Universität Ulm
Institut für Theoretische Informatik
Leiter: Prof. Dr. Uwe Schöning

Engineering Stochastic Local Search for the Satisfiability Problem

Dissertation
zur Erlangung des Doktorgrades Dr. rer. nat.
der Fakultät für Ingenieurwissenschaften und Informatik der
Universität Ulm

vorgelegt von
Adrian Balint
aus Temeschburg

2013

Amtierende Dekanin:	Prof. Dr. Tina Seufert
Gutachter:	Prof. Dr. Uwe Schöning
	Prof. Dr. Jacobo Torán
	Prof. Dr. Armin Biere
Tag der Promotion:	19.12.2013

Summary

This thesis describes new algorithms for the Propositional Satisfiability Problem (SAT), a fundamental problem in theoretical and practical computer science. Given a formula in propositional logic (also called Boolean formula) over a set of Boolean variables, the SAT problem consists of answering the question whether an assignment to the variables exists, so that the formula evaluates to true. Besides the theoretical relevance of the SAT problem, many practical applications ranging from circuit verification to planning and scheduling corroborate the importance of the SAT problem.

There are several well established solving approaches for the SAT problem. Stochastic Local Search (SLS) is one of the most simple and elegant ones. Within this thesis, we provide different improvements for SLS solvers, and also propose new SLS solving techniques for the SAT problem. By means of empirical evaluations, we compare our solving methods with available state-of-the-art methods and show the superiority of the former. The results of our solvers within different SAT competitions further confirm their state-of-the-art performance.

First, we propose a new technique to analyze the search behavior of an SLS solver, which examines parts of the search trajectory of the SLS solver to detect missed solution. We show that this approach can be used to construct hybrid solvers using an arbitrary SLS solver as the main solver and a complete solver (*e.g.*, CDCL solver) as a sub-solver. The novel hybrid approach is implemented in three different solvers: *hybridGM*, *hybridGP* and *hybridPP*, which we analyze on different types of SAT problems and show their predominance over their SLS component.

We present a new type of heuristic for SLS solvers based on the concept of probability distributions. The new heuristic entails several desirable properties for SLS search heuristics. By replacing the *Novelty+* heuristic in the solver *gNovelty+* with our new heuristic, we create the solver *Sparrow*, which reaches state-of-the-art performance on a wide range of randomly generated SAT problems.

To improve the applicability of SLS solvers on structured problems, we analyze different preprocessing techniques in combination with the solver *Sparrow*. We are able to show that the performance of SLS solvers can be significantly improved on structured problems, when using appropriate preprocessing techniques.

Within our final study, we propose and analyze a solver based solely on probability distributions, which is the result of a dismantling process of the *Sparrow* solver. Our new solver, named *probSAT*, allows a detailed analysis of the role of *make* and *break* for SLS solvers. Within comprehensive evaluations, we analyze *probSAT* on different SAT problems, and show that it establishes new state-of-the-art standards.

Finally we present an advanced framework for the empirical evaluation of algorithms, named EDACC, which provides a plethora of functionalities for the design, execution and analysis of experiments with all kind of algorithms.

Zusammenfassung

Diese Arbeit beschreibt neue Algorithmen für das Erfüllbarkeitsproblem der Aussagenlogik (SAT), welches ein fundamentales Problem der theoretischen und angewandten Informatik ist. Gegeben sei eine Formel der Aussagenlogik (auch Boolesche Formel genannt) über eine Menge von Booleschen Variablen. Das SAT Problem besteht darin, zu beantworten, ob es eine Belegung für die Variablen gibt, sodass die Formel wahr ist. Neben seiner theoretischen Bedeutung hat das SAT Problem viele praktische Anwendungen, die von der Verifikation von digitalen Schaltkreisen bis in die Bereiche der Planung reichen.

Es gibt einige bewährte Lösungsansätze für das SAT Problem. Die auf Stochastischer Lokaler Suche (SLS) basierenden Methoden gehören zu den einfachsten und elegantesten. Innerhalb dieser Arbeit werden sowohl Verbesserungsansätze für bereits bekannte SLS Algorithmen als auch neu entwickelte SLS Algorithmen vorgestellt. Mithilfe von empirischen Evaluierungen werden diese Methoden und Algorithmen mit den aktuell besten verglichen und dabei wird gezeigt, dass die hier vorgestellten bessere Ergebnisse erziehlen. Die Überlegenheit wird auch durch die Ergebnisse in verschiedenen Wettbewerben (SAT Competitions) weiter untermauert.

Zunächst wird eine neue Methode vorgestellt, mit der die Qualität der Suche von SLS Algorithmen analysiert werden kann. Dieser Ansatz wird anschließend verwendet, um neue hybride Algorithmen zu entwickeln, die aus einem SLS Algorithmus und einem vollständigen Algorithmus (z.B. CDCL Algorithmus) bestehen. Drei neue auf diesem Ansatz basierende hybride Algorithmen: *hybridGM*, *hybridGP* und *hybridPP* werden auf verschiedenen SAT Problemen getestet. Dabei kann gezeigt werden, dass die hybriden Versionen in den meisten Fällen der SLS-Komponente alleine weit überlegen sind.

Des weiteren wird eine neue Heuristik für SLS Algorithmen vorgestellt, die auf Wahrscheinlichkeitsverteilungen basiert und einige Vorteile besitzt. Die neue Heuristik wird im Algorithmus *Sparrow* eingesetzt, der auf dem *gNovelty+* Algorithmus basiert. Durch empirische Evaluierungen wird gezeigt, dass die neue Heuristik für zufällig erzeugte SAT Probleme herausragende Performanz besitzt.

Um die Anwendbarkeit von *Sparrow* auch auf strukturierten SAT Problemen zu verbessern, werden als nächstes verschiedene Präprozessortechniken untersucht. Es kann gezeigt werden, dass die Anwendung von geeigneten Präprozessormethoden und ihre Parametrisierungen die Performanz von SLS Algorithmen auf strukturierten Problemen signifikant verbessert.

Motiviert von den guten Ergebnissen von *Sparrow* wird eine neue Klasse von SLS Algorithmen namens *probSAT* entwickelt und vorgestellt, die nur auf Wahrscheinlichkeitsverteilungen basiert. Mit Hilfe von *probSAT* kann auch die Rolle von *make* und *break* in SLS Algorithmen detailliert untersucht werden. Mit Hilfe von empirischen Evaluierungen kann gezeigt werden, dass *probSAT* auf zufällig erzeugten SAT Problemen neue Performanzmaßstäbe setzt.

Im letzten Teil dieser Arbeit wird EDACC vorgestellt, ein System für die empirische Evaluierung von Algorithmen. EDACC bietet eine Vielzahl von Funktionalitäten für den Entwurf, die Ausführung und die Analyse von Experimenten mit Algorithmen.

Table of Contents

Acknowledgments

I am very grateful to Prof. Dr. Uwe Schöning, for awaking my interest in the satisfiability problem already during my Diploma courses and for supporting my projects. Special thanks goes also to Prof. Dr. Armin Biere, for sharing his deep knowledge about SAT solving with me and giving me hints on how to write more efficient SAT solvers.

I thank my collaborators for sharing with me their interest in the SAT problem and in the experimental evaluation and analysis of algorithms. Thanks goes also to my former students. Without their coding skills and engagement, the EDACC system would still be a bunch of framework sketches in my note pad.

This work was partially supported through grants to Prof. Dr. Uwe Schöning by the German Research Foundation (DFG grant Scho302/9-1). I thank the Baden-Württemberg Grid (bwGRiD) project for providing the computational resources for performing the experiments within this thesis. Here I would like to specially thank the local coordinator in Ulm of the bwGRiD project, Christian Mosch, for his help.

I am very grateful to my parents, especially my father for awaking in me the interests in computer science and mathematics. Without our first 286 based computer, which he had built from spare parts, I would probably never be interested in computer science.

I would like to express my endless gratitude to my wife Elisabeth for being so patient during my Ph.D. time, taking care of the kids when I was working towards the next paper deadline, and for encouraging and supporting me whenever possible. Last but not least, I thank my three boys: Johannes, Christian and Jakob for prolongating my Ph.D. time; it was a good time.

1 Preface

1.1 Motivation

The Propositional Satisfiability Problem (SAT) aroused my interest during a course on Boolean functions held by Uwe Schöning. The simple formulation of the problem and at the same time its complexity is probably its most fascinating factor. The SAT problem has many practical and theoretical applications in the fields of computer science and electronics, making the problem even more interesting to solve. Advancing solving methods for the SAT problem also yields improvements for solving other related problems. Nowadays, it is often the case that researchers do not directly solve a combinatorial problem by writing a dedicated solver, but encode it to a SAT problem and then solve it with modern SAT solvers.

Stochastic Local Search (SLS) solvers, the key topic of this thesis, are probably the most lightweight and elegant solvers for the SAT problem. Their simple structure allows even a rigorous analysis of their run time, which was performed first in [Sch99]. Even though the complexity of SLS solvers can increase with the addition of different heuristics, overall they are all based on the very simple principle of local search. SLS solvers are the best known solving method for several classes of SAT problems like the uniform random class and several hard combinatorial problem classes like the computation of Van der Waerden numbers.

1.2 Goals and Contributions of this Thesis

The major goal of this thesis is to find better (practical) ways to solve different types of SAT problems by designing new state-of-the-art local search solvers. A deeper understanding of the inner working of SLS solvers and the development of new analysis methods is also an objective of this thesis. Equipped with a better comprehension of the solving techniques we hope to be also able to design more simple and efficient SLS solvers, being able to better understand the way SLS solvers work. Furthermore we want to advance the applicability of SLS solvers on structured problems, which is still a challenging task today, as it was ten years before [KS03, Challenge 6].

We have gradually reached our goals by starting with the analysis of an existing state-of-the-art SLS solver and modifying it step by step according to our needs. In more detail the contributions of this thesis are as follows:

1. Starting with the state-of-the-art solver *gNovelty+*, we have developed a new analysis method for the intensification of the search of local search solvers. By constructing partial assignments from the search trajectory of the SLS solver, we are able to check with a complete solver whether the SLS solver has missed near solutions during search. Our analysis approach can be easily transformed into an incomplete hybrid solver that is built on top of an SLS solver and uses a complete solver to verify the partial assignments. We can show that such hybrid solvers are in most cases faster than their single SLS component (Chapter 4).

2. By replacing the *Novelty* heuristic in the *gNovelty+* solver with a probability based decision heuristic, a new SLS solver named *Sparrow* was developed, which reached state-of-the-art performance on uniform randomly generated problems (Chapter 5).

3. By analyzing a wide range of preprocessing techniques in combination with SLS solvers, we were able to boost the performance of the *Sparrow* solver and of SLS solvers in general, on satisfiable hard combinatorial problems. We were also able to show that appropriate preprocessing techniques for SLS solvers are different from the ones used for complete solvers (Chapter 6).

4. Starting with the original *Sparrow* solver, we have gradually dismantled it by removing heuristics that were not strictly necessary to reach good performance. The outcome of this process is a very simple solver named *probSAT* that is based solely on probability distributions. *probSAT* sets new state-of-the-art standards for solving uniform random problems. With the help of automatic configuration methods and the *probSAT* solver we were able to show for the first time the different roles of the *make* and *break* values within SLS heuristics (Chapter 7).

5. As all our work is based on experimental analysis, we have also developed an advanced experimentation framework named *EDACC* that can accelerate the execution of experiments by using parallel systems like clusters or grids. *EDACC* was extended with a multitude of analysis tools that were used throughout this thesis and leveraged our findings. Supporting also the organization of competitions, *EDACC* was used for the SAT competitive events in 2012 and 2013 (Chapter 8).

All developed solvers within this thesis have been submitted to the SAT competitions where they reached state-of-the-art performance, always being placed among the best three top solvers within their category.

1.3 Statement of Collaborations

Most of the work presented in this dissertation is the result of collaboration with outstanding researchers or students that I have supervised in practical courses or in bachelor and master thesis. Many of the results presented in this work would probably not exist without their contributions for which I am very grateful.

The work presented in Chapter 4 is mainly based on work published in [BHG09] which was a collaboration with Oliver Gableske and Michael Henn. Oliver who was at that time supervised by me studied the construction of search space partitions and their possible usage in hybrid solvers in his Diploma thesis. He analyzed different construction methods and their usage. The construction described in Chapter 4 and published in [BHG09] is my main contribution and was partially implemented by Michael Henn in a practical course that he was doing in our department. The *hybridGM* solver was the result of further adaptation and tuning. The new solvers *hybridGP* and *hybridPP* are completely reimplemented from scratch.

Chapter 5 is mainly based on work published in [BF10] and is the result of joint work with Andreas Fröhlich who at that time analyzed local search solvers for his Diploma thesis under my supervision. Enhancing the flip heuristic of the solver *gNovelty+* with a probability based decision heuristic was proposed by Andreas after a series of intense discussion about the inner workings of SLS solvers in general. The optimization of *Sparrow*'s parameters was performed in collaboration with Holger Hoos and Dave Tompkins. The obtained parameter settings were published in [TBH11] and in [BFTH11] and are the basis of the results presented in Sections 5.3.2 and 5.3.3.

Chapter 6 is based on work published in [BM13a] which was done in collaboration with Norbert Manthey. Norbert implemented the preprocessor and described the techniques, while I designed and performed the experiments.

Chapter 7 is based on joint work with Uwe Schöning published in [BS12] and in [BS13].

The work presented in Chapter 8 is the result of a series of collaborations with Daniel Diepold, Simon Gerber, Daniel Gall, Robert Retz and Gregor Kapler. While I designed the main parts of Experiment Design and Administration for Computer Cluster (EDACC), most of the implementations of EDACC was part of their practical courses, their bachelor or master theses.

The work published in [TBH11] is joint work with Dave Tompkins and Holger Hoos. This work is not directly included in this thesis. Still, many insights gained within this work were used to transform the *Sparrow* solver into the much more simple *probSAT*.

Several other SAT competition related publications [BBD^{+}12, BBHJ13] which are partially included in this thesis have been published in collaboration with Matti Järvisalo, Anton Belov, Marijn Heule and Carsten Sinz.

2 Background

In this chapter we will briefly introduce the SAT problem. After presenting a brief history of the problem, we will describe the most common sources of SAT instances used for the evaluation of so called "SAT solvers". Randomly generated problems will be described in more detail. A brief overview of the most common solving techniques used in modern SAT solvers ends this chapter.

2.1 The Satisfiability Problem

Given a formula in propositional logic (also called Boolean formula) over a set of Boolean variables, the SAT problem consists in answering the question whether an assignment to the variables exists, such that the formula evaluates to true. This formulation of the problem is also known as the *decision version* because the possible answers are only "yes" or "no". The *model finding* formulation additionally requires to provide an assignment to the variables in case the formula is satisfiable. Within this thesis we concentrate on the model finding formulation.

The SAT problem was the first problem that was shown to be $\mathcal{NP}$-complete by Steven Cook in 1971 [Coo71]. This means that a solution can be verified very quickly (in polynomial time) and that every other problem from the class $\mathcal{NP}$ is reducible to SAT in polynomial time.

As long as $\mathcal{P} \neq \mathcal{NP}$, we do not expect to find polynomial bounded algorithms to solve the SAT problem (though there are special cases of the SAT problem that can be solved in polynomial time [AGRY09]). The $\mathcal{NP}$-completeness property entails that every problem from the complexity class $\mathcal{NP}$ can be efficiently (polynomially) transformed into a SAT problem. This property was often used in the field of computational complexity theory to prove that other problems like the vertex cover problem or the traveling salesman problem are also $\mathcal{NP}$-complete. As a consequence there are many known transformations from and to SAT.

Even if the SAT problem is a hard problem, and it might look hopeless to try to solve it, the algorithms to solve the SAT problem, also called “SAT solvers”, underwent a huge progress in the last decades. Formulas that where thought unreachable ten years ago, can now be solved within seconds by modern SAT solvers. A major driving force of this progress can be accounted to the International SAT Competition (SC) [1], which is organized biennially since 2002. The SAT Competition has a strong engineering effect on the SAT community: ideas that are presented at conferences or in journals (mostly in a more theoretical manner) are implemented in a SAT solver. If the ideas have practical relevance (*i.e.* they improve the performance of a SAT solver), then they are probably going to be also integrated in other solvers. As a result modern SAT solvers can be seen as a clever community engineered collection of solving techniques that together result in a powerful piece of software able to tackle down even hard SAT problems with millions of variables.

The $\mathcal{NP}$-completeness property is based on a worst case analysis, meaning that we do not know how hard the problem is to solve in a general case, but only in the worst case. It might be even possible that most of the problems are easy to solve, except for some special cases. This seems to be the case for the randomly generated colorability problems [Tur88].

Besides the theoretical importance, the SAT problem occurs in many practical applications like hardware and software verification, and also in fields like planning and scheduling. Even in the field of cryptanalysis, it was shown [EPV08] that some stream ciphers can be broken with the help of SAT solvers. The powerfulness of SAT solvers together with the possibility to encode many types of problems as SAT problems led to the usage of SAT solvers in many domains where hard combinatorial problems have to be solved.

2.1.1 Definitions and Notations

Formally, we denote Boolean formulas with F, G, H. Variables are represented with $x_1, x_2, \ldots, x_n$, where n denotes the number of variables of a formula F. The domain of the variables is the set of Boolean values: $\{\mathit{false}, \mathit{true}\}$ which we can also represent by the binary values $\{0, 1\}$. The family of SAT problems has no restriction on the type of logical operators that are used. A certain Boolean formula is also called *instance* (because it is an instantiation of the SAT problem). An example of a SAT formula is:

$$F = (x_1 \vee x_2) \wedge (x_1 \rightarrow (x_3 \wedge x_4)) \wedge (x_2 \rightarrow (\overline{x_3} \wedge \overline{x_4})) \tag{2.1}$$

Without loss of generality we can restrict the SAT problem to a special form, the so called Conjunctive Normal Form (CNF), which is the most common format used by SAT solvers and for benchmarks problems. From now on, when we refer to

[1] `www.satcompetition.org`

SAT, we actually mean CNF-SAT. A CNF formula is represented as a conjunction of disjunctions (*i.e.* AND of ORs). An example for a CNF formula is:

$$G = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \overline{x_3}) \wedge (\overline{x_1} \vee x_2 \vee x_3) \wedge (x_1 \vee \overline{x_2} \vee x_3) \tag{2.2}$$

Within a CNF formula, the conjunctive elements are called *clauses* and are denoted with A, B, C. An element of a clause is called a *literal* and is a variable or its complement. A clause that contains only one single literal is called *unit clause*. If the length of all clauses within a formula is bounded by some number k, then the formula is called a k-CNF formula or k-SAT within this thesis.

A CNF formula F can also be represented as a set of clauses and a clause as a set of literals. The formula G from 2.2 has the following set representation:

$$G = \{\{x_1, x_2, x_3\}, \{x_1, x_2, \overline{x_3}\}, \{\overline{x_1}, x_2, x_3\}, \{x_1, \overline{x_2}, x_3\}\} \tag{2.3}$$

The mathematical definition of a set does not allow duplicate elements within a set. This property is also valid in the set representation of a CNF formula, as duplicate literals can be eliminated from a clause, and duplicate clauses can be eliminated from a formula without changing the satisfiability property of the formula. The size of a problem can be measured in terms of variables, which is denoted by $|F|_v = n$, or in terms of clauses, which is denoted as $|F|_c = m$.

