Helsinki University of Technology
Faculty of Information and Natural Sciences
Department of Information and Computer Science

Antti E. J. Hyvärinen

Approaches to Grid-Based SAT Solving

Licentiate's thesis submitted in partial fulfillment of the requirements for the degree of Licentiate of Science in Technology

Espoo, 4th March 2009

Supervisor:	Prof. Ilkka Niemelä
Instructor:	Tommi Junttila, D.Sc. (Tech.)

HELSINKI UNIVERSITY OF TECHNOLOGY
Faculty of Information and Natural Sciences
Department of Information and Computer Science

ABSTRACT OF LICENTIATE'S THESIS

Author	Antti E. J. Hyvärinen	Date	4th March 2009
		Pages	vii + 76
Title of thesis	Approaches to Grid-Based SAT Solving		
Professorship	Theoretical Computer Science	Code	T-119
Supervisor	Prof. Ilkka Niemelä		
Instructor	Tommi Junttila, D.Sc. (Tech.)		

In this work we develop techniques for using distributed computing resources to efficiently solve instances of the propositional satisfiability problem (SAT). The computing resources considered in this work are assumed to be geographically distributed and connected by a non-dedicated network. Such systems are typically referred to as computational grid environments.

The time a modern SAT solver consumes while solving an instance varies according to a random distribution. Unlike many other methods for distributed SAT solving, this work identifies the random distribution as a valuable resource for solving-time reduction. The methods which use randomness in the run times of a search algorithm, such as the ones discussed in this work, are examples of *multi-search*. The main contribution of this work is in developing and analyzing the multi-search approach in SAT solving and showing its efficiency with several experiments. For the purpose of the analysis, the work introduces a grid simulation model which captures several of the properties of a grid environment which are not observed in more traditional parallel computing systems.

The thesis develops two algorithmic frameworks for multi-search in SAT. The first, SDSAT, is based on using properties of the distribution of the solving time so that the expected time required to solve an instance is reduced. Based on the analysis of SDSAT, the thesis proposes an algorithm for efficiently using large number of computing resources simultaneously to solve collections of SAT instances. The analysis of SDSAT also motivates the second algorithmic framework, CL-SDSAT. The framework is used to efficiently solve many industrial SAT instances by carefully combining information learned in the distributed SAT solvers.

All methods described in the thesis are directly applicable in a wide range of grid environments and can be used together with virtually unmodified state-of-the-art SAT solvers. The methods are experimentally verified using standard benchmark SAT instances in a production-level grid environment. The experiments show that using the relatively simple methods developed in the thesis, SAT instances which cannot be solved efficiently in sequential settings can be now solved in a grid environment.

Keywords
Propositional Satisfiability, SAT Solving, Computational Grids, Distributed Search, Multi-search.

TEKNILLINEN KORKEAKOULU Informaatio- ja luonnontieteiden tiedekunta Tietojenkäsittelytieteen laitos	LISENSIATTITYÖN TIIVISTELMÄ

Tekijä	Antti E. J. Hyvärinen	Päiväys	4. maaliskuuta 2009
		Sivumäärä	vii + 76
Työn nimi	Lauselogiikan toteutuvuustarkastusmenetelmiä grid-ympäristössä		
Professuuri	Tietojenkäsittelyteoria	Koodi	T-119
Työn valvoja	Prof. Ilkka Niemelä		
Työn ohjaaja	TkT Tommi Junttila		

Tässä työssä tutkitaan ja kehitetään tehokkaita menetelmiä, joilla voidaan ratkoa lauselogiikan toteutuvuusongelman (SAT) lauseilmentymiä ympäristössä, joka koostuu hajautetuista laskentaresursseista. Tällaista ympäristöä, jossa laskentaresurssit on yhdistetty yleiskäyttöisellä tietoliikenneverkolla, kutsutaan usein grid-ympäristöksi.

Modernin toteutuvuustarkastimen yksittäisen lauseilmentymän ratkaisuun käyttämä aika voidaan kuvata satunnaisjakaumalla. Toisin kuin monissa muissa lauselogiikan lauseiden hajautettua ratkaisemista käsittelevissä töissä, tässä työssä keskitytään hyödyntämään tätä ajoajan satunnaisuutta ratkaisuun tarvittavan ajan lyhentämiseksi. Tämän *monihaun* periaatteen mukaisten menetelmien tehokkuutta analysoidaan työssä kehitetyssä simulaatioympäristössä, joka huomioi myös niitä grid-ympäristön piirteitä, joita ei esiinny tavanomaisissa rinnakkaislaskentaympäristöissä. Analyysin tulokset vahvistetaan aidossa grid-ympäristössä.

Monihakua varten työssä kehitetään kaksi algoritmista menetelmää. Ensimmäinen menetelmä perustuu toteutuvuustarkastimiin liittyvän ajoaikajakauman ominaisuuksien hyödyntämiseen ajoajan odotusarvon pienentämiseksi. Tämän menetelmän analyysin pohjalta työssä esitetään algoritmi, jolla suurta määrää hajautettuja laskentaresursseja voidaan hyödyntää lausejoukon lauseiden toteutuvuuden tarkastamisessa tehokkaasti. Ensimmäisen algoritmisen menetelmän pohjalta kehitetään toinen menetelmä, joka hyödyntää yksittäisten toteutuvuustarkastimien oppimaa informaatiota vaikeiden teollisuuslähtöisten lauselogiikan lauseiden toteutuvuuden tarkastamisessa. Työssä osoitetaan, että tietyin perustein valitut rinnakkain opitut apulauseet voivat huomattavasti tehostaa alkuperäisen lauseen toteutuvuuden tarkastamista.

Kaikkia työssä esiteltyjä menetelmiä voidaan soveltaa sellaisenaan grid-ympäristössä, ja menetelmien toteuttaminen vaatii vain pieniä muutoksia olemassaoleviin toteutuvuustarkastimiin. Menetelmien tehokkuutta arvioidaan ratkomalla niillä tunnettuja toteutuvuustarkastinten vertailuun käytettyjä lauseita tuotantokäytössä olevassa grid-ympäristössä. Työssä kehitetyillä kohtuullisen yksinkertaisilla menetelmillä pystytään ratkaisemaan toteutuvuusongelman ilmentymiä, joita ei ole aiemmin pystytty tehokkaasti käsittelemään olemassa olevilla lauselogiikan toteutuvuustarkastimilla.

Avainsanat	Lauselogiikka, lauselogiikan toteutuvuustarkastus, grid-ympäristöt, hajautettu haku, monihaku.

Contents

List of Figures

List of Tables

Preface

During the writing of this work as well as while carrying out the research presented in it I have had the privilege of receiving the overwhelming amounts of support, guidance, time, and advice from my supervisor, Professor Ilkka Niemelä, and my instructor, Doctor Tommi Junttila. I am grateful to Docent Tomi Janhunen for his valuable feedback to this work and insights to the research area, and would like to use this opportunity to express my hopes that these fruitful discussions will continue.

I thank my other co-authors Mikko Pitkänen, Xin Zhou, and Dr. Henning Müller. I have also enjoyed both professional support and warm friendship of Olli Tourunen, Michael Gindonis, and Dr. Ivan Degtyarenko from CSC, and numerous people, including Dr. Alexandr Konstantinov, Dr. Balázs Kónya and Dr. Oxana Smirnova, from NorduGrid. The research has been carried out in TKK Department of Computer Science and Engineering (Department of Information and Computer Science as of 2008), and I thank my colleagues here for creating the atmosphere where I so much enjoy working.

The work has been funded by TKK Department of Computer Science and Engineering, Helsinki Graduate School in Computer Science and Engineering, and Academy of Finland projects number 122399 and 211025. I have also received financial support from Jenny ja Antti Wihurin rahasto, Emil Aaltosen säätiö, CICM'08 Doctoral Programme, and the NorduGrid collaboration.

In addition to the valued professional contacts, I also want to thank Salla Hakkola, Heikki Hiltunen, Fredrik Löf and Kristian A. J. Meurman. Thank you for not letting me forget there are other worlds as well. I am grateful to everyone in my big family for their support. I am fortunate to have the best friends I could possibly imagine. And I thank my fiancée Hissu Mikkonen for everything she has given me while this work has progressed.

March, 2009

Antti E. J. Hyvärinen

Chapter 1

Introduction

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

The SAT problem belongs to the class of NP-complete decision problems [22], for which all known algorithms need in the worst case exponential number of steps with respect to the size of the problem instance. Furthermore, 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 [83].

Despite the notorious complexity of solving SAT instances, there are several implementations, called SAT *solvers*, capable of solving instances with a large number of variables efficiently. Since the first implementations of SAT solvers [25, 24], originally designed for first-order theorem proving, SAT solving has experienced tremendous enhancements in algorithm design, and recent solvers [28, 80] represent in many ways the state-of-the-art in solving NP-complete problems. Many engineering problems seem to be naturally transformable to a SAT instance, and can then be efficiently solved by a general purpose SAT solvers.

In recent years, SAT has experienced an increase of interest from the industry with applications such as planning [63], automated test pattern generation [67, 13], cryptanalysis [79] and bounded model checking [77]. The cumbersome and rather theoretical machinery developed for SAT solving is brought to the level of applications usually by a relatively simple process. Problems from the application domain are *encoded* as a SAT instance, and the instance is then solved using a general-purpose SAT solver. Crucial to the approach is, of course, that the satisfying truth assignment possibly obtained by the solver from the encoding can be mapped to a solution of the original problem in the application area. In this paradigm the programmer is relieved from algorithm design, a task already performed in building the solver. Instead, the emphasis is on how to build the encoding. This work does not directly address the issue of building the encoding, but rather studies how a SAT solver can efficiently be used for obtaining the satisfying truth assignment for the encoding.

The approach described above is an example of *declarative problem solving.* One of the most widely known examples of the approach is the Fifth Generation Project [96] which aimed at using massive amounts of parallel resources for efficient computing in artificial intelligence. The project built on ideas developed for Prolog systems [104], which, compared to SAT, carry a burden of their complicated procedural semantics. The complicated semantics render parallelizing Prolog systems more difficult [89].

While problem solving using Prolog systems resembles programming in traditional languages, a more declarative approach is that based on describing the applications as *constraint satisfaction problem* (CSP) instances [90]. Compared to SAT, CSP offers a wider spectrum of constraints over the variables for the programmer. In fact, SAT is viewed by many as a special case of CSP. However, this hierarchy is not always clear in practice: in some cases the constraints in CSP are compactly and more efficiently expressible as SAT instances [109, 50]. Therefore, comparing SAT and CSP is an active research area, general differences being studied, for example, in [18] and efficiencies for some commonly occurring encodings in [75].

An alternative to programming using Prolog systems is *Answer set programming* [81] (ASP), a logic programming paradigm also closely related to CSP, which uses the stable model semantics [38] as its basis. The paradigm has an established track record in planning [26], product configuration [102], formal verification [45], and even biology [30], among others. In part, the success of the paradigm is due to several highly optimized implementations [98, 69, 27]. Stable model semantics are closely related to SAT [59, 72, 12]. Recent experimental evaluations, such as [75], suggest that in some cases of practical relevance the machinery developed for SAT solvers is valuable in finding stable models of ASP programs.

The limitations placed by SAT and ASP encodings to the application domains that can be efficiently described are rather strict. While in theory any polynomial-time algorithm can be encoded as a SAT instance using the construction of Cook [83], the straightforward process is hopelessly inefficient for domains including for example integers or floating-point arithmetics. The relatively new approach of *satisfiability modulo theories* (SMT) [93, 19, 82, 37], combines the successful SAT solving and other 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 in a form where the theory-specific algorithms can be used while maintaining and even enhancing the powerful algorithms for propositional semantics. For example, an instance originating from domain considering integers together with scheduled planning is usually inefficient to express as a SAT problem. If the encoding is based on SMT, then the part of the domain considering integers can be encoded as an integer theory T while the propositional part is still efficiently solvable using algorithmic ideas that have proved useful in propositional theories.

SAT solvers can be roughly divided into two categories: incomplete solvers based on *local search* such as random walk or similar methods [95], and complete solvers usually based on *backtracking search*, such as those based on the Davis-Putnam-Logemann-Loveland (DPLL) [25, 24] algorithm. Outside of this categorization lie methods based on *knowledge compilation*, such as *binary decision diagrams* [20], and the more recently introduced *decomposable negation normal form* [23].

The focus of this work is on the DPLL solvers. Unlike the local search methods, DPLL solvers are able to establish unsatisfiability, and are less prone to exponential memory consumption occasionally observed in methods based on knowledge compilation. They are also largely observed to perform significantly better on many industrial SAT instances than the methods based on local search.

1.1 Contributions

This work analyzes and implements distributed SAT solving techniques which exploit two key elements of modern SAT solvers: randomness in time required to solve a particular instance, and the property of SAT solving algorithms that they learn clauses while solving an instance.

The work first defines an abstract distributed computing environment which is a model of a computing grid. This environment is used for studying the effect of delays and resource bounds on a framework called *simple distributed SAT solving* (SDSAT), based on distributed randomized SAT solvers. The SDSAT framework is studied in the context of several *restart strategies* [74]. 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 [83] 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 in a distributed environment. The framework is analyzed with respect to several learning strategies that can be implemented in such environments. The CL-SDSAT framework uses a limited form of communication, which is analyzed in controlled experiments and shown efficient for a large amount of distributed resources. The efficiency of CL-SDSAT is demonstrated by solving several well-known and hard SAT problems using an implementation of CL-SDSAT and a production level grid.

The results presented in Chapts. 5 and 6 show that the two relatively restricted frameworks are sufficient to yield concrete speed-up on many known hard SAT instances compared to state-of-the-art single-CPU SAT solvers. Furthermore, the experimental evaluation using instances from the SAT 2007 competition (`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 [9], the results reported here are one of the first for production-level grids.

1.2 Related Work

This work considers distributed solving as a method of reducing the solving time of a SAT instance[1]. The study is motivated by the fact that an abstract model for parallelism would allow at most linear speed-up with respect to resources; however, in a more realistic environment, delays associated with distributed solving result in lower than linear speed-up. As a result of this observation, many distributed algorithms focus on minimizing communication.

Different approaches to declarative problem solving enable various methods for obtaining speed-up via parallelization. The approaches are divided to *vertical* and *horizontal parallelism* in [10]. Vertical parallelism is based on obtaining speed-up by simultaneously studying different truth assignments of a SAT or ASP solver, and in simultaneous selection of different clauses which unify to the same goal in Prolog systems[2]. The use of vertical parallelism is based on operations on data structures recording the currently active and the yet unexplored search spaces of the algorithm. Examples of such data structures are *stacks* [88] used in Prolog systems, and *guiding paths*, which are used in ASP and SAT solvers [115, 15].

In contrast to vertical parallelism, *horizontal parallelism* refers to parallelizing *propagation rules* (e.g. [25]) when applied to SAT and ASP solving, and in principle should results in speed-up for computing the same truth assignment. There seems to be no direct counterpart for horizontal parallelism in Prolog systems. Since in many cases applying propagation rules is inexpensive, the use of horizontal parallelism is limited in high-delay distributed environments. However, horizontal parallelism can be efficiently used to some extent in certain cases for ASP [10]. The approach is even less applicable to SAT solving, unless it is obtained with special hardware [117, 76], or as a more computationally involved parallel *look-ahead* [86].

In the context of this work a more useful taxonomy of the distribution methods aiming at obtaining speed-up in solving SAT problem instances is presented in [17]. The taxonomy identifies two strategies: *distributed search* which involves the explicit partitioning of the search space, and *multi-search* with no such explicit partitioning, but instead some overlapping of parallel searches. This section overviews the two strategies and their relation to a realistic environment for distributed computing. The issue will be addressed in more detail later in the work.

1.2.1 Distributed Search

The main challenge in obtaining speed-up in distributed search is in constructing partitions having equally large search spaces. The task is particularly difficult for the type of search problems discussed in this work, which typically posses highly imbalanced search spaces [103]. For example, a partitioning of the search space of a SAT problem performed at the beginning of the solving usually results in some partitions being much easier to solve than others. Almost independent of how the partitioning is performed and what partitions are assigned to the computing resources, some of the

[1]Other applications of distributed solving, such as distributing sensitive information among independent parties [6], are out of the scope of the work.

[2]The method is referred to as *or-parallelism* in Prolog terminology [7].

resources run out of work before others. This results in a need to repartition the work of those resources which have not yet fully explored their partitions. Repartitioning requires communication, which decreases the amount of simultaneous computing and ultimately the profit obtained from distributing the solving. The phenomenon is often referred to as the *ping-pong* effect [62], and may be tackled with *work scheduling* which concerns the partitioning of the search space dynamically during the solving process. Most distributed search algorithms discussed in this work are loosely based on scheduling ideas presented in [32], where idle resources request work from those which are busy.

The use of distributed search builds usually on methods similar to those in vertical parallelism, and they use the same data structures recording the currently active search space and the yet unexplored search spaces of the algorithm. The relatively complicated semantics of Prolog, such as those related to execution order, unavoidably result in more overhead compared to SAT solving [89]. Nevertheless, at least for a relatively small number of CPUs the method of distribution based on *stack splitting* seems promising [88]. Guiding paths are an important tool for partitioning the search space of ASP problems, and there are several examples where it has proved useful [44, 33, 87, 68]. A method similar to guiding paths is *scattering* [53, 21], where search space is encoded to the problem instances instead of providing it more directly as a guiding path.

Distributed search has been used in high-delay distributed environments similar to the grid environment discussed in this work [21, 14, 53]. The two main challenges in applying the method is in the amount of communication required and an inherent, forced duplicate work in all methods based on distributed search. This duplicate work results from the approach typically followed in distributed search that two partitions are forced not to explore the same search space, therefore requiring the completion of all distributed solvers in order to guarantee completeness. This issue will be addressed in more detail in Chapt. 3.

1.2.2 Multi-Search

While distributed search has attracted much attention in developing methods for solving logic programs and performing constraint-based search, this work mainly develops approaches based on multi-search. Contrasted to distributed search, this approach has several advantages in environments involving high communication costs. Multi-search requires no communication between the solvers and therefore does not necessarily involve communication overhead typical to distributed search. However, such communication may significantly improve the speed-up that can be obtained in solving and will therefore be considered in this work. More importantly, multi-search seems to *avoid* the duplicate work that is inherent under certain circumstances in methods based on distributed search. This surprising observation will be addressed in Chapt. 3.

The multi-search method discussed in this work is based on using a randomized SAT solver together with clause learning and restarts, both discussed later in Chapt. 2. Another possibility is to employ different SAT solvers, such as in parallelizing algorithm portfolios [41]. Algorithm portfolios can also be incorporating with clause learning (e.g. [57]). While the algorithm portfolio method is more general than the one based on a single SAT solver, the method does not provide similar controlled environment for analysis that is of interest in this work. Instead of strict partitioning of the search space, it has been suggested that parts of the search space can be delegated to other parallel solvers in a less strict manner than what is done in distributed search. This method, referred to as *nagging* [34, 94], is a hybrid of multi-search and distributed search, and the results reported on the performance reflect the fact that nagging avoids the duplicate work observed in other distributed search algorithms.

1.2.3 Complementary Search Methods

A complementary approach to the distributed search and multi-search is based on partitioning the instance as opposed to partitioning the search space, and having a central solver query the constraints which are distributed. The method has been considered in the context of model checking extremely large encodings which otherwise do not fit into the memory of a single computer [36]. A somewhat similar approach is followed in [99], where solutions are computed independently for subsets of the constraints and later combined. These approaches suffer either from high communication rate or the increase of the problem complexity from NP to #P [99] and it is doubtful if they would be useful in obtaining speed-up in solving.

1.3 Outline

This work presents results based on multi-search. The main novelty is in showing that a highly efficient SAT solver can be combined with simple ideas originating from randomized restarts and parallelism to solve problems previously beyond the capabilities of the solver. This is achieved not only in simulations or using a dedicated super-computer, but also in practice using an existing, easily available and inexpensive grid environment.

Before presenting the main results, this work first describes the principles of a modern SAT solver in Chapt. 2. Using the previously introduced concepts, the work then describes differences between multi-search, scattering and guiding paths in Chapt. 3 and provides some intuition to the power of multi-search in SAT. The distributed computing environment and the corresponding simulation environment are described in Chapt. 4. The algorithmic frameworks of SDSAT and CL-SDSAT are presented in Chapts. 5 and 6, where they are also analyzed using the simulation environment and finally used for solving hard SAT instances in a grid. The work concludes in Chapt. 7, which also discusses some ideas for future work arising from the results.

Chapter 2

Propositional Satisfiability

This chapter presents in detail how modern, complete SAT solvers operate on solving instances of the propositional satisfiability problem. The chapter begins by describing the propositional satisfiability problem, continues to describe the search algorithm, gives a characterization of the clause learning used both to guide the search algorithm and limit the search space, and concludes with notes on randomization and restarts.

2.1 Propositional Satisfiability

Let $\mathcal{V}$ be a set of Boolean variables. A *propositional formula in conjunctive normal form* (CNF) over $\mathcal{V}$ is a conjunction of *clauses* which are disjunctions of positive and negative *literals* $a, \neg a$ where $a \in \mathcal{V}$. Both conjunctions of clauses and disjunctions of literals are convenient to present as sets and the work will use the set notation wherever convenient. For example, the CNF formula

$$\mathcal{F} = ((\neg e \vee b) \wedge (\neg d \vee a) \wedge (\neg a \vee c) \wedge (\neg c \vee a) \wedge (\neg a \vee \neg b \vee \neg d)) \tag{2.1}$$

can be presented as

$$\mathcal{F} = \{\{\neg e, b\}, \{\neg d, a\}, \{\neg a, c\}, \{\neg c, a\}, \{\neg a, \neg b, \neg d\}\} .$$

The *negation* of a literal a is $\neg a$, whereas the negation of a literal $\neg a$ is a, also denoted by $\neg\neg a$. Let $\mathcal{F}$ be a formula in CNF over $\mathcal{V}$ and $M \subseteq \mathcal{V}$ a *truth assignment.* The *truth value* of a variable $a \in \mathcal{V}$ is *true* in M if $a \in M$, and otherwise it is *false* in M. In short, we say that a variable is *true* or *false* if the truth assignment is clear from the context. The literal a is *true* if the variable a is *true*, while the literal $\neg a$ is *true* if the variable a is *false*. A clause C is *satisfied* by M if and only if C contains a *true* literal. A CNF formula $\mathcal{F}$ is satisfied by M if and only if all clauses in $\mathcal{F}$ are satisfied. For example, the CNF in (2.1) is satisfied by $M = \{b\}$. The problem of determining whether $\mathcal{F}$ has a satisfying truth assignment is called the *propositional satisfiability problem* (SAT). The SAT problem is NP-complete, and all known SAT solvers require in the worst case exponential amount of time with respect to the length of $\mathcal{F}$ to solve the problem [83].

A CNF formula $\mathcal{F}'$ is a *logical consequence* of $\mathcal{F}$ if each satisfying truth assignment of $\mathcal{F}$ also satisfies $\mathcal{F}'$. Two formulas $\mathcal{F}$ and $\mathcal{F}'$ are *logically equivalent*, denoted $\mathcal{F} \equiv \mathcal{F}'$, if they are logical consequences of each other.

