. 6.2). The results are presented in Fig. 6.2, where each cross represents a formula with partitioning constraints. The value for the vertical coordinate of the cross comes from direct encoding and horizontal coordinate from assumption tagging. The run time comparison in top left reveals that the assumption tagging approach is usually slower in solving formulas. The assumption tagging approach also fails to solve many instances solved by the directly conjoining approach, as indicated by the crosses on the horizontal line at the top of the graphs.

The number of decision literals taken by the respective approaches is shown in top right graph. The increase in run time seems not to result from an increase in number of decisions, as the number is roughly the same for both approaches. However, the memory consumption shown on the bottom graph is significantly higher in the assumption tagging approach. It seems that the increase in clause sizes demonstrated in Ex. 7 results in a substantial bottleneck in memory consumption for some instances.

As the overhead in assumption tagging is high, this work studies also a more light-weight approach for storing for each learned clause whether partitioning constraints were used in the conflict analysis. This *flag tag-*

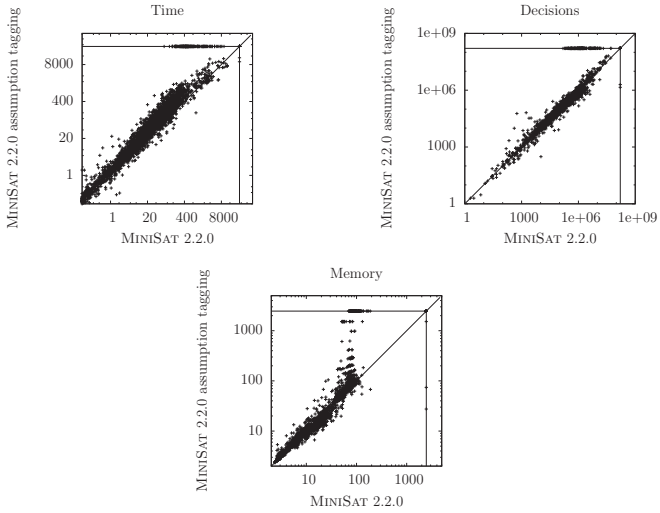


Figure 6.2. Comparison of the assumption tagging approach against the direct conjoining approach. The figure in the top left compares run times, the top right compares decisions and the bottom figure compares memory usage.

ging approach flags clauses *unsafe* if they are potentially not logical consequences of the original formula. The idea is that if an unsafe clause is used in the conflict analysis, the resulting conflict clause is also tagged unsafe. A similar idea has been used, for example, in [Wieringa et al. 2009] in bounded model checking. For performance reasons the flag tagging only classifies the clauses as either safe or unsafe. In practice all clauses in $\Pi_1, \ldots \Pi_k$ are tagged unsafe, and the clauses of the formula ϕ are untagged. To maintain the correctness of the learned clause sets $Lrnt(\phi)$ and $Lrnt(\Pi_1), \ldots, Lrnt(\Pi_k)$, if a learned clause is tagged unsafe it is added to the set $Lrnt(\Pi_k)$, while only clauses with no unsafe tag are added to the set $Lrnt(\phi)$. The following example clarifies the use of the flag tagging approach for the setting described in Ex. 6 and Fig. 6.1.

Example 8 *Let ϕ and Π_1 be as in Ex. 6, being solved with a flag tagging CACP solver. Initially the solver tags the partitioning constraint $(x_1 \vee x_3 \vee x_4)$ unsafe. As the clause is used in deriving $(x_1 \vee x_2 \vee x_3)$, also this clause is tagged correctly unsafe. This clause is then used to derive $(x_1 \vee x_2)$ which is therefore also tagged unsafe. The clause $(x_1 \vee x_2)$ is used in turn in a conflict analysis resulting in $(x_1 \vee x_3)$, and both clauses are finally used in analysis resulting in unsafe tagged clause (x_1). Finally in the conflict analysis where $(\neg x_1 \vee z_1)$ is derived, the clause (x_1) is tagged unsafe. Hence the simplification resulting in the unit clause (z_1) can now be performed,*

and also (z_1) *is tagged unsafe.*

The overhead caused by the flag tagging approach is minimal as exactly the same clauses are used in the analysis. The flag only requires one bit per each clause, and the clause representation of MINISAT 2.2.0 by chance contains one bit unused by the original implementation.

6.2 Cumulative Learning in Iterative Partitioning

The results in the preceding sections suggest that sharing the learned clauses between the derived formulas potentially improves the performance of the iterative partitioning approach. In what follows, the iterative partitioning approach discussed in Ch. 5 is extended with a cumulative learning similar to the CL-SDSAT approach in Ch. 4. The key points here are defining a procedure for sharing learned clauses between the derived formulas and defining a simplification process similar to the one used in the CL-SDSAT approach for the iterative partitioning approach.

The iterative partitioning approach with cumulative learning is based on a similar ϕ-rooted n-ary tree of partitions as the iterative partitioning approach without cumulative learning described in Ch. 5. Therefore only a small extension to Def. 3 suffices.

Definition 5 *A* learning partition tree $\mathcal{L}_\phi$ *of a formula* ϕ *is a finite* n*-ary tree rooted at* ν_0*. The nodes* ν_i *are associated with constraints* $Constr(\nu_i)$ *and sets of learned clauses* $Lrnt(\nu_i)$ *such that*

1. $Constr(\nu_0) := \phi$,

and given a node ν_i*, its children* $\nu_{i,1}, \ldots, \nu_{i,n}$*, and the rooted path* $\nu_0, \ldots, \nu_i$ *in the learning partition tree, the partitioning constraints of the child nodes are*

2. $Constr(\nu_{i,k}) := \Pi_k$ *where* $\Pi_k \in P(\bigwedge_{j=0}^{i}(Constr(\nu_j) \wedge Lrnt(\nu_j)), n)$

Finally, each node ν_i *represents the derived formula*

3. $\phi_{\nu_i} := \bigwedge_{j=0}^{i}(Constr(\nu_j) \wedge Lrnt(\nu_j))$

For now we simply assume that $Lrnt(\nu_i)$ is a set of clauses C such that $Constr(\nu_0) \wedge \ldots \wedge Constr(\nu_i) \models C$. In the iterative partitioning approach with cumulative learning, the solving of each node ν_i is attempted so that the partitioning constraints $Constr(\nu_0), \ldots, Constr(\nu_i)$ and heuristically promising subsets of the learned clauses $Lrnt(\nu_0), \ldots, Lrnt(\nu_i)$ are given as the constraints to a CACP solver, which will upon termination return new clauses for updating the sets $Lrnt(\nu_0), \ldots, Lrnt(\nu_i)$.

Due to the potentially massive amounts of learned clauses produced by the CACP solver, there is a need for a size limit $MaxDBSize$ to the learned clause sets in the nodes of the learning partition tree. A process similar to the one for CL-SDSAT is used to determine the clauses that are heuristically most likely to speed up the solving of the derived formulas. Let $Lrnt(\nu_0)', \ldots, Lrnt(\nu_i)'$ be clause sets learned by the CACP solver. Then the sets $Lrnt(\nu_j)$, $0 \leq j \leq i$ in the learning partition tree are updated so that

$$Lrnt(\nu_j) := Merge(U, Lrnt(\nu_j), Lrnt(\nu_j)', MaxDBSize),$$

where $U = \mathrm{UP}(\phi_{\nu_j} \wedge Lrnt(\nu_j)')$ is the set of literals obtained by unit propagation from the formula ϕ_{ν_j} related to the node ν_j, and the function $Merge$ is as defined in Sect. 4.4.