When a SAT solver tries to solve a problem, it works with *partial* or *complete assignments*. Assignments are denoted with α, β. A complete assignment α for a formula F assigns to every variable from F the values 0 or 1. A partial assignment, in turn, can assign values only to a subset of the variables from F. An example of a complete assignment for the formula G from 2.3 is: $\alpha = \{x_1 = 1, x_2 = 1, x_3 = 0\}$, which would also satisfy the formula. A partial assignment is $\beta = \{x_1 = 0, x_2 = 1\}$.

There are also other ways to represent assignments. One possibility is to specify the set of literals that have to be true (*i.e.* set to 1). The assignment α, mentioned before, can be represented as: $\alpha = \{x_1, x_2, \overline{x_3}\}$. If we assume that the ordering of the variables is well defined, then we can also represent a complete assignment as a binary vector. Assignment α from above is represented then by $\alpha = 110$. In case of partial assignments, we can introduce a special character $*$, which denotes variables not set to 0 or 1. Assignment β is represented by the vector $\beta = 01*$. The application of an assignment α on a formula F is denoted with $F|\alpha$. When applying a complete assignment to a formula, the result is 1 or 0, depending if the assignment satisfies F or not. On the other side, the application of a partial assignment results in a simplified formula, *i.e.* $F|\alpha = F'$. Of course there can be partial assignments that already render a formula to be 0 or 1. The application of an (unsatisfying) assignment α to the formula F splits the set of clauses into two sets: the set of unsatisfied clauses and the set of satisfied clauses.

The application of an assignment on a formula follows the rules of Boolean logic. If we set a literal within a clause to be true (*i.e.* 1), then the clause is satisfied and can be removed from the formula. If the formula is empty (*i.e.* no more clauses to satisfy), then the formula is satisfied. If the literal is set to false, then we can remove the literal from the clause. If, as a result, the clause is empty, then the clause can no longer be satisfied any more and the simplified formula is unsatisfiable.

Another important concept related to the structure of SAT problems is the neighborhood of a variable x_i, which is denoted by $N(x_i)$, and is defined as the set of variables that occur together with x_i in at least one clause.

Though all solvers presented in this thesis do not make direct use of the resolution calculus, several extensions and simplifications make use of it. Given two clauses A and B, where we have $l \in A$ and $\neg l \in B$ the resolution step notated with $A \otimes B$ produces a *resolvent clause* $C = (A \setminus \{l\}) \cup (B \setminus \neg\{l\})$.

The resolution operator can be overloaded by allowing a resolution between two sets of clauses S_1 and S_2, where all clauses from the former contain a literal l and all clauses from the latter contain $\bar{l}$. In this case, the set of resolvents S is defined as:

$$S = S_1 \otimes S_2 = \{A \otimes B | A \in S_1, B \in S_2\}$$

2.2 Types of SAT Instances

Since SAT solving is a practical domain, we need instances of the SAT problem to test different algorithmic approaches. SAT instances have a standardized format, which is also referred as the The Center for Discrete Mathematics and Theoretical Computer Science (DIMACS) format, which was first introduced in [DIM93]. SAT instances come from very different application domains and can have different generation methods. They can be loosely categorized in three main categories: *Application, Hard Combinatorial* and *Random*. This categorization was established by the organizers of the SAT Competition over the years, but there are controversies whether some instances should belong to one class or to the other. The categorization depends also on the encoding of the problem from its original specification language to CNF. Problems also tend to be categorized by the type of solvers by which they are solved best (*i.e.* if a solver which is strong for application problems can solve the problem better than a solver for crafted problems, most probably the problem is categorized as an application problem).

Instance Properties There is a huge amount of properties (also called features in the AI community) that can be defined and computed for SAT problems. SAT features are essential in the classification of SAT problems and in the prediction of the run time of SAT solvers on these. To our knowledge the most recent and extended list of SAT properties can be found in [HXHLB12, p. 19]. We are listing and explaining only a few of them that are simple to understand and will be often referred in this thesis.

1. n - number of variables
2. m - number of clauses
3. r - clause to variable ratio $r = \frac{m}{n}$ (also called density[2])
4. $|F|_l$ - number of clauses of length l (*e.g.*, $|F|_2$ denotes the number of binary clauses, $|F|_3$ denotes the number of ternary clauses)

SAT Instance Visualization The combinatorial properties of SAT problems can sometimes be better understood by visualizing the problem or even visualizing the solving process of the solver on the problem as shown in [SD05].

The two best known visualization types are the graph representation of the variable graph and the clause graph. In the clause graph, the nodes of a graph are represented by the clauses. Two vertices c_i and c_j are connected by an edge if they contain the same variable (or literal). In the variable graph representation, every variable x is represented by a vertex x. Two vertices x_i and x_j are connected by an edge, if the corresponding variables occur together in at least one clause. Since the number of clauses is in most cases considerably higher than the number of variables, the variable graph representation yields more compact visualizations. The neighborhood $N(x_i)$ of a variable x_i in the variable graph refers to the set of adjacent vertices of x_i.

2.2.1 Application Problems

Application instances normally come from fields where SAT solvers are "applied" to solve different combinatorial problems that are transformed to SAT problems. Two predominant classes of instances come from the field of hardware and software verification. Another class of problems comes from the domain of crypt-analysis, where different types of stream- and block-ciphers breaking problems are encoded as SAT problems. Additionally, the application domain also contains instances coming from bio-informatics, strip packing and diagnosis of systems. For an overview of a classification of problems the reader may refer to [BBD+12, p. 70].

Compared to other classes of instances, the application problems are generally very large in terms of the number of variables and the number of clauses. For example, the largest instance[3] used in one of the latest competitive events, the SC12[4], had over 13 million variables and more than 53 million clauses. The uncompressed size of the

[2]In the graph representation of SAT problems, the clause to variable ratio is highly correlated with the density of the variable graph

[3]Name of instance: esawn_uw3.debugged.cnf. Details can be found at `http://edacc2.informatik.uni-ulm.de/SATChallenge2012/instance/4833`

[4]`http://baldur.iti.kit.edu/SAT-Challenge-2012/index.html`

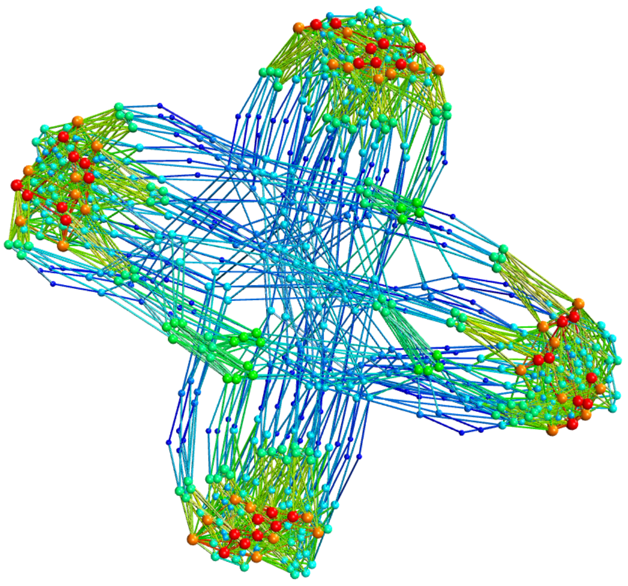


Figure 2.1: The variable graph representation of an application instance with 596 variables and 2780 clauses, which encodes a small scale variant of the block cipher AES. Nodes represent the variables. Two variables are connected by an edge if they occur in the same clause. The color of the nodes encodes the degree of the node (blue=low degree to red=high degree).

instances was 1.4 Gigabyte. Regardless of the size of the instance, almost half of the solvers participating at the competition solved the problem in less than 900 seconds. The application problems are known to be combinatorially very sparse, meaning that the number of decisions that a solver needs to solve the problem is relatively small when compared to the combinatorial search space that the problem could span. The number of binary clauses (clauses with only two literals) and the number of variables having few occurrences within an instance is relatively large.

Figure 2.1 shows a variable graph[5] for a small application instance[6] coming from the crypt-analysis domain [BBD+12, p. 74]. The color of the nodes and the edges represents the degree of the node and the degree of connected nodes, respectively (varying from blue to red). As we can see from the graph, the instance exhibits symmetrical structures and different patterns can be recognized.

[5]The visualizations have been created with *Graphinsight* `http://www.graphinsight.com/`.

[6]Name of the instance: *aes_64_1_keyfind_2.cnf*. Details can be found at: `http://edacc2.informatik.uni-ulm.de/SATChallenge2012/instance/4352`

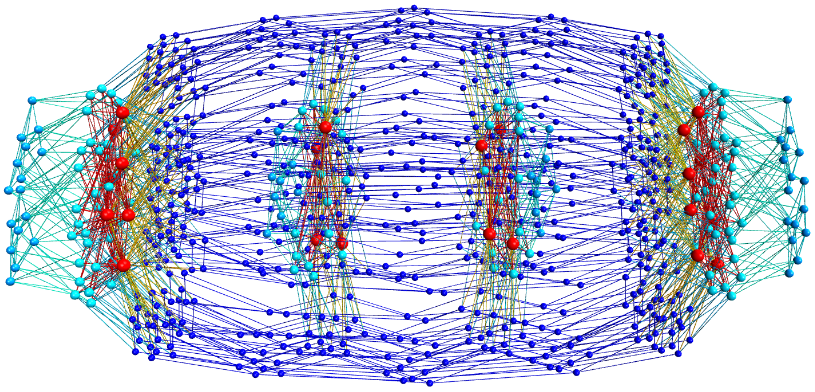


Figure 2.2: The variable graph representation of a hard combinatorial instance used in SC12 with 744 variables and 2464 clauses. See Figure 2.1 for interpretation help.

2.2.2 Hard Combinatorial Problems

The hard combinatorial problems are also called crafted, because they were “crafted” to be very difficult for certain types of solvers. Which of the names matches better is difficult to decide; probably both of them have their justification.

This class can be divided into two major sets. The first one contains the SAT encoded problems of different combinatorial games like Sudoku, Hidoku, parity games and pebbling games. The second set contains hard combinatorial mathematical problems like factorization, quasigroup, Ramsey cube, pigeon hole principle or the computation of Van der Waerden numbers. Problems that are thought to be extremely hard to solve, not because of their size but due to their combinatorial properties, are also classified within this class. A remarkable example is an instance[7] with only 95 variables that was not solved by any solver in the SC12. This instance was generated in 2003, and could be solved only recently by the parallel SAT solver *treengeling* [BBHJ13, p. 51]. This type of instance can be seen as a gentle reminder that we are dealing with a hard $\mathcal{NP}$-complete problem. Figure 2.2 shows the variable graph representation of a hard combinatorial problem[8] that encodes the question whether there exists a matrix multiplication involving only six multiplicative terms [LJPJ02]. This instance also exhibits obvious structural properties.

[7] Name of instance: *SGI_30_70_19_80_8-log.shuffled-as.sat03-143.cnf*. Details can be found at `http://edacc2.informatik.uni-ulm.de/SATChallenge2012/instance/279`

[8] Name of the instance: *contest02-Mat26.sat05-457.reshuffled-07.cnf*. Details can be found at: `http://edacc2.informatik.uni-ulm.de/SATChallenge2012/instance/1621`

2.2.3 Randomly Generated Problems

From a historical point of view, this class of problems is probably the oldest one. At the beginning of the SAT solving area it was very difficult to find appropriate benchmarks to test a SAT solver, forcing researchers to work with randomly generated problems tha follow different generation schemes. Since the collection of benchmarks like the ones in the before mentioned two classes was so successful in the last decade, the class of randomly generated problems lost importance. Nevertheless, it is one of the most challenging set of problems.

We are going to describe the class of randomly generated problems in more detail, as it is the class of problems that are solved best by the solving techniques presented in this thesis. The most prominent and also most studied class of randomly generated problems are the uniform randomly generated ones. The algorithm that generates this type of problems is listed in Algorithm 1.

Algorithm 1: Uniform Random k-SAT Generator

```
   Input  : $n$=number of variables, $m$=number of clauses, $k$=clause length
   Output: uniform random $k$-SAT instance
1  $F = \emptyset$ ;
2  $i = 0$;
3  while $i < m$ do
4      $C_i = \emptyset$;
5      $j = 0$ ;
6      while $j < k$ do
7          $v$ = variable index chosen uniformly at random from $\{1 \dots n\}$;
8          $l$ = literal chosen uniformly at random from $\{v, \overline{v}\}$;
9          if $(l \notin C_i) AND (\bar{l} \notin C_i)$ then
10             $C_i = C_i \vee l$ ;
11             $j$++;
12     if $(C_i \notin F)$ then
13         $F = F \wedge C_i$;
14         $i$++;
15 return $F$;
```

The input of the generator is the number of variables n, the number of clauses m and the clause length k. Note that the generator is only able to generate uniform k-CNF problems (*i.e.* all clauses have the same length k). In the main loop starting at line 3, clauses are being iteratively generated by selecting uniformly at random k distinct literals (this is guaranteed by the condition in line 9). To assure that exactly

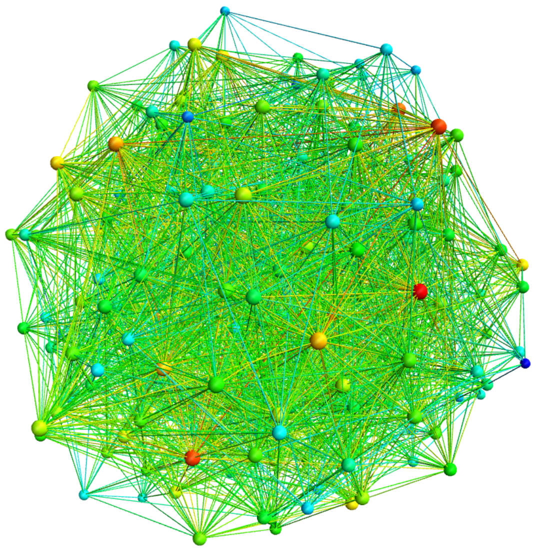


Figure 2.3: The variable graph representation of a randomly generated satisfiable 3SAT problem with 250 variables and 1065 clauses. See Figure 2.1 for interpretation help.

m distinct clauses are generated, the newly generated clause is added only if it is not part of the formula F yet (line 12). When generating instances of low density, a further test whether all variables have been used should be performed[9].

Figure 2.3 shows the variable graph of a uniform randomly generated SAT instance from the SATLIB benchmark set[10]. As can be seen from the visualization, no clear structure is recognizable.

Phase Transition Phenomenon

The class of random uniform fixed clause length mentioned before was studied heavily also because its hardness and satisfiability status are characterized by only one single parameter: the clause-to-variable ratio r [MSL92]. The probability of a random problem to be satisfiable is directly correlated with the clause-to-variable ratio of the problem. Below a critical value of r_k, which depends on k, almost all generated problems are satisfiable and above almost all problems are unsatisfiable. The sharpness of the threshold increases with the number of variables. The latest results analyzing this phenomenon and providing the most accurate values for the threshold values can be found in [MMZ06], and are listed in Table 2.1. These values are only valid when n is

[9]We thank Oliver Gableske for pointing out this problem.

[10]Name of instance: *uf250-0100.cnf* available at `http://www.cs.ubc.ca/~hoos/SATLIB/Benchmarks/SAT/RND3SAT/uf250-1065.tar.gz`

k	3	4	5	6	7
r_k	4.267	9.931	21.117	43.37	87.79

Table 2.1: Threshold values r_k for different values of k when n is large, computed with methods from statistical physics in [MMZ06].

"large" (*i.e.* when small-world effects can be neglected). This values have been also used in the latest Competitions from 2012 [BBD+12, p. 73] and 2013 [BBHJ13, p. 97].

Increasing the clause-to-variable ratio of a problem was shown to correlate with the difficulty of the problem. This phenomenon is also known as the "easy-hard-easy" pattern [MSL92], because for complete solving methods (that can prove satisfiability and unsatisfiability), problems get harder as the ratio increases, but get easier as soon as a given threshold value has been exceeded. For SLS methods, the problem are hardest around the threshold ratio. For random problems, it can be stated that the less constrained they are (*i.e.* the smaller the clause-to-variable ratio), the easier it is to find a satisfying assignment. On the other side, the more constrained a problem is, the simpler it is to show that it is unsatisfiable with complete methods [MSL92].

Categorizing Problems by their Satisfiability Status

A randomly generated problem around the phase transition has an unknown status, which in some cases can be a drawback (*e.g.*, when evaluating incomplete solving techniques). Now let us suppose that we are interested in the satisfiable instances, which are the target set of instances used throughout this thesis. There are several possibilities to categorize the instances into the set of satisfiable and unsatisfiable ones.

1. Use complete solvers to rule out unsatisfiable ones.
2. Use incomplete solvers to find the satisfiable ones.

Generally both approaches are very costly in terms of time and computational resources.

The first approach is strongly limiting the size of the generated problems, as complete solvers can only prove the unsatisfiability of randomly generated problems with a relatively small number of variables (*e.g.*, in case of 3-SAT instances, an efficient look-ahead solver needs several hours to solve a problem with 650 variables at a ratio of $r = 4.26$).

The second approach results in a biased set of instances towards the solvers used for filtering. If the generated problems are intended to be used within a competition or a solver evaluation, special care should be taken on how the selection is performed and which solvers are used for the filtering procedure.

To exemplify this problem in more detail let us assume that we generate a set I of n instances, and we are going to filter the satisfiable ones with s solvers. Now let us suppose that $n/2$ of the instances denoted by I_s can be solved by the s solvers and will be selected for an evaluation or a competition. A new solver (probably based on a different solving paradigm as our initial s solvers) that is able to solve the instances from the set $I \setminus I_s$ but not the ones from I_s will never be categorized as an efficient solver when evaluated on the filtered set I_s.

This scenario might look artificial, still this is possible in the random category of the SAT Competition. Randomly generated instances are being filtered with SLS solvers, while survey propagation solvers are not taken into consideration. The final set will thus contain problems for which SLS solving approaches work well.

Planted Solution Problems

Generating only satisfiable instances can be achieved by planting (also called hiding in the literature) one or even more predetermined solutions within the formula. Instances without hidden assignments are called *0-hidden*, while an instance with x hidden solutions is called x*-hidden*. Algorithm 1 can be easily changed to plant a solution α or a set of solutions A in the generation process. In line 12, we have to add the additional condition: clause C_i is going to be added to the formula only if it is not falsified under α or one of the assignments in A (in case multiple assignments are to be hidden).