2.2 SAT Solvers

Most current complete SAT solvers, such as zChaff [80] and MiniSAT [28] are based on the Davis-Putnam-Logemann-Loveland algorithm [25, 24] extended with *clause learning* techniques [78, 116], and their implementations follow the ideas described in this section. Such solvers are referred to as *conflict driven clause learning* (CDCL) solvers [116]. The CDCL algorithm will first be described in sufficient detail in Sect. 2.2.1 to then describe the clause learning techniques in Sect. 2.2.2. The chapter closes with remarks on restarts and randomization.

2.2.1 Backtracking Search

The CDCL algorithm described in this section searches a satisfying truth assignment for a formula $\mathcal{F}$. If there is such a truth assignment, the algorithm returns *sat*. Otherwise, the algorithm returns *unsat*. The search corresponds roughly to a depth-first search, which is performed in the set of all truth assignments. This section describes the algorithm in detail, but prior to that introduces some additional terminology.

A *partial truth assignment* is a set of literals P with the additional restriction that for no a, both $a \in P$ and $\neg a \in P$. A partial truth assignment corresponds to all truth assignments M, where a variable $a \in M$ if $a \in P$, and $a \notin M$ if $\neg a \in P$. The variables and literals $a \in \mathcal{V}$ such that $a \notin P$ and $\neg a \notin P$ are *unknown* in P.

Each variable a such that $a \in P$ or $\neg a \in P$ is associated with a unique *decision level* $dl(a)$ which is determined by how and when the literal was included to the partial truth assignment. The definition is extended to literals $\neg a \in P$ so that $dl(\neg a) = dl(a)$. In the CDCL algorithm, there are two rules for including literals to P, and they also determine the decision levels:

- The *branching rule* states that a partial truth assignment P can be extended with a *decision literal* l to $P \cup \{l\}$ if l is *unknown* in P. The decision level $dl(l)$ of such decision literal l is one more than the number of decision literals in P.

- The *unit propagation rule* (or simply *propagation rule*) states that if a SAT instance $\mathcal{F}$ contains a clause $(l_1 \vee \cdots \vee l_k)$ such that for some $1 \leq j \leq k$, l_j is *unknown* and for all l_i such that $i \neq j$ and $1 \leq i \leq k$, l_i are *false* in P, then P can be extended to a set $P \cup \{l_j\}$. The decision level $dl(l_j)$ of such l_j is the number of decision literals in P.

The process of iteratively applying the propagation rule is called *propagation*, and the literals obtained by propagation are called *implied literals*. The propagation is performed until P cannot be extended further with the propagation rule, or applying the rule results in *a conflict*, which is a clause $(l_1 \vee \cdots \vee l_k) \in \mathcal{F}$ such that all literals l_i in the clause are *false*.

The CDCL algorithm performs the search by extending an initially empty partial truth assignment with propagation and branching rule until (i) propagation results in a conflict, or (ii) no *unknown* variables remain for branching. In (i), the CDCL algorithm *backtracks* to some previous decision level d by removing all literals associated with decision level greater than d, and continues the search with the resulting partial

```
Input: $\mathcal{F}$, a SAT formula; $s$, a random seed
1  Initialize prng with seed $s$
2  let $\mathcal{C} = \emptyset$
3  while (true)
4     while (propagate($\mathcal{F} \cup \mathcal{C}$) $\neq$ false)
5        if (branch(prng()) = false)
6           return sat
7     if $P$ contains no decision literals
8        return unsat
9     if restartPoint()
10       backtrack to decision level 0
11    let $C$ = analyze()
12    backtrack($C$)
13    let $\mathcal{C} = \mathcal{C} \cup \{C\}$
```

Figure 2.1: The CDCL algorithm

truth assignment. If (*i*) happens when P contains no decision literals, the algorithm has found the instance unsatisfiable and terminates. In (*ii*), the algorithm has found a partial truth assignment corresponding to a satisfying truth assignment completely determined by the partial truth assignment and also terminates.

When the CDCL algorithm results in a conflict in propagation, it creates a *learned clause*. The learned clause is a logical consequence of $\mathcal{F}$ having a unique literal associated with the highest decision level and all of its literals *false* in the partial truth assignment where the conflict occurred. The CDCL algorithm uses the learned clause to backtrack to the second highest decision level associated with any literal in the learned clause. After the backtracking, the literal previously associated with the highest decision level becomes *unknown*, and therefore becomes implied on the new decision level after propagation. Learned clauses are collected to a set $\mathcal{C}$ which is considered as a part of the SAT instance in propagation.

The CDCL algorithm is described in Fig. 2.1. For simplicity, the inclusion and removal of literals from the partial truth assignment P are not handled explicitly in the pseudo-code, but are instead performed implicitly at lines 4, 5, 10 and 12, as explained later.

Lines 1 and 2 consist of an initialization of the CDCL algorithm. The initialization addresses the set of learned clauses and randomization, which are two aspects of the algorithm that we will extensively use in the development in this work. At line 1, the algorithm seeds the pseudo-random number generator prng using the seed s provided as input. The pseudo-random number generator is used at line 5, which corresponds to application of the branching rule. Line 2 initializes the set of learned clauses $\mathcal{C}$ to the empty set.

The search is performed in the loop at lines 3 to 13. The propagation, performed at line 4 by the function propagate(), returns true if no conflict arises. In this case, the algorithm proceeds to line 5, where the pseudo-random number generator is used in the function branch(n) to obtain a decision literal. The function branch(n) returns false if all variables have a value. From this and the fact that propagation resulted in no conflicts it follows that the problem is satisfiable and P corresponds to a satisfying truth assignment for the SAT instance.

In case propagation results in a conflict, the propagation is interrupted and analyzed at lines 7 to 11. If the conflict occurs with no decision literals, the instance has been shown unsatisfiable. Otherwise, the algorithm may either perform a restart or continue the analysis from line 11. In the latter case, analysis results in a new learned clause C. The learned clause is used to direct backtracking (line 12) and then included to the set of learned clauses at line 13, so that the unique literal previously associated with the highest decision level in C is implied at line 4 unless this results in another conflict. In the former case, where the algorithm performs a restart, all literals l with decision level $dl(l) > 0$ are removed from the partial truth assignment P and the execution of the algorithm continues from line 4.

2.2.2 Constructing a Learned Clause

This section describes the construction of learned clauses which a CDCL SAT solver uses to guide its search and prune the search space while solving an instance. The discussion is slightly more declarative but essentially the same as the standard discussion, e.g., in [78], which uses *conflict graphs.*

Propagation described in the previous section results in implied literals being included to the partial truth assignment P, or in a conflict, which is a clause $(l_1 \vee \cdots \vee l_k)$ that is either part of the original SAT instance $\mathcal{F}$ or one of the previously learned clauses from the set $\mathcal{C}$ such that all its literals are *false* in P. Given a clause where no literal is *unknown*, let $\mathrm{maxdl}(C)$ be the set of literals with highest decision level in C. All literals $l \in \mathrm{maxdl}(C)$ of a conflict C except the decision literal are *conflict literals.* All conflicts contain a conflict literal since a decision literal is selected after propagation and therefore is never the only *unknown* literal in a clause.

The learned clauses are constructed using information related to how literals were included to P. To construct the learned clause, each implied literal l is associated with a unique *reason* $r(l) \in \mathcal{F} \cup \mathcal{C}$ such that

- $r(l) = (l_1 \vee \cdots \vee l_k)$ such that for all i, $1 \leq i \leq k$ and $l_i \neq l$, it holds that $\neg l_i \in P$, and
- $l \in \mathrm{maxdl}((l_1 \vee \cdots \vee l_k))$.

Intuitively, the set of reasons represents the order in which the implied literals have been obtained by propagation rule while running the CDCL algorithm. To capture this, we need an additional restriction on the set of reasons: the set of all reasons $\{r(l) \mid l \text{ is implied}\}$ must have a partial ordering $\prec$ such that for all literals $l_i \in r(l)$ such that neither l_i nor $\neg l_i$ is a decision literal and $l_i \neq l$, the ordering satisfies $r(\neg l_i) \prec r(l)$.

We will now describe how the learned clause is constructed algorithmically using the reason clauses by describing the analyze() algorithm in Fig. 2.2.

Line 1 (non-deterministically) chooses a conflict literal l which has a reason $r(l)$. If also $\neg l$ has a reason, then the reasons are *resolved* at line 3 by the function

```
1 **let** $l$ be a conflict literal
2 **if** $\neg l$ is not a decision literal
3     **let** $C$ = resolve($r(l)$, $r(\neg l)$, $l$)
4 **else**
5     **let** $C = r(l)$
6 **while** $|\text{maxdl}(C)| > 1$
7     **let** $C = \text{resolve}(C, r(l), l)$ for some implied $l$ such that $\neg l \in C$
8 **return** $C$
```

Figure 2.2: The (non-deterministic) analyze() function for clause learning

$\text{resolve}(C_1, C_2, l) = (C_1 \cup C_2) \setminus \{l, \neg l\}$, which takes two clauses C_1, C_2 where one contains literal l and other $\neg l$, and results in a new clause. The resulting clause is a logical consequence of $\mathcal{F}$, since both C_1 and C_2 are logical consequences of $\mathcal{F}$ (either by induction hypothesis or since they are part of $\mathcal{F}$), and the function resolve, when provided with two logical consequences produces a clause which is a logical consequence of the two clauses. If $\neg l$ is a decision literal, it has no reason and the algorithm sets C to the reason of l at line 5. Line 6 uses the function $\text{maxdl}(C)$ to determine when C satisfies the criterion for learned clauses, that is, C has a single literal at the highest decision level. At line 7, the algorithm resolves clause C with the reason $r(l)$ using one of its literals $\neg l \in C$. There always is at least one such literal, since initially all literals of C are false, and after the resolution step, only false literals are added from $r(l)$. Furthermore, the argument C at line 7 never consists of only decision literals since $|\text{maxdl}(C)| > 1$ by line 6 and each decision level greater than 0 is associated with exactly one decision literal. The loop at lines 6 and 7 is guaranteed to terminate because of the partial ordering of the reason clauses. Finally, C is returned as the learned clause at line 8.

The following example illustrates the construction of a learned clause in the middle of an execution of the CDCL algorithm.

Example 1 *The CDCL algorithm is provided with the CNF presented in Eq. (2.1) and the partial truth assignment P of the algorithm is $\{e, b\}$ where e is a decision literal and $dl(e) = dl(b) = 1$. Assume that the function branch() results in $P = \{e, b, d\}$, where $dl(d) = 2$. The propagation (at line 4) has now, for example, the two following outcomes: either (i) $P = \{e, b, d, a, c\}$ or (ii) $P = \{e, b, d, a\}$. Both outcomes are terminated by the conflict $(\neg a \vee \neg b \vee \neg d)$.*

The conflict clause $(\neg a \vee \neg b \vee \neg d)$ contains two conflict literals, $\neg a$ and $\neg d$ both with the conflict as the reason. For implied literal a, the reason $r(a)$ cannot be $(\neg c \vee a)$ since this cannot be ordered by $\prec$ with the unique reason $r(c)$. Therefore, $r(a) = (\neg d \vee a)$ and the ordering, when conflict literal is $\neg d$, becomes essentially $r(a) \prec r(c)$, $r(a) \prec r(\neg d)$ and $r(b) \prec r(\neg d)$.

The analyze() function continues to line 5, to obtain $C = (\neg a \vee \neg b \vee \neg d)$, and produces by resolving with $r(a)$ the clause $C = (\neg b \vee \neg d)$ having $\text{maxdl}(C) = \{\neg d\}$, and therefore C is the new learned clause.

The CDCL algorithm then backtracks to decision level 1 resulting after propagation in $P = \{b, \neg e, \neg d\}$, with $dl(b) = dl(\neg e) = dl(\neg d) = 1$. Branch on $\neg a$ results after *propagation in $P = \{b, \neg e, \neg d, \neg a, \neg c\}$ corresponding to a satisfying truth assignment $\{b\}$ with which the CDCL algorithm terminates.*

In addition to directing the backtracking of the algorithm, learned clauses also contribute to the efficiency of propagation. On one hand, the learned clauses result in at least as much propagation with a given partial truth assignment as the original instance. This helps to reduce the size of the search space covered by the algorithm. On the other hand, propagation consumes time proportional to the number of learned clauses. The minimum set of learned clauses the CDCL algorithm requires consists of the clauses which are reasons for currently implied literals. Modern CDCL solvers tend to keep some other clauses in memory as well by employing various heuristics. Later in Chapt. 6 we will study certain heuristics for collecting such clauses in other settings.

2.2.3 DPLL SAT Solvers

The conflict analysis described here corresponds to what most modern SAT solvers, such as MiniSAT, employ in guiding their search. In contrast, earlier implementations [25, 71] record their search state more directly using *guiding paths* [115]. To make the distinction clear, we refer to these solvers as DPLL solvers as opposed to the CDCL solvers described above. We will overview such solvers here since their approach for storing the search state is used when describing some modern parallel solving techniques in Chapt. 3. Guiding paths store the decision literals of a DPLL search tree as a sequence $GP = (\langle l_1, \delta_1 \rangle, \ldots, \langle l_i, \delta_i \rangle)$ of *branches* $\langle l_j, \delta_j \rangle$, where for all $1 \leq j \leq i$, l_j is the j:th decision literal and either $b(l_j) = 0$ or $b(l_j) = 1$. The value of $b(l_j)$ is 1 if the SAT solver has proved that there are no satisfying truth assignments corresponding to a partial truth assignment $\{l_1, \ldots, l_{j-1}, \neg l_j\}$. A branch $\langle l_j, 1 \rangle$ is called a *closed* branch. A branch $\langle l_j, 0 \rangle$, called an *open* branch, signifies that the solver has not proved that there are no satisfying truth assignments corresponding to a partial truth assignment $\{l_1, \ldots, l_{j-1}, \neg l_j\}$. Hence, the backtracking needs to be performed only to decision levels corresponding to open branches.

Upon reaching a conflict, a DPLL solver always backtracks to the highest decision level which is open, say, decision level j. The backtracking, in addition to the standard removal of literals from the partial truth assignment, also removes the branches corresponding to the decision levels $j+1, \ldots, i$ and changes the branch $\langle l_j, 0 \rangle$ to $\langle \neg l_j, 1 \rangle$. Therefore the search of a DPLL solver can be viewed as traversing a tree where nodes are labeled with decision variables and the two children of a node correspond to truth assignments where the decision variable is *true* and *false*.

The conflict analysis of a CDCL algorithm can be restricted to produce a similar search. To achieve this, the learned clause must always contain the negation of the most recent decision literal and at least one *false* literal from the previous decision level. When such clause is included to the set of learned clauses, the propagation will result in a truth assignment where literals from higher decision levels are removed and the previous literal is included negated. Intuitively, such search will be essentially the same as the search of the corresponding DPLL solver[1].

[1]The decision levels will be completely different, however.

2.2.4 Restarts and Randomization

In addition to clause learning, most modern SAT solvers also apply search restarts and some form of randomization to avoid getting stuck at hard subproblems [42].

The function branch() in Fig. 2.1 relies on actual implementation to some of the numerous heuristics described in the literature when selecting the decision literal (e.g., [66, 46, 58, 97, 48, 71, 61, 47]). 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* [41] 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. While the introduction of randomness might seem counter-intuitive as the predictability of run time is usually a desired property of an algorithm, it turns out that the randomness can in fact be used to decrease the expected time required to solve a SAT instance.

Restarts are motivated by the empirical observation that some SAT instances, when solved by a CDCL algorithm, obey a *heavy-tailed run time distribution* [42]. Such distributions have a large probability of producing "outlier" samples, that is, run times which are far from median run times. The distributions have an infinite standard deviation or even an infinite mean[2]. For instances obeying the heavy-tailed distribution it is useful to interrupt the search procedure after some time and start the search again from the beginning. It can be shown that restarts together with randomization eliminate heavy-tailed distributions [42].

Technically, restarts are events where the CDCL algorithm decides (in Fig. 2.1 at line 9) to backtrack to decision level 0 and continue the search as if it were started from the very beginning. Often a restart also involves removing some of the clauses learned by the solver during the search. Randomization and restarts will be covered in more detail in Chapts. 3 and 5, but it is useful to state here that restarts together with clause learning do increase the strength of the CDCL algorithm when compared to a traditional DPLL algorithm as described in [24]: it can be shown that clause learning together with restarts can be as powerful as general resolution, resulting in some cases exponentially smaller proofs (e.g., the total number of decision literals covered during the search) than those achievable with the traditional DPLL SAT solvers [11, 49].

[2]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 [43].

Chapter 3

Parallelization Methods in SAT Solving

This chapter describes how the guiding path method [115, 15] described in the previous chapter is used in distributed search, contrasts it to scattering [53], points out the pitfalls of methods based on distributed search and gives an intuition why multi-search performs sometimes better in practice than methods based on explicit partitioning [53]. Finally, the chapter discusses and reviews work studying restart strategies, a framework closely related to multi-search.

3.1 Distributed Search with Guiding Paths

A guiding path records the search state of a DPLL SAT solver, as described in Chapt. 2. Guiding paths can be used in connection with restarts to avoid performing the same search multiple times, as in [114], but are more commonly used to express partitions of search spaces for parallelizing search [115, 15].

Guiding paths direct the parallel search in a fashion similar to the sequential case. Each parallel solver maintains its own guiding path which, in addition to storing information about the search state, is used to delegate parts of the search space to other solvers. If a branch j is closed for some $1 \leq j < i$, then either some other SAT solver is exploring truth assignments corresponding to a partial truth assignment $\{l_1, \ldots, l_{j-1}, \neg l_j\}$, or, as in the sequential case, such truth assignments have been proved unsatisfiable. Therefore, after proving a branch unsatisfiable the SAT solver needs only to explore the open branches as discussed in Chapt. 2.

A SAT solver whose search state is represented by a guiding path GP may delegate parts of its search space to a new solver by constructing a delegate guiding path GP_d and continuing the search on an altered guiding path GP_a representing the remaining search space of GP not contained in GP_d. This is done by (1) selecting a set of open branches to be delegated from GP, (2) copying the contents of GP to GP_a, (3) copying the contents of GP to GP_d up to and including the last open branch to be delegated, (4) closing the previously selected open branches from GP_a, (5) closing the branches in GP_d remaining open in GP_a, and (6) replacing the last (open) branch $\langle l_j, 0\rangle$ of GP_d by $\langle \neg l_j, 1\rangle$.

Example 2 *For sake of an example, assume that a DPLL SAT solver with guiding path*

$$GP = (\langle l_1, 0\rangle, \langle l_2, 1\rangle, \langle l_3, 0\rangle, \langle l_4, 1\rangle, \langle l_5, 0\rangle, \langle l_6, 1\rangle)$$

wishes to delegate the parts corresponding to open branches l_3 and l_5 by constructing a guiding path GP_d for the delegated part and maintain a guiding path GP_a consisting of the remaining search space to itself. The solver closes the two open branches resulting in

$$GP_a = (\langle l_1, 0\rangle, \langle l_2, 1\rangle, \langle l_3, 1\rangle, \langle l_4, 1\rangle, \langle l_5, 1\rangle, \langle l_6, 1\rangle),$$

and delegates the guiding path

$$GP_d = (\langle l_1, 1\rangle, \langle l_2, 1\rangle, \langle l_3, 0\rangle, \langle l_4, 1\rangle, \langle \neg l_5, 1\rangle).$$

Guiding path creation is dynamic in the sense that if a SAT solver has closed all branches of its guiding path, the search space is fully explored and the solver may request a new guiding path from some other solver. The parallel search based on guiding paths terminates in two cases: if a solver finds a satisfying truth assignment, which then shows that the instance is satisfiable, or if no guiding paths with branches remaining to be explored exist, which shows that the problem is unsatisfiable.

3.1.1 Scheduling in Guiding Paths

While the construction of a single guiding path is not expensive, the communication associated with delegating guiding paths might consume a significant portion of SAT solving time. Therefore it is important to minimize the number of dividing operations. Scheduling in the context of guiding paths addresses the problem of determining which parts of a guiding path should be delegated in order to result in a balanced partition, ultimately leading to small number of dividing operations.

The types of scheduling in guiding paths can be classified to *top scheduling* and *bottom scheduling* [10]. In top scheduling, only the search space corresponding to the first open branch is delegated to a new solver. In this case, it suffices to delegate the beginning of the guiding path up to and including the first previously open branch to the new solver. Bottom scheduling, on the other hand, corresponds to delegating the last open branch, and possibly other branches as well. The two scheduling types are compared in [10] for ASP, where bottom scheduling performs better in most cases. Similar results are obtained for Prolog [88]. It remains unclear whether bottom scheduling is effective in SAT, although some negative results have been claimed [14]. It seems that there is no comprehensive study for this question in SAT.

Distributed search is almost exclusively implemented with guiding paths and some form of scheduling. Results in [100], albeit based on only few experiments, suggest that also static partitioning of the search space may result in speed-up.

3.1.2 Learning and Guiding Paths

Most parallel SAT solvers which share learned clauses among parallel processes use distributed search based on guiding paths [101, 14, 70, 92, 31]. The two similar approaches based on distributed environments in [101, 14] and tightly coupled environments, such as multi-core CPUs, in [70, 92, 31] are both able to produce speed-ups in some problems, although [31] reports mostly negative results. However, [70] provides empirical evidence that a careful design of data structures overcomes the problems mainly related to memory bandwidth and thread locking identified in [31]. Since the amount of learned clauses is usually overwhelming even in sequential SAT solving, parallel solvers need to employ different forms of filtering for the shared clauses so that the memory exhaustion and excess overhead related to propagation is avoided. A simple yet efficient filtering criterion is the length of clauses [101, 14, 70]. More complex filtering mechanisms involve the amount of backtracking resulting from the inclusion of the learned clause [92].

In many implementations, the learned clauses are shared dynamically, while the solvers are running. Therefore, a shared learned clause might be in any of the following modes for the receiving SAT solver, identified by the partial truth assignment of that solver:

1. satisfied, in which case the clause does not affect the partial truth assignment in any way;

2. conflict, in which case backtracking must be performed until at least one of the literals in the clause is no longer *false*;

3. implying, i.e., one of the literals is unknown and all others are false. In this case the remaining literal needs to be propagated, together with possibly backtracking to the highest decision level of the false literals in the clause;

4. none of the above, when at least two of the literals in the clause are unknown.

While cases 1 and 4 have no direct effect on the search state, 2 and 3 might result in behavior very similar to restarting. For example, clauses consisting of a single literal not appearing in the partial truth assignment result in backtracking to the first decision level and might significantly affect the run of the algorithm [92].

Implementing the above functionality involves the tuning of the inner loop of the DPLL-based SAT solver, which in addition to the delays associated with communication might slow down the execution of the solver more than what is gained from the learned clauses. Therefore, some parallel solvers based on distributed search avoid involved changes in the solver. For example, a parallel SAT solver described in [21] splits the search space by fixing the first decision literal and producing another instance where the literal is fixed with an opposite value. Hence, the scheduling corresponds to top scheduling guiding path. The learned clauses are shared so that only those which are not satisfied on the new instance on decision level $d = 0$ are transferred to the solver solving the new instance. Since delegating guiding paths is relatively infrequent, the overall price resulting from special handling of the learned clauses is low. Unlike in the more involved parallel solvers which modify the inner loop of the SAT solver, this is the only cost associated with clause sharing since no learned clauses are shared between solvers during the execution of the inner loop of the SAT solvers.