Given a node ν_i to be solved with a CACP solver, the number of learned clause sets for this node is i. As the number of usable learned clause sets increases with the length of the rooted path, it is also necessary to limit the size of and devise a heuristic for selecting the learned clauses provided to the CACP solver.

The design choice taken here is to learn as "general" clauses as possible based on earlier shared clauses. The $Merge$ function removes from the input clauses the literals that are false under the set U. Hence its use would result in learned clauses becoming dependent on the constraints used for this simplification. Therefore the clause sets $Lrnt(\nu_j)$, $0 \leq j \leq i$, provided to the CACP solver are only simplified by removing the clauses satisfied by $Constr(\nu_0) \wedge \ldots \wedge Constr(\nu_i)$. However, the sizes of the learned clauses is computed as if the simplification removing also false literals were performed. More technically, let $U = \mathrm{UP}(\bigwedge_{j=0}^{i}(Constr(\nu_j) \wedge Lrnt(\nu_j)))$ be the unit propagation closure, and $\overline{U} = \{\neg l \mid l \in U\}$. The formula provided to the constraint aware solver consists of constraints and corresponding learned clauses $Constr(\nu_0) \wedge Lrnt(\nu_0)', \ldots, Constr(\nu_i) \wedge Lrnt(\nu_i)'$ such that $Lrnt(\nu_j)' \subseteq Lrnt(\nu_j)$ for $0 \leq j \leq i$, and

(i) no clause is satisfied by the unit propagation closure, that is, if $C \in Lrnt(\nu_j)'$, then $C \cap U = \emptyset$, and

(ii) the sum of the sizes of the simplified learned clauses $\sum_{j=0}^{i} ||\{C \setminus \overline{U} \mid C \in Lrnt(\nu_j)'\}||$ is less than or equal to a constant $SubmSize$.

Example 9 *The example in Fig. 6.3 illustrates, similar to Fig. 5.2, how the learning partition tree is constructed on-the-fly in breadth-first order starting from the root using eight CPU cores in a grid and when the arity*

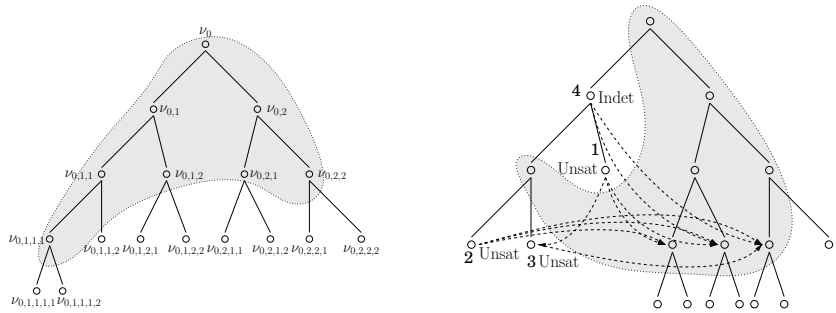


Figure 6.3. The iterative partitioning approach with cumulative learning. The nodes represent the derived formulas, and the nodes in the shaded area are being solved simultaneously. Terminated jobs are marked either Indet or Unsat depending on whether they run out of resources or prove unsatisfiability, and annotated with the termination order (1 terminates first and 4 last). Some learned clauses from earlier terminated jobs can be transferred to the newly submitted jobs, illustrated by the dashed arrows. The tree is constructed in breadth-first order.

$n = 2$. *In the left tree the derived formulas at nodes are sent to the environment to be solved (in parallel) with a SAT solver, and the nodes are further partitioned into child nodes at the same time. At this point no clauses are learned yet, and thus the sets* $Lrnt(\nu_i) = \emptyset$ *for all nodes. The process continues as in Ex. 3 until all eight computing resources are used. Similar to Ex. 3, the solving of* $\nu_{0,1,1}$ *could be finished once* $\nu_{0,1,1,1}$ *and* $\nu_{0,1,1,2}$ *are shown unsatisfiable. The solving is not terminated in the example as the clauses learned there might still prove useful in other parts of the partition tree, and instead the next node* $\nu_{0,2,1,2}$ *is submitted. Learned clauses can be transferred to subsequent jobs, indicated by the dashed arrows. Cumulative learning can be seen in learned clauses transferred to* $\nu_{0,2,2,1}$*, as these potentially include clauses learned in* $\nu_{0,1,1,2}$ *using clauses learned in* $\nu_{0,1,2}$.

6.3 Effect of Learned Clauses with Tagging

Based on the results of the CL-SDSAT approach in Ch. 4, one can expect that learned clauses should speed up solving once a sufficient number of high quality clauses have been obtained. It is interesting to compare the two tagging approaches in this respect. On one hand the clauses shared in the assumption tagging approach should be more numerous. As the number of clauses is higher, more search space can be pruned compared to the flag tagging approach. On the other hand the related overhead seems to be higher in the assumption tagging approach.

This section studies the combined effect of learned clauses and the tagging approaches. The benchmark instances used for these experiments are given in the lower half of Table. 6.1. The formulas are constructed based on these instances with the iterative partitioning approach with cumulative learning. The shortest from all available learned clauses are used so that the total number of literals in these clauses is at most 100 000. The same number was used in the CL-SDSAT experiments. Both the DPLL based partitioning with the lookahead heuristic and the scattering based partitioning with the vsids heuristic were used in producing the derived formulas.

Unlike in Fig. 6.2, where the goal is to measure the overhead of the assumption tagging approach, the experiments in this section also consider the gain obtained with the learned clauses. The value on the vertical axis is obtained from a CACP solver and a number of constraints consisting of learned clauses and partitioning constraints. The value on the horizontal axis is obtained from MINISAT 2.2.0 with the partitioning constraints but without the learned clauses. The learned clauses are not included in the formulas corresponding to the values on the horizontal axis, as a solver conjoining the partitioning constraints as such cannot, of course, transfer learned clauses between two arbitrary derived formulas after terminating without risking the correctness of the approach.

The comparisons are shown in Fig. 6.4 for the assumption tagging approach (graphs (a) and (b)), and for the flag tagging approach (graphs (c) and (d)). The overhead caused by the assumption tagging is often high compared to the reduction in decisions gained from the higher number of short learned clauses. In particular the amount of failed executions (the crosses on the horizontal line on top of the graphs) is high for the approach. The graphs (c) and (d) in Fig. 6.4 show that the gain from the clauses learned with the flag tagging approach is significantly better. For these formulas the flag tagging approach results in sufficient number of learned clauses while keeping the overhead related to tracking the partitioning constraint dependency sufficiently low.

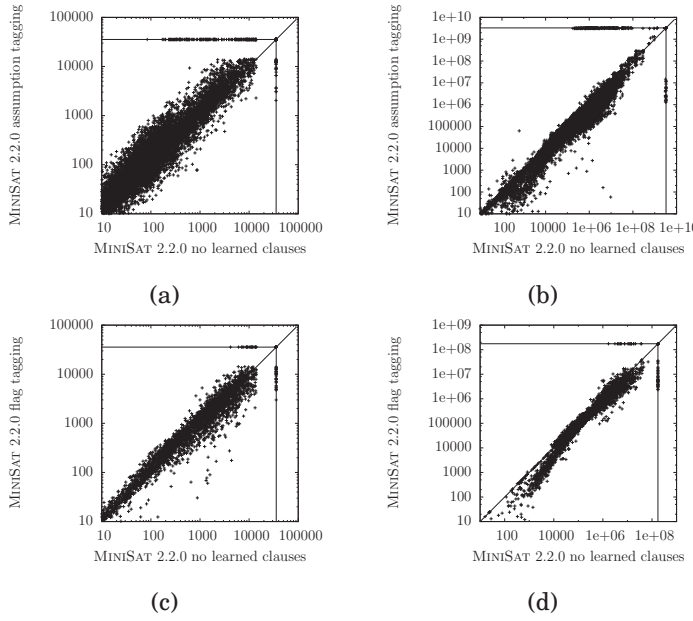


Figure 6.4. Comparing non-learning and learning approaches for iterative partitioning. The graphs (a) and (c) show run times and the graphs (b) and (d) show the number of decisions. The graphs (a) and (b) compare the assumption and the figures (c) and (d) the flag tagging approaches.