Hiding only one single assignment without any further restriction on the type of added clauses gives rise to instances that are generally easier to solve than similar 0-hidden instances of the same size [AJM04]. This can be easily explained by the fact that the generated clauses point to the hidden solution [AJM04]. In case of uniform 3-SAT problems, if a variable x_i is set to $x_i = 0$ in the hidden assignment, then $\frac{4}{7}$ of the occurrences of variable x_i will occur as literal $\overline{x_i}$ and $\frac{3}{7}$ as literal x_i. This property can also be extended to the more general k-SAT problem accordingly. If the ratio of the formula is high enough (*i.e.* the formula has a high density), then we can use this bias to solve the problem by simply setting each variable to the value corresponding to the literal with maximal occurrence (*i.e.* set $x_i = 0$ if $\overline{x_i}$ occurs more often than x_i and $x_i = 1$ else). This type of algorithm is also called the democratic algorithm or majority voting algorithm and was shown in [BSBG02] to always find a valid assignment when the density of the formula (*i.e.* clause to variable ratio) equals $r = \Omega(log(n))$.

To generate satisfiable instances with planted solutions that yield hardness similar to that of 0-hidden instances several approaches have been proposed. Achlioptas proposed in [AJM04] to hide two assignments that are the opposite of each other (*i.e.* hide some randomly generated α and $\overline{\alpha}$), which resulted in instances more similar to the 0-hidden instances in terms of the hardness as well for SLS and for Conflict Driven Clause Learning (CDCL) solvers. This approach was used in [SB08] to generate satisfiable forced shape instances for non-clausal SLS solvers.

Another approach proposed in [BHL+01] is to hide only one single assignment, and alter the probabilities for the type of clauses that will be accepted, such that the number of occurrences of positive and negative literals of the same variable is balanced. Without loss of generality we can restrict the analysis by hiding the solution $\alpha = 1^n$ and considering only the 3-SAT case for the moment. Clauses can be categorized by the number of literals that are being satisfied under the planted solution. In case of 3-SAT we have seven possible clauses that have one, two or three satisfied literals in the planted solution. For each type of clause we assign the probabilities p_1 for 1-satisfied and accordingly p_2 and p_3. Due to probability normalization we have the equation:

$$3p_1 + 3p_2 + p_3 = 1 \tag{2.4}$$

For an arbitrary position in the clause we have to balance the probabilities of negative and positive occurrence of a variable. Without loss of generality, we restrict the discussion to the first position in the clause. The probability of a positive literal is the sum of probabilities of the 1-satisfied, 2-satisfied and 3-satisfied cases where the first position is positive (*i.e.* true). Within 1-satisfied clauses, only one single assignment has the first literal set to true: 100. For a 2-satisfied clause, there are two cases: 110 and 101. For a 3-satisfied clause, there is obviously only one single case: 111. The probability of a negative literal is also computed accordingly. The 1-satisfied clauses have two cases, where the first literal is false: 010 and 001. The 2-satisfied clauses have only one single case: 011. Setting the sum of these probabilities to be equal, we can add the additional equation:

$$p_1 + 2p_2 + p_3 = 2p_1 + p_2 \tag{2.5}$$

Solving the equation system containing 2.4 and 2.5, we get a parametrized probability distribution in p_3 of the form:

$$((1 + 2p_3)/6, (1 - 4p_3)/6, p_3)$$

With methods coming from the field of statistical mechanics, Barthel *et al.* [BHL+01] showed that for the generation of high quality instances p_3 should lie in the following interval:

$$0.077 \leq p_3 < 0.25$$

Another way to define the probability distribution of the different types of clauses was proposed in [JMS05]. Here a single parameter q is defined and a randomly generated clause that has t satisfied literals in the planted solution is accepted with a probability proportional to q^t. The authors call this model the q-hidden model. To generate balanced instances (positive and negative literal probability should be the same) the authors show that q should be set to $q = (\sqrt{5} - 1)/2 = 0.618$, which is the

k-SAT	3	4	5	6	7
q	0.6180	0.8392	0.9275	0.9659	0.9835

Table 2.2: The positive roots of Equation 2.8 for different values of k.

golden ratio. Adding to the Equations 2.4, 2.5 the additional equations:

$$p_2/p_1 = p_3/p_2 = q \tag{2.6}$$

and then solving this equation system, we will get the same value for q as mentioned above.

The value of q, which depends on k, can be computed as the positive root of the equation:

$$(1-q)(1+q)^{k-1} - 1 = 0 \tag{2.7}$$

To compute the values of p_t we can use the general form of the normalization Equation 2.4, which then results in:

$$p_t = \frac{q^t}{(1+q)^k - 1} \tag{2.8}$$

Table 2.2 shows the values of q for typical k-SAT. Smaller values for q will result in problems where local search algorithms are more likely to move away from the hidden solution then being attracted by it. This type of problems is also called *deceptive* formulas. A systematic construction of deceptive formulas that were shown to be very hard for local search solvers are also presented in [Hir00].

We can change the uniform k-SAT generator presented in Algorithm 1 to generate planted solution instances according to the q-hidden model by adding after line 12 an additional condition that checks if the clause is to be accepted or should be dropped depending on the planting model. After computing the number of true literals in the clause, the clause will be accepted according to an acceptance probability, which can be computed from the probability distribution $(p_1, \ldots, p_k)$ by normalizing the p-values with respect to p_1. This will result in an acceptance probability distribution of $(1, q, q^2, \ldots, q^{k-1})$ where the value of q can be found in Table 2.2.

Quality of planted solution problems

Planted solution problems were intended to be used in experiments where only satisfiable problems are of interest. They should not be distinguishable from 0-hidden problems of the same size and ratio. An intuitive quality measure for a planted solution model is to measure the difference in the run time distribution of solvers when executed on the planted solution problems and on the 0-hidden problems. This concept is similar to the one used in the quality measure of random number generators, where given two different sources of random numbers (where one of them is truly random), it should not be possible to distinguish between the two sources in polynomial time. In the field of cryptography this concept is called "computational indistinguishability" and was first introduced by Yao in [Yao82].

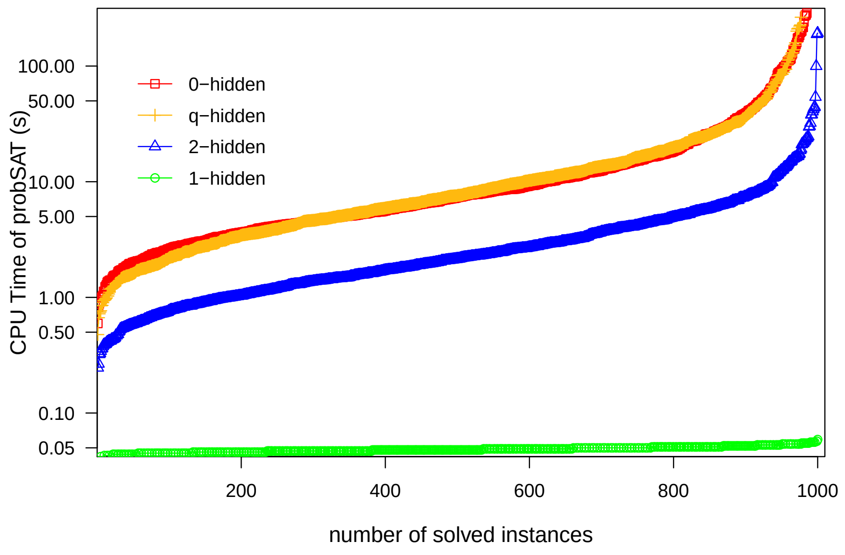
Within a small experiment we have evaluated the methods mentioned before to hide solutions for 3-SAT problems. We have generated 100 3-SAT problems with 10000 variables at a ratio of 4.2 (lower than the threshold) and a set of 100 problems with 1000 variables at a ratio of 4.267 (on threshold) according to the 0-hidden, 1-hidden, 2-hidden [AJM04] and q-hidden [JMS05] model. We have run the SLS solver *probSAT* (see Chapter 7 for details about this solver) ten times on each problem with a cutoff of 300 seconds, resulting in a total of 1000 evaluation points.

Figure 2.4a shows the run time distribution of *probSAT* on the four different instance sets. The 1-hidden set is by far the most simple one, and the run time of *probSAT* is always below 0.1 seconds. The 2-hidden problems are indeed much harder, though still significantly easier than the remaining two sets of 0-hidden and q-hidden problems, which do not exhibit large difference in the run time distributions. At a ratio of 4.2 and 10000 variables, these instances are expected to contain many solutions, so that hiding a solution to guarantee the satisfiability of the problems does not make much sense.

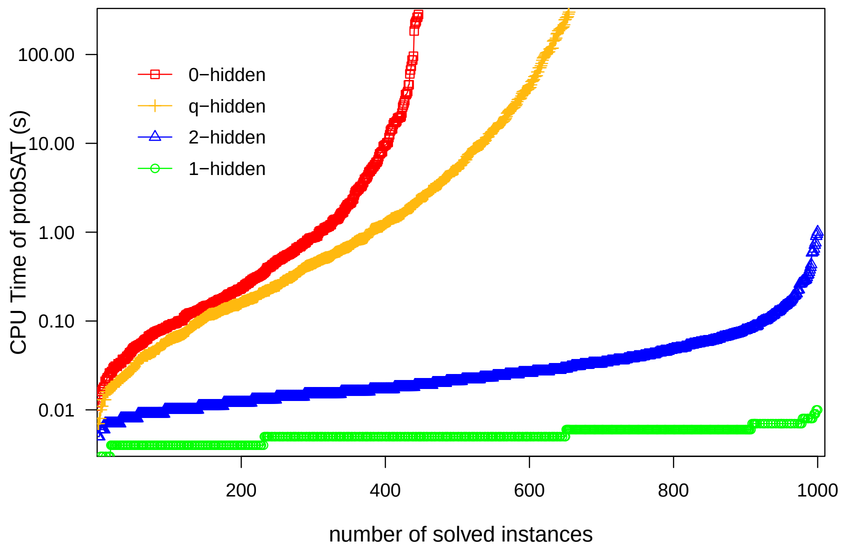
Instances generated at the phase transition where half of the instances are expected to be unsatisfiable constitute a more appropriate evaluation scenario. Figure 2.4b shows the run time distribution of *probSAT* on instances generated at the phase transition. As in the previous scenario the 1-hidden set is the simplest one followed by the 2-hidden set. This time there is a clear difference between the 0-hidden and the q-hidden problems. Within the time limit of 300 seconds almost half of the runs were successful (44.6%) in the 0-hidden model, while in the q-hidden model considerably more than half of the instances were solved successfully (65.5%). The difference in the run time distribution is expected to be proportional to the increase of the clause to variable ratio. We have also evaluated another SLS solver on these sets: the solver *WalkSAT* from the *UBCSAT* framework. The differences were consistent with the ones obtained with *probSAT*.

2.3 SAT Solving Techniques

SAT solving techniques can be first categorized by the type of answers they are able to provide. If a SAT-solver is able to find solutions *and* also to prove unsatisfiability, then the solver is called *complete*. If it is able to find only a solution, or only prove unsatisfiability, then it is called *incomplete*. Further we can also categorize SAT solvers by the way they perform search into *global* and *local* search solvers. These terms can



(a) 3-SAT with 10000 variables and $r = 4.2$



(b) 3-SAT with 1000 variables and $r = 4.26$

Figure 2.4: Run time distribution of the SLS solver *probSAT* on the four sets of 3-SAT problems, randomly generated according to the 0-hidden, 1-hidden, 2-hidden and q-hidden model. The x-axis represents the number of solved problems, while the y-axis represents the run time of the solver measured in seconds on a log scale.

be confusing, as local search solvers use a global state (*i.e.* they work on complete assignments), while global search algorithms like CDCL work with local states (*i.e.* they work with partial assignments).

2.3.1 Complete Methods

The two most prominent categories of complete SAT solving methods are the Conflict Driven Clause Learning (CDCL) solvers and the Look-Ahead solvers, which are both backtracking algorithms for the SAT problem. They are both based on the Davis-Putnam-Logemann-Loveland (DPLL) solving method presented in [DP60, DLL62].

The DPLL algorithm works on partial assignments. Starting with the empty assignment, the DPLL solver tries to extend its current consistent partial assignment to a complete consistent assignment by choosing a free variable, giving it a truth value and simplifying the formula. A (partial) assignment is consistent if it is not falsifying any of the clauses of the formula. If within this process the simplified formula gets unsatisfiable (an empty clause is detected), a backtracking step occurs, where the last decision is being undone, and the search continues on the other search branch, or backtracks further. If a backtracking to the root node occurs twice (*i.e.* both branches of the root variable do not contain a solution), the problem is unsatisfiable. If all clauses are satisfied by the current assignment, the solver returns this satisfying assignment. DPLL solvers are mainly depth-first search algorithms, though some solvers can perform a limited number of breadth search steps between depth search steps.

The simplification rules used by the original DPLL algorithm [DLL62] are only the propagation of unit clauses and the elimination of pure literals. Note that many degrees of freedom are present in the DPLL algorithm. Which unassigned variable should be picked next? Which value should be tried first? There has been a huge amount of research in the last decades to answer these question, and different heuristics have been proposed (see [BHvMW09, Chapter 4,5] for an overview).

Conflict Driven Clause Learning (CDCL)

This class of solvers is arguably the most important class of SAT solvers, since these solvers are used in daily practice to solve SAT problems occurring in different application domains. The main framework of CDCL solvers is based on the DPLL solver presented in [DLL62] which was extended with non-chronological backtracking [BS97], conflict-driven clause learning [MSS99] and heuristics for variable selection and phase selection (the selection of the truth values for a variable). During the search of a CDCL solver, the number of learned conflict-clauses can increase very fast, consequently CDCL solvers use different heuristics to reduce their conflict-clause database. Another major ingredient of modern CDCL solvers is the use of restarts. Instead of trying to extend the current partial assignment to a complete one, after a certain number of search steps, the solver will start from scratch with the empty assignment, while maintaining learned information (*e.g.*, the conflict-clauses).

Besides these algorithmic approaches, CDCL solvers also make use of highly efficient data structures (like the lazy data structures of watched literals) to speed up their search. Most of these data structures are designed such that unit propagation is performed as efficiently as possible. Unit propagation is the key operation performed by CDCL solvers. Another current major ingredient of CDCL solvers is the use of preprocessing and inprocessing (*i.e.* the use of simplification rules before and during search). The use of preprocessing can also boost other types of solvers too (see Chapter 6).

CDCL solvers are best suited to solve application problems, which is their main playground. Hard Combinatorial problems can also be tackled with these solvers. Generally, instances that have many binary clauses and few variables with many occurrences can be solved by CDCL solvers.

Look-Ahead

Look-Ahead solvers are also based on the DPLL algorithm. The key components of look-ahead solvers are the variable selection and value selection procedure. As their name already suggests, before taking decisions within these two procedures they perform a look-ahead by computing the possible reductions when a free variable is set to a certain Boolean value. The important part here is that when testing values of variables, necessary assignments can be detected, which are called failed literals (*i.e.* if setting l to true results in an unsatisfiable clause, then l is a failed literal and the literal $\bar{l}$ can be added to the formula). A look-ahead solver measures the importance of a variable x_i with a decision heuristic and then measures which branch of x_i to take with the direction heuristic. The heuristics are based on the reduction values computed during the look-ahead.

Besides these heuristics (which play the key role), other techniques like double look-ahead, local learning, back-jumping or autarky reasoning have been added to improve their performance. Look-ahead solvers are known to be efficient on unsatisfiable random problems and on small hard combinatorial problems. An overview of techniques employed in look-ahead solvers can be found in [BHvMW09, Chapter 5].

2.3.2 Incomplete Methods

The two major classes of incomplete methods are the local search solvers and the message passing solvers. Both methods are not able to prove the unsatisfiability of a problem. Though, there are some local search solvers designed to solve unsatisfiable problems [PL06].

Stochastic Local Search

As this type of algorithms is the major topic of this thesis, a thorough description of this algorithmic approach is given in Chapter 3.

Message Passing

Message Passing methods are derived from the field of statistical physics of disordered systems. These methods work on the factor graph representation [MZ02] of SAT formulas where variables and clauses are represented by vertices in a graph. Variable nodes are connected to all clauses where they appear, and clauses are connected to all variables they contain. Each variable has a value between $[-1, 1]$, which represents its tendency to take a certain polarity (*i.e.* false or true).

Message passing algorithms are based on the same principle: passing messages from variables to clauses, and from clauses to variables until messages converge (*i.e.* there is no significant change in their values). Once this happens, a decimation process starts to set the most polarized variables to their (converged) value and to simplify the problem by removing these variables. Then, the message passing procedure starts over again on the simplified formula until the formula is so simple that it can be easily solved by other faster approaches (usually a local search solver is used here).

The messages sent by variables to clauses can be interpreted as "I can satisfy you with probability ...", while the messages sent from clauses to variables (also called warnings) would mean "The probability that none of my variables will be satisfying me is ...". The major difference between the different algorithms forming this class is the message update procedure. The work presented in [Gab13] gives an overview on the different message update procedures and possible interpolations of these.

The survey propagation solver of [BMZ02, BMZ05] is the best known and also the most successful solver from this class. Survey propagation based solvers are able to solve very large random problems up to millions of variables in acceptable time (see Chapter 7 for an evaluation of the solver).

The underlying mathematics used in message passing algorithms have also been used for empirically determining the values of the threshold ratio of random k-SAT problems [MMZ06]. These values are still the best known values in the literature.

3 Stochastic Local Search

Compared to other search methods, Stochastic Local Search (SLS) is more closely related to the search that we perform in life for the best way of living. Like local search solvers, we are living in full state and try to improve our state of life by doing small local changes. It is often the case that we have to endure suboptimal states (like being a Ph.D. student) to reach a better state like earning a Ph.D. degree.

In this chapter, we will give an overview on the major SLS procedures proposed for SAT. There are plenty of other methods and heuristics used in SLS solvers. An overview of the older methods can be found in [HS05].

SLS solvers operate on complete assignments and try to find a solution by flipping variables (changing a variable x to $\overline{x}$). They can change only one variable per step or multiple variables (multi-flip), even though the latter is rarely used.

An SLS solver can also be viewed as an optimization procedure that tries to optimize an objective function. Since the SAT problem consists of finding an assignment such that all clauses are being satisfied, the objective is defined as minimizing the number of unsatisfied clauses or maximizing the number of satisfied clauses. From a mathematical point of view, a SAT problem can be formulated as an *Integer Linear Program.*

3.1 General Framework

The heuristics used by SLS solvers can differ considerably, however the main solving framework and data structures are the same and can easily be generalized. This has motivated the development of a unified SLS solver framework in which most known SLS solvers are implemented. The platform is called *UBCSAT*[1] [TH05] and was developed by Dave Tompkins at the University of British Columbia.

[1] `http://ubcsat.dtompkins.com/`

Algorithm 2: Generic SLS solver

```
Input  : Formula $F$, $maxTries$, $maxFlips$
Output: satisfying assignment $\alpha$ or UNKNOWN
1 for $i = 1$ to $maxTries$ do
2     $\alpha \leftarrow$ randomly generated assignment
3     for $j = 1$ to $maxFlips$ do
4         if ($\alpha$ is model for $F$) then
5             return $\alpha$
6         $var \leftarrow$ pickVar()
7         flip($var$)
8 return UNKNOWN;
```

Algorithm 2 presents the general solving procedure of SLS solvers. It has three types of inputs: the SAT formula and two limits, which specify how many times to try to solve the problem and how many flips to perform per try. The last two limits are needed to implement the concept of restarts, which at least from a theoretical point of view is very important. In practice, SLS solvers seldom use restarts. Almost all SLS solvers start with the generation of a random assignment. Then, they evaluate the assignment. If all clauses are satisfied, then the solution has been found. If not, one of the variables from the problem is selected according to a variable selection heuristic, which we further call *pickVar*. The value of this variable is flipped, the corresponding data structures are updated and the process starts over again with the selection of a new variable to flip.