3.2 Scattering

Guiding paths are limited in the sense that they can only express a single path in the DPLL search tree and the literals which are still available for branching on the path. A more general approach is presented in [53, 52]. The approach, called *scattering*, is based on expressing the division of the search space of a SAT instance F to n SAT instances $F_1, \ldots, F_n$ such that

$$F_i = \begin{cases} F \wedge T_1 & \text{if } i = 1 \\ F \wedge \neg T_1 \wedge \cdots \wedge \neg T_{i-1} \wedge T_i & \text{if } 1 < i < n \\ F \wedge \neg T_1 \wedge \cdots \wedge \neg T_{n-1} & \text{if } i = n. \end{cases} \tag{3.1}$$

Each T_i is a heuristically selected conjunction $l_1^i \wedge \cdots \wedge l_{d_i}^i$ of literals, and $\neg T_i$ is the clause $(\neg l_1^i \vee \cdots \vee \neg l_{d_i}^i)$. The amount of literals d_i in T_i is determined so that the search space corresponding to F_i is approximately equal for all $1 \leq i \leq n$. Solving of the SAT instance F then reduces to either showing that one of the SAT instances F_i has a satisfying truth assignment or that none of the SAT instances is satisfiable.

Scattering can be seen as a generalization of guiding paths where each parallel SAT solver is given a set of guiding paths instead of a single guiding path. The number of guiding paths corresponding to a SAT instance F_i is in the worst case exponential in i. For example, to express the search space of the instance F_n as a set of guiding paths, there will in the worst case be a guiding path for each satisfying truth assignment for $\neg T_i \wedge \cdots \wedge \neg T_{n-1}$. Note that scattering can simulate guiding paths: a guiding path

$$GP = (\langle l_1, \delta_1 \rangle, \ldots, \langle l_n, \delta_n \rangle)$$

corresponds to the scattered instance

$$F = (l_1 \wedge \cdots \wedge l_n) \vee \bigvee_{1 \leq i \leq n \text{ s.t. } \delta_i = 0} (l_1 \wedge \cdots \wedge l_{i-1} \wedge \neg l_i).$$

The formula F can be presented in conjunctive normal form with size polynomial in n by introducing new variables [107].

In [53], the resulting SAT instances $F_1, \ldots, F_n$ are recursively scattered to form a *scattering tree*, which is then solved by either showing one of the instances corresponding to the nodes in the tree satisfiable, or showing a set of instances corresponding to a cut in the tree unsatisfiable. Recursive scattering can be seen as a form of scheduling, since only problems not already shown unsatisfiable are scattered. Unfortunately, the results reported in [53] are obtained using only a very limited form of clause learning. It is currently an open question whether scattering outperforms methods based on guiding paths in practice.

3.3 Multi-Search in SAT

Despite the intuitive appeal of distributed search, all methods based on partitioning the search space are prone to *partitioning on don't cares*. To give a concrete example on the subject, let the *Simple Distributed SAT Solving* (SDSAT) be the simple method of running the same randomized SAT solver on the same instance in parallel with no communication between the solvers until one of the solvers solves the instance. Suppose a SAT instance contains an unsatisfiable *core*, which involves only a subset of the variables but showing the unsatisfiability of the core suffices to show the full instance unsatisfiable. If the search space is partitioned, for example using guiding paths, with a variable not in the core, the distributed search has to solve the same core twice to show the instance unsatisfiable. A sequential solver clearly needs to solve the problem only once. For example, let $q_F(t)$ be the probability that the SAT instance F is solved within time t by a single solver, and let $F_x, F_{\neg x}$ be the SAT instances $F \wedge (x)$ and $F \wedge (\neg x)$ for some literal pair $x, \neg x$ not in the core of F. Then the probability that F is solved within time t using SDSAT with two CPUs is $1 - (1 - q_F(t))^2$ whereas the probability that problems F_x and $F_{\neg x}$ are both solved is $q_F(t)^2$, assuming that $q_{F_x} = q_{F_{\neg x}} = q_F$. If F is solved in one hour with probability 0.8, then the corresponding probabilities are 0.96 for SDSAT and 0.64 for the search space partitioning method.

There is some empirical evidence that the SDSAT method is competitive compared to distributed search (for example, [53] and the results reported in the SAT Race organized as part of the SAT 2008 conference). One reason for this counter-intuitive result could be the tendency of distributed search to solve a core several times as exemplified above.

As it turns out, for many SAT problems the search space seems to be so huge that the explicit partitioning of the search space is not required because the overlap in the searches of independent randomized SAT solvers is small.

Typically an implementation of a randomized constraint solver, such as a (randomized) SAT solver, has a highly erratic run time behavior [74, 42, 108]. The SDSAT framework is a natural way of harnessing the variance in run times to obtain speed-up in a parallel environment. Continuing the previous example, assume that the probability that the instance F is solved in exactly one hour is α and otherwise the instance is solved in exactly 10,000 hours. The expected run time for the solver is then $\alpha \times 1 + (1 - \alpha) \times 10000$ hours. Using again SDSAT with two CPUs, the expected run time becomes $(1 - (1 - \alpha)^2) \times 1 + (1 - \alpha)^2 \times 10000$ hours. For $\alpha = 0.8$ this results in expected running time of 2000.8 hours for single solver and 400.96 hours for two solvers. Perhaps surprisingly, the speed-up is super-linear with respect to the number of resources. Figure 3.1 shows the speed-up for the discrete distribution for different values of α.

The speed-up phenomenon seems to be related to particular type of distributions, called *heavy-tailed distributions*, typical to search problems such as SAT [42, 108] and especially satisfiable instances [40]. The variability in run times of a randomized backtracking search is significant enough to be taken into account in designing of SAT solvers [43], leading to the concept of *restart strategies* [74], *interleaving strategies*, and other algorithm portfolios [41]. A restart strategy defines a sequence of time periods $(t_1, t_2, \ldots)$, and is employed by running an algorithm for time period $t_i, 1 \leq i$, and — if the instance is not solved — restarting the algorithm and running it for the time period t_{i+1}. It can be shown that a simple restart strategy where t_i is some constant for all i results in lowest expected run time over all restart strategies [74]. However, determining the optimum restart strategy requires obtaining the run time distribution

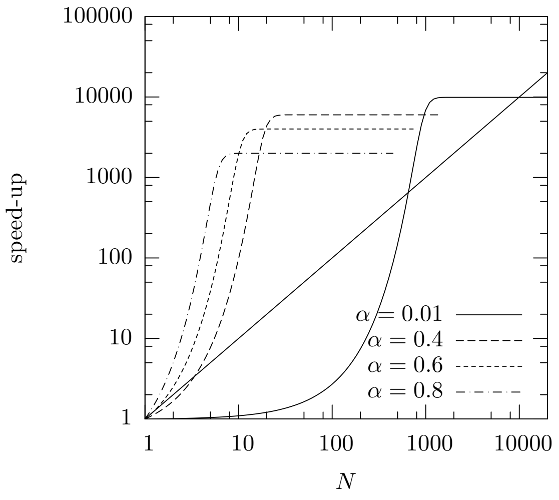


Figure 3.1: Speed-up on two-valued discrete distribution, where probability of finding a solution in exactly one hour is α and in exactly 10,000 hours is $1 - \alpha$, as a function of number of resources N

of the instance, which is seldom available in practical settings. A similar argument shows that there is a universal restart strategy which guarantees an expected run time within logarithmic factor from the optimal restart strategy [74]. However, this factor is in practice high, and comparisons between other strategies are undetermined, as can be seen by contrasting [42, 49] and the results reported in this work. Parallel restart strategies are studied, for example, in [73, 41] and this work extends the results by studying a realistic parallel environment with various instances and restart strategies. Overall, restart strategies together with clause learning significantly increase the power of DPLL-based SAT solvers [49, 11].

The problem of determining restart strategies becomes more tractable if some information is available from the run time distributions of instances. Using a large number of learning data, run time information can be used to determine good restart strategies for similar instances [64, 110] or statistically dependent sequence of instances [91]. Most of the methods require an initial, expensive training phase, although restart strategies can be learned on-the-fly with reasonable performance early in the process [35]. Furthermore, there is some evidence that the run-time distribution of an instance can be classified as heavy-tailed based on the shape of the search tree of a backtracking search [40]. The overall impact of restart strategies is analyzed, for example, in [111], where various restart strategies are contrasted both analytically and experimentally. Similar methods have been used in the algorithm portfolio setting in [112], where characteristics of an instance are analyzed in an initial phase and, based on these results, a set of algorithms expected to perform well are run on the instance. The performance of the method in this setting seems promising.

The problem of computing an approximation of the optimal portfolio is NP-hard, meaning that the worst-case complexity of all known methods of determining an optimal portfolio is at least exponential in the number of available algorithms for the portfolios. Fortunately, this number is usually relatively low, and there are algorithms which perform well in practice [105, 84]. The problem of determining optimal restart strategy seems to be hard as well [106].

3.4 Remarks

This chapter studies different methods for obtaining speed-up in solving SAT problems. The analysis provides insight to distributed search by comparing different methods for implementing search space partitioning. The chapter also gives a condition when multi-search is preferable to distributed search in practice. This motivates the study of algorithmic frameworks for multi-search that forms large part of the rest of this work.

While there is a substantial amount of research dedicated to the design of restart strategies, the results obtained later in this work do not completely justify these efforts in distributed environments. It seems that simultaneous solving of instances is sufficiently powerful to provide the run-time reducing effect obtained by restart strategies, and therefore the methodology for learning a good restart strategy for a particular instance is irrelevant. The effect of restart strategies to the run time in a distributed environment is further diminished for many SAT instances by the high latencies in the environment.

Chapter 4

Computational Grids

This chapter describes the grid computing environment that will be employed in the upcoming chapters. Most of the methodology presented in the following chapters are designed based on the assumption that the underlying environment provides the capabilities of a grid as described here, and nothing more. More specifically, the underlying environment provides

- a *master-worker architecture*, where all communication is performed between a unique master and a set of workers and no communication takes place between the workers, and
- *high-latency communication*, where all communication between the master and the workers takes several orders of magnitude more time than the single steps of the algorithm.

After motivating the environment, this chapter describes a simulation environment that will be used for studying the efficiency of the methodology described in later chapters in order to avoid the unpredictability of actual grid environments arising from, e.g., network glitches and hardware failures.

4.1 The Grid Environment

The parallel solving environment used in this work is that provided by computational grids. The environments considered as grids in this work consist of large number of geographically distributed computers which are available through a uniform interface. Such environments are readily available, and examples include UNICORE (`http://www.unicore.eu/`) and NorduGrid (`http://www.nordugrid.org/`), the latter being used in some of the experiments of this work.

Grids can be contrasted to the more commonly used computing clusters consisting of a large number geographically localized computers also available through a uniform interface. The geographical proximity allows building systems with somewhat lower-latency communication, making cluster computing more competitive in terms of computational efficiency. However, a single cluster owned by a single organization may suffer from high-utilization peaks during which the capacity of the cluster is not sufficient for the use of the organization. Avoiding such peaks is achieved, in general, by larger investments to computing hardware. This is both expensive and environmentally unfriendly because large numbers of resources are under-used outside of the peak utilization. An improvement to this is to share a number of clusters among collaborating organizations. Grid computing addresses this issue by providing a uniform interface for the clusters and usually at least an option for a form of load balancing between the clusters.

Grid environments usually compose of several clusters. Therefore they are characterized by a large number of relatively efficient CPUs, very much like the more traditional cluster computing environments. Since a grid can be formed by several independent but collaborating organizations which decide to share the computing resources, it is common that two jobs submitted to the grid are not guaranteed to be able to communicate with each other at all. For example, such limitations are typically posed by the networks of the organizations in NorduGrid used in the experiments. Therefore, in the algorithms developed in this work we assume that jobs cannot communicate directly with each other. This assumption leads naturally to a *master-worker* architecture, where all communication takes place between the workers and the master, whereas the workers do not communicate directly with each other.

This work builds largely on the observation that the computations submitted to a grid can be described as *jobs* consisting of programs and their inputs, needing no user input during their execution. A job places requirements on the CPU resources, amount of memory, and disk space on the computers where it can run. When the requirements are not excessively restrictive, the number of available CPUs for the job is higher and therefore the job can usually be executed sooner in the grid.

The jobs submitted to a grid environment are executed on a collection of computing resources called *computing elements* (CEs). A CE corresponds to one or more CPUs depending on the requirements of the job. Each CE executes a single job at a time and usually corresponds to a CPU or a set of CPUs in one of the clusters in the grid.

The entry point of a submission of a job to CE is a queue which accepts jobs with requirements matching those provided by the queue. The submission of a job consists of selecting the target queue and transferring the job to the selected queue. The time required to submit the job to the queue is the *submission delay* of a job. In the model described here the submission cannot be parallelized, meaning that the submission delay places an upper limit on the number of jobs that can be submitted in a unit of time.

Each queue is associated with a set of CEs corresponding to a set of CPUs, usually within one cluster. A job starts executing when the queue assigns the job to a CE. The time between finishing the submission and assignment to a CE is the *queuing delay* of a job. This delay does not affect the rate at which jobs can be submitted, but instead contributes to the *efficiency of computing*, that is, the number of operations the program can effectively perform on the input per unit of time. Usually there is also some delay associated with the actual finishing of the execution and the detection that a job is finished. This can be seen as contributing to the queuing delay since there is little that can be done from the point of view of the master process during this time.

The two delays described above result from several causes. Firstly, if a job involves transmitting a large amount of data, the amount of network bandwidth may greatly affect the submission delay [85]. The submission delay is also affected by the possibility of jobs disappearing due to maintenance breaks or various random faults in the CEs. Secondly, the queuing delay depends on the amount and types of previously submitted jobs still in the queue, and the remaining run times of the jobs currently executing in the CEs. Thirdly, the efficiency of the computing is also affected by the fact that the run time of a job in a CE depends on the load potentially placed by other jobs on the other CPUs that share the same memory or disk resources, as well as the types of the CPUs in the CE.

The two delays described above together with the CPU selection pose a major challenge on maximizing the speed-up obtained by using a grid. This is performed usually by job management, sometimes referred to as scheduling or brokering. Efficient job management in grids is a non-trivial task and is typically handled by special tools (for example [29, 60, 51]), often designed for particular applications since a generic method seems to be difficult to achieve [65]. In those experiments of this paper that are run in NorduGrid, we use a fault-tolerant and efficient job management system called the Grid Job Manager (GridJM) [51]. The system incorporates ideas from monitoring the historical behavior of the grid environment as well as information provided by the clusters composing the grid, an approach shown useful in [113].

The requirement placed on CPU time by a job affects in some cases the queuing delay experienced by the job. For instance, most clusters support a mechanism called *reservation*, where a complicated task requesting a CE of several CPUs will force a queue to start reserving CPUs. This means that no new jobs requesting a CE will be assigned from the queue, unless the run time of the job is short enough to finish before the time expected for the requested CE of several CPUs to become available. On the other hand, the restrictions of the run time of a job together with the submission delay dictate the number of CEs that can be simultaneously obtained. Therefore it would be preferable to submit sufficiently long running jobs so that the submission delays do not dominate the total run time. Based on our previous experience, in the experiments in NorduGrid that are presented in this work we use jobs where the run time is limited to one hour.

4.2 The Simulation Environment

In order to compare the performances of SAT solving algorithms developed in this work, there is a need for a benchmarking system. Realistic grid systems pose certain challenges for rigorous algorithm benchmarking, since both the delays and the efficiency of computing vary, rendering the reproduction of results difficult. To overcome these challenges, this work employs a simple grid model based on the following components:

(1) A unique central process M initiating new jobs and monitoring old jobs, and a set of N CEs receiving jobs from and reporting the results to M.

(2) A submission delay describing the amount of time required to submit a job to the grid. The delay $d(N)$ can be modeled as a random variable depending on

the number of CEs employed. The delay is simulated by M and results in a bottleneck when initiating new computations.

(3) A queuing delay which is the sum of two components: the time spent queuing to a CE, and the time spent receiving the results after the job has finished. The delay $d_q(N)$ can be modeled as a random variable depending on the number of CEs employed. The delay is experienced by the job and does not form a bottleneck for submission.

(4) A maximum resource limit T_c describing the amount of time a job is allowed to execute in a CE before the job is terminated and the CE becomes ready to accept a new job.

We believe that this environment provides a realistic model for distributed computing in grids, as justified here: (1) A central process managing jobs provides a natural synchronization mechanism. (2&3) Most such systems have a delay associated with the synchronization, and specifically shared distributed environments require certain communication in selecting the CE to be employed. (4) Batch systems such as grids usually limit the resources available to a single job, for example, to provide fairness in scheduling.

The model does not directly consider the effect of various CPU models and the load on the CPUs on the run time. Such effects can be obtained by adjusting the queue delay and the resource limit accordingly. The simulation environment provides large enough flexibility to study the effect of different delays. This is needed to achieve reliable results on the different methodologies that do not depend excessively on the job management aspects left outside of the scope of this work.

We may study an application submitting jobs to the grid through a central process M as a time line, illustrated in Fig. 4.1. Time advances to the right in the figure and the abstract CEs can be seen as N bands placed on top of each other. The filled rectangles represent jobs, and the dark areas inside the jobs represents the CPU time, while the rest of the rectangle consists of the queuing delay. The time in the figure starts when the first job (the long rectangle at the bottom of the figure) has been submitted into a queue of a CE. The second job is submitted immediately after this, and it reaches the queue immediately after the submission delay $d(N)$. Meanwhile, the first job has reached the CE. After ten finished submissions, the first job finishes execution, and the finishing is finally observed by the master after the finishing delay.

When performing the actual simulations, we make the following simplifying assumptions on the model:

- submit delay $d(N) = d$ is constant for every CE and does not depend on N, and
- queue delay $d_q(N) = d_q$ is constant for every CE and does not depend on N.

The assumptions are necessary since the actual distributions would be difficult to obtain and the focus of this work is not on these distributions but instead on the run time distributions of the SAT solver when solving a given instance. If the effect of the number of CEs is taken into account, the delays will increase since in practice the

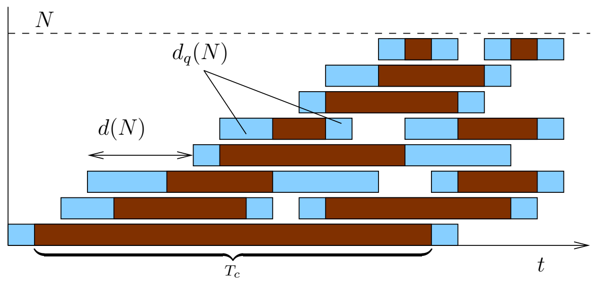


Figure 4.1: A time line of an execution in grid representing the number N of CEs, queue delay $d_q(N)$, and the submit delay $d(N)$. In the example, the first job has executed the maximum allowed time T_c on a CE.

jobs will interfere with each other. This means that using the simplifying assumptions the resulting run time is underestimated and this error increases with the number of CEs employed. Hence, the model with the simplifying assumptions gives overly optimistic results on speed-ups for larger numbers of CEs which needs to be taken into consideration when interpreting the results. Nevertheless, these assumptions allow us to study the effect of delays in a simple yet reasonably realistic environment.

Chapter 5

Randomized Parallel Solving in Grids

In this chapter we develop strategies for solving collections of hard instances of the propositional satisfiability problem (SAT) with a randomized SAT solver run in a grid. The results reported here have been published earlier in [55]. We study alternative strategies by using a simulation framework described in Chapt. 4 together with run-time distributions of a randomized solver, obtained by running a state-of-the-art SAT solver on a collection of hard instances. In terms of the taxonomy discussed in Chapt. 3, this corresponds to multi-search with no information exchanged between the solvers. While all the experimental results are obtained using a conflict-driven clause learning SAT solver similar in design to that discussed in Chapt. 2, the results and techniques are in principle not specific to SAT solving, but can be generalized to any technique where similar run time distributions are observed.

The results are experimentally validated in a production level grid that was used as a model for the simulation framework. When solving a single hard SAT instance, the results show that in practice only a relatively small amount of parallelism can be efficiently used; the speed-up obtained by increasing parallelism thereafter is negligible. This observation leads to a novel strategy of using a grid to solve collections of hard instances. Instead of solving instances one-by-one, the strategy aims at decreasing the overall solution time by applying an alternating distribution schedule.

One of the design criteria of the techniques developed here is to use state-of-the-art SAT solvers with no or only minor modifications. To achieve this, we use the *Simple Distributed SAT* (SDSAT) framework, whose basic version consists of simply running *N randomized SAT solvers* in parallel until one of them finds the solution. We consider extensions of the basic version obtained by incorporating different *restart strategies* and study their effects in a specifically built simulation environment.

This chapter first studies the effect of applying several restart strategies on a benchmark set of hard SAT instances in the sequential setting. The results show that there are instances on which the optimal fixed restart strategy provides a substantial reduction in the expected run time. The two universal strategies considered can also reduce the expected run time on some instances but result in a significant increase on other instances. The reason is that the universal strategies can spend too much time in trying to find a short run; when an instance has none, all that time is wasted.

Based on results in the sequential case, the chapter considers ways to parallelize restart strategies in the SDSAT framework and use the simulation model to benchmark them. The results give rise to two major observations. First, parallelism seems to be an effective “luck enhancer”; when randomized solvers are run in parallel, the probability that one of them finds a short run grows quite quickly. This seems to render elaborate restart strategies practically useless in the parallel setting as the simple approach with no restarts tends to provide quite good results consistently. The second observation is that only a relatively small amount of parallelism seems to be effectively exploitable; after a certain amount, adding more parallel solvers does not seem to give any significant performance gain. There seems to be two reasons for this: (i) the probability that a short run is found is already quite high with a smallish number of parallel solvers, and (ii) the delays in the grid environment reduce the effect of restart strategies.

The above results suggest that when solving a *set of instances*, a good speed-up is not obtained by solving instances of the set one-by-one in a grid. Instead, the instances should be solved in parallel by reserving a smallish amount of computing resources for each instance. This idea will be validated in Sect. 5.3 both with the simulation model and by using a production level grid.

5.1 Restart Strategies in a Sequential Setting

The key idea we will exploit is that a complete SAT solver can be viewed as a *randomized search procedure* (RSP) when it is implemented as described in Chapt. 2. For example, MiniSAT [28] 1.14 makes by default 2% of its heuristic choices pseudo-randomly; thus, a natural modification to turn MiniSAT into a RSP is to seed its pseudo-random number generator differently for each run. Such a randomized search procedure, when provided with an input x, is guaranteed to give a correct result $RSP(x)$ when the computation of the procedure finishes. However, due to the randomization, the time required for computing $RSP(x)$ is not known in advance but is described by a random variable $T_{RSP(x)}$. The random variable $T_{RSP(x)}$, and thus the run time of $RSP(x)$, is completely characterized by its cumulative *run time distribution* function, $q_{RSP(x)}(t)$, giving the probability that the computation on the input x will terminate before or at time t. This randomization of a SAT solver may sound counter-intuitive as one usually tries to remove all non-determinism in order to make runs reproducible to ease benchmarking and debugging. However, in the SDSAT framework as well as when employing restart strategies to a RSP (discussed below), the goal is to exploit the *short runs* (if any) in the distribution to decrease the *expected run time* of the overall system.