6.4 Experiments on the Iterative Partitioning with Cumulative Learning

This section studies the run time of the iterative partitioning approach with cumulative learning using the m-grid environment discussed in Ch. 3. The maximum run time of the jobs in this environment are randomly selected between 60 and 90 minutes, the number of simultaneously running jobs is limited to 64 and each job can consume at most 2GB of memory. The full work flow was limited to 6 hours. The underlying CDCL solver, modified to a CACP solver, is MINISAT 2.2.0 [Eén and Sörensson 2004].

The first experiment studies the efficiency of the lookahead DPLL and the vsids scattering partitioning functions (see Sect. 5.5). Results for a third partitioning function based on combining scattering and lookahead are presented in [PIV]. All partitioning functions produce in these experiments eight derived instances. The maximum size $MaxDBSize$ of the sets of non-unit learned clauses $Lrnt(\nu_0)$ is limited to 100 000 literals, while the sizes of the sets of learned clauses was zero for all other nodes. This choice was made to keep the experiments as simple as possible. The same limit of 100 000 literals is used as the maximum size $SubmSize$ of the set of

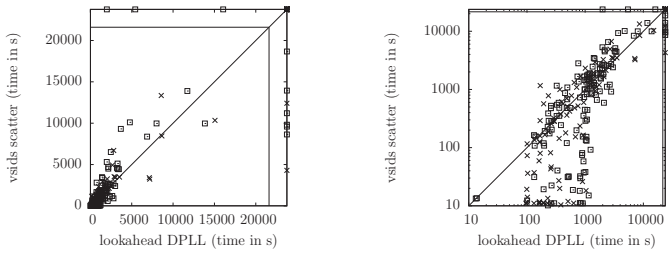


Figure 6.5. Comparing the vsids scattering heuristic against the lookahead DPLL heuristic. Crosses (×) represent satisfiable and boxes (□) unsatisfiable instances.

learned clauses provided for each CACP solver. Similar to the CL-SDSAT approach, no limits are placed on the number of learned unit clauses.

The first comparison is done on the partitioning functions based on the vsids scattering and the DPLL lookahead approaches. The application category benchmarks from SAT-Comp 2009 were tried using flag based tagging in the underlying constraint aware solver. Both partitioning functions were allowed to run at most 300 seconds while producing the eight derived formulas. The results are shown as scatter plots in Fig. 6.5 with both logarithmic (right) and linear (left) scale. The results show that the vsids scattering partition function usually performs better than the lookahead DPLL partitioning function in "easy" benchmarks solvable in less than 100 seconds, but also in the most difficult benchmarks where the iterative partitioning using the lookahead DPLL partitioning function is not able to determine satisfiability. The fast solving times for easy instances can be explained by the nature of the partitioning function and the delays in m-grid. The vsids scattering partitioning function runs essentially a local CDCL SAT solver on the formulas and can therefore find solutions without waiting for the results coming from the grid. The lookahead DPLL partitioning function, on the other hand, is not tuned towards finding solutions and therefore the formulas get solved in the grid. The good performance of the vsids scatter partitioning function in the most difficult instances is more difficult to explain, and could even be an artifact of the benchmark set. Since the grid and cloud based computing approaches are naturally tuned towards solving the difficult benchmarks because of the high delays, a conclusion can be drawn that the vsids scatter partitioning function performs better than the lookahead DPLL partitioning function in this context.

The iterative partitioning approach (Part-Tree) is compared to the iter-

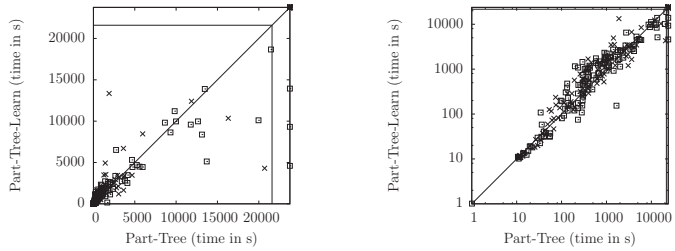


Figure 6.6. Comparing the Part-Tree approach against the Part-Tree-Learn approach. Crosses (×) represent satisfiable and boxes (□) unsatisfiable instances.

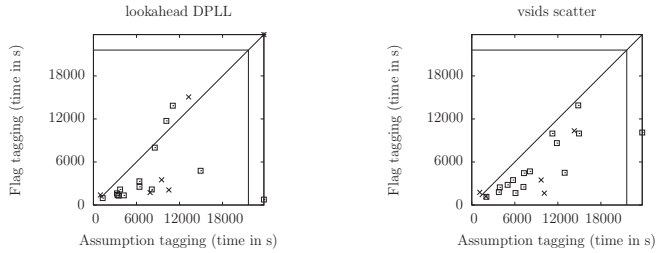


Figure 6.7. Comparing the assumption and flag tagging approaches for the lookahead DPLL partitioning function (left) and vsids scatter partitioning function (right).

ative partitioning approach with cumulative learning (Part-Tree-Learn) in Fig. 6.6. The benchmarks that can be solved in roughly an hour are faster to solve without learning, whereas the more difficult benchmarks can be more efficiently solved when learning is enabled. It is enlightening to compare these results to the results, for example, in Figs. 4.4 and 4.5 for the CL-SDSAT approach. It is possible that the slow down in solving these "mid-range" instances results from the initial low-quality clauses observed in cumulative learning and CL-SDSAT.

Finally, the comparison in Fig. 6.7 shows the difference between the assumption tagging and flag tagging approaches on some of the more challenging instances from SAT-Comp 2009. The assumption tagging approach performs usually worse than the flag tagging approach, a result that could already be extrapolated from Fig. 6.4. However, it is important to note that the assumption tagging is not used to its full potential in this work, since learned clauses are only updated to the root of the partition tree. The performance of the assumption tagging approach depends on the partitioning function. Significantly better results are obtained with the lookahead DPLL, while the results on vsids scatter are less encourag-

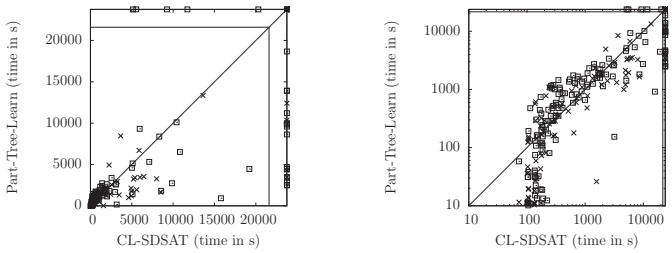


Figure 6.8. Comparing the CL-SDSAT approach against the iterative partitioning approach with cumulative learning. Crosses (×) represent satisfiable and boxes (□) unsatisfiable instances.