To keep the description of the following SLS solvers as simple as possible, we will omit issues related to implementation details, like data structures or the methods to keep these updated. Nevertheless, we have to point out that these details are very important when it comes to the practical application of the solvers or the empirical evaluation and comparison (see Section 7.6 for discussion on this topic). Often, a solver can be improved by a factor of two only by using clever data structures, implementation tricks and by compiling the code with efficient compiler optimizations. All the SLS algorithms presented in the following differ only in the *pickVar* procedure that implements the selection heuristic.

Local Information used by SLS Solvers

Local search based solvers use within their *pickVar* method different types of information (also called properties or functions) or combinations of them, also called *evaluation functions* to guide the search. We will present the most common types of information. A more detailed list including some analysis can be found in [TBH11]. Given a formula F, a full assignment α and a variable x, we define the make value of x denoted with $make(x, F, \alpha)$ as the number of clauses that will be additionally satisfied when changing the value of x to $\overline{x}$. The *break* value $break(x, F, \alpha)$ is defined as the number of clauses that will get unsatisfied when flipping x. The score value is defined as $score(x, F, \alpha) = make(x, F, \alpha) - break(x, F, \alpha)$. Whenever the full assignment α and the formula F are obvious from the context, we will reduce the notation of the functions to only $make(x, \alpha)$, $break(x, \alpha)$, $score(x, \alpha)$ or even $make(x)$, $break(x)$, $score(x)$. If before a flip of x_i we have $break(x_i) = b_i$, $make(x_i) = m_i$ and $score(x_i) = s_i$, then after flipping x_i we will have $break(x_i) = m_i$, $make(x_i) = b_i$ and $score(x_i) = -s_i$. These types of properties are called *greedy*-properties or also *intensification* properties.

On the other side, properties related to the search history of a solver are called *diversification* properties. The age property $age(x)$, one of the most studied of this class, is defined as the number of flips since the last flip of x. In case x has never been flipped, then the $age(x)$ is defined as the number of flips since the beginning of the solving process.

The Probabilistically Approximate Complete Property

An important property of SLS solvers is the Probabilistically Approximate Complete (PAC) property [Hoo99]. An SLS solver has the PAC property if and only if the probability to reach a solution gets arbitrary close to one, when allowed to perform sufficiently many steps. This property implies certain behavior of the solver, like that the solver will not get trapped in cycles that do not contain the complete search space. This is actually the usual way to show that a certain solver does not have the PAC property: by providing a hopefully small example, where the solver gets trapped into a short cycle. Some examples can be found in [Hoo99]. On the other side, if a solver has a probability larger than zero in each step to perform a random step, then the solver will have the PAC property.

3.2 Uniform and Focused Random Walk

The probably most simple SLS solver, also called *Uniform Random Walk*, selects within the *pickVar* method one variable at random. A more elaborated version of this algorithm, called *Focused Random Walk* [Pap91], selects uniformly at random one unsatisfied clause first and then selects randomly a variable from this clause. Papadimitriou showed in [Pap91] that this algorithm can solve a satisfiable 2-CNF problem in $O(n^2)$ steps. Generally, all SLS solvers that pick variables from unsatisfied clauses can be considered focused SLS solvers, as they focus the search on the obvious trouble maker. Schöning presented in [Sch99] a variant of this algorithm that uses restarts after every $3n$ steps (n is the number of variables in the problem). He showed that this algorithm solves satisfiable 3-CNF problems in $O(1.334^n)$ steps, which is a remarkable theoretical performance for such a simple algorithm.

From a practical point of view, Schöning's algorithm has a relatively weak performance. For the price of losing the theoretical guarantee, a slight modification of this algorithm will yield state-of-the-art performance. The new algorithm called *probSAT*, which was developed within this thesis will be presented and analyzed in more detail in Chapter 7.

3.3 The WalkSAT Architecture

A further representative of the focused random walk algorithms is the *WalkSAT* solver by Selman *et al.* [SKC94]. After randomly picking an unsatisfied clause, *WalkSAT* picks randomly one of the variables from the clause that has *break* value of zero. In this case, due to the focused search, the *make* value of these variables is always at least one. If there is no variable with $break(x) = 0$, then with a probability of p a random variable is picked, else with probability $1 - p$ the best variable with respect to the *break* value is picked. The parameter p, also called noise, determines the ratio between random steps and greedy steps. The performance of *WalkSAT* is known to be highly dependent on this noise parameter [KSS10].

Another representative class of SLS solvers is the family of *Novelty* solvers [MSK97], which use within their *pickVar* method the *score* and the *age* property of variables. If the best variable is not the youngest (*i.e.* does not have minimal age), then this variable is picked. If the best variable has minimal age, then with a probability p the second best variable is selected, else with probability $1 - p$ the best variable is selected. Ties are always broken by the age of variables. Hoos showed that the *Novelty* solver does not have the PAC property, and introduced the solver *Novelty+* [Hoo99], which adds a random walk step to the *Novelty* solver. Different other improvements or extensions have been proposed for the *Novelty* solver: [MSK97, Hoo99, LH05, LWZ07].

The performance of the *Novelty* solvers is also highly dependent on the parameter p. Hoos proposed a self adapting scheme for the noise parameter in *Novelty* solvers [Hoo02] which he implemented in the *AdaptNovelty+* solver. The key observation of his study, which we will also confirm in Chapter 7, is that noise should be approximated starting with low values and should only be increased when a search stagnation is detected. If the objective function does not improve over $\theta \cdot m$ steps, the noise is increased with: $p' = p + (1 - p) \cdot \phi$. The noise is decreased with $p' = p - p \cdot \frac{\phi}{2}$ whenever an improvement in the objective function is detected. The objective function is the number of unsatisfied clauses. Hoos showed in experiments reported in [Hoo02] that the values of θ, ϕ can be fixed to $\theta = \frac{1}{6}$ and $\phi = \frac{1}{5}$. The *AdaptNovelty+* solvers have a stable performance on a wide range of problems, even if they can not reach the performance of the *Novelty+* solvers with optimal noise.

3.4 The GSAT Architecture

From a historical point of view, the *GSAT* architecture is probably the oldest class of SLS solvers and was first presented by Selman in [SLM92]. The name *GSAT* is motivated by the greedy approach the solver takes to solve a problem. The name would also match "global", which would also fit the description of these solvers, as they take all variables into consideration within the *pickVar* method. *GSAT* computes the *score* for all variables from the formula, and then selects from these the best one to flip. If more than one variable has the best *score*, then one of them is selected randomly. To keep the overhead of the *score* computation as low as possible, the *score* value of all variables is computed from scratch only at the beginning of the solving procedure. Afterwards, these values are only updated for variables where the *score* changes. To be more precise: if the variable x_i has been flipped by *GSAT*, then only the *score* value of variables occurring in $N(x_i)$ have to be updated.

The different variants of the G^2WSAT solvers, first presented by Li in [LH05] and then extended in [LWZ07, WLZ08], belong also to the class of greedy SAT solvers. These solvers are actually hybrids between *GSAT* and *WalkSAT* solvers because in a first step they use a restrictive variant of greedy steps and after reaching a local minimum *WalkSAT*-style steps are being used until the solver escapes from these. The key idea behind the G^2WSAT solvers is the variable property called *promising decreasing variable*, or for short only *promising*. A variable x_i is *promising* if $score(x_i) > 0$ and if this was not the result of flipping x_i when $score(x_i) < 0$ (which always renders the score of the variable to become positive). Variable x_i can get *promising* if during the search trajectory of the solver its score drops below zero and then gets again above zero as a consequence of flips of other variables, but not itself. The G^2WSAT solvers work as follows: as long as *promising* variables are present, the best one is picked (breaking ties by the *age* of variables, *i.e.* pick the oldest variable). If there are no *promising* variables, then G^2WSAT behaves similar to the *WalkSAT* variants. The G^2WSAT solvers actually differ only in the used *WalkSAT* variant, and their performance is indeed very sensitive to these heuristics.

3.5 The DLS Architecture

Dynamic Local Search (DLS) solvers can also be categorized to the class of greedy solvers, because they generally pick the best overall variable with respect to an alternative *score* value. Clauses are additionally augmented with weights (also called penalties in the literature), which can vary from clause to clause and are changing during search. The evaluation function used in DLS solvers takes these *clause weights* into account. If we define the weight of clause c_j as $clw(c_j)$, then the weighted score of a variable x_i from a formula F under assignment α can be defined as:

$$wscore(x_i) = score(x_i) + \sum_{C_j \in CU(F,\alpha)} clw(C_j) - \sum_{C_j \in CS(F,\alpha)} clw(C_j)$$

Here $CU(F, \alpha)$ denotes the unsatisfied clauses of F under assignment α and $CS(F, \alpha)$ the set of satisfied clauses respectively. The key idea behind DLS is to guide the search dynamically by increasing (scaling) or decreasing (smoothing) the weights of clauses continuously or periodically. This results in a guidance of the search into the direction of the higher weights.

The main difference between the different DLS methods is the way weights are being scaled, smoothed and used. The number of DLS methods proposed in the last two decades is quite large. For the work developed in this thesis only two methods are of interest: the *Scaling and Probabilistic Smoothing* method used in the SLS solver *SAPS* [HTH02] and the *Pure Additive Weighting Scheme* used in the solver *PAWS* [Tho05].

For an overview and analysis of DLS methods, the reader is referred to the PhD thesis of Dave Tompkins who studies in depth the field of Dynamic Local Search (DLS) [Tom10]. The first DLS methods were proposed for the *GSAT* solver by Selman and Kautz in [SKC94], followed by other methods proposed by Morris [Mor93], Frank [Fra96, Fra97] and Mills [MT99]. DLS methods inspired by the *Discrete Lagrangian Methods* (coming from the operation research area) have been proposed in [SW98, WW99, WW00, SS00]. The probably best representative and best performing Lagrangian method is the *ESG* algorithm proposed by Schuurmans *et al.* in [SSH01], which then motivated the work of Hutter *et al.* on the DLS solver *SAPS*, which plays an important role in this thesis.

SAPS

The core search procedure of *SAPS* is based on a variation of *GSAT*. The algorithm starts by initializing all clause weights to one. In each search step, *SAPS* picks the best variable according to the weighted score, if the score exceeds a certain threshold ϵ. Ties are broken randomly. If no such variables exist, then with probability $wp \in [0 \dots 1]$ it performs a random flip, else it will perform the scaling and smoothing of the clause weights. Scaling is performed only on the unsatisfied clauses according to the following scheme: $clw'(C_j) = clw(C_j) \cdot a$, where $a > 1$ is a parameter of the solver (originally a was denoted α, but this conflicts with our notation). The smoothing step is performed only with a probability of $sp \in [0 \dots 1]$ (also called smoothing probability). A smoothing step is performed only for clauses that have weights greater than one according to: $clw'(C_j) = clw(C_j) + (1 - \rho) \cdot \overline{clw}$, where $\overline{clw}$ denotes the average clause weight over all clause weights and $\rho \in [0 \dots 1]$. The performance of *SAPS* depends on its parameters a, sp and ρ. To overcome this sensitivity problem, Hutter *et al.* proposed *RSAPS* [HTH02](a reactive version of *SAPS*), which tries to automatically adapt these parameters during search, similar to the adaptive methods in the *AdaptNovelty* [Hoo02] solver.

SPAWS

Thornton analyzed in [Tho05] in more detail the *SAPS* solver and proposed a pure additive clause weighting scheme called *PAWS*. The major difference between *PAWS* and *SAPS* is that the scaling and smoothing steps are restricted to linear operations. The weights of all false clauses are scaled up linearly with the update rule: $clw(C_j) = clw(C_j) + 1$. Weighted clauses are smoothed after Max_{inc} steps (which renders this operation to be deterministic) with the rule: $clw(C_j) = clw(C_j) - 1$. *PAWS* also differs from *SAPS* in the way it picks a variable when no improving steps are possible. Instead of the random walk step, *PAWS* probabilistically (with probability p_{flat}) performs a flat move (a flip that does not change the number of unsatisfied clauses).

3.6 gNovelty+

The *gNovelty+* solver was proposed by Pham in [PTGS07] and was one of the best SLS solvers a couple of years ago when the studies for this thesis started. We will describe this solver in more detail, as it is the basis of several solvers developed within this thesis. Though there are several versions of *gNovelty+*, they all can be characterized as being a hybrid SLS solver that combines the methods used in G^2WSAT, *AdaptNovelty*, *SAPS* and *PAWS*. The key idea is to combine the gradient steps of G^2WSAT with a new clause weighting scheme similar to that of *SAPS* and *PAWS*.

Algorithm 3 contains the pseudo code of *gNovelty+* in the version it was first presented in [PTGS08]. The first decision taken by *gNovelty+* is whether to select a variable according to a pure focused random walk or not, which is done with probability $wp = 0.01$ (line 7). If no random walk is performed, the solver tries to pick the best *promising* variable (according to the definition of G^2WSAT) breaking ties by the *age* of the variables (line 10). In case no *promising* variables are available, it will pick a variable according to the *AdaptNovelty* heuristic (line 12). In this case, the weights of all unsatisfied clauses are scaled, and with probability *sp* smoothed with a linear scheme, like in *PAWS*. The selected variable is being flipped, and then the noise and the list of promising variables is updated.

All selection steps used in *gNovelty+* use the weighted *score* of variables and not the real *score*. Accordingly, variables can get *promising* if weights are being scaled or they can loose this property if weights of clauses are being smoothed. As the authors of the solvers also noted in [PTGS08], the smoothing probability *sp* is the key parameter

Algorithm 3: *gNovelty+* solver

```
   Input  : Formula $F$, $maxTries$, $maxFlips$, $wp = 0.01$, $sp$
   Output: satisfying assignment $\alpha$ or UNKNOWN
1  for $i = 1$ to $maxTries$ do
2      $\alpha \leftarrow$ randomly generated assignment
3      for $j = 1$ to $maxFlips$ do
4          if ($\alpha$ is model for $F$) then
5              return $\alpha$
6          if (rand() < wp) then
7              var=randomly picked variable from one randomly selected
               unsatisfied clause
8          else
9              if $\exists$ promising variable then
10                 var=best promising variable (break ties by age)
11             else
12                 var = pick variable like AdaptNovelty heuristic
13                 update weights of all unsatisfied clauses (like PAWS)
14                 if (rand()< sp) then
15                     smooth weights of all weighted clauses (like PAWS)
16         flip(var)
17         update noise $p$ for AdaptNovelty
18         update the promising variable list of $G^2WSAT$
19 return UNKNOWN;
```

of *gNovelty+* having high influence on its performance. The setting of *sp* also depends on the type of problem being solved.

For $sp = 1$, the clause weights are disabled and *gNovelty+* behaves almost like G^2WSAT with an *AdaptNovelty* component. The authors determined empirically that *sp* should be set to $sp = 0.4$ for random 3-SAT problems and to $sp = 1.0$ for 5-SAT and 7-SAT problems. With this setting *gNovelty+* was able to win the random satisfiable track of the SC07, which attracted the attention of several researches on this solver.

4 A Novel Approach to combine an SLS and a DPLL Solver for SAT

Within this chapter, we will present a generic approach on how to analyze the search of an SLS solvers and how to combine an SLS and a DPLL solver to create an incomplete hybrid SAT solver. The SLS solver gets supported by a DPLL solver, which results in a boost of its performance. We will first define the term of a Search Space Partition (SSP), which is the key concept in our hybridization and provide some explanation for its construction and use. We implement our hybridization approach in several solvers, and show within empirical studies that the performance of the hybrid solvers is exceeding the performance of the plain SLS solvers on most tested benchmarks.

The work presented in this chapter is partially based on work presented in [BHG09].

4.1 Introduction

Motivated by the performance of modern SLS solvers, we started the analysis of SLS solvers presented in Chapter 2. Our main goal was to analyze if the search of an SLS solver (in our particular case that of *gNovelty+*) is thoroughly, or if there are some ways to improve it.

Two issues play a crucial role here: *intensification* and *diversification* of the search. There is no clear mathematical definition of these terms and researchers often mean slightly different things. We characterize the search of a solver as being *intensified* when the solver restricts its decision to a small set of variables and changes only these, guiding its decision on information that will improve the overall objective function (the number of satisfied clauses). Guiding the search based on greedy properties usually results in an intensified search. The search of a solver is *diversified* when the solver is taking almost all variables into consideration for flipping, and when it tries to avoid repeating previous steps by taking diversification properties like *age* or *tabu* into consideration within the selection heuristic.

A solver has to find the optimal balance between intensification and diversification to reach state-of-the-art performance. This balance can be controlled in the *WalkSAT* style solvers with the noise parameter p. High noise leads to diversified search, while low noise values intensify the search.

4.2 Search Analysis

The first observation looking at the results of SLS solvers in the random category of the SC07 was that the run time of an SLS solver on problems of the same size can vary greatly. To analyze this effect we focused our attention on the winning SLS solver of the SC07 random satisfiable category, *gNovelty+* [PTGS08].

Our assumption was that the search space structure of the hard problems contained many attractive local minima that were visited by the SLS solver very often, which means that the diversification in the solver is not sufficient. To verify this hypothesis we tried to cluster all points from *gNovelty+*'s search trajectory. This can be done by saving all assignments visited by the solver during its search. This approach, though, was unsuccessful because no cluster algorithm could cope with the huge amount of high dimensional (every variable represents one dimension) data we had. Note that an SLS solver can visit up to one million assignments per second and it takes up to several hundreds of seconds to solve a problem.

Another approach was to analyze only the points where the objective function had very low values (*i.e.* local minima and their close neighborhood). Since the amount of data was still too large, we did not directly save the assignments but used instead a bloom filter.

A bloom filter is a probabilistic data structure [Blo70] which allows to efficiently check whether an element is already part of a set or not. A bloom filter uses a binary array (*i.e.* an array that can contain only 0 and 1) as a data structure and provides two operations: adding and querying of elements. When initialized, the binary array contains only zeros. The add operation takes an element as input and computes with k different hash functions k positions in the array, which are then set to 1. The query operation will also use the same k hash functions to compute the positions in the array. If one of the computed positions in the array is 0, then the element is not part of the set. If all positions are already 1, then there is a high probability that the element is contained in the set, but this is no guaranteed, as the positions might have been set to 1 by the add operation of other elements. The query operation thus can produce false positives (*i.e.* return true even when the element is not part of the set).

With the help of the bloom filter we checked how many of the assignments visited by *gNovelty+* fell in the neighborhood of the saved local minima. The maximum found matching was lower than 2%. This indicates that the diversification of *gNovelty+*'s search is sufficient for the analyzed problems.