The expected run time of a randomized search procedure can often be substantially reduced by periodically restarting the procedure [42]. For example, assume that $T_{RSP(x)} = 1\text{s}$ with probability 0.3 and $T_{RSP(x)} = 10\text{s}$ with probability 0.7. Then the expected run time $\mathbb{E}T_{RSP(x)}$ is $0.3 \times 1\text{s} + 0.7 \times 10\text{s} = 7.3\text{s}$. If the RSP is modified so that it restarts itself immediately after time $t = 1\text{s}$, the expected run time becomes $\sum_{i=1}^{\infty} 0.7^{i-1} \times 0.3 \times i \approx 3.3\text{s}$. Such a modification, where the procedure is forced to start from the beginning after running t_1 seconds, then after t_2 seconds, and so forth, is called a *restart strategy* $S = (t_1, t_2, \ldots)$ and the time t_i the i:th *restart limit.* When a restart strategy is employed to a RSP, the result is a randomized *algorithm* that also has a run time distribution and an expected run time. The restart strategy employed in the previous example is a special case of a *fixed restart strategy* $S^t = (t, t, \ldots)$ and the algorithm corresponding to the fixed restart strategy S^t employed on a RSP is denoted by $\textsc{Fixed}_{t,RSP}$ (or simply $\textsc{Fixed}_t$ when the RSP is implicitly known). Fixed restart strategies are important in our analysis, since if $q_{RSP(x)}(t)$ is known, then t can be chosen so that the expected run time of $\textsc{Fixed}_t(x)$ is the minimal among all the algorithms obtainable from $RSP(x)$ by employing *any* restart strategy [74]. However, in practice $q_{RSP(x)}(t)$ is not known: obtaining information about $q_{RSP(x)}(t)$ in general requires solving $RSP(x)$, which is the overall goal in many applications. To circumvent this problem, several *universal* restart strategies have been suggested [74, 108]: they do not depend on the instance x and let the restart limits grow arbitrary large in order to preserve the completeness of the algorithm.

Given a randomized search procedure RSP and a problem instance x, it is possible to associate a distribution $q_{RSP(x)}(t)$ with the run time of $RSP(x)$. Employing a restart strategy S on a RSP results in a new algorithm with a potentially different run time distribution. In the following the instance x and the randomized search procedure RSP are both clear from the context, and therefore we often use notation $q(t) = q_{RSP(x)}(t)$. This section discusses the effect of using several such algorithms on a collection of SAT instances presented shortly by comparing the run time distributions $q_{RSP(x)}(t)$ with the run time distributions of the new algorithms. We use the following restart strategies and corresponding algorithms:

- $\textsc{Optimum}$. The fixed restart strategy S^t and the corresponding algorithm $\textsc{Fixed}_t$ have the property that there is a restart limit t^* which is optimal for a given RSP and instance x [74]. If the cumulative distribution function $q(t)$ of the instance is known, the optimal restart limit t^* may be determined by minimizing the expected run time $\mathbb{E}T_{\textsc{Fixed}_t(x)}$ as a function of the restart limit t,

$$\mathbb{E}T_{\textsc{Fixed}_t(x)} = \frac{t - \int_{t'=0}^{t} q(t')dt'}{q(t)}, \tag{5.1}$$

  i.e., $t^* = \mathrm{argmin}_t(\mathbb{E}T_{\textsc{Fixed}_t(x)})$. Determining an approximation of t^* can be done in our simulation environment but not usually in practice as the distribution $q(t)$ is typically not known.

- $\textsc{Luby}$. Luby et al. [74] define the universal strategy $S^{\mathrm{L}} = (l(1), l(2), \ldots)$ where

$$l(i) = \begin{cases} 2^{k-1}, & \text{if } i = 2^k - 1 \text{ for some } k \in \mathbb{N} \\ l(i - 2^{k-1} + 1), & \text{if } 2^{k-1} \le i < 2^k - 1 \text{ for some } k \in \mathbb{N}. \end{cases}$$

  For example, the first few terms of S^{L} are

$$(1, 1, 2, 1, 1, 2, 4, 1, 1, 2, 1, 1, 2, 4, 8, \ldots).$$

  When the strategy S^{L} is employed on a RSP, the corresponding algorithm is called $\textsc{Luby}$. In [74] it is further shown that the expected run time of $\textsc{Luby}(x)$ is within a logarithmic factor from the expected run time of $\textsc{Optimum}(x)$ independently of x.

- WALSH. Another universal strategy is the strategy $S^{\mathrm{W}} = (w(1), w(2), \ldots)$, where $w(i) = 2^{1.2i}$, presented in [108]. The strategy differs from the Luby strategy S^{L}, for example, in the rate of growth. Clearly, the restart limits in S^{W} grow exponentially, whereas those in S^{L} grows only linearly with respect to i. The corresponding algorithm will be referred to as WALSH.

5.1.1 Run Time Distributions

As a representative collection of SAT instances we use a set of benchmarks from the SAT 2007 Competition (see `http://www.satcompetition.org/2007/`). The instances, with the full name, abbreviated name, and satisfiability, are listed below.

- mod2-rand3bip-sat-250-3.shuffled-as.sat05-2220, `mod2-250`, satisfiable.
- mod2-rand3bip-sat-280-1.sat05-2263.reshuffled-07, `mod2-280`, satisfiable.
- 999999000001nc.shuffled-as.sat05-446, `99999900`, unsatisfiable.
- clqcolor-10-07-09.shuffled-as.sat05-1258, `clqcolor`, unsatisfiable.
- cube-11-h14, `cube`, satisfiable.
- dated-10-13-s, `dated`, satisfiable.
- mizh-md5-48-5, `mizh-md5`, satisfiable.
- vmpc_28.shuffled-as.sat05-1957, `vmpc_28`, satisfiable.
- AProVE07-16, `AProVE07`, unsatisfiable.

The set is selected so that it covers both industrial and hand-crafted instances, and the run times of the instances are typically thousands of seconds for a state-of-the-art SAT solver. The run times are obtained by running all the instances from the SAT 2007 competition once in a heterogeneous environment with MiniSAT 1.14 [28].

The SAT solver run time distributions are approximated by using a collection of samples for each instance. The samples are obtained by 100 separate randomized runs of a state-of-the-art SAT solver (MiniSAT version 1.14 with its pseudo-random number generator initialized differently for each run). The randomized runs are used as a basis for constructing a distribution of run times with linear interpolation between the sample points, assuming probability 0 for runs shorter than the minimum sampled time and for runs longer than the maximum sampled time[1].

Table 5.1 documents for each instance the abbreviated names and the SAT solver run times for minimum, fifth percentile, median, 95th percentile and maximum of the samples (the other columns can be ignored for now). Also provided is the average of the samples, i.e., an approximation of the expected run time of the solver on the instance, in the *RSP* column. At this point, of particular interest are the large dynamics in certain distributions, such as `vmpc_28` with over 19000-fold difference

[1]Omitting the linear interpolation and using discrete distribution did not significantly affect the results.

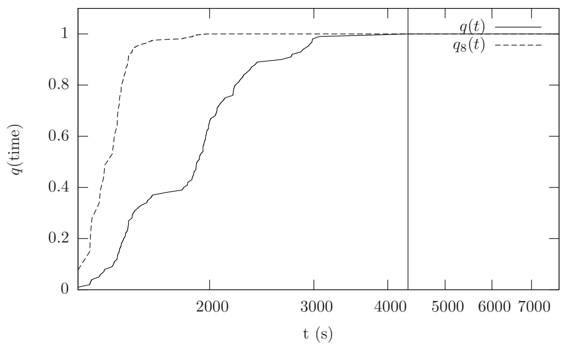


Figure 5.1: Run time distributions for `clqcolor`

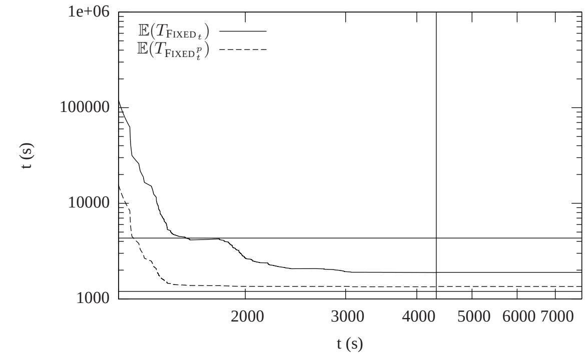


Figure 5.2: Expected run times for `clqcolor`. The horizontal lines correspond to maximum and minimum run times of the distribution.

between minimum and maximum run time. The cumulative run time distributions for two of the test instances are presented in Figs. 5.1 and 5.3, where the distribution is the increasing graph $q(t)$. The horizontal lines in Figures 5.2 and 5.4 indicate the maximum and minimum run times of the instance and the vertical line indicates the maximum run time on the x-axis, i.e., the value of t where $q(t) = 1$.

It can be argued that 100 samples is not enough to give us a realistic view of the run time distribution of an instance. In order to estimate the magnitude of the error introduced to the finite distribution, the distributions of `cube` with 100 samples and 1000 samples are compared in the first two rows of Table 5.2. Even though the minimum run time decreases and the maximum run time increases, the distribution seems to remain relatively stable when increasing the number of samples. To have an impression on how, for example, a short run would affect the results, an artificial short time not obtained while solving the instance is inserted to the samples. The resulting distribution seem similar to the distribution of `vmpc_28`.

Table 5.1 compares the three algorithms against the run time of RSP. Column RSP reports the expected run time of $RSP(x)$ for different instances x. The optimum restart limit t^* is computed using the run time distribution $q_{RSP(x)}(t)$ for each instance and minimizing Eq. (5.1). The resulting expected run time is reported on column OPTIMUM and the corresponding restart limit in column t^*. The value ∞ is used to mark the cases when run times for OPTIMUM(x) and $RSP(x)$ are equal. In this

Table 5.1: Characteristics of the run times for the test instances

Instance	Min	5%	Median	95%	Max	*RSP*	Optimum	t^*	Luby	Walsh
`mod2-250`	40.16	97.16	1,210	2,675	3,088	1,181	1,181	∞	2,715	1,510
`mod2-280`	9.184	55.71	1,732	6,611	7,775	2,382	918.4	9.184	1,274	1,718
`99999900`	1,072	1,204	2,056	3,101	3,725	2,065	2,065	∞	25,070	4,560
`clqcolor`	1,198	1,300	1,922	2,955	4,329	1,900	1,900	∞	23,060	4,158
`cube`	2,629	2,896	4,708	7,936	10,049	4,832	4,832	∞	106,200	18,500
`dated`	10.09	46.53	803.0	12,550	37,930	2,279	716.1	29.08	901.5	993.3
`mizh-md5`	49.76	128.7	861.7	5,784	9,489	1,660	1,236	899.3	3,403	1,471
`vmpc_28`	0.1370	3.905	394.7	1,730	2,720	623.3	12.71	0.2560	137.4	279.6
`AProVE07`	879.4	1,071	1,471	2,713	2,855	1,564	1,564	∞	17,330	3,381

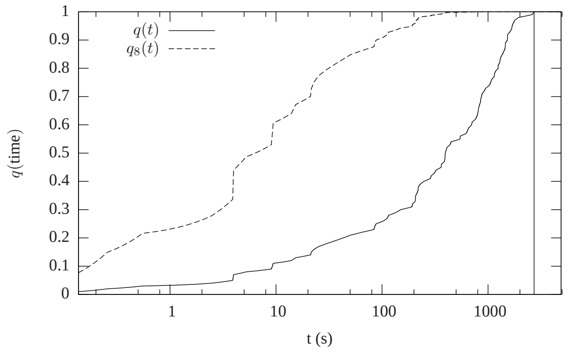


Figure 5.3: Run time distributions for `vmpc_28`

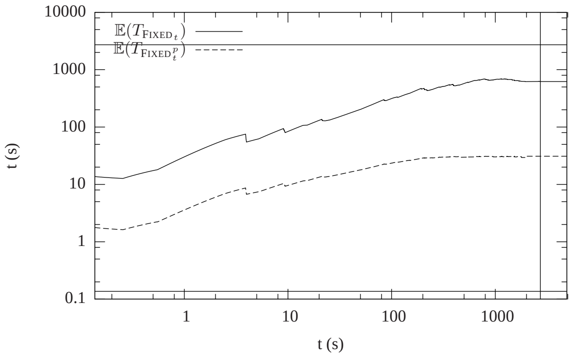


Figure 5.4: Expected run times for `vmpc_28`. The horizontal lines correspond to maximum and minimum run times of the distribution.

Table 5.2: Comparison of the distributions for cube with 100 samples ($\texttt{cube}_{100}$), 1000 samples ($\texttt{cube}_{1000}$), and a modified distribution with one artificial short run inserted ($\texttt{cube}_{1001m}$).

Instance	Min	5%	Median	95%	Max	*RSP*	Optimum	t^*	Luby	Walsh
$\texttt{cube}_{100}$	2,629	2,896	4,661	7,617	8,821	4,832	4,832	∞	106,200	18,500
$\texttt{cube}_{1000}$	1,441	2,990	4,914	7,664	14,050	5,067	5,067	∞	97,360	31,510
$\texttt{cube}_{1001m}$	0.7352	2,990	4,914	7,647	14,050	5,061	725.9	0.735	5,101	30,280

collection of instances, in five cases out of nine the expected run time of $\textsc{Optimum}(x)$ is equal to that of $RSP(x)$. Some of the satisfiable instances, though not all, seem to profit from employing a fixed restart strategy with small restart limit. As an example, the expected run time for the algorithm $\textsc{Fixed}_t$ with input `vmpc_28`, is shown in Fig. 5.4 as a function of the restart limit t (graph labeled $\mathbb{E}T_{\textsc{Fixed}_t}$). In other cases, the expected run times of algorithms with larger restart limits compare favorably to those with smaller restart limits. An example is shown in Fig. 5.2 (again $\mathbb{E}T_{\textsc{Fixed}_t}$).

The results for the two universal strategies are shown in columns Luby and Walsh of Table 5.1. Based on the results, it seems that in most cases the instances where $\mathbb{E}T_{\textsc{Optimum}(x)}$ is less than $\mathbb{E}T_{RSP(x)}$ also profit of more complex strategies. The strategy Luby performs very badly on many instances with a high minimum run time. This is a consequence of the slow growth of the restart limit in the strategy S^{L}. In general, the algorithm Walsh seems to offer a relatively robust approach, resulting in good speed-up where such would be obtainable with $\textsc{Fixed}_{t^*}$ given that t^* is known, and still performing usually well in cases where $\mathbb{E}T_{\textsc{Optimum}(x)} = \mathbb{E}T_{RSP(x)}$. This is a slightly surprising result, since it can be shown that unlike the Luby strategy, S^{W} is exponentially worse than the best strategy for some distributions [111][2].

5.2 Parallel Solving of a Single Instance

In the previous section we discussed several restart strategies and resulting sequential algorithms when the strategies are employed to a RSP. In this section we develop a number of *parallel algorithms* for grid environments based on the restart strategies. Here we consider a grid environment as an efficient distributed system for running jobs. Hence, the algorithmic design boils down to approaches to constructing a sequence of jobs $j_1, j_2, \ldots$ to be submitted to the grid for execution based on a RSP and a restart strategy. Since each job has a resource limit T_c limiting the execution time, we employ a *finite restart strategy* (discussed below) on the RSP which guarantees that the run time of the resulting algorithm is not more than T_c. Hence, each job j_i consists of the RSP, the input x to be solved and a finite restart strategy.

A *finite restart strategy* $S = (t_1, t_2, \ldots, t_n)$ is a finite sequence of restart limits which, when employed on a RSP, will terminate the resulting algorithm unless a solution is found by the end of the restart limit t_n. The *length* of the finite restart strategy S, denoted by $|S|$, is n. Given a restart strategy $S = (t_1, t_2, \ldots)$ and a resource limit T_c, we define an operator $\mathrm{finite}_{T_c}(S)$ for constructing finite restart strategies from S as

$$\mathrm{finite}_{T_c}(S) = \begin{cases} (T_c) & \text{if } t_1 > T_c \\ (t_1, t_2, \ldots, t_m) & \text{where } m \text{ maximizes } \sum_{i=1}^{m} t_i \le T_c \text{ otherwise.} \end{cases}$$

For any restart strategy S, the run time of the algorithm obtained by employing $\mathrm{finite}_{T_c}(S)$ on a RSP is less than or equal to T_c.

[2]The example distribution used in [111] is a pathological example of a discrete distribution and seems to be rare in practice.

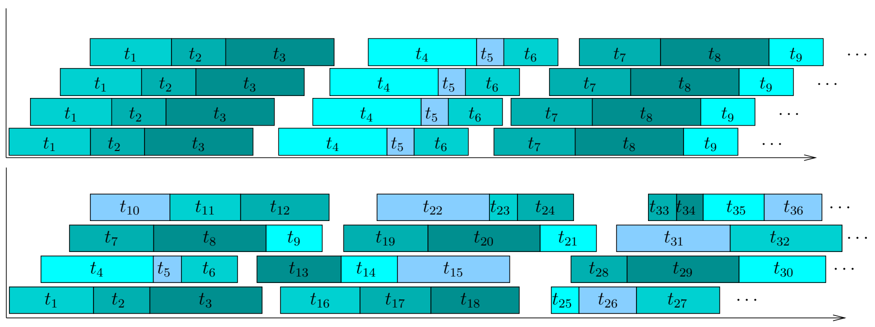


Figure 5.5: Illustration of the faithful (bottom) and straightforward (top) schemes with the same exemplary restart strategy. The x-axis in the figures correspond to time and y-axis to the parallel resources. See Fig. 4.1 and related text for more details on the grid environment. The queuing delays are omitted in the figures.

The most intuitive way of constructing jobs from a restart strategy $S = (t_1, t_2, \ldots)$ is to assign the job j_i the restart strategy (t_i) for $i = 1, 2, \ldots$. In practice this approach performs very poorly due to the high delays in actual grid environments. Therefore, the parallel algorithms are based on two general *schemes* for constructing a sequence of jobs, given a restart strategy S.

- *Straightforward scheme.* Given a restart strategy S for constructing jobs we define a sequence of restart strategies $S_1, S_2, \ldots$ in the following way: let $S_1 = S$ and given a strategy S_i, the restart strategy S_{i+1} is constructed from S_i by removing the first $|\text{finite}_{T_c}(S_i)|$ restart limits from S_i. Given an environment with N computing elements, in the straightforward scheme jobs are constructed from the sequence $S_1, S_2, \ldots$ by assigning the restart strategy $\text{finite}_{T_c}(S_1)$ for the jobs $j_1, \ldots, j_N$, then $\text{finite}_{T_c}(S_2)$ for the jobs $j_{N+1}, \ldots, j_{2N}$ and so forth. This strategy is discussed in [73].

- *Faithful scheme.* In this scheme given a restart strategy S we construct the sequence $S_1, S_2, \ldots$ as above and then assign the job j_1 the restart strategy $\text{finite}_{T_c}(S_1)$, the job j_2 the restart strategy $\text{finite}_{T_c}(S_2)$, and so forth.

Both schemes are illustrated in Fig. 5.5

Parallel Algorithms. Given the randomized search procedure and the distributed environment, the parallel algorithm is uniquely determined by the used scheme (introduced above) and the restart strategy. Furthermore, for all fixed restart strategies, the straightforward and faithful schemes result in the same parallel restart strategy, and thus the same algorithm. We will discuss six parallel algorithms:

- The *maximum parallel algorithm* $\text{Fixed}^p_{T_c}$ is formed from the fixed restart strategy S^{T_c}.

- The *parallelized optimal algorithm* $\text{Fixed}^p_{t^*}$ is formed by finding a value t^* which minimizes the parallel run time distribution

$$\mathbb{E}T_{\text{Fixed}^p_t(x)} = \frac{t - \int_{t'=1}^{t}(1 - (1 - q(t'))^N)dt'}{1 - (1 - q(t))^N} \tag{5.2}$$

for $RSP(x)$ with the run time distribution $q(t)$. Equation (5.2) is obtained from Eq. (5.1) by substituting $q(t)$ with the corresponding parallel distribution $1-(1-q(t))^N$. However, as shown in [73], there are run time distributions for which $\textsc{Fixed}^p_{t*}$ does not result in minimum expected run time over all parallel algorithms.

- The *faithful parallel Luby and Walsh algorithms* $\textsc{Luby-f}^p$ and $\textsc{Walsh-f}^p$ are constructed by using the faithful scheme on the strategies S^{L} and S^{W}, respectively.

- The *straightforward parallel Luby and Walsh algorithms* $\textsc{Luby-s}^p$ and $\textsc{Walsh-s}^p$ are constructed by using the straightforward scheme on the strategies S^{L} and S^{W}, respectively.

Zero-Delay Parallel Environment. In this subsection we consider an idealized grid environment captured by the grid model, where we set the delays $d = d_q = 0$ and the resource limit $T_c = 3600$s. This provides us with a lower bound on the run times achievable in more realistic grid environments.

The results for the maximum parallel algorithm are reported in column $\textsc{Fixed}^p_{T_c}$ of Table 5.3 for 16 and 64 CEs. For comparison, the column $\textsc{Fixed}^p_{t*}$ reports the results when using the parallelized optimal algorithm $\textsc{Fixed}^p_{t*}$. These results have been computed with no limits on the resources. Therefore t^* may be greater than T_c.

The speed-up is in most cases linear with respect to the added resources, and for `vmpc_28` even super-linear, for both $\textsc{Fixed}^p_{T_c}$ and $\textsc{Fixed}^p_{t*}$. For some instances, however, the speed-up is negligible. It seems that there are certain distributions for which the parallelization method does not result in speed-up after a certain amount of CEs has been reached. Two different examples of this phenomenon are closer studied in Figs. 5.2 and 5.4 for $N = 1$ and $N = 8$. The graphs labeled $\mathbb{E}T_{\textsc{Fixed}^p_t}$ in the figures are the expected run times of the algorithm $\textsc{Fixed}^p_t$ as a function of the restart limit t. In Fig. 5.2, the run time of the algorithm $\textsc{Fixed}^p_t$ with large values of t is almost equal to that of the shortest sampled run (the lower horizontal line) which can also be seen from the run time distribution of the algorithm $\textsc{Fixed}^p_t$ when $N = 8$, $q_8(t)$, in Fig 5.1. The situation is different in Fig 5.4, where the shortest run is much shorter than the expected run also when $N = 8$.

The difference between $\textsc{Fixed}^p_{T_c}$ and $\textsc{Fixed}^p_{t*}$ becomes insignificant when N increases. The intuitive explanation for this is that the benefit of aggressive restarting can be obtained by running several solvers in parallel, as discussed in Chapt. 3. The important consequence of the phenomenon is that with a large number of CEs, the significance of the restart strategies decreases.