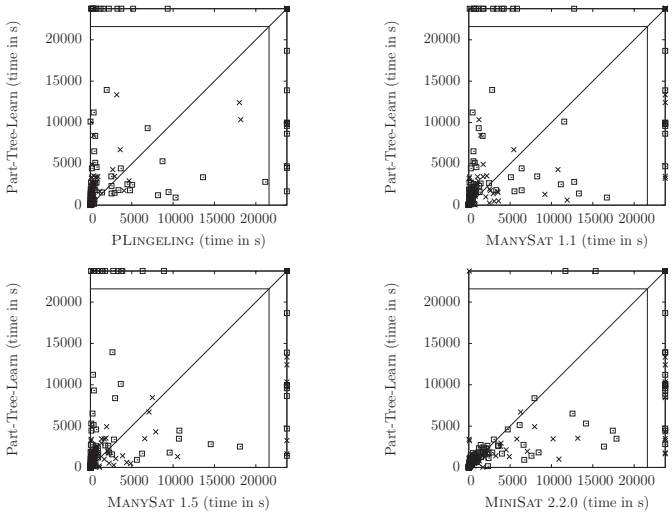


Figure 6.9. Comparing iterative partitioning with cumulative learning against PLINGELING (top left), MANYSAT 1.1 (top right), MANYSAT 1.5 (bottom left), and MINISAT 2.2.0 (bottom right).

ing. As mentioned in Sect. 6.2, a CACP solver which uses the assumption tagging approach can determine unsatisfiability of nodes that are on the path leading to the node being solved. This can potentially speed up the solving of the original formula with the iterative partitioning approach with cumulative learning. Unfortunately, in our experiments this hardly ever happened, a result that is reflected also in the comparison in Fig. 6.7.

The comparison in Fig. 6.8 reports how the iterative partitioning approach with cumulative learning (Part-Tree-Learn) performs against the CL-SDSAT approach. The Part-Tree-Learn approach performs particularly well again on the more difficult instances, whereas CL-SDSAT is able to solve faster many instances from the instances solvable in roughly one hour.

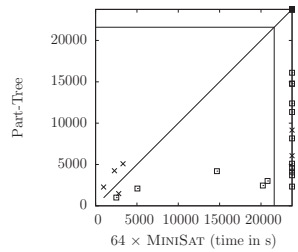


Figure 6.10. Comparing the iterative partitioning approach against the minimum over 64 runs of MINISAT.

Figure 6.9 collects comparisons against several well performing SAT solvers that are not designed to be run in grids or clouds, but instead use one or more cores sharing the memory of a single computer. All solvers were run on a 12-core AMD Opteron 2435 so that the whole computing node was reserved solely for one process, time limit was six hours and memory limit was 24 GB. The solver PLINGELING version 276, described in [Biere 2010], was run using the full 12 cores available, whereas MANYSAT 1.1 and MANYSAT 1.5 [Hamadi et al. 2009b] were run with the default setting using four cores. The comparison to MINISAT 2.2.0 is interesting, as it is the underlying solver in the partition tree approach. MINISAT 2.2.0 uses only a single core.

Finally Fig. 6.10 shows the performance of the iterative partitioning approach against MINISAT in an arrangement where 64 copies of MINISAT are run in parallel. The benchmark set consists again of some of the more challenging instances of SAT-Comp 2009. The results are in line with those in Ch. 4, showing that for these benchmarks a pure parallel portfolio approach with no clause sharing is not competitive.

It would be interesting to compare the iterative partitioning approach against a guiding path based parallel solver. Unfortunately, such solvers were not found at the time of this writing and therefore must be left for further work.

7. Conclusions

This work studies parallel SAT solving in a grid or cloud computing environment, where resources consist of several computing clusters that are distributed over a large geographical area. Several SAT solving approaches are developed for the environment and experimented using a large benchmark set consisting of instances from recent SAT solver competitions. The results are encouraging, as several instances that could not be solved with current state-of-the-art solvers within reasonable time limits could be solved with the presented approaches within hours. One of the most interesting future directions for the work started in here is in studying the behavior of the presented approaches in the important multicore computing environments. To this end, the experiments are presented in the work so that the results should, to some extent at least, generalize beyond the still emerging grids and clouds.

7.1 Summary of the Contributions

The work first defines an abstract model of a computing grid, based on the ideas in [Jensen et al. 2005] and experiences on the NorduGrid system [Ellert et al. 2007]. Throughout the experiments of this work the grid environment is operated through a job submission system, a job manager, running in the user's computer. The task of the job manager is to ensure that a computation, called a *job*, requested by the user is executed and the results are reported back within reasonable time. The job manager is also described in [Pitkanen et al. 2008], where it is used in medical image processing. Some central assumptions in the development of the job manager, as well as the model of computing, are that in a large grid jobs are bound to sometimes fail, the distances to resources cause unavoidable

delays in job and result transmission, and the resources might sometimes be overloaded causing high queuing times.

The grid environment is used in studying the effect of delays and resource bounds on the *simple distributed SAT solving* (SDSAT) framework, based on solving a single instance with several randomized SAT solvers in parallel. The SDSAT framework is studied in the context of several *restart strategies* [Luby et al. 1993]. Based on the experimental evaluation, the work describes a method for efficiently solving a set of SAT instances in a grid. This method is general in the sense that it works on all so called Las Vegas type algorithms [Babai 1979; Papadimitriou 1994] and instances which can be associated with a run time behavior similar to those of SAT instances.

Based on the results, the work devises the *Clause-Learning Simple Distributed SAT Solving* (CL-SDSAT) framework which incorporates the powerful clause learning techniques of modern SAT solvers to the SDSAT approach. The CL-SDSAT framework is analyzed with respect to several learning strategies using controlled experiments and shown to efficiently scale to a large amount of distributed resources in a setting where clauses are *cumulatively learned* in parallel running solvers. The efficiency of CL-SDSAT is further demonstrated by solving several well-known and hard SAT problems using an implementation of CL-SDSAT and a production level grid.

Many parallel SAT solvers are based on dividing the search space of a formula by inserting *partitioning constraints* and solving the resulting partitions in parallel. An idealized version of the approach, called *plain partitioning*, is studied analytically using a natural model for constructing the partitions. An analysis of plain partitioning shows that for unsatisfiable instances the approach is "risky" in the sense that increasing the number of parallel partitions increases the expected run time of the approach. This observation motivates firstly the study of alternate forms of dividing search spaces, resulting in the *safe* and *repeated partitioning approaches*, and the *iterative partitioning approach*. Secondly, the success in producing partitions with equally sized search spaces is critical to avoiding the risks in the plain partitioning approach. Different efficient *partitioning functions* for this task are developed and studied in particular on the most challenging benchmarks.

The final topic of the work is integrating cumulative, parallel clause learning, studied for CL-SDSAT, to the iterative partitioning approach.

The problem is substantially more difficult here, as the clauses learned in solving one partition are not necessarily logical consequences of another partition. The study results in the *assumption* and *flag tagging* approaches able to efficiently track the dependency of the learned clauses on the partitioning constraints enabling sharing of the learned clauses between partitions in a sound way.

The results of this work show that the presented, relatively restricted frameworks are sufficient to yield concrete speed-up on many known hard SAT instances compared to state-of-the-art SAT solvers. Furthermore, the experimental evaluation using instances from both the SAT competition 2007 and SAT competition 2009 (`http://www.satcompetition.org/`) resulted in solving several problems which were not solved by any SAT solver in the competition, and even problems that could not be solved using no time limitations at all. The literature reports few positive results obtained on parallel SAT solving when the actual solving time is measured, and therefore the significance of the results presented in this work is also in showing that high-latency grid environments can be efficiently used in algorithms that are not trivially distributable. The author of this work sees this as an important contribution, since grid-based computing has been gaining more popularity among those in possession of computational resources, and will therefore be of interest to a wider audience in the future. While similar results have been obtained for highly controlled grid environments [Bal and Verstoep 2008], the results reported here are one of the first for production-level grids.