The next thing to analyze was if the intensification of *gNovelty+*'s search around these local minima was sufficient to assure with a high probability that there are no solutions. One possibility to check this is to search the complete neighborhood of a local minimum within a certain Hamming distance. This is possible for small problems, but for instances having thousands of variables the neighborhood is by far too large to be computed in foreseeable time. However, this research seemed promising, because Zhang showed in [Zha04] that the Hamming distance between a qualitative local minimum and the nearest solution is correlated with the quality of that local minimum.

The key contribution of this chapter is a method to check the intensification of SLS solvers around local minima (in general around any arbitrary point in the search space visited by the solver). The idea is to build a partial assignment around one complete assignment from the search trajectory of the solver. This partial assignment is then applied to the formula resulting a simpler and smaller formula that can be solved with a complete solver (introducing the hybridization approach). If the complete solver finds a solution for the simplified formula, the intensification was not sufficient. As a side effect, the problem can be solved, resulting in a speed up of the hybrid solver over the plain SLS solver.

4.2.1 Search Space Partitions

Search space partitions can be easily defined with the help of partial assignments and the flip trajectory of an SLS solver. The necessary definitions and notations have been partially presented in Section 2.1.1. Recall that the application of a partial assignment β on a formula F results in a simplified formula F' (*i.e.* $F|\beta = F'$).

Definition 4.2.1 (Flip Trajectory)**.** Given an SLS solver S with an input formula F and a complete starting assignment α_s, we define the flip trajectory of $S(F, \alpha_s)$ as $T_S = t_1, \ldots, t_w$, where $t_i \in \{x_1, \ldots, x_n\}$ denotes the variables being flipped by the SLS-algorithm S, and w is the total number of flips made starting with the formula F and the initial assignment α_s.

The flip trajectory is a vector of variables recording the history of the search of an SLS solver.

Definition 4.2.2 (Search Space Partition (SSP))**.** We define an SSP by construction: Given a complete assignment α_j, which was visited by S in the j'th flip on its flip trajectory, we construct the SSP by starting with $k = 0$ and $\beta = \alpha_j$. We repeat setting $\beta[t_{j+k}] = *$ and $\beta[t_{j-k}] = *$, where $t_{j\pm k} \in T_S$ and $*$ denotes unassigned variables, and increasing k by 1 until the number of unassigned variables in β exceeds $c \cdot n$, where c is some fixed constant $c \in (0, 1)$ (to be determined later).

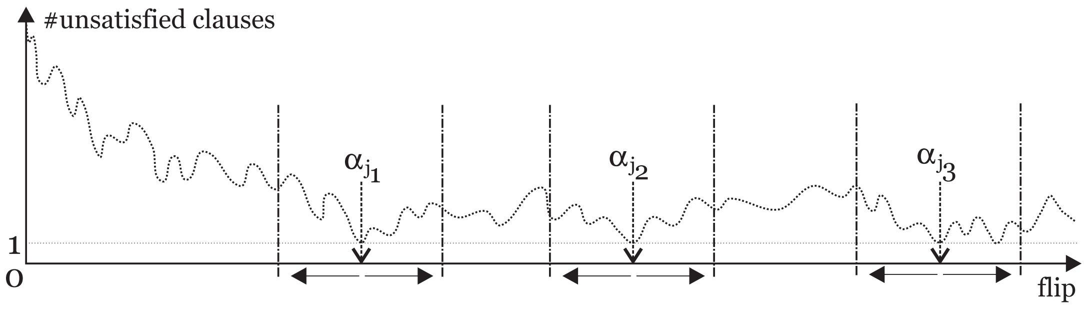


Figure 4.1: An exemplary evolution of the objective function of an SLS solver during its search. The x-axis denotes the step (flip) number and the y-axis the number of unsatisfied clauses. α_{j_i} represents the start point of the construction of several SSPs around these points.

An SSP defines a subspace of the complete search space that an SLS solvers has traversed. It can be constructed from the flip trajectory of the SLS solver, and its parameters are the start position and size. To exemplify the construction of an SSP let $\alpha_7 = 0011010111$ be a complete assignment for a formula F with 10 variables that was visited by the solver after the 7th flip. Let the surrounding flip trajectory be:

$$T_S = x_2, x_6, x_1, x_9, x_1, x_6, \underline{\mathbf{x_1}}, x_3, x_9, x_1, x_1, x_8, x_3, \ldots$$

If we set $c = 0.5$ and start to construct an SSP from position $j = 7$ from $T_S(F, \alpha_s)$, then the first variable that is unassigned in β is x_1 ($k = 0$). In the next step, x_3 and x_6 get unassigned ($k = 1$) according to T_S. This procedure is repeated until five variables get unassigned. After five steps the process will stop with $\beta = *0 * 10 * 0 * *1$.

Figure 4.1 describes how SSPs are created around local minima α_{j_i} with exactly one unsatisfied clause. Note that an SSP can entail more than one local minimum.

The notion of search space partitions was also used by Wu and Hsiao in [WH06], where they propose a simulation-based algorithm for checking the safety property of digital systems. Although the notion is similar, the concepts and the construction are quite different.

4.2.2 Construction and Use of Search Space Partitions

We are interested in the study of local minima and their neighborhood and whether there are solutions in that part of the search space that have been missed by an SLS solver. The α_j mentioned in the definition of an SSP would ideally be a local minimum. An SSP created as defined above overlaps with the Hamming neighborhood of α_j, but is not identical.

A local minimum exists because several variables imply a complex conflict that cannot be directly resolved by the SLS solver. It might be the case that conflicting variable assignments are partially detected by the SLS solver, but the order and the values used for these to change the current assignment will not lead to a solution. Therefore, it might be worth to monitor the flips performed around local minima to detect a set of possibly conflicting variables.

Using this knowledge, we can create a partial assignment by unassigning the flipped variables around the local minimum. The result is an SSP. Given such an SSP, our hybrid strategy consists in calling a DPLL style solver, which will try to resolve the conflict by finding an assignment for the unassigned variables. In case the DPLL solver finds an assignment that resolves the conflict using the unassigned variables, a solution for the complete formula is found. If a satisfying assignment does not exist, the conflict can not be resolved by using only the free variables in the SSP (*i.e.* the conflict variables have not been unassigned in the SSP). In this case, the search of the SLS solver must continue.

All in all, a generic algorithm implementing the above described idea will use an SLS solver to localize qualitative local minima, build an SSP, apply the partial assignment of the SSP on the formula, and try to find a solution for the simplified formula with a DPLL solver. This process will be repeated until a solution is found or until some stopping criteria is met. This solving approach can not prove the unsatisfiability of the problem, however it could speed up the search process by finding solutions sooner.

4.3 hybridGM

To check if our approach is promising we implemented a hybrid SAT solver called *hybridGM*. The SLS solver used is *gNovelty+*. We conjecture that an SSP does not contain a solution in the majority of cases, thus, we needed a solver that can prove the unsatisfiability of a given (sub-)formula fast. The solver *march_ks* was the winner of the UNSAT random category of the SC07, and therefore, we have chosen *march_ks* for the DPLL component of *hybridGM*.

When implementing *hybridGM*, two questions arose. First, which local minima should be used to create an SSP? Using every appearing local minimum in the objective function leads to an overwhelming workload. Numerous local minima are discovered by the SLS solver as we have noticed when analyzing the search trajectory of *gNovelty+*. To reduce the workload, we confine ourselves to using only those assignments that leave exactly one clause in the formula unsatisfied.

Second, how large should the SSP be? In other words how shall we set parameter c when constructing the partial assignment β? The more variables in β are unassigned, the higher the probability to resolve all conflicts in the corresponding SSP. But on the other hand, the less variables in β are unassigned, the faster the DPLL solver can

Algorithm 4: *hybridGM* solver

```
Input  : Formula $F$, $maxTries$, $maxFlips$, $barrier = 1$
Output: satisfying assignment $\alpha$ or UNKNOWN
for $i = 1$ to $maxTries$ do
    $\alpha \leftarrow$ randomly generated assignment
    for $j = 1$ to $maxFlips$ do
        if ($\alpha$ is model for $F$) then
            return $\alpha$
        $var$=pickVar()
        flip($var$)
        update data structures
        append($T_S, var$)
        if ($|CU(F, \alpha)| \leq barrier$) then
            $\beta$=construct SSP of size $c \cdot n$ from $T_S$ starting at position $j$
            $\gamma$=DPLL($F, \beta$)
            if $\gamma$ is model for $F$ then
                return $\gamma$
            if DPLL did not perform search then
                c=c+0.05
return UNKNOWN;
```

solve the simplified problem. Therefore, we have to find a compromise between the probability to resolve all conflicts and the run time of the DPLL solver.

Within preliminary experimental studies we started with an initial size of $0.5 \cdot n$, which seems to be an appropriate size. In several cases the DPLL solver detected conflicts without taking any decisions, which means that the pure application of β on the formula F following the rules of Boolean logic already renders the formula as being unsatisfiable. In such cases we gradually increase the SSP size by 5% of the number of variables until the DPLL solver is performing search.

Algorithm 4 shows the pseudo-code of *hybridGM*. The solving concept is actually not restricted to an particular SLS or DPLL solver. The main search paradigm is that of an SLS solver with the only difference that when the objective function has reached a certain *barrier*, the solver builds an SSP and calls the DPLL solver with the constructed partial assignment and the formula. If the DPLL solver is able to solve the problem, the algorithm terminates. Else, the SLS solver takes over again and continues search while monitoring its search trajectory.

Within our concrete implementation of the framework we have used as SLS solver *gNovelty+* and as DPLL solver the look-ahead solver *march_ks*. Details about this type of solvers can be found in more detail in Section 2.3.

4.4 Empirical Evaluation

For the implementation of *hybridGM* we used the *gNovelty+* code in the version submitted to the SC07, with a changed smoothing probability from 0.4 to 0.33. The barrier was fixed to one, so that SSPs are built only when exactly one clause remains unsatisfied. An adaptive scheme is subject of future research. In some cases when calling *march_ks* and returning the result to *gNovelty+*, we observed that part of the memory allocated by *march_ks* data structure is not released. This memory leak is very small, however if the number of *march_ks* calls gets too large, we get an out of memory exception. To avoid this case we limited the number of *march_ks* calls to 5000.

We also tested different settings for the size of the SSP (*i.e.* the number of variables in an SSP). However, we found out that an initial value larger than $0.5 \cdot n$ just increases the computation times of *march_ks*, without improving its success rate.

Soft- and Hardware The *gNovelty+* and the *adaptg2wsat0* code we used for the comparison was the one submitted to the SC07.[1] The *march_ks* code we used was a bug-fixed version of the SC07.[2] The code of *hybridGM* was version 3 of the one submitted to the SC09.[3] The solvers were run on the bwGRiD (see page 123 for more details).

The Benchmark Problems We used problems from two benchmark sets to compare *hybridGM* with *gNovelty+* and *adaptg2wsat0*: the SC07 benchmark[4], and the SATLIB benchmark[5].

Concerning the SC07 benchmark, we randomly selected satisfiable randomly generated problems. Our set contains $2 + p$ problems of different sizes, small uniform random k-SAT problems generated on the threshold and large-size k-SAT problems that are underconstrained, being generated with a clause to variable ratio lower than the threshold value. Concerning the SATLIB benchmark, we confined ourselves to only crafted/industrial instances, because the provided random instances can be solved by *gNovelty+* and *hybridGM* in less than ten seconds.

[1] `http://www.satcompetition.org/2007/winners.tgz`
[2] `http://www.st.ewi.tudelft.nl/sat/Sources/sat2007/march_ks.zip`
[3] `http://www.cril.univ-artois.fr/SAT09/solvers/SAT2009sources.7z`
[4] `http://satcompetition.org/2007/random.tar.bz2`
[5] `http://www.satlib.org`

Results Table 4.1 presents the results of our first empirical study, which reveals that *hybridGM* dominated *gNovelty+* on the $2 + p$ and the large 3-SAT problems. Both types of problems seem to be underconstrained, containing many solutions. The intensification of *gNovelty+* is not sufficient to find the solutions in the area where the search is performed. *hybridGM* can compensate this problem with its DPLL component, which causes almost no overhead when called. The larger the number of times *march_ks* finds the solution, the better is the speed up when compared to *gNovelty+*.

Instance	***gNovelty+***	**adaptG2-WSAT0**	***hybridGM* (gNov,March)**	**Gain**
SAT 2007 Competition random				
unif2p-p0.7-v3500-c9345-S1568322528-08	10%	2.91 | 1.63	9.63 | 7.82 (9, 91)	**> 1**
unif2p-p0.7-v4500-c12015-S1973057201-08	25%	138.52 | 42.71	121.52 | 82.44 (23, 77)	> 1
unif2p-p0.7-v5500-c14685-S915337037-05	95%	3.33 | 1.60	20.51 | 5.79 (24, 76)	**> 1**
unif2p-p0.7-v6500-c17355-S152598520-02	226.64 | 168.00	10.23 | 1.90	2.17 | 1.66 (27, 73)	**104.44**
unif2p-p0.8-v1295-c4027-S1762612346-15	136.06 | 94.03	1.13 | 0.79	2.65 | 2.20 (9, 91)	**51.34**
unif2p-p0.8-v1665-c5178-S1404069132-16	265.54 | 196.49	5.70 | 3.10	23.56 | 17.24 (13, 87)	**11.27**
unif2p-p0.8-v2035-c6328-S316347254-19	0.75 | 0.48	0.18 | 0.14	0.23 | 0.17 (40, 60)	**3.26**
unif2p-p0.8-v2405-c7479-S183991542-09	33%	33.50 | 21.10	92% (5, 87)	> 1
unif2p-p0.9-v1170-c4235-S2131244303-19	52.26 | 31.82	1.45 | 1.21	2.80 | 2.07 (30, 70)	**18.66**
unif2p-p0.9-v630-c2280-S2099846342-04	0.35 | 0.27	0.14 | 0.10	0.33 | 0.23 (88, 12)	**1.06**
unif2p-p0.9-v810-c2932-S1274825698-06	0.27 | 0.21	0.06 | 0.05	0.15 | 0.10 (59, 41)	**1.80**
unif2p-p0.9-v990-c3583-S461590508-14	0.28 | 0.21	0.10 | 0.08	0.23 | 0.18 (56, 44)	**1.22**
unif-k3-r4.261-v650-c2769-S1089058690-02	0.25 | 0.16	0.17 | 0.10	0.38 | 0.28 (61, 39)	0.66
unif-k3-r4.261-v650-c2769-S1172355929-14	0.04 | 0.02	0.02 | 0.02	0.05 | 0.04 (79, 21)	0.80
unif-k3-r4.2-v10000-c42000-S1173369833-06	73.87 | 50.01	11%	7.28 | 5.61 (23, 77)	**10.15**
unif-k3-r4.2-v10000-c42000-S421554531-04	98.25 | 79.86	7%	11.66 | 9.52 (30, 70)	**8.43**
unif-k3-r4.2-v13000-c54600-S1054448974-13	97%	0%	44.89 | 38.74 (36, 64)	> 1
unif-k3-r4.2-v13000-c54600-S161446644-14	55%	0%	99% (33, 66)	**> 1**
unif-k3-r4.2-v16000-c67200-S1099746708-06	23%	0%	79% (24, 55)	> 1
unif-k3-r4.2-v16000-c67200-S1600965758-04	18%	0%	73% (18, 55)	**> 1**
unif-k3-r4.2-v19000-c79800-S1106616038-10	74%	0%	92.02 | 70.75 (39, 61)	**> 1**
unif-k3-r4.2-v19000-c79800-S1299985238-16	2%	0%	60% (16, 44)	> 1
unif-k3-r4.2-v4000-c16800-S1178874381-13	8.11 | 6.02	184.52 | 110.63	4.99 | 4.00 (46, 54)	**1.63**
unif-k3-r4.2-v4000-c16800-S1588170820-15	94%	14%	11% (5, 6)	≤ 1
unif-k3-r4.2-v7000-c29400-S102550125-14	42.92 | 31.46	82%	6.85 | 5.70 (28, 72)	**6.27**
unif-k3-r4.2-v7000-c29400-S2051531193-03	45.65 | 27.33	56%	9.84 | 7.48 (30, 70)	**4.64**
unif-k5-r21.3-v100-c2130-S455021619-18	0.02 | 0.02	0.05 | 0.04	0.03 | 0.03 (100, 0)	0.67
unif-k5-r21.3-v100-c2130-S744612847-12	0.06 | 0.04	0.06 | 0.06	0.07 | 0.05 (100, 0)	0.86
unif-k5-r21.3-v110-c2343-S1019153514-04	0.66 | 0.47	0.33 | 0.26	0.66 | 0.46 (100, 0)	1.00
unif-k5-r21.3-v110-c2343-S1869272420-19	0.05 | 0.04	0.06 | 0.05	0.05 | 0.04 (100, 0)	1.00
unif-k5-r21.3-v120-c2556-S1191693850-19	0.14 | 0.11	0.12 | 0.08	0.15 | 0.11 (100, 0)	0.93
unif-k5-r21.3-v120-c2556-S1615006153-07	0.48 | 0.32	0.40 | 0.26	0.48 | 0.33 (100, 0)	1.00
unif-k5-r21.3-v130-c2769-S1109841921-18	0.50 | 0.40	0.39 | 0.28	0.51 | 0.41 (100, 0)	0.98
unif-k5-r21.3-v130-c2769-S1284937235-05	0.33 | 0.24	0.27 | 0.18	0.33 | 0.24 (100, 0)	1.00
unif-k7-r89-v70-c6230-S1106151685-15	0.64 | 0.38	1.43 | 1.26	0.64 | 0.38 (100, 0)	1.00
unif-k7-r89-v70-c6230-S1635684145-01	0.52 | 0.38	1.28 | 1.15	0.53 | 0.39 (100, 0)	0.98
unif-k7-r89-v75-c6675-S1299158672-14	11.33 | 8.29	10.93 | 6.86	11.37 | 8.33 (100, 0)	1.00
unif-k7-r89-v75-c6675-S1572638390-17	9.99 | 7.29	10.98 | 8.39	9.98 | 7.24 (100, 0)	**1.00**
SATLIB industrial				

Instance	*gNovelty+*	adaptG2-WSAT0	*hybridGM* (gNov,March)	Gain
bw_large.c	4.31 | 2.94	3.49 | 2.72	9.09 | 6.24 (100, 0)	0.47
bw_large.d	21.34 | 15.30	16.11 | 10.63	64.25 | 41.74 (100, 0)	0.33
g125.17	11.00 | 7.50	2.17 | 1.53	10.26 | 7.69 (100, 0)	**1.07**
g125.18	0.13 | 0.13	0.22 | 0.22	0.14 | 0.14 (100, 0)	0.93
g250.15	0.25 | 0.25	1.07 | 1.07	0.27 | 0.27 (100, 0)	0.93
g250.29	13.92 | 10.11	8.96 | 7.96	14.31 | 11.10 (100, 0)	0.97
qg1-08	202.40 | 166.45	4.53 | 4.31	88% (88, 0)	≤ 1
qg2-08	43%	10.77 | 9.07	9% (9, 0)	≤ 1
qg5-11	2%	64%	6% (6, 0)	**> 1**
qg6-09	92%	2.95 | 1.64	99% (99, 0)	**> 1**
qg7-13	0%	18%	0% (0, 0)	≤ 1
SATLIB crafted				
par16-1-c	4.49 | 2.96	12.20 | 8.56	70% (66, 4)	≤ 1
par16-2-c	51.64 | 40.79	101.65 | 81.98	66% (52, 14)	≤ 1
par16-3-c	18.24 | 14.20	30.18 | 22.65	99% (96, 3)	≤ 1
par16-4-c	19.77 | 11.30	24.91 | 17.13	97% (96, 1)	≤ 1
par16-5-c	19.21 | 12.32	17.20 | 13.45	98% (95, 3)	≤ 1
par32-1-c	0%	0%	0% (0, 0)	≤ 1
par32-2-c	0%	0%	0% (0, 0)	≤ 1
par32-3-c	0%	0%	0% (0, 0)	≤ 1
par32-4-c	0%	0%	0% (0, 0)	≤ 1
par32-5-c	0%	0%	0% (0, 0)	≤ 1

Table 4.1: Each solver performed 100 runs per instance (*gNovelty+* and *hybridGM* were started with the same seed). Run times are given in seconds: mean | median. If the solver was not able to succeed 100 times (each within 2000 seconds), we give the success rate for the instance in percent. The *hybridGM* column also contains a percentage tuple that specifies how often the solution was found by each component. The gain column represents the speed up of *hybridGM* over *gNovelty+* (> 1 indicates that *hybridGM* was faster or had a better success rate and is typed bold).