The remaining columns in Table 5.3 show the behavior of the strategies S^{L} and S^{W}. The results are obtained by simulating 100 runs of the parallel algorithms and reporting the mean time required to find the solution. The columns $\textsc{Luby-s}^p$ and $\textsc{Walsh-s}^p$ correspond to the straightforward parallel restart strategy for S^{L} and S^{W}. This scheme has the benefit that small restart limits are attempted often. However, especially $\textsc{Luby-s}^p$ suffers from the repeating of the short runs in cases where the smallest run time is high. The results corresponding to the faithful scheme are

Table 5.3: Results for different strategies and the zero-delay parallel environment

Instance	N	FIXED$^p_{t^*}$	FIXED$^p_{T_c}$	LUBY-S^p	WALSH-S^p	LUBY-F^p	WALSH-F^p
mod2-250	16	105.7	116.2	334.2	177.5	171.8	114.0
	64	47.25	47.25	194.6	84.86	50.23	45.32
mod2-280	16	61.82	84.52	71.44	76.65	67.65	79.32
	64	19.36	21.55	22.29	25.69	21.44	24.58
99999900	16	1,219	1,219	14,657	2,910	1,620	1,238
	64	1,097	1,097	14,530	2,784	1,213	1,094
clqcolor	16	1,293	1,293	14,730	2,963	1,553	1,301
	64	1,223	1,223	14,660	2,899	1,287	1,224
cube	16	2,891	2,891	33,600	6,777	8,105	2,996
	64	2,682	2,682	33,410	6,570	3,086	2,687
dated	16	48.44	64.12	59.30	53.29	63.46	60.15
	64	15.89	16.33	15.92	16.05	14.69	19.26
mizh-md5	16	133.8	133.8	525.8	116.6	162.1	125.4
	64	73.23	73.23	259.2	126.1	84.53	81.76
vmpc_28	16	0.834	7.293	4.694	6.065	4.366	11.22
	64	0.251	0.539	0.6507	0.7994	0.6550	0.5003
AProVE07	16	1,049	1,049	11,040	2,285	1,299	1,064
	64	918.8	918.8	7,823	1,823	1,056	915.4

reported in columns LUBY-F^p and WALSH-F^p. In most cases the faithful scheme performs significantly better than the straightforward scheme, and when this is not the case, the difference is relatively small.

A natural generalization of the strategies S^{L} and S^{W} is to multiply the restart limits of the strategies by a constant factor f. Table 5.4 shows the results for different values of f and 64 CEs. Based on these results, the factor does not seem to have a significant effect on the run times. The runs in Table 5.3 (as in Table 5.5) are measured with $f = 15.0$.

Table 5.5 studies the effect of a larger sample base similar to the case in Table 5.2 in the zero-delay environment. For this particular instance, the strategy FIXED$^p_{t^*}$ is equal to the maximum strategy both when the amount of samples is 100 and 1000. In this case, when the number of samples is increased, the expected solving time decreases for most algorithms. There is no significant difference between WALSH-F^p and FIXED$^p_{T_c}$ whereas LUBY-F^p suffers from a larger number of short unsuccessful runs (even though not visible in Table 5.2, the distributions are significantly different when $t \leq T_c$; e.g. $q(3600\text{s}) \approx 0.24$ in the 100 samples distribution but only approximately 0.14 in the 1000 samples case). Since `cube` is a satisfiable instance, it is possible that there is a short run time for the randomized SAT solver. Since the 1000 samples did not reveal a short run time, it might be that the run is extremely improbable. To study the effect of such a short successful run we may modify the sample set of the instance `cube` to include a single, artificial, short run. The resulting run times are given in the row labeled $\texttt{cube}_{1001m}$. In this case, LUBY-F^p is better than FIXED$^p_{T_c}$ because of the higher probability of finding the short run. Based on the two first rows of the table, may conclude that hundred is usually sufficiently large amount of samples to determine the trends for restart strategies on an instance, even though outliers, when such exist, will significantly alter the result.

Non-Zero Delay Parallel Environment. The simulation results from the parallel environment with zero submission delay and zero queuing delay provide some insight to how the parallelization method based on randomizing algorithms can perform on the benchmark set. However, realistic parallel environments in general, and grid environments in particular, always include some overhead related to initializing the computations. As described in Sect. 4.2, we divide the delays into two categories: submit delay d and queue delay d_q. Typical values in NorduGrid are currently $d = 12\text{s}$ and $d_q = 125\text{s}$. However, the two values seem to vary strongly. The simulated experiments are presented in Table 5.6 under the title "large delay". All results are obtained by computing the mean run time over 100 samples using $T_c = 3600\text{s}$ for the jobs.

The results show that almost always the maximum parallel algorithm FIXED$^p_{T_c}$ outperforms those based on universal restart strategies on these instances. It is worth noting that increasing the number of CEs four-fold brings next to nothing in speed-up, a consequence of the long queuing delays.

It is possible that the submission and queue delays are significantly shorter in, say, some other grid environments. The effect of smaller delays can be simulated by using submission delay $d = 5\text{s}$ and queue delay $d_q = 30\text{s}$. The results are reported under the caption "small delay". Even though the strategies S^{L} and S^{W} are now more

Table 5.4: Comparison of 64-CE LUBY-F^p and WALSH-F^p with $f = 1$, $f = 15$, and $f = 100$

	LUBY-F^p			WALSH-F^p		
Instance	$f = 1$	$f = 15$	$f = 100$	$f = 1$	$f = 15$	$f = 100$
mod2-250	68.09	50.23	47.19	46.54	48.85	48.55
mod2-280	34.71	21.44	20.16	23.82	21.69	18.55
99999900	1,372	1,213	1,166	1,093	1,096	1,105
clqcolor	1,345	1,287	1,262	1,220	1,224	1,222
cube	3,950	3,086	2,977	2,696	2,688	2,677
dated	28.43	14.69	18.71	18.74	15.85	18.57
mizh-md5	98.35	84.53	74.76	82.65	72.48	79.45
vmpc_28	0.5140	0.6550	0.6560	0.4717	0.5401	0.4870
AProVE07	1,088	1,056	992.2	930.2	936.2	914.76

competitive, their effectiveness still suffers from the high delays and it can be argued that the maximum timeout is a sufficient approximation of the optimum. The super-linear speed-up observed in zero-delay environment cannot be observed in either of the delayed environments. For certain instances, such as `99999900` and `cube`, already a smallish number of parallel runs suffices to find a short run from the samples. As a result, obtainable speed-up is small.

These results are confirmed by repeating them for two instances in the NorduGrid grid environment. Two instances are selected which, according to the simulated results, are illustrative examples on the techniques used in parallel solving. The instance `vmpc_28` shows super-linear speed-up in simulations in zero-delay environments, but only a moderate speed-up for larger number of CEs in delayed environments using the techniques we have studied. The instance `AProVE07`, on the other hand, has a less dynamic distribution in the simulations and yields no significant speed-up at the transition from 16 to 64 CEs even in the zero-delay environment. The results are presented in Table 5.7. The submission delays seem to be below the average delay of 12 seconds, but the results correspond approximately to the simulated results. No speed-up seems to be achieved when the number of CEs is increased.

We may draw the following conclusions from the experiments described in this section:

- Outlier samples greatly affect the strategy that is most effective in solving a given instance, but hundred samples seems to be enough in practice to provide reliable information on the distribution.
- Linear scaling of the restart strategies does not significantly change their behavior.
- Usually the faithful scheme is more efficient than the straightforward scheme, which occasionally increases the solving time for some instances significantly compared to other parallel algorithms.
- The significance of restart strategies to the expected run times of the instances is large for small amounts of CEs and decreases as the amount of CEs increases.
- The differences between restart strategies are small when delays are considered.

5.3 Parallel Solving of a Set of Instances

Often in realistic solving scenarios we are not interested in solving a single SAT instance, but instead a set of previously created SAT instances. The results from the

Table 5.5: Effect of additional samples on the zero-delay solving of `cube` with 64 CEs

Instance	$\text{Fixed}^p_{t^*}$	$\text{Fixed}^p_{T_c}$	Luby-f^p	Walsh-f^p
$\texttt{cube}_{100}$	2,682	2,682	3,086	2,687
$\texttt{cube}_{1000}$	2,364	2,364	3,760	2,270
$\texttt{cube}_{1001m}$	11.86	2,175	969.8	2,185

Table 5.6: Results for different strategies and delayed parallel environments. The two rows for each instance correspond to $N = 16$ (top) and $N = 64$ (bottom)

		small delay				large delay			
Instance	N	Fixed$^p_{t^*}$	Fixed$^p_{T_c}$	Luby-f^p	Walsh-f^p	Fixed$^p_{t^*}$	Fixed$^p_{T_c}$	Luby-f^p	Walsh-f^p
`mod2-250`	16	177.0	145.1	232.7	164.4	352.8	379.3	399.8	399.3
	64	161.5	157.7	182.7	133.5	364.4	355.7	422.7	350.4
`mod2-280`	16	125.8	159.0	137.4	150.7	306.8	331.0	321.4	350.0
	64	118.1	126.2	135.2	132.4	296.3	327.7	320.9	340.1
`99999900`	16	1,242	1,268	1,672	1,306	1,431	1,477	1,984	1,527
	64	1,208	1,246	1,401	1,253	1,432	1,485	1,756	1,490
`clqcolor`	16	1,340	1,353	1,455	1,378	1,506	1,525	1,846	1,577
	64	1,328	1,351	1,448	1,352	1,508	1,536	1,777	1,554
`cube`	16	2,882	2,960	9,209	3,067	3,094	3,117	9,233	3,195
	64	2,792	2,840	3,489	2,842	3,050	3,121	4,159	3,145
`dated`	16	112.1	140.5	138.2	126.1	272.2	323.8	281.6	312.1
	64	104.7	114.1	116.6	117.4	284.3	309.2	293.8	305.8
`mizh-md5`	16	181.4	190.4	268.7	199.4	352.6	391.2	445.0	395.3
	64	190.4	186.3	208.5	195.0	379.8	385.2	464.8	392.0
`vmpc_28`	16	43.27	67.35	62.70	65.49	155.7	206.7	198.4	214.0
	64	42.18	68.06	62.59	64.30	155.5	218.3	200.0	212.0
`AProVE07`	16	1,073	1,089	1,313	1,127	1,262	1,289	1,569	1,310
	64	1,073	1,065	1,205	1,061	1,292	1,299	1,568	1,300

Table 5.7: Experimental results in grid for selected instances. Reported is the average over 10 runs using the strategy $\textsc{Fixed}^{p}_{T_c}$. The column d reports the measured submission delay.

Instance	CEs	Time	d
`vmpc_28`	8	105.4	3.333
	16	125.7	7.668
	64	134.5	5.189

Instance	CEs	Time	d
`AProVE07`	8	1,624	5.917
	16	1,574	9.714
	64	1,271	8.555

Table 5.8: Expected solving times in parallel solving of a set of instances

Instance set	Single CE	64 CEs / instance one-by-one (simulated)	8 CEs/instance simultaneously
All but `cube`	13,650	5,916	1,865
All	18,486	9,037	5,136

following experiment show that in such case it is more efficient to limit the number of CEs reserved for each instance and solve simultaneously several instances from the set than it is to use all available CEs to solve each instance one-by-one.

For this experiment, 8 problems from the benchmark set of 9 problems are selected and run in parallel with 64 CEs, reserving at most eight CEs per problem. This enables comparing the results of this experiment against a strategy where 64 CEs are dedicated for a single instance at a time.

Initially the instance `cube` is excluded from the set of instances, since this problem is in the limit of solvable problems within 3600 seconds in the grid environment, having expected run time of 4708 seconds in the simulation environment. The run times for the set is reported on the first row on Table 5.8. The columns report in order the expected solving time when each instance is solved one-by-one using single CE (results on Table 5.1), when 64 CEs are used for solving the instances one-by-one (results on Table 5.6 and long delays) and when at most 8 CEs are reserved for each instance and all are solved simultaneously (results on NorduGrid). The speed-up when switching from one-by-one solving to simultaneous solving is over three, while the total speed-up when switching from using a single CE to the simultaneous solving is slightly more than seven. Therefore, the effect of simultaneous solving is significant even compared to the effect of introduction of more computing elements.

However, the results can be significantly worse if a difficult instance, such as `cube`, is included in the set of problems to solve. The bottom row of Table 5.8 compares the results now using resource limit $T_c = 7200$ seconds and including `cube` to the set of problems to solve. This resulted in a speed-up less than two with average solving time compared to the expected solving time when solving instances one-by-one with long delays and 64 CEs. When these results are compared against a simple strategy of running the problems on a single CE with no delays, the speed-up is 3.60.

The maximum obtainable speed-up in solving instances simultaneously in the setting discussed in this paper is the maximum of minimum run times over all instances in the set. Based on the results in Table 5.1, this speed-up is slightly over 11 for the first row and slightly over 7 for the second row. Therefore the speed-ups obtained with the simultaneous solving are relatively good in a realistic widely distributed computing environment.

Based on these results, the design of an algorithm for solving a collection of SAT instances efficiently in a grid environment should consider the two observations:

(i) an increase in the number of CEs does not result in a corresponding speed-up when solving a single instance, and

(ii) for a large number of problems to solve, a good speed-up is not obtained by using all the resources for solving a single problem at a time, but rather by dedicating only a certain amount of CEs for a single problem and solving multiple problems simultaneously instead.

These observations lead to the following *locally-aided fair-share algorithm*: Given a collection of instances, the instances are sent for solving in a round-robin manner by using the maximum parallel algorithm $\text{F{\scriptsize IXED}}^{p}_{T_c}$. In addition, the problems are also solved locally at the same time using an algorithm similar to $\text{L{\scriptsize UBY}}$ with the modified strategy $S^{\text{L},C} = (\min\{l(1), C\}, \min\{l(2), C\}, \ldots)$, where C is a *maximum local run time constant.* Local solving should be performed in a round-robin manner. The effect of the addition of local solving into the above results does not significantly alter the results, since they are dominated by the run times of the difficult instances.

5.4 Remarks

In this chapter we have developed techniques for solving collections of hard SAT instances in a grid using a randomized SAT solver. The idea of randomization and restarts, introduced in Chapt. 2 and widely used in sequential SAT solvers as well as forming the basis of some multi-search methods as discussed in Chapt. 3, were extensively studied as a means of obtaining speed-up in grid environments. The results show that modern SAT solvers already perform quite well with their restarts. This can be seen from the relatively high degree of instances for which the expected run time, when the optimum restart strategy is employed, equals the expected run time of the underlying SAT solver. They also show that randomization is a good source of speed-up when relatively small number of computing elements are used simultaneously.

We have compared different approaches using a simulation framework consisting of the grid model capturing the communication and management delays, and obtained a representative collection of run-time distributions by running a randomized SAT solver on a set of instances. The results are experimentally confirmed also in NorduGrid which is a European-wide distributed production-level grid. When solving a single hard SAT instance, the results show that in practice often (i) a relatively small number of parallel jobs suffices to increase the probability of finding a short run in the distribution to a significant level and (ii) the non-negligible delays in a grid eliminate super-linear speed-ups that could be obtained in an ideal environment without any delays. Hence, attempts to decrease the overall expected run time by using clever universal restart strategies or by finding optimal restart limits do not lead to significant improvements compared to using the resource limit implied by the grid environment as the restart limit.

These observations lead to a novel strategy of using grid to solve *collections* of hard instances. Instead of solving instances one-by-one, the strategy aims at decreasing the overall solution time by applying an alternating distribution schedule called locally-aided fair-share algorithm: Given a collection of instances, the instances are solved in parallel in a straightforward manner, but in addition also locally using a restart strategy which is a slightly modified version of the Luby strategy described in [74].

Speed-up obtainable on an instance by using exclusively restart strategies is limited by the minimum run time required to solve the instance. The following chapter discusses an approach based on the learned clauses constructed by a CDCL SAT solver during its search. This allows altering the minimum run time of an instance and therefore lifts the limitation on the obtainable speed-up.

Chapter 6

Techniques for Parallel Learning

This chapter extends the techniques described in the previous section by incorporating clause learning of the conflict-driven clause learning SAT algorithms, known to yield significant speed-ups in the sequential case, in grid environments similar to those described in Chapt. 4. The results in this chapter have previously been published in [54][1]. Compared to the SDSAT approach discussed in the previous section, the techniques aim at solving a wider range of SAT instances. The techniques exploit the principle of extending the instance using the results obtained from the jobs which were not able to determine the satisfiability of the instance. The approach exploits existing state-of-the-art CDCL SAT solvers which are modified so that the set of learned clauses they maintain can be extracted after they are finished executing as a result of reaching a resource limit. Some of the learned clauses obtained from such solvers are included to the original SAT instance so that the search space of the original SAT instance is reduced, hopefully resulting in an instance with lower solving time. Such modifications preserve the logical equivalence between the original and the altered instance and therefore the techniques are examples of multi-search with a limited form of communication according to the taxonomy discussed in Chapt. 3. Faithful to the grid environment described in Chapt. 4, the approach observes the master-worker architecture.

A substantial amount of controlled experiments demonstrates that the CL-SDSAT framework enables a form of efficient clause learning which is not directly available in the underlying sequential SAT solver. Finally, an implementation of the algorithm is run in a production level grid where it solves several problems not solved in the SAT 2007 solver competition.

6.1 Motivation

This chapter describes an enhancement for the SDSAT approach called *Clause Learning Simple Distributed SAT* (CL-SDSAT). The basic idea is quite straightforward: a master process submits jobs consisting of a randomized state-of-the-art clause learning SAT solver and a SAT instance to the distributed environment until one of the jobs solves the problem. In order to solve hard problems in the presence of resource limits

[1]The chapter is based on the article [56], currently in peer-review.

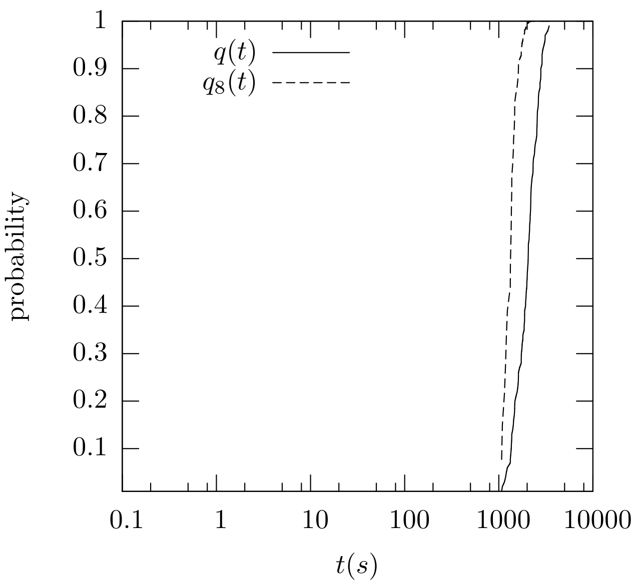


Figure 6.1: The run time distribution of a SAT instance for single (the $q(t)$ plot) and eight (the $q_8(t)$ plot) randomized SAT solvers.

imposed on jobs, the approach exploits the work done in *unsuccessful* jobs (i.e. those that exceeded the resource limits without finding a solution) by transferring some of the clauses learned by the solver back to the master process. When new jobs are submitted later, some of the learned clauses are passed to the jobs to constrain the search of the solver. This approach enables a form of clause learning which is not directly available in the underlying sequential SAT solver: on one hand, learned clauses from multiple independent unsuccessful jobs are combined and, on the other hand, the clauses learned from such combinations are *cumulated*. The proposed approach is designed to fit well the distributed environment provided by standard grid environments as (i) the amount of data transferred back to the master process is relatively low and can be performed at the end of the computing, (ii) the jobs have no need to communicate with each other, (iii) each job has predefined resource limit for the time and memory it is allowed to consume, and (iv) CE failures do not affect the correctness or completeness of the approach. Although more involved than in the SDSAT approach, the modifications required in CL-SDSAT to the SAT solver are still relatively small; therefore it should be easy to exploit the future improvements in clause learning SAT solvers in the CL-SDSAT approach. However, the benefits of CL-SDSAT when compared to SDSAT are clear. Although the SDSAT approach can reduce the expected time to solve an instance, it cannot reduce it below the minimum running time (i.e. the smallest t for which $q(t) > 0$). For an example, observe the sequential run time distribution $q(t)$ of a SAT instance given in Fig. 6.1 (a SAT instance encoding prime number factorization); the variation of the run time is relatively small and the instance seems to have no short running times. Contrasting this to the run time distribution in Fig. 5.3 it is clear that the speed-up obtained will be much less impressive for the distribution in Fig. 6.1. Consequently, running eight SAT solvers in parallel (the plot $q_8(t)$) does not reduce the expected running time significantly;

in numbers, the (approximated) expected running time for this instance is 2,065 seconds with one solver and 1,334 seconds (i.e., only less than two times faster) for eight parallel solvers. Even more importantly, the *minimum running time stays the same irrespective of how many parallel solvers are employed.* This is a serious drawback when solving *hard SAT problems* in a grid-like environment where the computing elements usually impose an upper limit for the computing time available for a single job. For example, if the computing elements only allow four hours of CPU time for each job, the basic SDSAT approach simply *cannot solve any problem with a longer minimum running time* because none of the SAT solvers running in parallel can solve the problem within that time. Notice that the "straightforward" approach of storing the memory image of a solver execution just before the time limit is reached and then continuing the execution in a new job in another computing element is not a viable solution due to the amount of data that should be transferred between the jobs.

6.2 Simplifications

This chapter uses two standard concepts in order to remove some redundancy in large clause sets. First, given a formula (i.e., a set of clauses) $\mathcal{F}$, the set $\mathrm{Prop}(\mathcal{F})$ of *unit clauses implied by unit propagation* is the smallest set U containing all literals obtained by propagation using the formula $\mathcal{F}$ and the empty partial truth assignment as unit clauses (see Sect. 2.2.1). We note that the set of unit clauses implied by unit propagation are exactly the implied literals obtained by the propagation rule of a SAT solver before applying the branching rule for the first time. A key property of these literals is that a formula $\mathcal{F}$ is logically equivalent to the formula augmented with the implied unit clauses, i.e. $\mathcal{F} \cup U$. Furthermore, if $\mathrm{Prop}(\mathcal{F})$ contains two mutually inconsistent unit clauses (v) and $(\neg v)$, then $\mathcal{F}$ is unsatisfiable.

Second, given a set U of unit clauses, a formula (i.e. a set of clauses) $\mathcal{F}$ can be *simplified* with respect to U by (i) removing all clauses that contain a literal appearing in U, and (ii) removing from all the remaining clauses every literal whose negation appears in U. Formally, letting $\hat{U} = \{l \mid (l) \in U\}$ we define

$$\mathcal{F}^U = \Big\{ C \setminus \Big\{ \neg l \mid l \in \hat{U} \Big\} \mid C \in \mathcal{F} \text{ and } C \cap \hat{U} = \emptyset \Big\}. \tag{6.1}$$

The key property of this simplification is that the formulas $\mathcal{F} \cup U$ and $\mathcal{F}^U \cup U$ are logically equivalent. In particular, we will use the property that if $\mathcal{F}$ is a formula, $\mathcal{C}$ is a set of clauses that are logical consequences of $\mathcal{F}$, and $U = \mathrm{Prop}(\mathcal{F} \cup \mathcal{C})$, then $\mathcal{F}$ and $\mathcal{F}^U \cup \mathcal{C}^U \cup U$ are logically equivalent.

Example 3 *Given a formula* $\mathcal{F} = (x) \wedge (\neg x \vee \neg y) \wedge (y \vee \neg x \vee z) \wedge (x \vee v) \wedge (y \vee v \vee \neg w)$, *the set of unit clauses implied by unit propagation is* $\mathrm{Prop}(\mathcal{F}) = \{(x), (\neg y), (z)\}$, *and simplifying* $\mathcal{F}$ *with this set results in* $\mathcal{F}^{\mathrm{Prop}(\mathcal{F})} = (v \vee \neg w)$.