7.2 Further Work

Given the practical significance of constraint programming in general and the propositional satisfiability problem in particular, the topic of this work seems far from exhaustively researched and understood. Although grid or cloud computing might not be novel ideas and have been known with different names for a long time, their economical and practical values have been recognized only recently, due to advances both in algorithms and hardware. This section discusses some of the new intriguing research questions raised by the results of this work.

One of the most obvious questions is the scalability of the presented approaches. Although some results for this are presented in [PI] for the

SDSAT and in [PII] for the CL-SDSAT approaches, as well as for the plain partitioning approach in [PIII], the more complex repeated, safe, and iterative partitioning approaches are yet to be studied in this respect.

The initial results on the efficiency of the repeated partitioning approach reported in [PIII] are highly encouraging. It seems that combining ideas from the repeated partitioning to the iterative partitioning, while far from straightforward to implement, could provide a robust approach for solving formulas beyond the reach of current state-of-the-art.

The experiments in [PV] suggest that iterative partitioning approach can be substantially improved by sharing some of the clauses learned in different parts of the partition tree. Based on the results it is possible that a more general clause sharing scheme would increase the performance even more. For example, in the current implementation it was decided that the non-unit clauses are only shared if they are logical consequences of the original instance, while the described framework supports sound sharing of any clauses based on the tagging information.

As mentioned in the beginning of the chapter, the experiments of this work are to a large extent motivated by the grid and cloud computing environments. As the discussed approaches proved to be highly efficient in these experiments, it is also natural to ask how they would perform in multi-core environments. These environments differ from grid computing by being more predictable and enabling more efficient ways of communication, but provide a lower number of computing resources and have usually more congested, shared memory. Implementing, for example, the iterative partitioning approach to a multi-core environment is an interesting future challenge.

Finally the results here can be studied in more general context of other constraint programming paradigms. For example, much analytical work should immediately be applicable to parallelizing ASP, SMT, and more general constraint programming solvers.

References

ÁBRAHÁM, E., SCHUBERT, T., BECKER, B., FRÄNZLE, M., AND HERDE, C. 2011. Parallel SAT solving in bounded model checking. *Journal of Logic and Computation 21,* 1, 5–21.

ASÍN, R., NIEUWENHUIS, R., OLIVERAS, A., AND RODRÍGUEZ-CARBONELL, E. 2010. Practical algorithms for unsatisfiability proof and core generation in SAT solvers. *Advances in Computers 23,* 2–3, 145–157.

AUDEMARD, G. AND SIMON, L. 2009. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*. 399–404.

BABAI, L. 1979. Monte-Carlo algorithms in graph isomorphism testing. Tech. Rep. DMS 79-10, Université de Montréal.

BAL, H. AND VERSTOEP, K. 2008. Large-scale parallel computing on grids. *Electronic Notes in Theoretical Computer Science 220*, 3–17.

BALDUCCINI, M., PONTELLI, E., ELKHATIB, O., AND LE, H. 2005. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing 31,* 6, 608–647.

BEN-ELIYAHU, R. AND DECHTER, R. 1994. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence 12,* 1-2, 53–87.

BIERE, A. 2008. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation 4,* 2–4, 75–97.

BIERE, A. 2010. Lingeling, Plingeling, PicoSAT and PrecoSAT at SAT race 2010. Technical Report 10/1, Institute for Formal Models and Verification, Johannes Kepler University.

BIERE, A. AND KUNZ, W. 2002. SAT and ATPG: Boolean engines for formal hardware verification. In *Proceedings of 20th IEEE/ACM International Conference on Computer Aided Design (ICCAD 2002)*. Association for Computing Machinery, 782–785.

BLOCHINGER, W., SINZ, C., AND KÜCHLIN, W. 2003. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing 29,* 7, 969–994.

BLUMOFE, R. D. AND LEISERSON, C. E. 1994. Scheduling multithreaded computations by work stealing. In *Proceedings of the 35th Annual Symposium on Foundations of Computer Science (FOCS 1994)*. IEEE Press, 356–368.

BÖHM, M. AND SPECKENMEYER, E. 1996. A fast parallel SAT-solver: Efficient workload balancing. *Annals of Mathematics and Artificial Intelligence 17,* 4–3, 381–400.

BONACINA, M. P. 1999. A model and a first analysis of distributed-search contraction-

based strategies. *Annals of Mathematics and Artificial Intelligence 27,* 1–4, 149–199.

BONACINA, M. P. 2000. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence 29,* 1–4, 223–257.

BONACINA, M. P. 2001. Combination of distributed search and multi-search in Peers-mcd.d. In *Proceedings of the 1st International Joint Conference on Automated Reasoning (IJCAR 2001)*. Lecture Notes in Artificial Intelligence, vol. 2083. Springer, 448–452.

BORDEAUX, L., HAMADI, Y., AND SAMULOWITZ, H. 2009. Experiments with massively parallel constraint solving. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*. 443–448.

BORDEAUX, L., HAMADI, Y., AND ZHANG, L. 2006. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys 38,* 4.

BOZZANO, M., BRUTTOMESSO, R., CIMATTI, A., JUNTTILA, T., VAN ROSSUM, P., SCHULZ, S., AND SEBASTIANI, R. 2005. MathSAT: Tight integration of SAT and mathematical decision procedures. *Journal of Automated Reasoning 35,* 1-3, 265–293.

BRYANT, R. 1986. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers 35,* 8, 677–691.

CHRABAKH, W. AND WOLSKI, R. 2006. GridSAT: a system for solving satisfiability problems using a computational grid. *Parallel Computing 32,* 9, 660–687.

CHU, G., SCHULTE, C., AND STUCKEY, P. J. 2009. Confidence-based work stealing in parallel constraint programming. In *Proceedings of the 15th International Conference on Principles and Practice of Constraint Programming (CP 2009)*. Lecture Notes in Computer Science, vol. 5732. Springer, 226–241.

COOK, S. 1971. The complexity of theorem-proving procedures. In *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing*. Association for Computing Machinery, 151–158.

DARWICHE, A. 2001. Decomposable negation normal form. *Journal of the ACM 48,* 4, 608–647.

DAVIS, M., LOGEMANN, G., AND LOVELAND, D. 1962. A machine program for theorem proving. *Communications of the ACM 5,* 7, 394–397.

DAVIS, M. AND PUTNAM, H. 1960. A computing procedure for quantification theory. *Journal of the ACM 7,* 3, 201–215.

DE MOURA, L. M. AND BJØRNER, N. 2008. Z3: An efficient SMT solver. In *Proceedings of the 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS 2008)*. 337–340.

DECHTER, R. 1990. Enhancement schemes for constraint processing: Backjumping, learning, and cutset decomposition. *Artificial Intelligence 41,* 3, 273–312.

DEQUEN, G., VANDER-SWALMEN, P., AND KRAJECKI, M. 2009. Toward easy parallel SAT solving. In *Proceedings of the 21st IEEE International Concerence on Tools with Artificial Intelligence (ICTAI 2009)*. IEEE Press, 425–432.