On the small highly constrained k-SAT problems, *hybridGM* cannot take advantage of its DPLL component that is not able to solve the problem in any of its calls. The same holds also for most of the industrial and crafted problems.

hybridGM was submitted to the SC09 in the random satisfiable category where it won the third place (bronze medal). Its performance on the large 3-SAT problems exceeds that of all other competitors, being very often the only solver that was able to solve a problem. On 5-SAT and on 7-SAT instances, its performance dropped significantly loosing points against the competitors.

4.5 Extended Analysis

To show that the hybridization concept can be applied also to other solvers we have implemented two other hybrid solvers, namely *hybridGP* respectively *hybridPP*. The

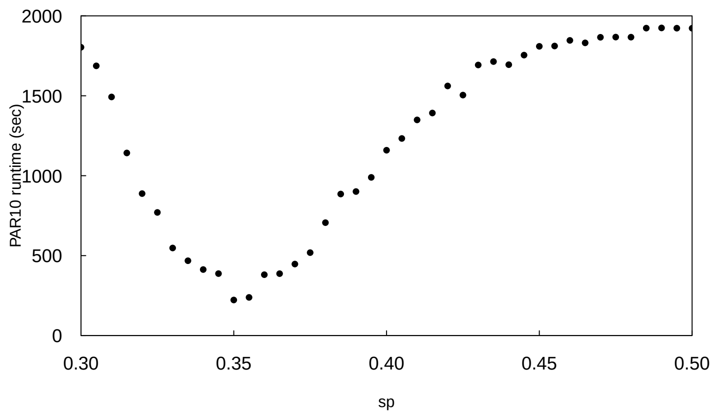


Figure 4.2: The performance of *hybridGP* solver with varying values for *sp* (x-axis) measured as PAR10 of the run time in seconds (y-axis). The lower the PAR10 value the better the performance of the solver.

former uses as an SLS component a reimplementation of *gNovelty+* and as a DPLL component the CDCL solver *picosat* [Bie08], which is memory efficient and has an easy to use Application Programming Interface (API). *hybridPP* uses the state-of-the-art SLS solver *probSAT* (see Chapter 7) and the CDCL solver *picosat* [Bie08]. In both solvers the SSP size is limited to exactly half of the number of variables of the input formula. To limit the solving time of *picosat* we allow it to perform up to $5 \cdot 10^4$ decisions. This value seems to be sufficient for *picosat* to solve the constructed sub-problem in almost all tested cases. One call of *picosat* needs on average only 0.05 seconds.

Optimal *sp* parameter for hybridGP As a first step we want to know what is the optimal value of the smoothing probability parameter *sp* in the SLS component for different types of k-SAT problems. We have randomly selected a set of 100 3-SAT problems with 10^4 variables and a ratio of 4.2. These instances come from [TBH11] and are also used in Chapter 7 for the optimization of the parameters of the *probSAT* solver. We have evaluated *hybridGP* with different values of $sp \in [0.3 \ldots 0.5]$ in steps of 0.005 on these instances with a cutoff time of 200 seconds. The performance is measured as PAR10[6] value of the run time. Figure 4.2 shows the results of the evaluation.

A value of $sp = 0.35$ seems to be the best choice for this type of instances, and not $sp = 0.4$ as mentioned in [PTGS07]. The shape of the performance curve is very similar to the one of the *probSAT* solver studied in Chapter 7 (see Figure 7.1 right

[6]PAR10 (Penalized Average Run time) is the average run time of the solver, while runs that reached the cutoff time are penalized with a factor of 10.

side). The performance degradation when moving away from the optimal value is more pronounced on the left side (lower *sp* values) than on the right side (higher *sp* values). We can thus conclude that the *sp* parameter indirectly controls the greediness of the solver and can thus be compared to the noise parameter in the *WalkSAT* solver [SKC94] and the *cb* parameter in the *probSAT* solver [BS12].

We have performed similar experiments to determine appropriate values of *sp* for 5-SAT ($n = 500$, $r = 20$) and 7-SAT ($n = 90$, $r = 85$) problems, also coming from [TBH11]. Beneficial values for the *sp* parameter for our target problems are as follows:

k-SAT	size	ratio	*sp*
3-SAT	$n = 10^4$	$r = 4.2$	0.35
5-SAT	$n = 500$	$r = 20$	0.90
7-SAT	$n = 90$	$r = 85$	0.90

Having the optimal *sp* parameter, we can measure the difference between *hybridGP* with and without the CDCL component. The same evaluation is also done for *hybridPP* to see if the hybridization can boost also a state-of-the-art solver. The parameters of *hybridPP* are the default ones reported in [BS13] and are also presented in Chapter 7. We start both pairs of solvers with the same seed, which means that both solvers have the same flip trajectory in each run. The only difference is that the hybrid version will stop on the flip trajectory once a solution has been found by *picosat* within an SSP, while the version without hybridization (the plain SLS) will continue search. For the evaluation, we used sets of 250 instances with size and ratio reported before with a cutoff time of 900 seconds.

3-SAT Figure 4.3 (first row) shows the results of our evaluation for 3-SAT problems as a scatter plot. Points lying on or near the diagonal are runs where the problem was either solved by the SLS component or by the CDCL component just shortly before the SLS component found the solution. As no points lie below the diagonal, we can conclude that the overhead caused by the CDCL calls can be neglected. All points laying clearly above the diagonal represent runs where *picosat* finds solutions within a constructed SSP, which is often the case for *hybridGM*. In the case of *hybridPP*, the boost of the hybridization is not that prominent but still present. See Table 4.2a for detailed results. In terms of average run time, *hybridPP* can improve *probSAT* by 15%, reducing the average run time from 29.46 seconds to 25.59 seconds. From the total of 250 runs, *picosat* was able to find a solution in 112 cases, while in the other cases the SLS solver found the solution.

5-SAT The second row of Figure 4.3 shows the results of the evaluations on the 5-SAT problems. The performance of *gNovelty+* can be improved in several cases as can be seen from Figure 4.3 (second row, left side) where several points lay above the diagonal

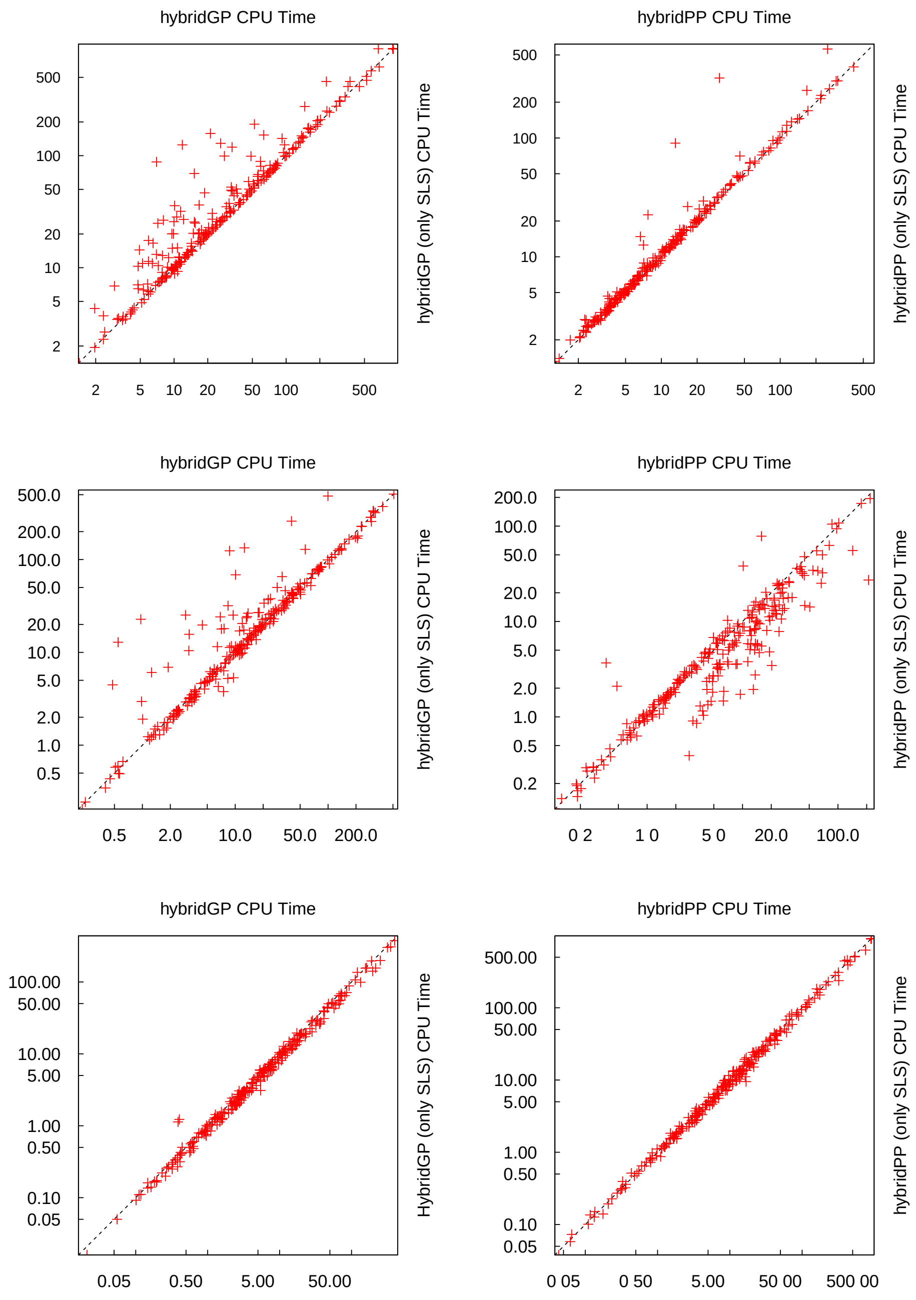


Figure 4.3: Comparison of the performance (CPU-time measured in seconds) on a log scale between the hybrid solver *hybridGP* (left side) and *hybridPP* (right side) represented on the x-axis and its SLS components represented on the y-axis, when started on the same flip trajectory on 3-SAT (first row), 5-SAT (second row) and 7-SAT problems (third row).

and only few of them below. This improvement can be seen in the average run time (see Table 4.2b) and also more pronounced in the median run time. The *probSAT* solver can not profit from the hybridization, being rather hindered by the CDCL component. Here we have to notice that *probSAT* reaches remarkable performance on 5-SAT problems when compared to other SLS solvers. Further, the number of calls of *picosat* is relatively high, however no solution can be found in the constructed SSPs.

7-SAT The results of the 7-SAT evaluation can be seen in Figure 4.3 last row. Similar to *hybridGM* no improvement can be achieved for this type of problems. The number of constructed SSPs is relatively high, however none of them contains a solution. As a result the run time of the solver is extended without benefit (see Table 4.2c for detailed run times).

SC12 Hard combinatorial We have performed similar evaluations also for hard combinatorial problems. We have used the satisfiable instances used in SC12 Hard Combinatorial Track. The results of the evaluation can be seen in Figure 4.4. Besides some cases where the CDCL component is able to find a solution (around 10%), in all other cases the numerous calls of *picosat* increase the run time of the solver considerably, both for *hybridGP* and *hybridPP*. As opposed to the k-SAT problems, on the hard combinatorial problems *picosat* is sometimes not able to solve the sub-problem within the $5 * 10^4$ decisions that it is allowed to perform[7]. In the other cases the SSP does not contain a solution. This was also the case for 7-SAT problems. The similarity with respect to the behavior of SLS solvers on large k-SAT problems and hard combinatorial problems was also observed in [BS12] and will be discussed in Chapter 7. Since the constructed SSPs do not contain a solution, though plenty of them can be constructed, the instances contain many high quality local minima found by the SLS solver, where it gets trapped.

Run time distribution of the SLS and CDCL components To show that the runtime of the CDCL component is almost negligible, we have computed for the set of 3-SAT instances the runtime of the SLS and CDCL components separately, and plotted them as a cactus plot in Figure 4.5. The run time of *picosat* is two orders of magnitudes lower than that of the *gNovelty+* solver, being in most cases lower than 0.1 seconds. Consequently, it might be worth increasing the size of the SSP and measure the gain in terms of CPU time.

[7]An adaptive scheme which adapts the size of the SSPs would probably be more appropriate for these types of problems.

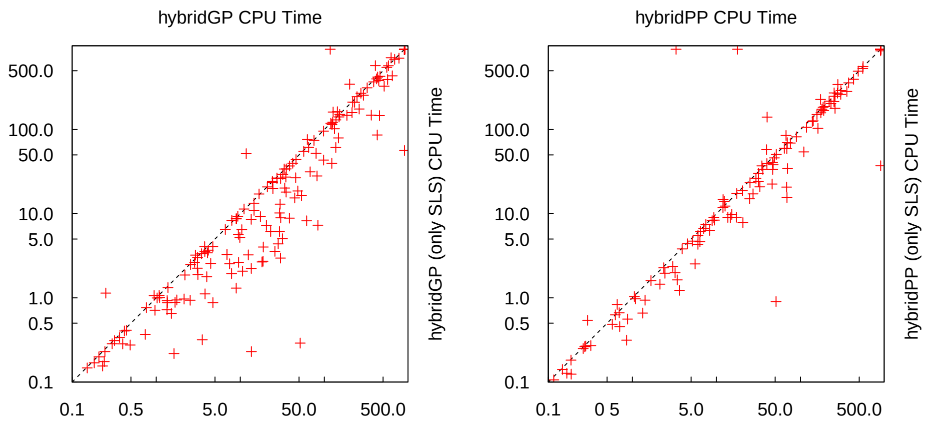


Figure 4.4: Comparison of the performance (CPU-time measured in seconds) on a log scale between the hybrid solver *hybridGP* (left side) and *hybridPP* (right side) represented on the x-axis and its SLS components represented on the y-axis, when started on the same flip trajectory on the satisfiable hard combinatorial problems from SC12.

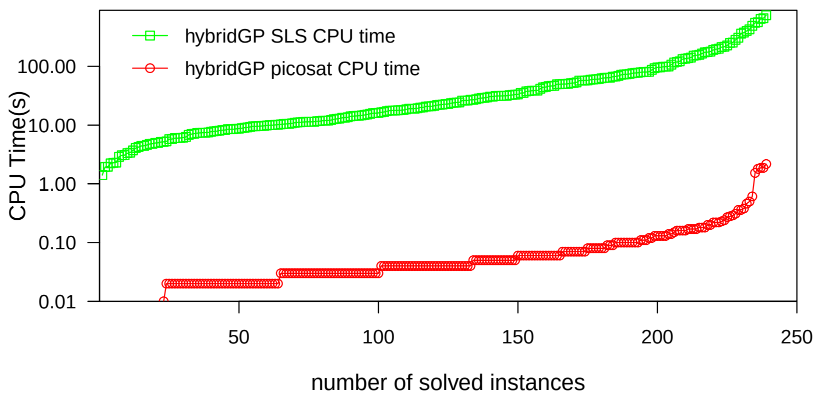


Figure 4.5: The run time of the CDCL component (*picosat*) and of the SLS component (*gNovelty+*) within the *hybridGP* solver on the set of 3-SAT instances. The y-axis is logarithmic scale.

(a) 3-SAT

	Solver	number solved	*total CPUtime*	*average CPUtime*	*median CPUtime*
1	hybridPP	250	6398.271	25.593	8.023
2	hybridPP (SLS)	250	7366.177	29.464	8.612
3	hybridGP	239	24398.630	97.594	22.940
4	hybridGP (SLS)	238	26662.139	106.648	28.294

(b) 5-SAT

	Solver	number solved	*total CPUtime*	*average CPUtime*	*median CPUtime*
1	hybridPP (SLS)	250	2768.550	11.074	4.454
2	hybridPP	250	3727.495	14.910	5.987
3	hybridGP	250	9382.189	37.528	12.635
4	hybridGP (SLS)	250	10488.255	41.953	15.822

(c) 7-SAT

	Solver	number solved	*total CPUtime*	*average CPUtime*	*median CPUtime*
1	HybridGP (SLS)	250	4712.859	18.851	4.238
2	hybridGP	250	5298.739	21.195	4.782
3	hybridPP (SLS)	244	14412.445	57.649	8.714
4	hybridPP	244	14874.343	59.497	9.147

Table 4.2: Ranking of the hybrid solvers and their SLS components on k-SAT problems with a cutoff time of 900 seconds.

4.6 Related Work

Considerable effort has been undertaken to create hybrid SAT solvers for more than a decade now. In general, three different approaches to create such hybrid SAT solvers have emerged.

The first approach uses an SLS solver to support a DPLL solver [Cra96, MSG98, FF04, HD04, FH07, Gab09]. Such a support can come in various ways. In [Cra96], an SLS solver is used to derive weights for clauses, which are then used by a DPLL solver to preferably branch on variables that occur more often in clauses with higher weights. In [MSG98], an SLS solver is used to find local inconsistencies in a formula. This knowledge then allows to narrow the search of a DPLL solver to the inconsistent part of a formula. As a result, the global unsatisfiability of a formula can be shown in less computational time. In [FF04], an SLS solver is used to determine an ordering of the branching variables that a DPLL solver should follow. In [Gab09], an SLS solver is used to identify areas of the search space that are more likely to contain a solution. These areas are represented as partial assignments which are used to start a DPLL solver. The solver *SparrowToRiss* [BM13b] submitted to the SC13 uses the SLS solver *Sparrow* for $5 \cdot 10^8$ flips, and then passes the last assignment in chronological order (*i.e.* the oldest flipped variable first) to the CDCL solver *Riss*, which initializes the phase saving of the variables according to this information. More details on this solver can be found in Section 6.6 of this thesis.