6.3 The CL-SDSAT Framework

The basic idea of the proposed CL-SDSAT approach is to exploit the master-worker architecture described in Chapt. 4 so that the most CPU intensive work is performed in the workers, while the master computes less expensive work based on the information produced by the workers. A *master process* submits *jobs* consisting of a randomized SAT solver $\mathcal{S}$ and the SAT instance $\mathcal{F}$ to be solved into a grid-like distributed environment DE(r), which consists of r computing elements (CEs) performing computations dictated by the jobs. Each job occupies a CE for a time depending on the background load of the DE(r), the properties of the job, and the *resource limits* of the job which determine the amount of CPU time and memory the job can use on a CE.

If a job solves the problem (that is, the satisfiability of $\mathcal{F}$ is decided) within the resource limits, the whole algorithm terminates with the solution. If the solution is not found in the job, some of the clauses the solver has learned during its search are transferred back to the master process. The master process maintains a database of such clauses, and whenever a new job is submitted, a subset of the clauses in the current database is conjuncted with $\mathcal{F}$ in the submitted SAT instance.

Given a clause database at a particular time, the jobs which are constructed by conjoining $\mathcal{F}$ with some of the clauses in the clause database are called *subsequent* to that clause database. Conversely, a job *precedes* a given clause database if the learned clauses of the job have at some point been previously included to the clause database. The CL-SDSAT algorithm aims at pruning the search space of subsequent jobs by using the learned clauses of preceding jobs. The jobs are are randomized, and therefore the clauses returned by jobs subsequent to the same clause database usually differ. The following explains the proposed approach in more detail; the next sections then study different aspects of the approach using controlled experiments.

The framework for the approach is presented in pseudo-code in Fig. 6.2. The learned clauses are collected to an initially empty database of clauses, denoted by *ClauseDB*. The database is allowed to vary in size and its current maximum size is imposed by the variable *MaxDBSize*. From this database, a subset of size at most *SubmSZ* is provided to each solver instance $\mathcal{S}$ together with the original SAT instance $\mathcal{F}$. The unit clauses U are stored separately. The set is used for simplification (see Eq. (6.1)) of the clause database.

The main loop of the framework consists of two concurrent tasks (for clarity described as a sequential algorithm): submitting subsequent jobs to idle CEs in the distributed environment and receiving the results of the finished jobs. The submitting of subsequent jobs is described on lines 2–4. If there are idle CEs in the environment (line 2), then a job $\langle \mathcal{S}, \mathcal{F} \cup U \cup \text{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ}) \rangle$ is submitted to it (line 4). The function Choose selects a subset of the clauses in the current database *ClauseDB* so that the size of the subset is at most *SubmSZ*.[2] The size of the subset is restricted for two reasons: transferring data in a widely distributed environment takes non-negligible time and having an excessive amount of learned clauses can slow down the inner loop of the SAT solver. In order to keep the approach complete (that is, the approach is eventually able to determine the satisfiability of any SAT instance), the size limit *SubmSZ* may have to be increased during the search (line 3); this issue is discussed at the end of this section.

The results received from the DE are handled on lines 5–11 with two cases.

[2]The size of a set of clauses means here and in the following the sum of the number of the literals in the clauses in the set.

```
Input: $\mathcal{F}$, a SAT instance; $\mathcal{S}$, a randomized SAT solver;
  DE, the environment containing CEs.

  let $\mathit{ClauseDB} = \emptyset$
  let $\mathit{MaxDBSize}$ be the initial size limit of the clause database
  let $\mathit{SubmSZ}$ be the initial size of the learned clauses submitted with the job
  let $U = \mathrm{Prop}(\mathcal{F})$
1 while (True):
2   if there are idle CEs in DE:
3     update $\mathit{SubmSZ}$
4     submit the job $\langle \mathcal{S}, \mathcal{F} \cup U \cup \mathrm{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ})\rangle$ to an idle CE
5   if $\langle \mathrm{result}, \mathcal{C}\rangle$ is received from DE:
6     if result is in $\{\mathrm{SAT}, \mathrm{UNSAT}\}$:
7       return result
8     else
9       update $\mathit{MaxDBSize}$
10      let $U = \mathrm{Prop}(\mathcal{F} \cup U \cup \mathit{ClauseDB} \cup \mathcal{C})$
11      let $\mathit{ClauseDB} = \mathrm{Merge}(U, \mathit{ClauseDB}, \mathcal{C}, \mathit{MaxDBSize})$
```

Figure 6.2: A general framework for CL-SDSAT

- If the result is either *SAT* or UNSAT, the algorithm terminates with that result (line 7). The correctness of the result in this case, that is, the soundness of the framework, follows directly from the properties of learned clauses: a SAT instance $\mathcal{F} \cup U \cup \mathrm{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ})$ submitted to a CE is satisfiable if and only if the original instance $\mathcal{F}$ is satisfiable because all the clauses in $U \cup \mathrm{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ})$ are (simplified) learned clauses and, thus, logical consequences of $\mathcal{F}$.

- If a job is unsuccessful, the clause database *ClauseDB* is updated with the set $\mathcal{C}$ of learned clauses returned from the job (lines 10 and 11). The function Merge takes the sets U', *ClauseDB* and $\mathcal{C}$ and the size *MaxDBSize* as input. Here *ClauseDB* is the clause database and *MaxDBSize* is its maximum size, $\mathcal{C}$ are the learned clauses, and U' is the new set of unit clauses obtained by propagation from the SAT instance $\mathcal{F}$, old unit clauses U, *ClauseDB* and $\mathcal{C}$. The result of Merge is a new database $\mathit{ClauseDB}' \subseteq (\mathit{ClauseDB} \cup \mathcal{C})^U$ of size at most *MaxDBSize*.

When instantiating the framework into a concrete implementation, of special interest are the heuristics used in the two operators Choose and Merge. The next section analyzes key aspects of the operators.

Completeness. If the resource limits for the jobs as well as the size limits for the clause database (*MaxDBSize*) and submitted learned clause sets (*SubmSZ*) are fixed, the framework is not complete (that is, there are SAT instances for which the framework does not terminate). The range of problems solvable with the framework can be extended by increasing the parameters *MaxDBSize* and *SubmSZ* periodically during the search until a solution is found. Naturally, the Choose and Merge operators must use this increased space by returning clause sets of analogously increasing size. Observe the similarity to clause learning SAT solvers: they also usually increase the limit for the number of stored learned clauses gradually during the search. However, for very large problems and clause sets, it is of course possible that the resource limits are exceeded before a job is able to produce any new learned clauses; this results in incompleteness. In practice this seems not to be a concern.

6.4 Analyzing the Key Aspects of CL-SDSAT

This section studies empirically the key aspects of the parallel learning strategies in the CL-SDSAT algorithm in Fig. 6.2. The study is divided into three parts. Part A experiments on four heuristics for selecting learned clauses to a subsequent job. This is done in a controlled setting involving one *round of learning* where a number of independent jobs are run with a randomized SAT solver on the same SAT instance and the resulting learned clauses are used to construct a *derived instance.* The four heuristics are compared by studying the run time distribution of a SAT solver on a SAT instance and the corresponding derived instances where the learned clauses selected by the heuristics have been included. Part B studies how the run time distribution of the derived instance behaves as the number of CEs is increased and, hence, the number of independent jobs in the round grows. Part C studies the cumulative effect of learned clauses by increasing the number of rounds.

All the experiments of this section are conducted in a controlled environment without background load, using Intel Xeon 5130 2GHz CPUs with 16GB of memory.

The benchmark instances are selected from a representative set of benchmarks from the SAT 2007 Solver Competition, formed in three phases as follows. In the first phase, an initial set of problems was selected such that it consisted of the instances which were solved by MiniSAT in the 2007 competition, but required at least 2000 seconds for solving. This set consists of 53 problems. Second, each of these problems were solved 100 times using MiniSAT v1.14 using different seeds, and only those for which the minimum run time was more than 1000 seconds were qualified. This set consists of 28 instances, of which all but one are unsatisfiable. From the resulting set, a representative subset consisting of 17 instances was formed so that from each instance family there is only one representative. The instance families were identified based on the names of the instances. Table 6.1 reports the names together with the short labels used later in the experiments. The only satisfiable instance is labeled `cube`.

A. Heuristics for the Operator Choose. Central to the CL-SDSAT framework presented in Fig. 6.2 is the criterion for selecting clauses in line 4. In many of the sequential as well as distributed CDCL solver implementations which address the problem of selecting learned clauses, such as [78, 101], this criterion is the length of the clauses. In the actual implementation we will use a criterion which prefers the shortest learned clauses, and call this heuristic $\text{Choose}_{\text{len}}$. This approach is well justified in the CL-SDSAT framework: length based heuristic is efficiently implementable, and in

Table 6.1: Benchmark instances from SAT 2007 competition and their short labels

Name	Label
`AProVE07-09`	`AProVE`
`contest03-SGI_30_50_30_20_3-dir.sat05-440.reshuffled-07`	`contest`
`cube-11-h14-sat`	`cube`
`dated-10-11-u`	`dated`
`emptyroom-4-h21-unsat`	`emptyroom`
`eq.atree.braun.11.unsat`	`atree`
`hwb-n28-02-S818962541.sat05-492.reshuffled-07`	`hwb`
`linvrinv5.sat05-564.reshuffled-07`	`linvrinv`
`manol-pipe-f9b`	`manol`
`mod2c-3cage-unsat-10-2.sat05-2567.reshuffled-07`	`mod2c`
`pmg-12-UNSAT.sat05-3940.reshuffled-07`	`pmg`
`pyhala-braun-unsat-40-4-02.sat05-459.reshuffled-07`	`pyhala`
`QG7-gensys-ukn003.sat05-3346.reshuffled-07`	`GQ7`
`s101-100`	`s101-100`
`sortnet-6-ipc5-h11-unsat`	`sortnet`
`total-10-13-u`	`total`
`unsat-set-b-fclqcolor-10-07-09.sat05-1282.reshuffled-07`	`fclqcolor`

this form guarantees the progress of the search because new clauses are included to the database as long as the database size limitation is not exceeded.

It is possible to vision other types of heuristics as well, which could be based, for example, on the number of occurrences of certain clauses in all learned clauses obtained from the unsuccessful jobs. Such heuristics are less straightforward to implement. For example, the heuristic $\text{Choose}_{\text{freq}}$, preferring the most commonly learned clauses in the preceding jobs, would require centralizing all learned clauses to a single place. This approach does not scale well, as an excessive amount of memory is required when the number of clauses increases. Even more importantly, it is unclear how the historical frequency of a clause should be interpreted as new frequent clauses are included to the SAT instance to be solved. Nevertheless, we experiment with the heuristic $\text{Choose}_{\text{freq}}$ in a sufficiently simple framework to study how such a heuristic differs from $\text{Choose}_{\text{len}}$.

The heuristic $\text{Choose}_{\text{len}}$ is also contrasted to two other heuristics: Choose_{123} which only considers clauses of length at most three and $\text{Choose}_{\text{rand}}$ which selects random clauses from the clause database. The former only selects a subset of the clauses selected by $\text{Choose}_{\text{len}}$, and helps to study the effect of longer clauses when contrasted to $\text{Choose}_{\text{len}}$. Since Choose_{123} does not consider clauses longer than three, it cannot guarantee progress (and therefore, completeness) in instances where short learned clauses are rare. Therefore it is not generic enough for the purposes we are considering. The heuristic $\text{Choose}_{\text{rand}}$ is useful as a reference, as all learned clauses are already carefully selected by the solver and it is not a priori clear if heuristics are required to obtain better results.

We repeat the definitions of the four heuristics below.

- $\text{Choose}_{\text{len}}$ prefers short learned clauses within *ClauseDB*. Short clauses are potentially effective in pruning the search space.

- Choose$_{123}$ returns only clauses of at most three literals. Such clauses are even more effective in pruning the search space but might be rare in practice in some cases.
- Choose$_{\mathrm{freq}}$ prefers the most common learned clauses. Such clauses are intuitively good since they are encountered in many jobs. From equally frequent clauses, the shorter ones are preferred. It is not clear how this heuristic could be realized in the full CL-SDSAT framework.
- Choose$_{\mathrm{rand}}$ returns a set of clauses which are randomly picked from the set of learned clauses so that each learned clause is returned with equal probability.

In the experiments the clause database is simplified before the heuristics are used for selecting the set of clauses. This has several subtle consequences: the length of a clause in the clause database may decrease as a result of simplification and if two longer clauses reduce to the same short clause, the frequency of the short clause increases. Also none of the heuristics need to return unit clauses, as unit clauses are included into the submitted instance separately.

The aim of the first experiment is to study the effect of including learned clauses to benchmark SAT instances when doing one round of learning. A fixed number of SAT solvers produce a large candidate set of learned clauses for each benchmark. The candidate set acting as the clause database is then given to several different heuristics which produce the derived instances consisting of the benchmark SAT instance and a subset of the clause database. Finally the derived instance is solved using a SAT solver. All solvers used in the experiments are modified versions of MiniSAT v1.14 with the capability of accepting as input a seed for internal random number generator or, in addition to that, terminating the search at a given time and outputting the learned clauses held at that time.

In order to construct the clause database, the minimum run time of each benchmark is determined by solving it hundred times. The clause database is then constructed by running a number of solvers for one fourth of the minimum run time of the instance and collecting the learned clauses.

More formally, let $\mathcal{C}_1, \ldots, \mathcal{C}_r$ be the learned clauses returned by the solvers. Then let the clause database

$$\mathit{ClauseDB} = (\mathcal{C}_1 \cup \cdots \cup \mathcal{C}_r)^{\mathrm{Prop}(\mathcal{F} \cup \mathcal{C}_1 \cup \cdots \cup \mathcal{C}_r)}. \tag{6.2}$$

A derived instance is constructed by employing the respective Choose heuristic to select a subset of the clause database, that is, $\mathcal{F} \cup U \cup \mathrm{Choose}(\mathit{ClauseDB}, \mathit{SubmSZ})$ (line 4 of the algorithm in Fig. 6.2), with *SubmSZ* set to 100,000 literals and $r = 8$. The size limitation of *ClauseDB* is effectively ignored by setting *MaxDBSize* to infinity in these experiments.

Table 6.2 gives an overview of the results by comparing the expected run times over fifty runs of the derived instances when using the heuristics. For comparison, the table reports expected run times for the original instances without additional learned clauses (Base). The table also reports the expected number of decisions made by the

SAT solver, which measures the expected size of the search space for the instance, below the run time. The lowest of these numbers on each row is printed in bold face.

Based on these results, the expected number of decisions is lowest when using the length-based heuristic ($\text{Choose}_{\text{len}}$), and the expected run time is lowest when only clauses of length two and three are considered (Choose_{123}). If all clauses are considered in the length based heuristic, the run time of the instance can be high, being often higher than when no clauses are included. The experiment allows us to conclude that while length is an efficient criterion for selecting clauses, the inclusion of longer clauses results in more overhead than what is gained by reduction of the size of the search space. On the other hand, using frequent clauses ($\text{Choose}_{\text{freq}}$) results also in good speed-up in expected run times. These clauses are at least as long as the clauses preferred by $\text{Choose}_{\text{len}}$, suggesting that carefully selected long clauses can be used to speed up the solving. The results from $\text{Choose}_{\text{rand}}$ show also reduction in both expected number of decisions and run time. The clauses returned by MiniSAT seem to be good in reducing the number of decisions even when no particular heuristic is used in selecting the clauses. The comparison to other heuristics reveals though that an appropriate heuristic can significantly lower the expected run time of a derived instance.

The results on this benchmark set are surprisingly consistent: in 13 cases, $\text{Choose}_{\text{len}}$ results in lowest expected number of decisions, while in ten cases, Choose_{123} results in lowest expected run time. Using eight sources of learned clauses, in every case at least one heuristic succeeds in reducing the expected time required to solve the instance compared to Base.

Interestingly, $\text{Choose}_{\text{freq}}$ seems to perform well when compared to $\text{Choose}_{\text{len}}$. The relatively good performance might be an indication that an approach based on clause frequencies has potential. However, further studies are required to determine if this actually is the case. If so, more work is needed to obtain an efficiently implementable realistic approximation of the $\text{Choose}_{\text{freq}}$ heuristics.

B: The Effect of Increasing the Number of CEs. In many realistic scenarios, the number of CEs used in a computation can grow much higher than that used in Table 6.2. This corresponds to increase of r in Eq. (6.2), and should intuitively result in decrease of the expected run time for the derived instance.

This experiment confirms this intuition for the heuristic $\text{Choose}_{\text{len}}$ by showing the decrease of expected run time of a job subsequent to a clause database after a single round of learning with r preceding jobs as r is increased.

Starting from an empty database, one round of learning is performed by submitting r subsequent jobs. The jobs are guaranteed to be unsuccessful as the run time of each job is again limited to 25% of the previously measured minimum run time. The resulting clauses are merged to clause database, denote by $\mathit{ClauseDB}_r$. While performing these experiments, used sizes are $\mathit{MaxDBSize} = 10,000,000$ and $\mathit{SubmSZ} = 100,000$. As discussed above, the experiments use the $\text{Choose}_{\text{len}}$ heuristic. The benchmark instances are selected using the results of Table 6.2 as a criterion.

The run time distributions for instances `contest` is shown in Fig. 6.3, and for `manol` in Fig. 6.4, where the former seems to scale well using all heuristics, and the latter does not clearly benefit from any heuristic and is especially bad for $\text{Choose}_{\text{len}}$

Table 6.2: Expected run times for selected benchmarks from SAT 2007 competition

Name	Base	$\text{Choose}_{\text{len}}$	$\text{Choose}_{\text{freq}}$	Choose_{123}	$\text{Choose}_{\text{rand}}$
`AProVE`	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
`atree`	3,096	2,967	2,152	**1,439**	2,481
	22,311,255	**7,831,761**	13,263,105	9,034,391	14,404,941
`contest`	1,432	**70**	485	211	343
	1,240,001	**165,721**	541,943	357,978	467,458
`cube`	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`	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`	5,205	**1,498**	1,631	1,704	1,954
	1,885,355	**688,156**	813,027	853,642	1,052,777
`fclqcolor`	2,027	**1,153**	1,388	1,196	1,864
	41,172,989	**13,696,945**	29,946,390	25,945,033	26,103,961
`hwb`	4,654	14,128	5,001	**4,454**	10,211
	125,472,477	**68,950,042**	123,220,119	97,041,128	82,550,196
`linvrinv`	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`	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`	3,020	3,827	2,659	**2,496**	4,392
	271,766,780	**62,714,188**	221,568,484	195,269,018	87,430,751
`pmg`	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`	2,641	887	1,086	**782**	1,348
	2,775,304	**1,001,999**	1,855,329	1,436,653	2,245,269
`QG7`	1,594	760	1,196	**513**	1,506
	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`	4,886	1,521	2,893	**1,507**	4,694
	2,743,833	**900,265**	1,842,295	980,166	2,607,584
`total`	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

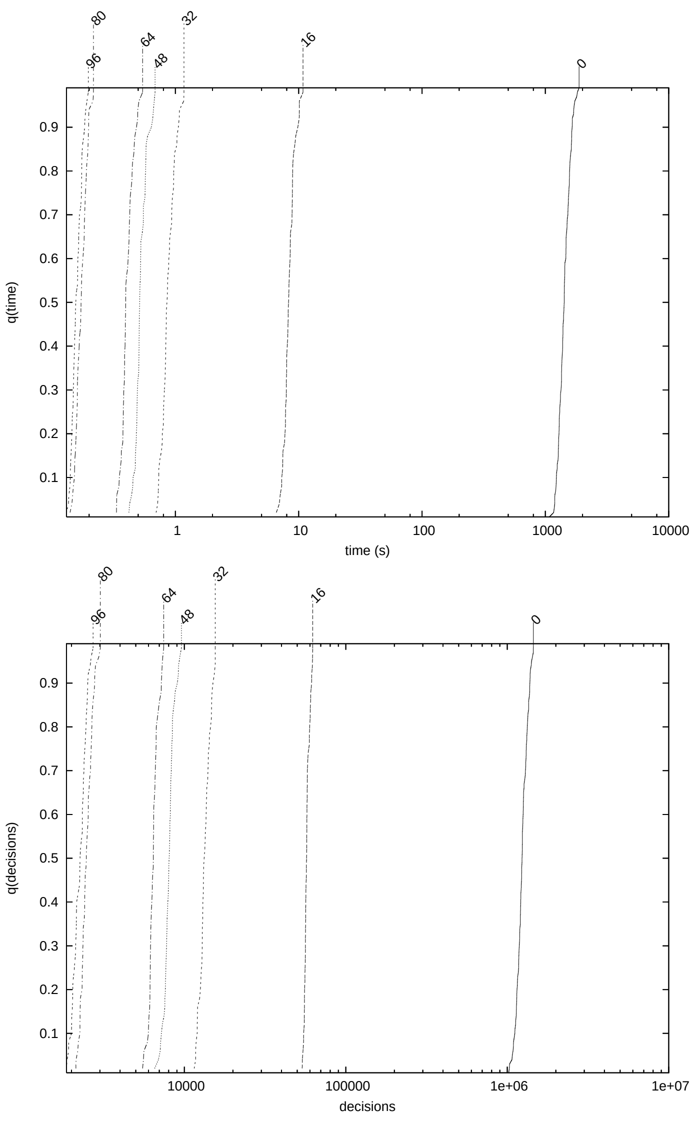
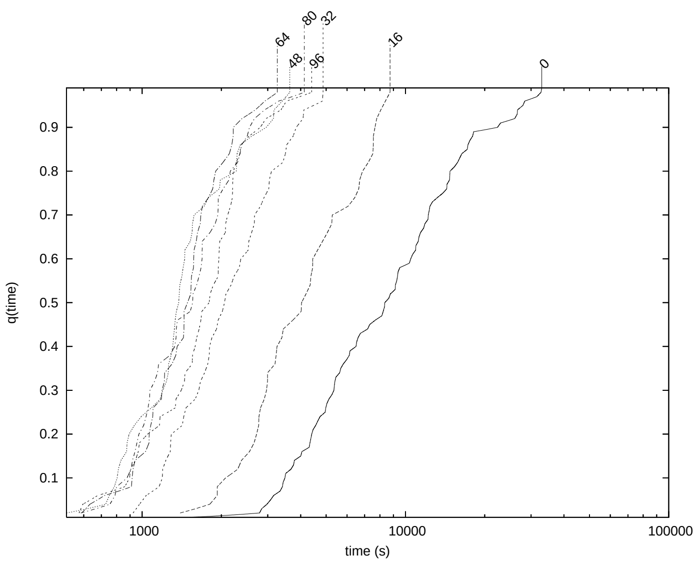


Figure 6.3: Distributions for run time and number of decisions for `contest` using different values of preceding jobs r and $\text{Choose}_{\text{len}}$

in Table 6.2. For comparison, the figures also give the distributions for the number of decisions. Table 6.3 provides statistics for these and two other instances.

In all of the experiments, the number of decisions and the run time decrease when the amount of preceding jobs increases sufficiently. The instance `contest` becomes easy to solve relatively soon. As the typical run time reaches values less than one second, it becomes difficult to see if the added clauses help in solving the problem further. Some of the instances show a slowdown in the decrease of the run time. The slowdown is well illustrated by Fig. 6.4 for `manol`. The expected run time for the instance first decreases gradually from almost three hours to slightly over 25 minutes reaching the minimum when the number of preceding jobs is 64. Increasing the amount of preceding jobs does not help to decrease the expected run time, which indeed seems to slightly increase, as the expected run time is almost five minutes higher when there are 96 preceding jobs.