DIMOPOULOS, Y., NEBEL, B., AND KOEHLER, J. 1997. Encoding planning problems in nonmonotonic logic programs. In *Proceedings of the 4th European Conference on Planning (ECP 1997)*. Lecture Notes in Artificial Intelligence, vol. 1348. Springer, 169–181.

DRESCHER, C., GEBSER, M., GROTE, T., KAUFMANN, B., KÖNIG, A., OSTROWSKI, M., AND SCHAUB, T. 2008. Conflict-driven disjunctive answer set solving. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning (KR 2008)*. AAAI Press, 422–432.

EÉN, N., MISHCHENKO, A., AND AMLA, N. 2010. A single-instance incremental SAT formulation of proof- and counterexample-based abstraction. *Computing Research Repository abs/1008.2021*.

EÉN, N. AND SÖRENSSON, N. 2003. Temporal induction by incremental SAT solving. *Electronic Notes in Theoretical Computer Science 89,* 4, 534–560.

EÉN, N. AND SÖRENSSON, N. 2004. An extensible SAT-solver. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Selected Revised Papers*. Lecture Notes in Computer Science, vol. 2919. Springer, 502–518.

ELLERT, M., GRØNAGER, M., KONSTANTINOV, A., KÓNYA, B., LINDEMANN, J., LIVENSON, I., NIELSEN, J. L., NIINIMÄKI, M., SMIRNOVA, O., AND WÄÄNÄNEN, A. 2007. Advanced resource connector middleware for lightweight computational grids. *Future Generation Computer Systems 23,* 2, 219–240.

ERDEM, E. AND TÜRE, F. 2008. Efficient haplotype inference with answer set programming. In *Proceedings of the 23rd AAAI Conference on Artificial Intelligence (AAAI 2008)*. AAAI Press, 436–441.

FELDMAN, Y., DERSHOWITZ, N., AND HANNA, Z. 2005. Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science 128,* 3, 75–90.

GANZINGER, H., HAGEN, G., NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2004. DPLL(T): Fast decision procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification (CAV 2004)*. Lecture Notes in Computer Science, vol. 3114. Springer, 175–188.

GEBSER, M., KAMINSKI, R., KAUFMANN, B., SCHAUB, T., AND SCHNOR, B. 2011. Cluster-based ASP solving with Claspar. In *Proceedings of the 11th International Conference on Logic Programming and Nonmonotonic Reasoning (LPNMR 2011)*. Lecture Notes in Computer Science, vol. 6645. Springer, 364–369.

GEBSER, M., KAUFMANN, B., NEUMANN, A., AND SCHAUB, T. 2007. Clasp: A conflict-driven answer set solver. In *Proceedings of the 9th International Conference on Logic Programmin and Non-monotonic Reasoning (LPNMR 2007)*. Number 4483 in Lecture Notes in Computer Science. Springer, 260 – 265.

GELFOND, M. AND LIFSCHITZ, V. 1988. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming (ICLP 1988)*. MIT Press, 1070–1080.

GENT, I. P., JEFFERSON, C., KOTTHOFF, L., MIGUEL, I., MOORE, N. C. A., NIGHTINGALE, P., AND PETRIE, K. E. 2010. Learning when to use lazy learning in constraint solving. In *Proceedings of the 19th European Conference on Artificial Intelligence (ECAI 2010)*. IOS Press, 873–878.

GIL, L., FLORES, P., AND SILVEIRA, L. M. 2009. PMSat: a parallel version of MiniSAT. *Journal on Satisfiability, Boolean Modeling and Computation 6,* 1–3, 71–98.

GOMES, C. P. AND SELMAN, B. 2001. Algorithm portfolios. *Artificial Intelligence 126,* 1–2, 43–62.

GOMES, C. P., SELMAN, B., CRATO, N., AND KAUTZ, H. 2000. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning 24,* 1/2, 67–100.

GOMES, C. P., SELMAN, B., AND KAUTZ, H. 1998. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI 1998)*. AAAI Press, 431–437.

GRAMA, A. AND KUMAR, V. 1999. State of the art in parallel search techniques for discrete optimization problems. *IEEE Transactions on Knowledge and Data Engineering 11,* 1, 28–34.

GRESSMANN, J., JANHUNEN, T., MERCER, R. E., SCHAUB, T., THIELE, S., AND TICHY, R. 2005. Platypus: A platform for distributed answer set solving. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning, (LPNMR 2005)*. Lecture Notes in Computer Science, vol. 3662. Springer, 227–239.

GUO, L., HAMADI, Y., JABBOUR, S., AND SAIS, L. 2010. Diversification and intensification in parallel SAT solving. In *16th International Conference on Principles and Practice of Constraint Programming (CP 2010)*. Lecture Notes in Computer Science, vol. 6308. Springer, 252 – 265.

HAMADI, Y., JABBOUR, S., AND SAIS, L. 2009a. Control-based clause sharing in parallel SAT solving. In *Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009)*. 499–504.

HAMADI, Y., JABBOUR, S., AND SAIS, L. 2009b. ManySAT: a parallel SAT solver. *Journal on Satisfiability, Boolean Modeling and Computation 6,* 4, 245 – 262.

HELJANKO, K. 1999. Using logic programs with stable model semantics to solve deadlock and reachability problems for 1-safe Petri nets. *Fundamenta Informaticae 37,* 3, 247–268.

HERBSTRITT, M. AND BECKER, B. 2003. Conflict-based selection of branching rules. In *Proceedings of the 6th International Conference on Theory and Applications of Satisfiability Testing (SAT 2003), Selected Revised Papers*. Lecture Notes in Computer Science, vol. 2919. Springer, 441–451.

HEULE, M. AND VAN MAAREN, H. 2006. March_dl: Adding adaptive heuristics and a new branching strategy. *Journal on Satisfiability, Boolean Modeling and Computation 2,* 1–4, 47–59.

HEULE, M. AND VAN MAAREN, H. 2009. Look-ahead based SAT solvers. In *Handbook of Satisfiability*. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press, 155–184.

HOOKER, J. N. AND VINAY, V. 1995. Branching rules for satisfiability. *Journal of Automated Reasoning 15,* 3, 359–383.

HUANG, J. 2008. Universal Booleanization of constraint models. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming (CP 2008)*. Lecture Notes in Computer Science, vol. 5202. Springer, 144–158.

HUBERMAN, B. A., LUKOSE, R. M., AND HOGG, T. 1997. An economics approach to hard computational problems. *Science 275,* 5296, 51–54.

HYVÄRINEN, A. E. J., JUNTTILA, T., AND NIEMELÄ, I. 2006. A distribution method for solving SAT in grids. In *Proceedings of the 9th International Conference on Theory*

and Applications of Satisfiability Testing (SAT 2006). Lecture Notes in Computer Science, vol. 4121. Springer, 430–435.

HYVÄRINEN, A. E. J., JUNTTILA, T., AND NIEMELÄ, I. 2008. Incorporating learning in grid-based randomized SAT solving. In *Proceedings of the 13th International Conference on Artificial Intelligence: Methodology, Systems, Applications (AIMSA 2008)*. Lecture Notes in Artificial Intelligence, vol. 5253. Springer, 247–261.

HYVÄRINEN, A. E. J., JUNTTILA, T., AND NIEMELÄ, I. 2009. Partitioning the search space of a randomized search. In *Proceedings of the 11th International Conference of the Italian Association for Artificial Intelligence (AI*IA 2009)*. Lecture Notes in Artificial Intelligence, vol. 5883. Springer, 243–252.