As a second approach, one is able to use information gathered by DPLL solvers on a certain formula to support the search of an SLS solver [JL02, FR04, HLDV02]. In [FR04], a DPLL solver is called whenever the SLS solver has moved into a local minimum in the search space. The approach adds implied clauses (learned by the DPLL solver) to modify the landscape of the search space of the SLS solver. This learning process is repeated until the SLS solver is able to move out of the (former) local minimum. In [HLDV02], a DPLL solver will derive implications between variables when arriving in a certain node of its search tree. With these implications, a reduced version of the currently investigated formula is created. When applying the SLS solver on the reduced formula, it will consider the equivalence classes rather than the original variables. This in turn helps the SLS solver to concentrate on actually different variables, when making its choice which variable to flip next. The SLS solver is then allowed to perform a maximum number of flips to search for a solution (under the DPLL provided preconditions). If the SLS solver finds a solution during its search, the algorithm terminates. If the SLS solver exceeds the number of allowed flips and did not find a solution, the DPLL solver comes back into play and continues traveling down its search tree.

The third approach on creating hybrid SAT solvers is peer-like, where SLS and DPLL solvers are supposed to benefit equally from each other as presented in [FH07, LMS08]. The *hbisat* solver [FH07] and its successor *hinotos* [LMS08] both use an SLS solver that first tries to solve the formula. When a certain criteria is met (*e.g.*, only a certain number of unsatisfied clauses remain), a DPLL solver is called, which is supposed to solve these clauses separately. When the DPLL solver finds a model for this partial set of clauses, it will return the corresponding assignment to the SLS solver, which will then use this assignment to continue its search. Eventually, the SLS solver will be able to find a solution. If this is not the case, the set of clauses that the DPLL solver investigates, grows over time. Eventually, the DPLL solver will be provided with enough clauses to find a contradiction and deduce the unsatisfiability of the formula or it can provide a solution for the complete formula.

4.7 Conclusion and Future Work

We have presented a novel and simple approach to analyze and improve the search of an SLS solver. Our approach is based on the construction and use of Search Space Partition (SSP). We defined this new concept, explained how such SSPs are constructed and how they are used. We implemented our novel approach in the hybrid SAT solver *hybridGM*, utilizing *gNovelty+* as the SLS component and *march_ks* as the DPLL component. For an extended analysis we have implemented the solver *hybridGP* and *hybridPP*, which use the SLS solver *gNovelty+* respectively *probSAT* and the CDCL solver *picosat*.

We performed an empirical study to test our approach against *gNovelty+* and *adaptg2wsat0*. Our study revealed that *hybridGM* outperforms *gNovelty+* on $2 + p$ and large size uniform random 3-SAT problems, without experiencing serious losses in other formula categories. We also showed that using a DPLL component to solve the SSPs yields no additional advantage on (the tested) crafted and industrial instances. We have also shown that the performance of a state-of-the-art SLS solver like *probSAT* can be improved on 3-SAT problems with the hybridization scheme.

Several findings of this study leave several questions unanswered. For example, on uniform random 5- and 7-SAT instances, the DPLL component rarely finds a solution. We hypothesize that this is due to the fact that the constructed SSP does not contain all conflicting variables. Moreover, SLS solvers have difficulties to find qualitative local minima for this type of problems.

The way the partial assignments (SSPs) are constructed plays a crucial role for our hybrid solving approach. There are several other ways to construct an SSP, which might also yield interesting results. The size of the SSP is also important and should be further analyzed. Large SSPs can increase the chance to find a solution, but also increase the complexity of the sub-formula and thus the solving time of the DPLL component. The barrier at which an SSP is built can also influence the performance of the solver. An adaptive scheme for the size of the SSP and the barrier might be a promising approach.

Another interesting research direction would be the use of our hybridization concept for survey propagation type solvers. The set of variables that converge during the message passing procedure can be frozen to their polarity. The remaining variables with low convergence or no convergence at all are left undefined. This partial assignment represents the SSP that is passed to the CDCL solver, which is then called to solve the sub-problem. In case of unsatisfiability, the SSP can be extended to other variables.

Acknowledgments We thank the bwGRiD [bhg10] project and especially the local coordinator of the project Christian Mosch. We also thank Marijn Heule for providing some major fixes to *march_ks* and Armin Biere for making available the *picosat* solver.

5 Improving SLS for SAT with a New Probability Distribution

In this chapter, we introduce a new SLS solver for the SAT problem. Our solver is based on *gNovelty+*, which uses an additive clause weighting scheme. When our solver reaches a local minimum, in contrast to *gNovelty+*, it computes a probability distribution for the variables from an unsatisfied clause. Then it flips a variable picked according to this distribution. Compared with other state-of-the-art SLS solvers this distribution needs neither noise nor a random walk to escape efficiently from cycles. We compare our algorithm, which we called *Sparrow* to several state of the art solvers on a broad range of SAT problems. Our results show that *Sparrow* is significantly outperforming all of its competitors on the random k-SAT problem establishing new state-of-the-art performance marks.

Further, we also show that by using appropriate preprocessing techniques the solver is able to solve satisfiable hard combinatorial problems, thus yielding also on this type of problems state-of-the-art performance

This Chapter is mainly based on work we have performed in [BF10], where the solver was presented for the first time. In [BFTH11], we have further analyzed the parameter space of the solver and optimized these for SC11. In [BM13a], *Sparrow* was used with a configured preprocessor and was evaluated on hard combinatorial problems.

5.1 Introduction

Most SLS solvers use different measures in their *pickVar* method. For example, *Novelty* uses the *score* and the *age* of variables (the number of steps since the variable was last flipped). The score of variable x_i is defined as the number of clauses that x_i will satisfy minus the number of clauses that will become unsatisfied by flipping x_i (see

Chapter 3 for more details). To choose a variable, *Novelty* picks a random unsatisfied clause, and then selects the best and the second best variable relative to their score. If the best variable is not the one with the lowest age-value within that clause, then this variable is always chosen. Otherwise, the best variable is chosen with probability $(1-p)$ and the second best variable with probability p. With probability wp a random walk is performed. Neither the difference between the scores nor the age-difference is taken into account, which is a disadvantage in our opinion.

In this chapter, we address this weakness, and improve a state-of-the-art solver like *gNovelty+*. We will replace the *AdaptNovelty* heuristic with a novel heuristic based on a probability distribution that takes into account the difference between the *score* and the *age* of variables. The resulting scheme is not dependent on noise settings and does not need an explicit random walk step to escape from cycles. We implemented these improvements in a solver called *Sparrow*[1]. To show its superior performance, we compare *Sparrow* with the winners of the SAT competitions from 2009 and 2011 on a wide range of k-SAT formulas. We show that *Sparrow* is able to outperform all winners of the SC09 and SC11 competitions.

5.2 Sparrow

The *Sparrow* SAT solver is based on the 2007 version (submitted to the SC07) of *gNovelty+* which is described in more detail in Section 3.6. *gNovelty+*'s *pickVar* heuristic works in four phases that all take into account the weights of the clauses. Phase zero is a random walk step, which is performed with probability 0.01. In the first phase it uses the gradient-walks until there are no more promising variables to flip. This state characterizes a local minimum. To escape from this local minimum, a variable is chosen according to the *AdaptNovelty+* heuristic (see Section 3.3 for more details about this heuristic). Whenever an *AdaptNovelty+* step is performed, the weights of the clauses are updated according to an additive weighting scheme. With probability $(1-sp)$ the weights of the unsatisfied clauses are increased by one and with probability sp the weights of the satisfied clauses are decreased by one. Note that the version submitted to the SC07 is slightly different to the version of the solver, as presented in [PTGS07].

The first two phases of *gNovelty+* without weights are the core of the G^2WSAT algorithms. The performance of this kind of algorithm heavily relies on the *AdaptNovelty+* component [LH05]. We suppose that by introducing a better differentiation the performance of the solver might improve.

Defining a probability distribution One drawback of SLS algorithms using *AdaptNovelty+*-like heuristics to escape from local minima is the lack of differentiation

[1]The sparrow is the emblem of the city of Ulm

between the variables from an unsatisfied clause. While always selecting the best variable in gradient-steps seems to work very well, a more advanced heuristic is needed when a local minimum is reached. We therefore keep the gradient-walk as well as the adaptive weighting scheme, but remove the *AdaptNovelty+* component and replace it with a new heuristic that is based on a probability distribution over the variables from a random unsatisfied clause.

When defining this probability distribution we focus on two aspects: First, we want to keep the features used by *AdaptNovelty*. Second, we want to allow a better differentiation in the heuristic between the variables taken into consideration for flipping.

Let $\{y_1, \ldots, y_u\}$ be the variables from a random unsatisfied clause C of size u. We now define the probability of choosing a variable y_i as

$$p(y_i) := \frac{f_s(y_i) * f_a(y_i)}{\sum_{k=1}^{u} f_s(y_k) * f_a(y_k)} \tag{5.1}$$

where $f_s(y_i)$ is a function of $score(y_i)$ and $f_a(y_i)$ is a function of $age(y_i)$. Recall that the age of a variable is the number of flips since it last changed its value.

We now have the possibility to directly let the *score* and the *age* of a variable influence its probability of getting flipped. This offers a better differentiation between the individual variables than just by deciding whether they have the best *score* or the lowest *age* in the clause. This type of function is similar to the Softmax activation function used in reinforcement learning[2] [SB98].

In particular we chose the following functions for our implementation:

$$f_s(y_i) = c_1^{score(y_i)} \tag{5.2}$$

$$f_a(y_i) = \left(\frac{age(y_i)}{c_3}\right)^{c_2} + 1 \tag{5.3}$$

This yields a probability distribution that grants higher values to variables with higher *score* and *age* (like in *AdaptNovelty+*). As it can easily be seen from the formula, small changes in score have a huge impact on the probability because of the exponential shape of the function. On the other hand the age of a variable only slowly starts to influence the probability, but is also able to have a great impact once an age of c_3 is exceeded. The degree of influence depends on c_2. The parameter *sp* inherited from *gNovelty+* together with c_1, c_2, c_3 represent the parameters of *Sparrow*. Appropriate values for these parameters will be specified in our empirical study.

[2] Reinforcement learning, a sub-topic of machine learning, is a collection of methods concerned with determining actions for agents that act in an environment, such that a given reward measure is optimized.

The resulting distribution exhibits the two properties mentioned before. It behaves similar to *AdaptNovelty* when it comes to preferring variables with best *score* and high *age*. Additionally, it differentiates between how good the *score* and how old the variable is compared to the other variables from the unsatisfied clause. Furthermore, we do not need an explicit implementation of a random walk step since the probability of being flipped is greater zero for all variables in the clause. If the solver would perform in each step only probability based decisions, then the solver would have the PAC property, but this is not the case, as the first type of decisions are those of the gradient heuristic inherited from *gNovelty+*.

Algorithm 5 describes the pseudo-code of *Sparrow*. The major difference to the *gNovelty+* solver (described in Algorithm 3) is the removal of the random walk step and of the *AdaptNovelty* heuristic. The probability based heuristic is described between the lines 9 and 19. First, we select randomly an unsatisfied clause C, and compute the function values for each variable occurring in this clause. By selecting a random position between 0 and the sum of the function values, and then finding the interval where this random position lies, we emulate picking a variable according to the constructed probability distribution. After determining the variable, the solver updates the weights of the variables.

Implementation details In the first implementation of *Sparrow* [BF10], we have adapted the scores of variables with $-10 > score(y_k) > 0$. Variables with a score less than -10 are treated as having $score(y_k) = -10$ and those with score larger than zero as $score(y_k) = 0$. Variables having score greater than zero would have already been selected by the G^2WSAT component if they would have been promising. If they have a positive score but are not promising (*i.e.* the positive score is a result of their own flip when they had a negative score), then their *score* is treated as zero (*i.e.* the function value will be one). To avoid computation of the score function each time a variable is picked we compute at the beginning a lookup table. The *age* function is computed every time on the fly because the age values have a very wide range.

5.3 Empirical Evaluation

Sparrow was empirically analyzed in different publications and also in different competitions. The first and preliminary evaluation was performed in [BF10], where *Sparrow* was presented for the first time and was evaluated solely on 3-SAT problems from the SC09. The second empirical evaluation was performed in [TBH11] and [BFTH11], where parameters of *Sparrow* have been also optimized for 5- and 7-SAT problems. This version of *Sparrow*, with optimized settings, was submitted to the SC11, where *Sparrow* won two gold medals in the random category sequential and parallel track (*Sparrow* is not a parallel solver, but its performance exceeded that of all other parallel

Algorithm 5: *Sparrow* solver

```
Input  : Formula $F$, $maxTries$, $maxFlips$, $sp$, $c_1$,$c_2$,$c_3$
Output: satisfying assignment $\alpha$ or UNKNOWN
1  for $i = 1$ to $maxTries$ do
2      $\alpha \leftarrow$ randomly generated assignment
3      for $j = 1$ to $maxFlips$ do
4          if ($\alpha$ is model for $F$) then
5              return $\alpha$
6          if $\exists$ promising variable then
7              $var$=best promising variable (break ties by age)
8          else
9              $C$ = randomly selected unsatisfied clause
10             $sum = 0$
11             for $k = 1$ to $size(C)$ do
12                 $prob(y_k) = f_s(y_k) \cdot f_a(y_k)$
13                 $sum+ = prob(y_k)$
14             $pos$ = uniform random number in $(0 \dots sum)$
15             for $k = size(C)$ downto 1 do
16                 $sum- = prob(y_k)$
17                 if $sum < pos$ then
18                     $var$=variable at position $k$ in clause $C$
19                     break
20             if (rand()< sp) then
21                 reduce weights of all weighted clauses by 1
22             else
23                 increase weights of all unsatisfied clauses by 1
24         flip($var$)
25         update the promising variable list of $G^2WSAT$
26 return UNKNOWN;
```

solvers, thus also winning the parallel track). Further we have also submitted *Sparrow* to SC12 as a reference solver. All these analysis were performed only on randomly generated problems. The latest analysis of *Sparrow* was performed in [BM13a], where the solver was slightly improved and augmented with the preprocessor *CP3* to be able to solve also hard combinatorial problems.

5.3.1 SC09 Random 3-SAT

To assess the performance of *Sparrow* on randomly generated 3-SAT problems we compare *Sparrow* with the best performing solvers from SC09 random sat category, *TNM*, *gNovelty+* and *hybridGM* (see Chapter 4 for more details on this solver). The solver *TNM* is using a two noise mechanism scheme within a G^2*WSAT* solver. The parameters of *Sparrow* have been determined manually, and only appropriate settings for 3-SAT problems were found. For *Sparrow* we used the following parameter settings for all runs: $sp = 0.35$, $c_1 = 2$, $c_2 = 4$, $c_3 = 10^5$. All evaluations were performed on a cluster of the bwGRiD. See page 123 for technical details.

The Benchmark Problems For our tests, we use two benchmark sets containing randomly generated 3-SAT problems with a clause to variable ratio of $r = 4.2$. They have been generated according to the uniform random generation model presented in Section 2.2.3. The first set contains 64 instances from the random satisfiable large set of SC09 with a number of variables ranging between 2000 and 18000. The second set contains all formulas of the additional benchmark from the same category with a number of variables ranging between 20000 and 26000.

All solvers were run 100 times (50 for the additional benchmark set, due to resource limitations) on each instance and the mean values for the running time and the number of flips were calculated. The time limit was set to 1200 seconds and 2400 for the second benchmark. On the instances on which one of the compared solvers did not finish all 100 (respectively 50) runs within the time limit we plot the number of successful runs.

Solver version For all solvers we have used the code submitted to the SC09. The version of *gNovelty+* submitted to SC09 uses a tabu scheme and we changed the name of the solver in *gNovelty+2T* not to confuse it with the original solver submitted to SC07.

Results The results of our evaluation are represented as scatter plots in Figure 5.1. *TNM* and *hybridGM* are compared to *Sparrow* on the regular benchmark first and on the additional benchmark in the following row. We do not plot the results on the additional benchmark for *gNovelty+2T*, because it has poor performance on these instances.

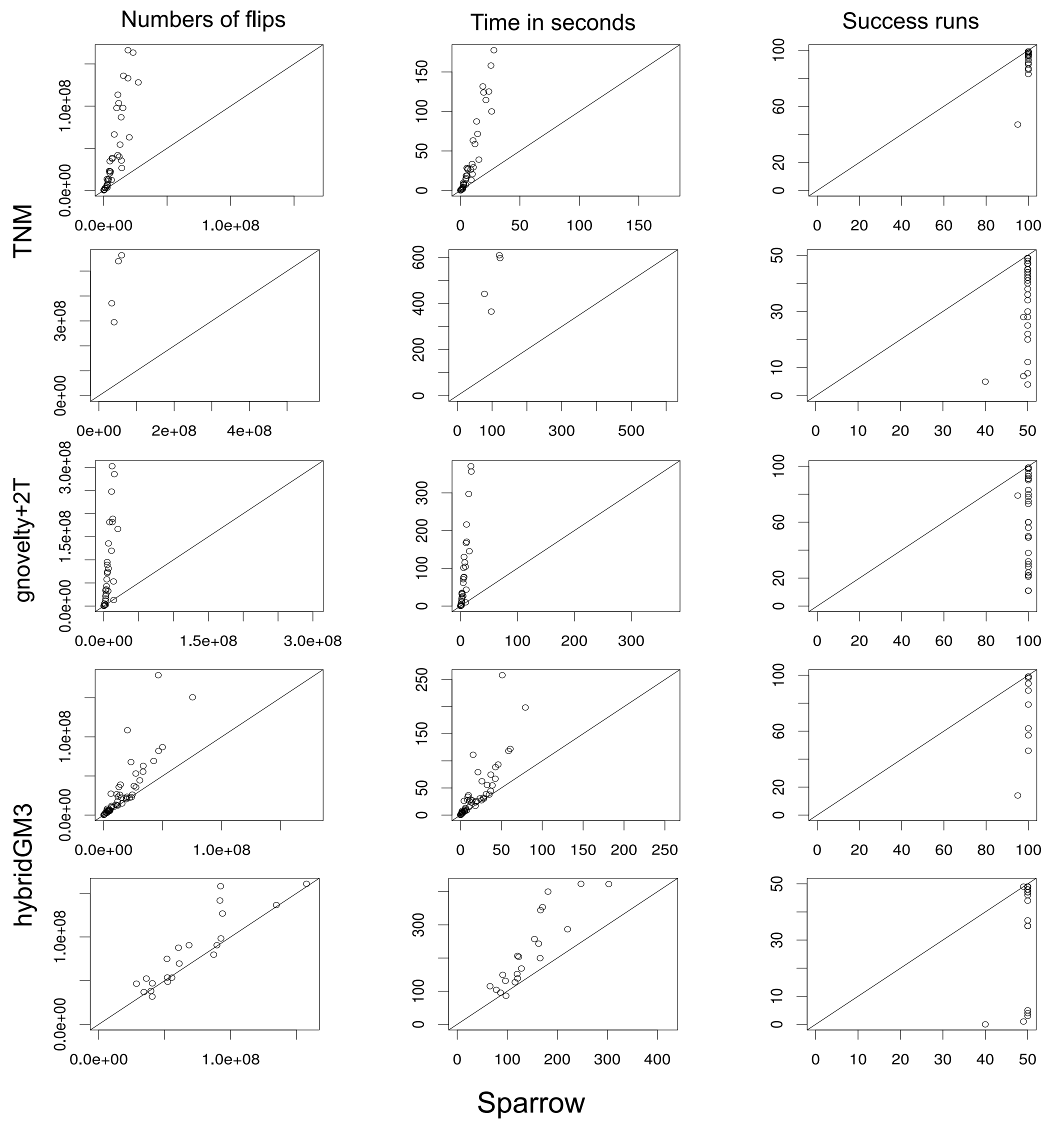


Figure 5.1: *Sparrow* compared to *TNM*, *gNovelty+2T* and *hybridGM* on 104 randomly selected large-size 3SAT-instances. The first column compares the number of flips, the second one the run time measured in seconds. The third column compares the success rate (*i.e.* the number of times the solver was able to solve the problem within the timeout limit). The first two rows compare *Sparrow* to *TNM* on the regular and then on the additional set. The third row compares *Sparrow* with *gNovelty+2T* only on the regular set. The last two rows compare *Sparrow* to *hybridGM* on both instance sets.