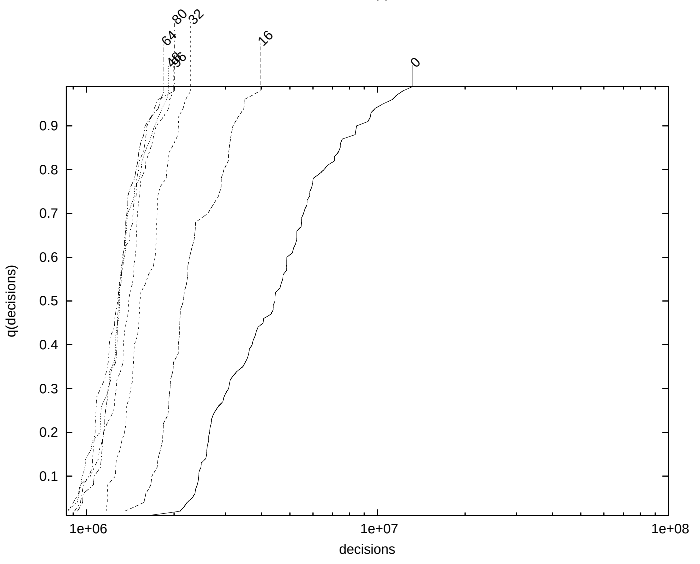


Figure 6.4: Distributions for run time and number of decisions for `manol` using different values of preceding jobs r and $\text{Choose}_{\text{len}}$

The other two instances in Table 6.3 show a nearly consistent decrease in all statistics. Interestingly, the expected run time of `hwb` in Table 6.3 reaches that of the original instance (see Table 6.2) only when there are more than 64 preceding jobs.

We note briefly that the expected total time required to solve, for example, the derived instance `manol` when $r = 48$ is 1,580 seconds. Even when the time required to obtain the derived instance is taken into consideration $(0.25 \times 1,510 + 1,580)$, the problem can be solved five times faster than the original instance.

However, these results show that there are instances such that after a single round of learning in CL-SDSAT, even the minimum run time of the derived instance does not become arbitrarily small no matter how much r is increased.

C: Cumulative Effect of Learned Clauses. As the previous experiment indicates, in many cases after a single round of learning the subsequent jobs do still

Table 6.3: Minimum, expected and maximum run times for different values of preceding jobs r and $\mathrm{Choose}_{\mathrm{len}}$

Label		0	16	32	48	64	80	96
`contest`	Min	1,080	6.58	0.70	0.53	0.41	0.17	0.16
	Exp	1,430	8.39	0.88	0.53	0.41	0.17	0.16
	Max	1,990	10.9	1.36	0.53	0.41	0.17	0.16
`fclqcolor`	Min	1,010	384	138	84.1	54.9	24.9	22.6
	Exp	2,030	595	281	161	102	55.9	51.1
	Max	3,650	1,450	529	297	187	120	162
`hwb`	Min	3,600	7,040	5,980	5,470	3,520	2,666	2,320
	Exp	4,650	8,770	7,880	6,710	4,680	3,390	3,000
	Max	6,190	11,300	9,990	9,045	6,590	4,780	4,100
`manol`	Min	1,510	1,394	924	516	589	584	576
	Exp	10,600	4,570	2,320	1,580	1,540	1,652	1,820
	Max	65,400	14,700	5,250	4,540	3,480	5,700	4,880

exceed the time limitation imposed by the DE. In the CL-SDSAT framework, the clauses are cumulated to overcome the problem. This means that not all jobs are subsequent to the same clause database, but the set of preceding jobs is allowed to grow arbitrarily as jobs are submitted. The effect is studied by continuing the previous experiment as follows. As previously, each clause database is assumed to include the unit literals. Let $ClauseDB_n^0$ denote the empty clause database, and denote by $ClauseDB_n^{i+1}$ the clause database after one round of learning from n SAT instances obtained from $ClauseDB_n^i$. The experiment studies the behavior of the job subsequent to $ClauseDB_n^i$ with fixed $n = 16$ as the number of rounds i increases. A resource limit is imposed on the subsequent jobs so that their run time is at most 25% of the experimental minimum run time of the original instance. The process terminates when the instance is solved within this time. Other parameters are as in the previous experiment.

Reported here are the results for the same problems as in the previous case, with the exception that `contest` is replaced by `total` since `contest` is easily solved in a job subsequent to $ClauseDB_{16}^1$. The results are shown in Fig. 6.5 and Table 6.4. We may compare the results illustrated for `manol` in Fig. 6.5 against those in Fig. 6.4 bearing in mind that each round corresponds to 16 jobs. When clauses are cumulated, the run times decrease at a consistent pace, as opposed to the slowdown illustrated in Fig. 6.4. Similar results are reported for several other instances in Table 6.4. For example, CL-SDSAT is able to overcome the increase in the run time of the derived instances of `hwb` relatively soon when the number of rounds increases.

The results support the hypothesis that hard instances with practical relevance can be made solvable within typical resource limits for individual jobs in realistic distributed environments considered in this work.

Table 6.4: Minimum, expected (Exp) and maximum run times for different number of rounds i in jobs subsequent to $\mathit{ClauseDB}^i_{16}, 0 \leq i \leq 4$ and $\text{Choose}_{\text{len}}$

Label		0	1	2	3	4
`fclqcolor`	Min	1,010	384	5.15		
	Exp	2,030	595	9.10		
	Max	3,650	1,450	30.4		
`hwb`	Min	3,600	7,040	4,520	1,060	2.77
	Exp	4,650	8,770	6,000	1,350	3.16
	Max	6,190	11,300	7,880	1,900	3.87
`manol`	Min	1,510	1,394	618	76.9	
	Exp	10,600	4,570	1,350	157	
	Max	65,400	14,700	3,350	313	
`total`	Min	1,190	892	370	6.19	
	Exp	3,280	1,480	568	7.01	
	Max	8,530	2,020	830	8.41	

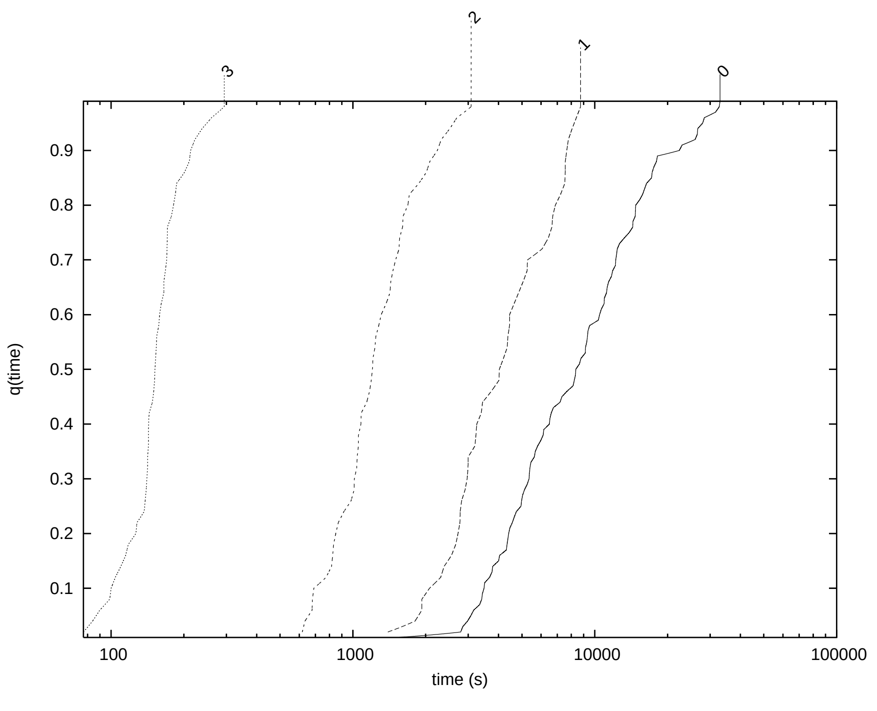


Figure 6.5: Run time distributions for jobs which are subsequent to $\mathit{ClauseDB}^0_{16}$, $\mathit{ClauseDB}^1_{16}$, $\mathit{ClauseDB}^2_{16}$ and $\mathit{ClauseDB}^3_{16}$ for `manol` and $\text{Choose}_{\text{len}}$

6.5 Grid Implementation

The ideas developed in this work have been implemented in a prototype of the proposed CL-SDSAT framework. The prototype uses NorduGrid (`http://www.nordugrid.org/`), a production level grid, as the distributed environment, and MiniSAT version 1.14 (with modifiable pseudo-random number generator seed) as the randomized SAT solver. The job management in the grid is handled by GridJM [51], and each job has a resource limit restricting the use of CPU time to one hour and the use of memory to one gigabyte. The implementation uses the heuristic $\text{Choose}_{\text{len}}$ preferring the shortest clauses for parallel learning; other heuristics discussed in Sect. 6.4 were not implemented in the prototype. Furthermore, the requirements needed for guaranteeing completeness are ignored by simply using a fixed clause database size of 1,000,000 literals. Similarly, unsuccessful jobs do not return all their learned clauses but only the shortest ones that together have at most 100,000 literals.

As the benchmark problems a set of hard SAT instances is selected such that there was little or no a priori information about the run time distribution. Such problems are available from the SAT 2007 solver competition (`http://www.satcompetition.org/`), where some of the instances were not solved by any of the competing solvers within the time bounds (10,000 seconds for the industrial and 5,000 seconds for the crafted category). Table 6.5 presents the results of running the CL-SDSAT prototype on a subset of these unsolved problems as well as on some other problems which were not solved by MiniSAT in the competition. Each instance was run for three days allowing the use of 64 CEs simultaneously. Column MiniSAT also reports the run times of the sequential MiniSAT v1.14 with no time limit but the memory usage restricted to two gigabytes. The runs were performed using an Intel Xeon 5130 2GHz CPU. It should be noted that the exact run times reported in the column Grid in the table are dependent on factors such as the background load of the grid environment and therefore difficult to reproduce.

Two phenomena can be observed from the results. Firstly, some problems, such as `vmpc_33`, are solved in less than one hour with the CL-SDSAT prototype and are, thus, clearly also solvable with the basic SDSAT method (see Chapt. 5) with no need for the learning-enhanced techniques of CL-SDSAT. Secondly, and more importantly, the prototype solves, with one hour time limit for each job, several problems which were not solved by *any* solver in SAT 2007 competition in 10,000 seconds. This suggests that the proposed CL-SDSAT framework also works for very hard problems and causes the run time distribution to “shift leftwards” (recalling Fig. 6.5 and the results from Table 6.4) as more learned clauses are seen. The other, much more unlikely explanation for this is that the problems have a very small but non-zero probability to be solved in less than one hour and, thus, would have been solved with the basic SDSAT method by using hundreds of parallel solvers.

Some of the instances were not solved in the grid within three days. The two instances the implementation was not able to solve suffered from the slow rate of change in the clause database. This in part was a result of the eventual high number of binary clauses in the clause database together with the property of the heuristic $\text{Choose}_{\text{len}}$ that it cannot differentiate clauses of the same length. When the clause database does not change, the subsequent jobs are similar to each other and the progress of the search

Table 6.5: Wall clock times for some difficult instances from SAT 2007 competition solved in grid and with standard MiniSAT v1.14. Memory outs are denoted by '*', time outs by '—'

Not solved by MiniSAT in SAT 2007			
Name	Type	Grid (s)	MiniSAT (s)
`ezfact64_5.sat05-452.reshuffled-07`	SAT	4,826	65,739
`vmpc_33`	SAT	669	184,928
`safe-50-h50-sat`	SAT	12,070	*
`connm-ue-csp-sat-n800-d-0.02-s1542454144-.sat05-533.reshuffled-07`	SAT	5,974	119,724
Not solved by any solver in SAT 2007			
Name	Type	Grid (s)	MiniSAT (s)
`AProVE07-01`	UNSAT	13,780	39,627
`AProVE07-25`	UNSAT	94,974	306,634
`QG7a-gensys-ukn002.sat05--3842.reshuffled-07`	UNSAT	8,260	127,801
`vmpc_34`	SAT	3,925	90,827
`safe-50-h49-unsat`		—	*
`partial-10-13-s.cnf`	SAT	7,960	*
`sortnet-8-ipc5-h19-sat`		—	*
`dated-10-17-u`	UNSAT	11,747	105,821
`eq.atree.braun.12.unsat`	UNSAT	9,072	59,229

is slow. This is of course a consequence of ignoring the completeness argument by not increasing the size of the clause database as the search progresses. Implementing this feature is an interesting direction of future work. We also note that it is often possible to simplify binary clauses with sophisticated techniques [8, 39]. However, experiments are required to determine whether such approaches are useful in this setting.

6.6 Remarks

This section has described a new approach to solving hard satisfiability problems in a distributed computing environment similar in properties to a typical grid. The approach can tolerate the severe restrictions imposed on the jobs executed in such an environment, e.g., it requires no inter-node communication and is inherently fault-tolerant. These restrictions correspond to the simulation environment described in Chapt. 4. The approach is based on combining (i) a natural method for solving SAT in parallel by independent randomized SAT solvers already considered in Chapt. 5, and (ii) the powerful conflict driven clause learning technique employed in many modern, sequential, CDCL SAT solvers described in Chapt. 2. This combination results in a novel parallel and cumulative clause learning approach which extends the capabilities of the previously discussed SDSAT framework. We have compared different heuristics for selecting learned clauses that are dynamically stored during the process, and demonstrated that the approach enables a form of clause learning that is not directly available in the underlying sequential clause learning SAT solver. Preliminary experimental results carried out in a production level grid indicate that the approach can indeed solve very hard SAT problems, including several that were not solved in the SAT 2007 competition by any solver, and one that we could not solve sequentially even without time limitations. This suggests that the developed algorithm is also useful in practical environments.

Chapter 7

Conclusions

This work considers logic programming and constraint-based search, which both have a long history in computer science and are actively studied in research communities as well as in the industry. The consideration in this work is in context of solving SAT instances and assuming a simultaneous access to a large quantity of computing resources. Specifically, the work concentrates on how distributed computing resources can be used to efficiently solve SAT instances in parallel, even though such resources unavoidably suffer from delays and even failures in completing the jobs assigned to them. In the work, it is generally assumed that the distributed resources are part of a computational grid environment and that the studied distributed solving techniques should be directly applicable in existing computational grids. Faithful to this assumption, the results are experimentally verified in a computational grid, the NorduGrid (`http://www.nordugrid.org/`).

The results support strongly the conclusion that many problems which are challenging for sequential SAT solvers can be solved in less time using a grid environment. This holds even though the execution in the environment is strongly restricted and the delays in communication are high. Furthermore, the results show that also instances which are not practically solvable using sequential SAT solvers can be solved using basically the same solvers and combining the results in a simple yet powerful way.

7.1 Distributed SAT Solving

This work identifies two aspects of SAT solving based on the taxonomy described in [17]: *distributed search* and *multi-search*, the former consisting of identifying and solving distinct partitions of the search space of a problem instance and the latter being based on running independent solvers on the same input instance with possibly some form of communication between the solvers and potential overlapping in the search spaces.

It is well-known that SAT instances are non-trivial to partition for efficient distributed search. Often the run time of the resulting partitions are highly imbalanced [103] resulting in reduced parallelism [15, 62]. On the other hand, multi-search, even with no communication, performs well on many SAT instances since SAT solver run times are inherently random and the randomness can be efficiently used in reducing the run time [74, 41].

There is some experimental evidence (for example, see [53]) that distributed search in context of SAT might have detrimental effect on the expected solving time of an instance since (i) an inefficient partitioning of the search space might result in instances which are essentially the same as the original instance and (ii) most distributed search methods expect that all partitions must be solved.

The results reported in this work show that the simple algorithmic framework, SDSAT, which is based on multi-search with no communication, provides surprisingly good speed-up in a grid environment. They also show that SDSAT can only be used to obtain significant speed-up for instances with run times that can be described by a certain type of distribution. Unfortunately, there are several, both satisfiable and unsatisfiable, difficult SAT instances which do not satisfy this condition and therefore SDSAT does not provide us with substantial speed-up when applied to these instances. The experiments allow us to conclude that while the run times of some instances vary significantly, there is in practice a minimum, non-negligible time required to solve many of the instances using the SDSAT framework. The grid environment discussed in this work places an upper bound on the computing resources a grid job can consume and therefore it seems there are instances which cannot be solved with any implementation of the SDSAT framework.

This limitation can be overcome with a simple yet powerful specialization of the SDSAT framework, called CL-SDSAT. The CL-SDSAT framework uses the property of some modern SAT solvers that as a side-effect of the solving, the solver produces *learned clauses* even though the solver is not able to find a solution to the instance. Such learned clauses can be combined and used to guide the searches of the subsequent SAT solvers. However, the task of combining the learned clauses is not straightforward and it is possible — in fact, quite common — that an instance containing learned clauses has a higher expected run time than the instance without the learned clauses. The effect is counter-balanced by the construction of more powerful learned clauses based on the previously obtained learned clauses. Gradually the instance is transformed using the learned clauses, so that solving within a given resource limit is possible.

The work reports the solving of instances not previously solved by any SAT solver, using an implementation of the CL-SDSAT framework. The results also show there are instances that cannot be solved with this implementation, and as the immediate reason for the failure suggest that the amount of learned clause information is limited in a very straightforward manner. The limit exists in order to avoid an excessive overhead which otherwise would result from the use of propagation. In many cases already this limit plays against the goal of reducing the time required to solve an instance. Hence, increasing the limit might not help to reduce the solving time, even though it seems to always decrease, for example, the amount of branching rule applications required to solve the instance. The work does not experimentally address this issue in the CL-SDSAT framework, but it is recognized as an interesting direction for future work.

7.2 Future Work

Excluding the anomalous super-linearity often observed in SAT solving, the maximum theoretical speed-up that can be obtained by parallelization is linear. This results from the fact that any parallel algorithm can be sequentialized with a linear overhead. Nevertheless, there are examples of instances which cannot be solved using a sequential solver but can be solved relatively fast using the CL-SDSAT framework and the same sequential solver. In the examples, the limiting factor seems to have been memory usage of the sequential solvers, and therefore it is doubtful if the approach could be sequentialized with only a linear overhead. The sequential version would involve transferring large amounts of learned clauses from memory to a medium with much higher latencies, e.g., a hard disk.

An explanation for the success of the CL-SDSAT framework in solving hard SAT problems is that — unlike sequential solvers — the memory used by the parallel solver is allowed to be distributed. Then the critical property of CL-SDSAT is its ability to compress the search it has performed into the information that is transmitted from the worker processes to the master process. The analysis on different approaches to filtering this information reveal that the length of the clauses is not the best possible criterion. The results presented in this work leave open certain questions, such as:

- Is it possible to efficiently implement filtering the learned clauses based on the frequencies they occur in a search.
- Could the simplification performed in the master process be efficiently implemented so that it would not be limited to using only unit clauses.
- In [16] the authors report good results on clustering related first-order lemmas based on some dependency. The search concentrates so that each distributed solver specializes on different subproblems, helping to improve the overall performance. Would such methods, when applied to learned clauses, be useful in the CL-SDSAT framework as well.
- Assuming that the amount of learned clause information is limited, it is evident that at some point the simplification will not be able to help progress the search by providing more space to the clause database, e.g., by using unit clauses. Would it be efficient to perform distributed search, by constructing new branches of the search, based on the information in the clause database. This approach could be seen as resorting to distributed search after multi-search has reached its limits.
- Could the methods described in this work be used to develop more efficient sequential algorithms?
- Would a more formal treatment of these presented methods result in more insight in how the efficient distributed solving should be performed?

The emphasis in this work is in multi-search. However, there are many open questions in distributed search as well; for example the relationship between guiding paths and scattering, shortly described in Chapt. 3, needs to be addressed with more mathematical rigor.

The grid environment and the corresponding simulation described in Chapt. 4 is still relatively simplistic. Issues related to scheduling of jobs in the grid are considered out of the scope of this work. However, there are still some issues related to the *interface* of the grid that should be further studied, and which could possibly help in efficiently implementing some of the ideas discussed in this section. For example, the information obtained by a SAT solver in a computing node usually consumes large amounts of memory and is therefore impractical to transfer to the master or even to store in the master. Therefore it might be useful for the master to query this information in order to obtain the most relevant information in the limited space that can be efficiently transmitted. Some grid environments do allow this type of interaction, but in the lack of experimental evaluation it is difficult to argue whether such features would be useful in grid-based SAT solving.

Bibliography

[1] *Proceedings of the 15th ACM/IEEE SC2003 Conference on High Performance Networking and Computing. Phoenix, AZ, November 15-21, 2003* (2003), IEEE Press. CD-ROM.

[2] *6th International Conference on Theory and Applications of Satisfiability Testing, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers* (2004), vol. 2919 of *Lecture Notes in Computer Science*, Springer-Verlag.

[3] *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing, SAT 2006. Seattle, WA, USA, August 12-15, 2006* (2006), vol. 4121 of *Lecture Notes in Computer Science*, Springer-Verlag.

[4] *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 07. Hyberabad, India, January 6-12, 2007* (2007). Online proceedings at `http://www.ijcai.org/proceedings07.php`.

[5] *Proceedings of the 22nd Conference on Artificial Intelligence, AAAI 2007. Vancouver, Canada, July 22-26, 2007* (2007), AAAI Press.

[6] Adjiman, P., Chatalic, P., Goasdoué, F., Rousset, M.-C., and Simon, L. Distributed reasoning in a peer-to-peer setting: Application to the semantic web. *Journal of Artificial Intelligence Research 25* (2006), 269–314.

[7] Ali, K. A. M., and Karlsson, R. Full Prolog and scheduling or-parallelism in Muse. *International Journal of Parallel Programming 19*, 6 (1990), 445–475.

[8] Bacchus, F., and Winter, J. Effective preprocessing with hyper-resolution and equality reduction. In *6th International Conference on Theory and Applications of Satisfiability Testing, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers* [2], pp. 341–355.

[9] Bal, H., and Verstoep, K. Large-scale parallel computing on grids. *Electronic Notes in Theoretical Computer Science 220* (2008), 3–17.

[10] Balduccini, M., Pontelli, E., Elkhatib, O., and Le, H. Issues in parallel execution of non-monotonic reasoning systems. *Parallel Computing 31*, 6 (2005), 608–647.

[11] Beame, P., Kautz, H. A., and Sabharwal, A. Towards understanding and harnessing the potential of clause learning. *Journal of Artificial Intelligence Research 22* (2004), 319–351.

[12] Ben-Eliyahu, R., and Dechter, R. Propositional semantics for disjunctive logic programs. *Annals of Mathematics and Artificial Intelligence 12*, 1-2 (1994), 53–87.

[13] Biere, A., and Kunz, W. SAT and ATPG: Boolean engines for formal hardware verification. In *Proceedings of the 20th IEEE/ACM International Conference on Computer Aided Design. San Jose, CA, November 10-14, 2002* (2002), Association for Computing Machinery, pp. 782–785.