INOUE, K., SOH, T., UEDA, S., SASAURA, Y., BANBARA, M., AND TAMURA, N. 2006. A competitive and cooperative approach to propositional satisfiability. *Discrete Applied Mathematics 154,* 16, 2291–2306.

IRGENS, M. AND HAVENS, W. S. 2004. On selection strategies for the DPLL algorithm. In *Proceedings of the 17th Conference of the Canadian Society for Computational Studies of Intelligence (CCAI 2004)*. Lecture Notes in Artificial Intelligence, vol. 3060. Springer, 277–291.

JANAKIRAM, V. K., AGRAWAL, D. P., AND MEHROTRA, R. 1988. A randomized parallel backtracking algorithm. *IEEE Transactions on Computers 37,* 12, 1665–1676.

JANAKIRAM, V. K., GEHRINGER, E. F., AGRAWAL, D. P., AND MEHROTRA, R. 1988. A randomized parallel branch-and-bound algorithm. *International Journal of Parallel Programming 17,* 3, 277 – 301.

JANHUNEN, T. 2006. Some (in)translatability results for normal logic programs and propositional theories. *Journal of Applied Non-Classical Logics 16,* 1-2, 35–86.

JENSEN, H. T., KLEIST, J., AND LETH, J. R. 2005. A framework for job management in the NorduGrid ARC middleware. In *Proceedings of the European Grid Conference (EGC 2005), Revised Selected Papers*. Lecture Notes in Computer Science, vol. 3470. Springer, 861–871.

JEROSLOW, R. G. AND WANG, J. 1990. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence 1*, 167–187.

JURKOWIAK, B., LI, C., AND UTARD, G. 2005. A parallelization scheme based on work stealing for a class of SAT solvers. *Journal of Automated Reasoning 34,* 1, 73–101.

KAUTZ, H. AND SELMAN, B. 1992. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI 1992)*. John Wiley and Sons, 359–363.

LAGOUDAKIS, M. G. AND LITTMAN, M. L. 2001. Learning to select branching rules in the DPLL procedure for satisfiability. *Electronic Notes in Discrete Mathematics 9*, 344–359.

LARRABEE, T. 1992. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design 11,* 1, 6–22.

LE, H. V. AND PONTELLI, E. 2007. Dynamic scheduling in parallel answer set programming solvers. In *Proceedings of the 2007 Spring Simulation Multiconference (SpringSim 2007) Volume 2*. Association for Computing Machinery, 367–374.

LEONE, N., PFEIFER, G., FABER, W., EITER, T., GOTTLOB, G., PERRI, S., AND SCARCELLO, F. 2006. The DLV system for knowledge representation and reasoning. *ACM Transactions on Computational Logic 7,* 3, 499–562.

LI, C. M. AND ANBULAGAN. 1997a. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), Volume 1*. Morgan Kaufmann, 366–371.

LI, C. M. AND ANBULAGAN. 1997b. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the 3rd International Conference on Principles and Practice of Constraint Programming (CP 1997)*. Lecture Notes in Computer Science, vol. 1330. Springer, 341–355.

LI, G.-J. AND WAH, B. W. 1990. Computational efficiency of parallel combinatorial OR-Tree searches. *IEEE Transactions on Software Engineering 16,* 1, 13–31.

LIN, F. AND ZHAO, Y. 2004. ASSAT: computing answer sets of a logic program by SAT solvers. *Artificial Intelligence 157,* 1-2, 115–137.

LUBY, M. AND ERTEL, W. 1994. Optimal parallelization of Las Vegas algorithms. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science (STACS 1994)*. Lecture Notes in Computer Science, vol. 775. Springer, 463–474.

LUBY, M., SINCLAIR, A., AND ZUCKERMAN, D. 1993. Optimal speedup of Las Vegas algorithms. *Information Processing Letters 47,* 4, 173–180.

MANCINI, T., MICALETTO, D., PATRIZI, F., AND CADOLI, M. 2008. Evaluating ASP and commercial solvers on the CSPLib. *Constraints 13,* 4, 407–436.

MARQUES-SILVA, J. P. 1999. The impact of branching heuristics in propositional satisfiability algorithms. In *Proceedings of the 9th Portugese Conference on Artificial Intelligence (EPIA 1999)*. Lecture Notes in Computer Science, vol. 1695. Springer, 62–74.

MARQUES-SILVA, J. P. 2008. Model checking with Boolean satisfiability. *Journal of Algorithms 63,* 1-3, 3–16.

MARQUES-SILVA, J. P. AND SAKALLAH, K. A. 1999. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers 48,* 5, 506–521.

MARTINS, R., MANQUIHO, V., AND LYNCE, I. 2010. Improving search space splitting for parallel SAT solving. In *Proceedings of the 22th IEEE International Conference on Tools with Artificial Intelligence (ICTAI 2010)*. IEEE Press, 336–343.

MICHEL, L., SEE, A., AND VAN HENTENRYCK, P. 2007. Parallelizing constraint programs transparently. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming (CP 2007)*. Lecture Notes in Computer Science, vol. 4741. Springer, 514–528.

MIRONOV, I. AND ZHANG, L. 2006. Applications of SAT solvers to cryptanalysis of hash functions. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing (SAT 2006)*. Lecture Notes in Computer Science, vol. 4121. Springer, 102–115.

MOSKEWICZ, M. W., MADIGAN, C. F., ZHAO, Y., ZHANG, L., AND MALIK, S. 2001. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference (DAC 2001)*. Association for Computing Machinery, 530–535.

NIEMELÄ, I. 1999. Logic programming with stable model semantics as a constraint programming paradigm. *Annals of Mathematics and Artificial Intelligence 25,* 3-4, 241–273.

NIEUWENHUIS, R., OLIVERAS, A., AND TINELLI, C. 2006. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to

DPLL(T). *Journal of the ACM 53,* 6, 937–977.

OHMURA, K. AND UEDA, K. 2009. c-sat: A parallel SAT solver for clusters. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*. Lecture Notes in Computer Science, vol. 5584. Springer, 524–537.

OKUSHI, F. 1999. Parallel cooperative propositional theorem proving. *Annals of Mathematics and Artificial Intelligence 26,* 1-4, 59–85.

PAPADIMITRIOU, C. H. 1994. *Computational Complexity*. Addison-Wesley, Boston, MA.

PETRIK, M. AND ZILBERSTEIN, S. 2006. Learning parallel portfolios of algorithms. *Annals of Mathematics and Artificial Intelligence 48,* 1-2, 85–106.

PITKANEN, M. J., ZHOU, X., HYVÄRINEN, A. E. J., AND MÜLLER, H. 2008. Using the grid for enhancing the performance of a medical image search engine. In *Proceedings of the 21st IEEE/ACM International Symposium on Computer-Based Medical Systems (CBMS 2008)*. IEEE Press, 367–372.

PONTELLI, E., VILLAVERDE, K., GUO, H.-F., AND GUPTA, G. 2007. PALS: Efficient or-parallel execution of Prolog on Beowulf clusters. *Theory and Practice of Logic Programming 7,* 6, 633–695.

PRESTWICH, S. AND MUDAMBI, S. 1995. Improved branch and bound in constraint logic programming. In *Proceedings of the 1st International Conference on Principles and Practie of Constraint Programming (CP 1995)*. Lecture Notes in Computer Science, vol. 976. Springer, 534–548.

RANJAN, D., PONTELLI, E., AND GUPTA, G. 1999. On the complexity of or-parallelism. *New Generation Computing 17,* 3, 285–307.