[14] Blochinger, W., Sinz, C., and Küchlin, W. Parallel propositional satisfiability checking with distributed dynamic learning. *Parallel Computing 29*, 7 (2003), 969–994.

[15] Böhm, M., and Speckenmeyer, E. A fast parallel SAT-solver: Efficient workload balancing. *Annals of Mathematics and Artificial Intelligence 17*, 4-3 (1996), 381–400.

[16] Bonacina, M. P. Experiments with subdivision of search in distributed theorem proving. In *Proceedings of the 2nd International Symposium on Parallel Symbolic Computation. Maui, Hawaii, July 20 - 22, 1997* (1997), Association for Computing Machinery, pp. 88–100.

[17] Bonacina, M. P. A taxonomy of parallel strategies for deduction. *Annals of Mathematics and Artificial Intelligence 29*, 1-4 (2000), 223–257.

[18] Bordeaxu, L., Hamadi, Y., and Zhang, L. Propositional satisfiability and constraint programming: A comparative survey. *ACM Computing Surveys 38*, 4 (2006), 12–12.

[19] Bozzano, M., Bruttomesso, R., Cimatti, A., Junttila, T. A., van Rossum, P., Schulz, S., and Sebastiani, R. MathSAT: Tight integration of SAT and mathematical decision procedures. *Journal of Automated Reasoning 35*, 1-3 (2005), 265–293.

[20] Bryant, R. Graph-based algorithms for Boolean function manipulation. *IEEE Transactions on Computers 35*, 8 (1986), 677–691.

[21] Chrabakh, W., and Wolski, R. GridSAT: A chaff-based distributed SAT solver for the grid. In *Proceedings of the 15th ACM/IEEE SC2003 Conference on High Performance Networking and Computing. Phoenix, AZ, November 15-21, 2003* [1], pp. 37–37. CD-ROM.

[22] Cook, S. The complexity of theorem-proving procedures. In *Proceedings of the Third Annual ACM Symposium on Theory of Computing* (1971), Association for Computing Machinery, pp. 151–158.

[23] Darwiche, A. Decomposable negation normal form. *Journal of the ACM 48*, 4 (2001), 608–647.

[24] Davis, M., Logemann, G., and Loveland, D. A machine program for theorem proving. *Communications of the ACM 5*, 7 (1962), 394–397.

[25] Davis, M., and Putnam, H. A computing procedure for quantification theory. *Journal of the ACM 7*, 3 (1960), 201–215.

[26] Dimopoulos, Y., Nebel, B., and Koehler, J. Encoding planning problems in nonmonotonic logic programs. In *Recent Advances in AI Planning, Proceedings of the 4th European Conference on Planning, ECP'97. Toulouse, France, September 24 - 26, 1997* (1997), vol. 1348 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 169–181.

[27] Drescher, C., Gebser, M., Grote, T., Kaufmann, B., König, A., Ostrowski, M., and Schaub, T. Conflict-driven disjunctive answer set solving. In *Proceedings of the 11th International Conference on Principles of Knowledge Representation and Reasoning, KR 2008. Sydney, Australia, September 16-19, 2008* (2008), AAAI Press, pp. 422–432.

[28] Eén, N., and Sörensson, N. An extensible SAT-solver. In *6th International Conference on Theory and Applications of Satisfiability Testing, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers* [2], pp. 502–518.

[29] Elmroth, E., and Tordsson, J. An interoperable, standards-based grid resource broker and job submission service. In *Proceedings of the 1st IEEE Conference on e-Science and Grid Computing. December 5 - 8, 2005, Melbourne, Australia* (2005), IEEE Press, pp. 212–220.

[30] Erdem, E., and Türe, F. Efficient haplotype inference with answer set programming. In *Proceedings of the 23rd Conference on Artificial Intelligence, AAAI 2008. Chicago, Illinois, July 13-17, 2008* (2008), AAAI Press, pp. 436–441.

[31] Feldman, Y., Dershowitz, N., and Hanna, Z. Parallel multithreaded satisfiability solver: Design and implementation. *Electronic Notes in Theoretical Computer Science 128*, 3 (2005), 75–90.

[32] Finkel, R. A., and Manber, U. DIB - a distributed implementation of backtracking. *ACM Transactions on Programming Languages and Systems 9*, 2 (1987), 235–256.

[33] Finkel, R. A., Marek, V. W., Moore, N., and Truszczynski, M. Computing stable models in parallel. Tech. Rep. SS-01-01, AAAI, 2001.

[34] Forman, S., and Segre, A. NAGSAT: A randomized, complete, parallel solver for 3-SAT. In *5th International Symposium on the Theory and Applications of Satisfiability Testing, SAT 2002. Cincinnati, Ohio, May 6-9, 2002* (2002). Online proceedings at `http://gauss.ececs.uc.edu/Conferences/SAT2002/sat2002list.html`.

[35] Gagliolo, M., and Scmidhuber, J. Learning restart strategies. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 07. Hyberabad, India, January 6-12, 2007* [4], pp. 792–797. Online proceedings at `http://www.ijcai.org/proceedings07.php`.

[36] Ganai, M. K., Gupta, A., Yang, Z., and Ashar, P. Efficient distributed SAT and SAT-based distributed bounded model checking. In *Correct Hardware Design and Verification Methods. Proceedings of the 12th Advanced Research Working Conference on Correct Hardware Design and Verification Methods. L'Aquila, Italy, October 21 - 24, 2003* (2003), vol. 2860 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 334–347.

[37] Ganzinger, H., Hagen, G., Nieuwenhuis, R., Oliveras, A., and Tinelli, C. DPLL(T): Fast decision procedures. In *Proceedings of the 16th International Conference on Computer Aided Verification, CAV 2004. Boston, MA, July 13-17, 2004* (2004), vol. 3114 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 175–188.

[38] Gelfond, M., and Lifschitz, V. The stable model semantics for logic programming. In *Proceedings of the 5th International Conference and Symposium on Logic Programming, ICLP 88. Seattle, Washington, August 15-19, 1988* (1988), MIT Press, pp. 1070–1080.

[39] Gershman, R., and Strichman, O. Cost-effective hyper-resolution for preprocessing CNF formulas. In *Proceedings of the 8th International Conference on Theory and Applications of Satisfiability Testing, SAT 2005. St. Andrews, Scotland, June 19-23, 2005* (2005), vol. 3569 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 423–429.

[40] Gomes, C. P., Fernández, C., Selman, B., and Bessière, C. Statistical regimes across constrainedness regions. *Constraints 10* (2005), 317–337.

[41] Gomes, C. P., and Selman, B. Algorithm portfolios. *Artificial Intelligence 126*, 1-2 (2001), 43–62.

[42] Gomes, C. P., Selman, B., Crato, N., and Kautz, H. A. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning 24*, 1/2 (2000), 67–100.

[43] Gomes, C. P., Selman, B., and Kautz, H. A. Boosting combinatorial search through randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence, AAAI 1998. Madison, Wisconsin, July 26-30, 1998* (1998), AAAI Press, pp. 431–437.

[44] Gressmann, J., Janhunen, T., Mercer, R. E., Schaub, T., Thiele, S., and Tichy, R. Platypus: A platform for distributed answer set solving. In *Proceedings of the 8th International Conference on Logic Programming and Nonmonotonic Reasoning, LPNMR 2005. Diamante, Italy, September 5-8* (2005), vol. 3662 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 227–239.

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

[46] Herbstritt, M., and Becker, B. Conflict-based selection of branching rules. In *6th International Conference on Theory and Applications of Satisfiability Testing, SAT 2003. Santa Margherita Ligure, Italy, May 5-8, 2003, Selected Revised Papers* [2], pp. 441–451.

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

[48] Hooker, J. N., and Vinay, V. Branching rules for satisfiability. *Journal of Automated Reasoning 15*, 3 (1995), 359–383.

[49] Huang, J. The effect of restarts on the efficiency of clause learning. In *Proceedings of the 20th International Joint Conference on Artificial Intelligence, IJCAI 07. Hyberabad, India, January 6-12, 2007* [4], pp. 2318–2323. Online proceedings at `http://www.ijcai.org/proceedings07.php`.

[50] Huang, J. Universal Booleanization of constraint models. In *Proceedings of the 14th International Conference on Principles and Practice of Constraint Programming, CP08. Sydney, Australia, September 14-18, 2008* (2008), vol. 5202 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 144–158.

[51] Hyvärinen, A. E. J. GridJM. A Computer Program. `http://www.tcs.hut.fi/~aehyvari/gridjm/`.

[52] Hyvärinen, A. E. J. SATU: A system for distributed propositional satisfiability checking in computational grids. Research Report A100, TKK, Laboratory for Theoretical Computer Science, Espoo, Finland, February 2006.

[53] Hyvärinen, A. E. J., Junttila, T., and Niemelä, I. A distribution method for solving SAT in grids. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing, SAT 2006. Seattle, WA, USA, August 12-15, 2006* [3], pp. 430–435.

[54] Hyvärinen, A. E. J., Junttila, T., and Niemelä, I. Incorporating learning in grid-based randomized SAT solving. In *Proceedings of the 13th International Conference on Artificial Intelligence: Methodology, Systems, Applications, AIMSA 2008. Varna, Bulgaria, September 4-6, 2008* (2008), vol. 5253 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 247–261.

[55] Hyvärinen, A. E. J., Junttila, T., and Niemelä, I. Strategies for solving SAT in Grids by randomized search. In *Proceedings of the 9th International Conference on Artificial Intelligence and Symbolic Computation, AISC 2008. Birmingham, UK, July 31 - August 1 2008* (2008), vol. 5144 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 125–140.

[56] Hyvärinen, A. E. J., Junttila, T., and Niemelä, I. Incorporating clause learning in grid-based randomized SAT solving. *Journal on Satisfiability, Boolean Modeling and Computation* (January 2009). Submitted.

[57] Inoue, K., Soh, T., Ueda, S., Sasaura, Y., Banbara, M., and Tamura, N. A competitive and cooperative approach to propositional satisfiability. *Discrete Applied Mathematics 154*, 16 (2006), 2291–2306.

[58] Irgens, M., and Havens, W. S. On selection strategies for the DPLL algorithm. In *Proceedings of the 17th Conference of the Canadian Society for Computational Studies of Intelligence, Canadian AI 2004. London, Ontario, Canada, May 17-19, 2004* (2004), vol. 3060 of *Lecture Notes in Artificial Intelligence*, Springer-Verlag, pp. 277–291.

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

[60] Jensen, H. T., Kleist, J., and Leth, J. R. A framework for job management in the NorduGrid ARC middleware. In *European Grid Conference. Amsterdam, The Netherlands, February 14-16, 2005, Revised Selected Papers* (2005), vol. 3470 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 861–871.

[61] Jeroslow, R. G., and Wang, J. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence 1* (1990), 167–187.

[62] Jurkowiak, B., Li, C., and Utard, G. A parallelization scheme based on work stealing for a class of SAT solvers. *Journal of Automated Reasoning 34*, 1 (2005), 73–101.

[63] Kautz, H., and Selman, B. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence, ECAI 92. Vienna, Austria, August 3-7, 1992* (1992), John Wiley and Sons, pp. 359–363.

[64] Kautz, H. A., Horvitz, E., Ruan, Y., Gomes, C. P., and Selman, B. Dynamic restart policies. In *Proceedings of the 18th National Conference on Artificial Intelligence, AAAI 2002. Edmonton, Canada, July 28-August 1, 2002* (2002), AAAI Press, pp. 674–681.

[65] Krauter, K., Buyya, R., and Maheswaran, M. A taxonomy and survey of grid resource management systems for distributed computing. *Software - Practice and Experience 32*, 2 (2002), 135–164.

[66] Lagoudakis, M. G., and Littman, M. L. Learning to select branching rules in the DPLL procedure for satisfiability. *Electronic Notes in Discrete Mathematics 9* (2001), 344–359.

[67] Larrabee, T. Test pattern generation using Boolean satisfiability. *IEEE Transactions on Computer-Aided Design 11*, 1 (1992), 6–22.

[68] LE, H. V., AND PONTELLI, E. Dynamic scheduling in parallel answer set programming solvers. In *Proceedings of the 2007 Spring Simulation Multiconference, SpringSim 2007, Norfolk, Virginia, USA, March 25-29, 2007, Volume 2* (2007), Association for Computing Machinery, pp. 367–374.

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

[70] LEWIS, M. D. T., SCHUBERT, T., AND BECKER, B. Multithreaded SAT solving. In *Proceedings of the 12th Conference on Asia South Pacific Design Automation, ASP-DAC 2007. Yokohama, Japan, January 23-26, 2007* (2007), IEEE Press, pp. 926–931.

[71] LI, C. M., AND ANBULAGAN. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence, IJCAI 97. Nagoya, Japan, August 23-29, 1997, Volume 1* (1997), Morgan Kaufmann, pp. 366–371.

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

[73] LUBY, M., AND ERTEL, W. Optimal parallelization of Las Vegas algorithms. In *Proceedings of the 11th Annual Symposium on Theoretical Aspects of Computer Science, STACS 94. Caen, France, February 24 - 26, 1994* (1994), vol. 775 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 463–474.

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

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

[76] MANOLIOS, P., AND ZHANG, Y. Implementing survey propagation on graphics processing units. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing, SAT 2006. Seattle, WA, USA, August 12-15, 2006* [3], pp. 311–324.

[77] MARQUES-SILVA, J. Model checking with Boolean satisfiability. *Journal of Algorithms 63*, 1-3 (2008), 3–16.

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

[79] MIRONOV, I., AND ZHANG, L. 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. Seattle, WA, USA, August 12-15, 2006* [3], pp. 102–115.

[80] Moskewicz, M. W., Madigan, C. F., Zhao, Y., Zhang, L., and Malik, S. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 38th Design Automation Conference, DAC 2001. Las Vegas, NV, June 18-22, 2001* (2001), Association for Computing Machinery, pp. 530–535.

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

[82] Nieuwenhuis, R., Oliveras, A., and Tinelli, C. Solving SAT and SAT modulo theories: From an abstract Davis-Putnam-Logemann-Loveland procedure to DPLL(T). *Journal of the ACM 53*, 6 (2006), 937–977.

[83] Papadimitriou, C. H. *Computational Complexity.* Addison-Wesley, Boston, MA, 1994.

[84] Petrik, M., and Zilberstein, S. Learning parallel portfolios of algorithms. *Annals of Mathematics and Artificial Intelligence 48*, 1-2 (2006), 85–106.

[85] Pitkanen, M. J., Zhou, X., Hyvärinen, A. E., and Müller, H. 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. Jyväskylä, Finland, June 17-19, 2008* (2008), IEEE Press, pp. 367–372.

[86] Plaza, S., Kountanis, I., Andraus, Z., Bertacco, V., and Mudge, T. Advances and insights into parallel SAT solving. In *International Workshop on Logic Synthesis* (January 2006). Online version at `http://www.gigascale.org/pubs/1093.html`.

[87] Pontelli, E., and El-Khatib, O. Construction and optimization of a parallel engine for answer set programming. In *Proceedings of the 3rd International Symposium on Practical Aspects of Declarative Languages, PADL 2001. Las Vegas, Nevada, March 11-12, 2001* (2001), vol. 1990 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 288–303.

[88] Pontelli, E., Villaverde, K., Guo, H.-F., and Gupta, G. PALS: Efficient or-parallel execution of Prolog on Beowulf clusters. *Theory and Practice of Logic Programming 7*, 6 (2007), 633–695.

[89] Ranjan, D., Pontelli, E., and Gupta, G. On the complexity of or-parallelism. *New Generation Computing 17*, 3 (1999), 285–307.

[90] Rossi, F., van Beek, P., and Walsh, T., Eds. *Handbook of Constraint Programming.* Elsevier Science Publishers Ltd., Amsterdam, The Netherlands, 2006.

[91] Ruan, Y., Horvitz, E., and Kautz, H. A. Restart policies with dependence among runs: A dynamic programming approach. In *Proceedings of the*

8th International Conference on Principles and Practice of Constraint Programming, CP01. Paphos, Cyprus, November 26-December 1, 2001 (2002), vol. 2470 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 573–586.

[92] Schubert, T., Lewis, M., and Becker, B. PaMira — a parallel SAT solver with knowledge sharing. In *Proceedings of the 6th International Workshop on Microprocessor Test and Verification, MTV'05, Common Challenges and Solutions. Austin, Texas, November 3-4, 2005* (2005), IEEE Press, pp. 29–36.

[93] Sebastiani, R. Lazy satisfiability modulo theories. *Journal on Satisfiability, Boolean Modeling and Computation 3* (2007), 141–224.

[94] Segre, A. M., Forman, S. L., Resta, G., and Wildenberg, A. Nagging: A scalable fault-tolerant paradigm for distributed search. *Artificial Intelligence 140*, 1/2 (2002), 71–106.

[95] Selman, B., and Kautz, H. A. An empirical study of greedy local search for satisfiability testing. In *Proceedings of the 11th National Conference on Artificial Intelligence, AAAI 1993. Washington, DC, July 11-15, 1993* (1993), AAAI Press, pp. 46–51.

[96] Shapiro, E. Y., Warren, D. H. D., Fuchi, K., Kowalski, R. A., Furukawa, K., Ueda, K., Kahn, K. M., Chikayama, T., and Tick, E. The fifth generation project: Personal perspectives. *Communications of the ACM 36*, 3 (1993), 46–103.

[97] Silva, J. P. M. The impact of branching heuristics in propositional satisfiability algorithms. In *Progress in Artificial Intelligence. Proceedings of the 9th Portuguese Conference on Artificial Intelligence, EPIA'99. Évora, Portugal, September 21-24, 1999* (1999), vol. 1695 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 62–74.

[98] Simons, P., Niemelä, I., and Soininen, T. Extending and implementing the stable model semantics. *Artificial Intelligence 138*, 1-2 (2002), 181–234.

[99] Singer, D., and Monnet, A. JaCk-SAT: a new parallel scheme to solve the satisfiablity problem (SAT) based on join-and-check. In *7th International Conference on Parallel Processing and Applied Mathematics, PPAM 2007. Gdansk, Poland, September 9-12, 2007, Revised Selected Papers* (2007), vol. 4967 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 249–258.

[100] Singer, D., and Vagner, A. Parallel resolution of the satisfiability problem (SAT) with OpenMP and MPI. In *6th International Conference on Parallel Processing and Applied Mathematics, PPAM 2005. Poznan, Poland, September 11-14, 2005, Revised Selected Papers* (2006), vol. 3911 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 380–388.

[101] Sinz, C., Blochinger, W., and Küchlin, W. PaSAT — Parallel SAT-checking with lemma exchange: Implementation and applications. In *Proceedings of the LICS 2001 Workshop on Theory and Applications of Satisfiability*

Testing, SAT 2001. Co-located with the 16th Annual IEEE Symposium on Logic in Computer Science, Boston, Massachusetts, June 16-19, 2001 (2001), vol. 9 of *Electronic Notes in Discrete Mathematics*, Elsevier Science Publishers Ltd., pp. 12–13.

[102] Soininen, T., and Niemelä, I. 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 the 26th ACM SIGPLAN-SIGACT Symposium on Principles of Programming Languages, POPL 1999, San Antonio, Texas, January 18-19, 1999* (1999), vol. 1551 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 305–319.

[103] Speckenmeyer, E., Böhm, M., and Heusch, P. On the imbalance of distributions of solutions of CNF-formulas and its impact on satisfiability solvers. In *Satisfiability Problem: Theory and Applications. Proceedings of the DIMACS Workshop, March 11-13, 1996* (1997), vol. 35 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, American Mathematical Society, pp. 669–676.

[104] Sterling, L., and Shapiro, E. Y. *The Art of Prolog.* MIT Press, Cambridge, MA, 1987.

[105] Streeter, M., Golovin, D., and Smith, S. F. Combining multiple heuristics online. In *Proceedings of the 22nd Conference on Artificial Intelligence, AAAI 2007. Vancouver, Canada, July 22-26, 2007* [5], pp. 1197–1203.

[106] Streeter, M., Golovin, D., and Smith, S. F. Restart schedules for ensembles of problem instances. In *Proceedings of the 22nd Conference on Artificial Intelligence, AAAI 2007. Vancouver, Canada, July 22-26, 2007* [5], pp. 1204–1210.

[107] Tseitin, G. S. On the complexity of derivation in propositional calculus. In *Studies in Constructive Mahtematics and Mathematical Logic, Part II, Volume 8 of Seminars in Mathematics, V. A. Steklov Mathematical Institute* (Leningrad, 1969), Consultants Bureau. Translated from Russian. Reprinted in J. Siekmann and G. Wrightson, editors, Automation of Reasoning 2: Classical Papers on Computational Logic 1967-1970, pages 466-483. Springer, 1983.

[108] Walsh, T. Search in a small world. In *Proceedings of the 16th International Joint Conference on Artificial Intelligence, IJCAI 99. Stockholm, Sweden, July 31 - August 6, 1999* (1999), Morgan Kaufmann, pp. 1172–1177.

[109] Walsh, T. SAT v CSP. In *Proceedings of the 6th International Conference on Principles and Practice of Constraint Programming, CP00. Singapore, September 18-21, 2000* (2000), vol. 1894 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 441–456.

[110] Wu, H., and van Beek, P. On portfolios for backtracking search in the presence of deadlines. In *Proceedings of the 19th IEEE International Conference on Tools with Artificial Intelligence, ICTAI 2007. Patras, Greece, October 29-31, 2007* (2007), IEEE Press, pp. 231–238.

[111] Wu, H., and van Beek, P. On universal restart strategies for backtracking search. In *Proceedings of the 13th International Conference on Principles and Practice of Constraint Programming, CP07. Providence, RI, September 23-27, 2007* (2007), vol. 4741 of *Lecture Notes in Computer Science*, Springer-Verlag, pp. 681–695.

[112] Xu, L., Hutter, F., Hoos, H. H., and Leyton-Brown, K. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research 32* (2008), 565–606.

[113] Yang, L., Schopf, J. M., and Foster, I. T. Conservative scheduling: Using predicted variance to improve scheduling decisions in dynamic environments. In *Proceedings of the 15th ACM/IEEE SC2003 Conference on High Performance Networking and Computing. Phoenix, AZ, November 15-21, 2003* [1], p. 31. CD-ROM.

[114] Zhang, H. A complete random jump strategy with guiding paths. In *Proceedings of the 9th International Conference on Theory and Applications of Satisfiability Testing, SAT 2006. Seattle, WA, USA, August 12-15, 2006* [3], pp. 96–101.

[115] Zhang, H., Bonacina, M., and Hsiang, J. PSATO: A distributed propositional prover and its application to quasigroup problems. *Journal of Symbolic Computation 21*, 4 (1996), 543–560.

[116] Zhang, L., Madigan, C. F., Moskewicz, M. W., and Malik, S. Efficient conflict driven learning in boolean satisfiability solver. In *Proceedings of the ICCAD 2001 International Conference on Computer-Aided Design. San Jose, CA, November 4-8, 2001* (2001), Association for Computing Machinery, pp. 279–285.

[117] Zhao, Y., Malik, S., Moskewicz, M. W., and Madigan, C. F. Accelerating boolean satisfiability through application specific processing. In *Proceedings of the 14th International Symposium on Systems Synthesis, ISSS 2001. Montréal, Canada, September 30 - October 3, 2001* (2001), Association for Computing Machinery, pp. 244–249.