RICE, J. R. 1976. The algorithm selection problem. *Advances in Computers 15*, 65 – 118.

ROSSI, F., VAN BEEK, P., AND WALSH, T., Eds. 2006. *Handbook of Constraint Programming*. Elsevier, Amsterdam, The Netherlands.

SCHUBERT, T., LEWIS, M., AND BECKER, B. 2009. PaMiraXT: Parallel SAT solving with threads and message passing. *Journal on Satisfiability, Boolean Modeling and Computation 6,* 4, 203–222.

SCHULZ, S. AND BLOCHINGER, W. 2010. Parallel SAT solving on peer-to-peer desktop grids. *Journal of Grid Computing 8*, 443–471.

SEBASTIANI, R. 2007. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation 3,* 3–4, 141–224.

SEGRE, A. M., FORMAN, S. L., RESTA, G., AND WILDENBERG, A. 2002. Nagging: A scalable fault-tolerant paradigm for distributed search. *Artificial Intelligence 140,* 1/2, 71–106.

SELMAN, B. AND KAUTZ, H. 1993. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI 1993)*. AAAI Press, 46–51.

SHAPIRO, E. Y., WARREN, D. H. D., FUCHI, K., KOWALSKI, R. A., FURUKAWA, K., UEDA, K., KAHN, K. M., CHIKAYAMA, T., AND TICK, E. 1993. The fifth generation project: Personal perspectives. *Communications of the ACM 36,* 3, 46–103.

SIMONS, P., NIEMELÄ, I., AND SOININEN, T. 2002. Extending and implementing the stable model semantics. *Artificial Intelligence 138,* 1-2, 181–234.

SOININEN, T. AND NIEMELÄ, I. 1999. Developing a declarative rule language for applications in product configuration. In *Proceedings of the 1st International Workshop on Practical Aspects of Declarative Languages (PADL 1999, Co-located with POPL 1999)*. Lecture Notes in Computer Science, vol. 1551. Springer, 305–319.

SOOS, M., NOHL, K., AND CASTELLUCCIA, C. 2009. Extending SAT solvers to cryptographic problems. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*. Lecture Notes in Computer Science, vol. 5584. Springer, 244–257.

SÖRENSSON, N. AND BIERE, A. 2009. Minimizing learned clauses. In *Proceedings of the 12th International Conference on Theory and Applications of Satisfiability Testing (SAT 2009)*. Lecture Notes in Computer Science, vol. 5584. Springer, 237–243.

SPECKENMEYER, E. 1989. Is average superlinear speedup possible? In *Proceedings of the 2nd Workshop on Computer Science Logic (CSL 1988)*. Lecture Notes in Computer Science, vol. 385. Springer, 301–312.

SPECKENMEYER, E., MONIEN, B., AND VORNBERGER, O. 1988. Superlinear speedup for parallel backtracking. In *Proceedings of the 1st international conference on Supercomputing (SC 1987)*. Lecture Notes in Computer Science, vol. 297. Springer, 985–993.

STERLING, L. AND SHAPIRO, E. Y. 1987. *The Art of Prolog*. MIT Press, Cambridge, MA.

WALSH, T. 1999. Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence (IJCAI 1999)*. Morgan Kaufmann, 1172–1177.

WALSH, T. 2000. SAT v CSP. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming (CP 2000)*. Lecture Notes in Computer Science, vol. 1894. Springer, 441–456.

WIERINGA, S., NIEMENMAA, M., AND HELJANKO, K. 2009. Tarmo: A framework for parallelized bounded model checking. In *Proceedings of the 8th International Workshop on Parallel and Distributed Methods in verifiCation (PDMC 2009)*. Electronic Proceedings in Theoretical Computer Science, vol. 14. 62–76.

WILLIAMS, R., GOMES, C. P., AND SELMAN, B. 2003. Backdoors to typical case complexity. In *Proceedings of the 18th International Joint Conference on Artificial Intelligence (IJCAI 2003)*. Morgan Kaufmann, 1173–1178.

WINTERSTEIGER, C. M., HAMADI, Y., AND DE MOURA, L. M. 2009. A concurrent portfolio approach to SMT solving. In *Proceedings of the 21st International Conference on Computer Aided Verification (CAV 2009)*. Lecture Notes in Computer Science, vol. 5643. Springer, 715–720.

ZHANG, H., BONACINA, M., AND HSIANG, J. 1996. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation 21,* 4, 543–560.

ZHANG, L., MADIGAN, C. F., MOSKEWICZ, M. H., AND MALIK, S. 2001. Efficient conflict driven learning in a Boolean satisfiability solver. In *Proceedings of the International Conference on Computer-Aided Design (ICCAD 2001)*. Association for Computing Machinery, 279–285.

ZHANG, L. AND MALIK, S. 2003. Validating SAT solvers using an independent resolution-based checker: Practical implementations and other applications. In

Proceedings of the 2003 Design, Automation and Test in Europe Conference and Exposition (DATE 2003). IEEE Press, 10880–10885.

DISSERTATIONS IN INFORMATION AND COMPUTER SCIENCE

TKK-ICS-D15 Heikinheimo, Hannes.
Extending Data Mining Techniques for Frequent Pattern Discovery: Trees, Low-Entropy Sets, and Crossmining. 2010.

TKK-ICS-D16 Hermelin, Miia.
Multidimensional Linear Cryptanalysis. 2010.

TKK-ICS-D17 Savia, Eerika.
Mutual Dependency-Based Modeling of Relevance in Co-Occurrence Data. 2010.

TKK-ICS-D18 Liitiäinen, Elia.
Advances in the Theory of Nearest Neighbor Distributions. 2010.

TKK-ICS-D19 Lahti, Leo.
Probabilistic Analysis of the Human Transcriptome with Side Information. 2010.

TKK-ICS-D20 Miche, Yoan.
Developing Fast Machine Learning Techniques with Applications to Steganalysis Problems. 2010.

TKK-ICS-D21 Sorjamaa, Antti.
Methodologies for Time Series Prediction and Missing Value Imputation. 2010.

TKK-ICS-D22 Schumacher, André
Distributed Optimization Algorithms for Multihop Wireless Networks. 2010.

Aalto-DD99/2011 Ojala, Markus
Randomization Algorithms for Assessing the Significance of Data Mining Results. 2011

Aalto-DD111/2011 Dubrovin, Jori
Efficient Symbolic Model Checking of Concurrent Systems. 2011

Today computing is ubiquitous, and the amount of data available for computing is overwhelming. This should enable us to make more educated decisions for achieving our goals, be it in economics, environmental or health care, constructing reliable software, or dating. One of the grand challenges for computing seems to be how to use this information in supporting our decision making. It is natural to start looking for the answers from the language of computing, that is, logic. This thesis studies algorithms for the propositional satisfiability problem. The algorithms are designed for computing exact solutions for a wide variety of such optimization and satisfiability questions. As these problems can be extremely difficult to solve, the algorithms are designed for large scale parallel computing in grids and clouds. The presented algorithms have succeeded in solving certain problems for the first time using a computing environment that is cost effective and sustainable.

Aalto-DD 118/2011

ISBN 978-952-60-4368-5 (pdf)
ISBN 978-952-60-4367-8
ISSN-L 1799-4934
ISSN 1799-4942 (pdf)
ISSN 1799-4934

Aalto University
School of Science
Department of Information and Computer Science
www.aalto.fi

BUSINESS + ECONOMY

ART + DESIGN + ARCHITECTURE

SCIENCE + TECHNOLOGY

CROSSOVER

DOCTORAL DISSERTATIONS