In the first two rows, we compare *Sparrow* to *TNM*. The *Sparrow* solver is superior to *TNM* (the winner of the Competition), considering the number of flips as well as considering the run time on all instances. This difference becomes even more pronounced on the additional set on which *TNM* is able to solve only four instances in all 100 runs, whereas *Sparrow* is able to solve almost all instances within the timeout limit.

The next row shows *Sparrow* compared to *gNovelty+2T*. *Sparrow* dominates *gNovelty+2T* on our benchmark in terms of number of flips as well as in terms of run time. The success rates show that many instances of our benchmark are indeed very difficult to solve, even for a state-of-the-art solver.

In the last two rows, we compare *Sparrow* to *hybridGM*. *hybridGM* can compete with *Sparrow* on several of the large instances, but looses ground with increasing difficulty of the instances.

Altogether there are only four instances on which *Sparrow* did not finish all 100 runs within the time limit. Except in one of them the success rate is almost 100%. This is also one of the most difficult instances for all solvers and their success rate was far below the one of *Sparrow*.

By the time these experiments have been conducted, no satisfying parameter configuration for 5- and 7-SAT instances was known, nor could easily be found. Consequently, our first experiments did not contain any other type of instances than 3-SAT.

5.3.2 SC11 Random SAT

During the studies performed in [TBH11] where *Sparrow* was part of the empirical evaluation, parameter settings for (c_1, c_2, c_3, sp) could be found, such that *Sparrow* reaches state-of-the-art performance also on 5- and 7-SAT problems. This motivated further analysis of the parameters by means of automated parameter configuration. *Sparrow* was first reimplemented in the *UBCSAT* [TH04] framework. This version was named *Sparrow2011* (not to be confused with the original implementation). *Sparrow2011* is based on a (preliminary) beta version of *UBCSAT* 1.2 (build 1.2b10). The UBCSAT implementation is semantically equivalent to the original *Sparrow* solver, but is more efficient with respect to CPU time, and provides better reports and statistics for empirical analysis. To find appropriate configurations for the parameters (c_1, c_2, c_3, sp), we used the automated configurator *ParamILS* [HHLBS09, HHS07]. As a basis for the automated configuration process we have generated 100 random 3-, 5- and 7-SAT satisfiable instances with characteristics similar to those used in the SC09 (see Table 5.1). Note that these instances are generated at a clause to variable ratio lower than the threshold values reported in Table 2.1.

For each training set, we performed 24 independent runs of *ParamILS* for four (CPU) days each. The parameter configurations found by *ParamILS* for each instance set were all evaluated on subsets of the instances from SC09 to find the best configu-

k-SAT	number of variables	ratio $r = \frac{m}{n}$
3-SAT	10000	4.2
5-SAT	600	20
7-SAT	100	85

Table 5.1: Characteristics of the training instances generated for the configuration of *Sparrow2011*.

ration. For the evaluation we have used the *EDACC* framework [BGKR10, BDG+11]. For 3-SAT, the best configuration found by *ParamILS* did not perform as well as the manually tuned configuration mentioned previously, so we used the latter instead. Starting from the best configuration for each set, we further hand-tuned the parameters by increasing the granularity of the parameter values and by attempting to interpolate the configurations from one class to the other. This manual tuning took approximately ten hours and was performed using *EDACC*. For the evaluation of the parameter configurations on the different instance classes, *EDACC* used between 300 and 400 CPUs from the bwGRiD computing grid (see page 123 for further details about the cluster).

The parameter settings found by this configuration procedure are listed in Table 5.2. The parameter configuration to be used by *Sparrow2011* on a given input instance is selected based on the maximum clause length found in that instance. The 3-SAT configuration is used when the maximum clause length is less than four, the 5-SAT configuration is used when the maximum clause length is less than six, otherwise the 7-SAT configuration is used. The 5-SAT configuration has a smoothing probability of $sp = 1.0$, which means that no clause weights are being used.

k-SAT	c_1	c_2	c_3	sp
3-SAT	2.15	4	10^5	0.347
5-SAT	2.85	4	$0.75 \cdot 10^5$	1.0
7-SAT	6.5	4	10^5	0.83

Table 5.2: Parameter settings of *Sparrow2011* used in the SC11.

Results *Sparrow2011* won the Random satisfiable track with respect to CPU time and wall clock time. It was able to solve 362 instances within the time limit, while the second best solver *sattime2011* [LL12] solved 334 and the third placed solver *EagleUp* [GH11] solved 328. Note that the *EagleUp* solver is a derivative of *Sparrow* with a unit propagation mechanism. We have evaluated the solver on the SC11 benchmarks on the bwGRiD cluster with the help of *EDACC*. The results are very similar (slightly less solved instances) to those produced during the competition as the hardware was

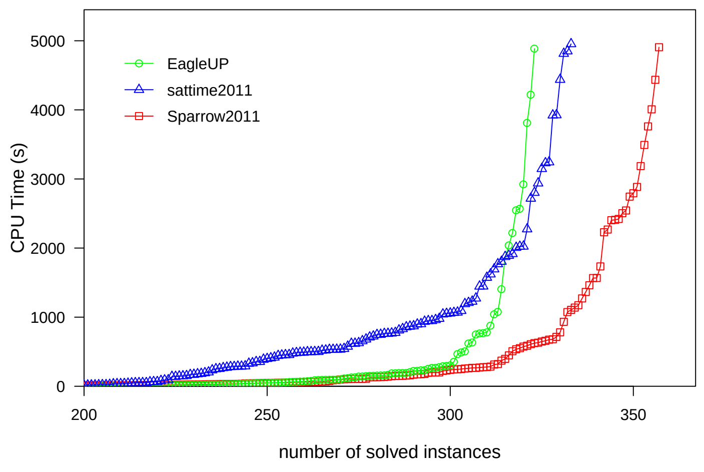


Figure 5.2: Comparison of the SLS solver *Sparrow2011*, *EagleUp* and *sattime2011* on the random satisfiable instances from SC11.

almost similar (slightly slower CPUs). The graphical representation of the results as a cactus plot can be seen in Figure 5.2.

The run time distribution of the solver is not distinguishable for the first 200 instances, as these can be solved in less than one second. While the performance of *EagleUp* is quite similar to that of *Sparrow2011* on the first 300 runs, *EagleUp* is not able to keep up with the other two solvers. We also have to notice that the difference between the best and second best solver was not so pronounced in any of the other tracks organized during the SC11.

5.3.3 SC12 Random SAT

As the author of this thesis was also a co-organizer of SC12 (which did not allow a participation of organizers' solvers), *Sparrow2011* (and also *probSAT*) was submitted to the SC12 Random satisfiable category only as a reference solver without further changes to the code or to the algorithm.

Figure 5.3 shows the run time distribution of the three best performing solvers from SC11 and SC12 on the SC12 random satisfiable benchmark set. Compared to the previous competitions, the SC12 random satisfiable benchmarks (generated according to the scheme presented in [BBJS12]) contained also 4-SAT and 6-SAT instances, for which *Sparrow2011* uses the parameter settings corresponding to 5-SAT respectively 7-SAT. *Sparrow2011* is able to outperform its competitors from SC11 corroborating again its superior performance. The new solvers submitted to SC12, *probSAT* [BS12] and *CCASat* [CS12], have been mainly developed to beat *Sparrow* and they indeed succeed on this benchmark set. The solver *probSAT*, a pure probability distribution

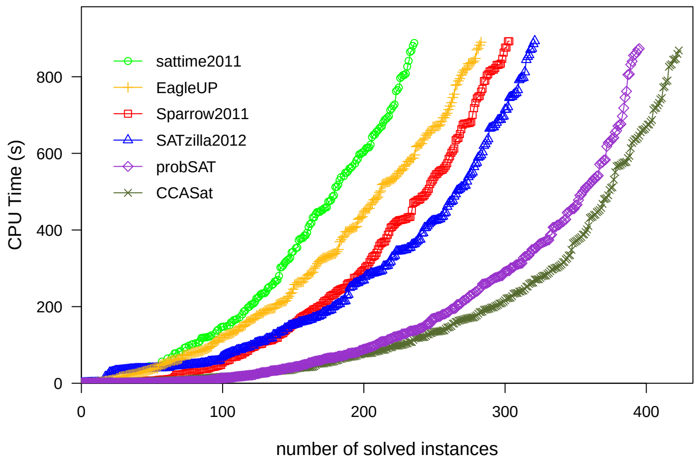


Figure 5.3: Comparison of the top three SLS solvers from the SC11 (*Sparrow*, *sattime2011*, *EagleUp*) and SC12 (*CCASat*, *probSAT*, *SATZilla*) on the SC12 random satisfiable set.

based solver, is described in detail in Chapter 7. It was submitted to the competition also only as a reference solver. The solver *CCASat* is very similar to *Sparrow*, because it is also using clause weights, but instead of using the gradient steps, *CCASat* is using a new technique called configuration checking to avoid flipping the same variables over and over again. Interestingly, the portfolio solver *SATZilla* is using *Sparrow* and *EagleUp* for almost all of its runs and thus takes advantage of the performance of *Sparrow* on lower k-SAT and that of *EagleUp* on higher k-SAT.

5.4 Related Work

There were many attempts to modify the *Novelty* heuristic to increase its performance on different benchmarks. The first time the difference between the score of the variables was taken into account was in the solver *R-Novelty* by McAllester *et al.* in [MSK97]. However, the variables taken into consideration were still only the first and the second best one. The third variable had always probability zero (when considering a 3-SAT problem). The *Novelty++* heuristic by Li [LH05] introduced an additional parameter *dp* (diversification probability) to the *Novelty+* heuristic to enable choosing the least flipped variable from a clause. This permits the heuristic to choose the third variable from a clause, but there is no differentiation between the *score* nor between the *age* of the variables. The solver *TNM* by Wei uses two noise mechanisms and switches between them whenever the weights of variables meet a given criteria. We are not aware of a heuristic that assigns probabilities to all variables depending on the difference between the score and between the age of variables.

5.5 Conclusion and Future Work

We presented in this chapter a probability distribution that is a function of the *score* and the *age* of all variables from a random unsatisfied clause and which takes into consideration the difference between these values. An advantage of such a heuristic is that it needs no noise nor a random walk, which are incorporated by definition. We have conducted several empirical studies on different k-SAT problems from different competitions and showed that our approach exhibits remarkable performance. The *Sparrow* solver was able to significantly exceed the performance of all major state-of-the-art solvers.

It has not been analyzed whether other probability distributions like polynomial or logistic would also be appropriate for use in the computation of the decision heuristic, and how the parameters of these functions should be set to achieve stable performance. Another solver, based on the analysis of *Sparrow* that also uses probability distributions is presented in Chapter 7.

Further, there is also the possibility to create a hybrid solver, like the ones described in Chapter 4, with *Sparrow* as an SLS component.

6 Analyzing the Utility of Preprocessing for SLS Solvers

This Chapter is based on the work presented in [BM13a] where Preprocessing Techniques (PPTs) have been analyzed in combination with SLS and CDCL solvers on hard combinatorial problems or on application problems respectively. Within this chapter, we will present only the work done for SLS solvers on hard combinatorial problems along with the results obtained in the International SAT Competition 2013 (SC13).

6.1 Introduction

As mentioned in Chapter 2, the area of pragmatic SAT solving is dominated by two types of solving techniques: Conflict Driven Clause Learning (CDCL) solvers and Stochastic Local Search (SLS). Each technique has its strength on different types of problems. While CDCL solvers are best suited for structured problems and unsatisfiable crafted problems, SLS solvers exhibit their strength on random and satisfiable crafted problems.

Besides solving techniques, Preprocessing Techniques (PPTs) (meanwhile also used during search and known as *inprocessing*) are crucial for SAT solvers and enable them to further increase performance. PPTs can be seen as transformation rules that take a formula as input and produce a transformed problem, which is satisfiability equivalent to the original one (*i.e.* the original problem is satisfiable if and only if the transformed problem is satisfiable).

Since the introduction of the *SatELite* [EB05] Preprocessor (PP), which is still one of the most used preprocessors in state-of-the-art SAT solvers, many new techniques for preprocessing have been proposed and implemented (*e.g.* [JBH10, MHB12, HJB10a, HJB10b, HJB11]). Although most of these new PPTs have found their way into SAT solvers, the preprocessor *Coprocessor2* [MHB12, Man12] was the first PPT framework like *SatELite* that provided all new techniques as a stand-alone tool. Within this work we use the latest version *Coprocessor3* (*CP3*).

Most of the new PPTs are developed only for CDCL solvers or at least with CDCL solvers in mind, a reason why the parameters of these techniques are predetermined to work well in collaboration with CDCL solvers. This raises up the question whether these techniques (maybe with different parameters) can also be useful to SLS solvers. Some PPTs have been analyzed in combination with SLS solvers [Gab12], still no improvement was detected for these. There is though evidence that some PPTs can help SLS solvers on crafted problems: the SLS solver *sattime2012* [LL12] (which uses failed literal probing and unit propagation as preprocessing) showed remarkable performance on the crafted problems during the latest SAT competitions.

6.2 Preprocessing Utility

In this work we are interested in analyzing the utility of PPTs for SLS solvers as opposed to the main stream where utility is analyzed only with respect to CDCL solvers.

Definition 6.2.1 (PPT utility)**.** A PPT P is considered to be useful (or utile) for a solver S on a set of instances I if the performance of S on I denoted by $\mathit{perf}(S(I))$ can be improved by first executing the preprocessor P on the instances and then running the solver S on the simplified problems $P(I)$, *i.e.* if $\mathit{perf}(S(P(I)) > \mathit{perf}(S(I))$, where *perf* is the statistical measure of interest (*e.g.*, the number of solved instances).

The same run time limitations are imposed to $S(I)$ and $S(P(I))$ (*i.e.* preprocessing time is considered as solving time). Note that the usefulness of a PPT highly depends on the solver S and on the instances I used in the analysis. We are not directly interested in the size of the reduction nor in the structural changes of PPTs, but more in the speed up preprocessing gives for the SAT solver. Therefore, we address the following questions related to the utility of PPTs.

1. *How useful is each PPT on its own?*
2. *Which combination and parametrization of PPTs yields the best improvement?*
3. *How far can the best PPT be improved with appropriate parametrization?*
4. *How sensitive is the performance gain when exchanging solvers?*

To answer these questions, we will use a modern preprocessor that entails all currently available simplification techniques, namely the preprocessor *Coprocessor3*[1].

[1]`http://tools.computational-logic.org`

Coprocessor3 as a PPT framework in combination with the state-of-the-art SLS solver *Sparrow* represents the experimental basis for our analysis. For the evaluation we use the hard combinatorial instances from SC12[2]. Question 1 is answered by evaluating each PPT individually in combination with the solver. To answer question 2 and 3, we use an automated algorithm configuration tool which searches for optimal combinations and parameterizations of PPTs such that the number of solved instances by *Sparrow* when prepended with the preprocessor is maximized. The results obtained are validated with another SLS solver, providing an answer to question 4.

6.3 Modern Preprocessing Techniques

Most modern SAT solvers still use the preprocessor *SatELite* based on work published in 2005 [EB05]. *SatELite* includes the PPTs *Unit Propagation* (also called Boolean Constraint Propagation), *Subsumption*, *Strengthening* (also called *self subsuming resolution*) and *bounded variable elimination.* Since 2005, many other new PPTs have been proposed [JBH10, MHB12, HJB10a, HJB10b, HJB11]. The implementation details or possible variations (parameterizations) of these PPTs are presented in these publications only briefly. Furthermore, the variations of each technique has not been analyzed in detail. In the following we provide a short description, along with possible parameterizations of the PPTs considered in this work. Let F be the input formula to a technique, and F' its output.

Unit Propagation (UP): If there is a unit clause in the formula of the form $C = \{l\} \in F$, then this literal will be set to *true* and the formula will be simplified by removing all occurrences of $\bar{l}$ from all clauses and removing all clauses that contain l (which are now satisfied).

Subsumption (SUB): [Rob65] If there is a clause $C \in F$ that is a subset of another clause $D \in F$, then D is removed.

Strengthening (STR): [EB05] Let D and E be a disjunction of literals. If there exist two (non-tautological) clauses $C_1 = \{l, D\}$ and $C_2 = \{\bar{l}, D, E\}$, then the clause C_2 can be replaced by the resolvent $C_1 \otimes C_2 = \{D, E\}$ which subsumes C_2, thus resulting in the elimination of one literal. STR is also known as self-subsumption in the literature. Within resolution steps, clause C_2 could produce several other resolvents that would also subsume C_2 or would be used elsewhere, however to the best of the author's knowledge, C_2 is replaced immediately by the resolvent. *Coprocessor3* has the option to keep C_2 during strengthening, which is enabled by the parameter *allStrength.* This

[2]`http://baldur.iti.kit.edu/SAT-Challenge-2012/index.html`

extension, though, is only active when the length of the clause is smaller than a certain limit (we have used three in our experiments), because large clauses can produce many resolvents [Bie05].

Bounded Variable Elimination (BVE): [DP60, SP05, EB05] Let S_x be the set of clauses that contain x, and $S_{\overline{x}}$ the set of clauses that contains $\overline{x}$. The set of all pairwise resolvents denoted with S is $S = S_x \otimes S_{\overline{x}} = \{A \otimes B \mid A \in S_x, B \in S_{\overline{x}}\}$. Variable x can be eliminated from the formula F by removing all clauses from S_x and $S_{\overline{x}}$ and adding S to F. The simplified formula F' has the form $F' = (F \setminus (S_x \cup S_{\overline{x}})) \cup S$.

Since the size of S can be very large when compared to the size of the sets S_x and $S_{\overline{x}}$, a limitation has to be imposed to this elimination process in form of a bound. Usually, the number of clauses can be used as a bound: if $|S| \leq |S_x| + |S_{\overline{x}}|$, the sets are replaced. Other limits are possible as well: for example the total number of literals in S has to be smaller than the total number of literals in S_x and $S_{\overline{x}}$. The related parameter in *Coprocessor3* is called *boundLits*. Since it is very unlikely that $|S| \leq |S_x| + |S_{\overline{x}}|$, if both $|S_x|$ and $|S_{\overline{x}}|$ are large, a cutoff can be imposed that does not apply BVE to a variable x if each of the sets has at least ten clauses, or if one set has at least five clauses and the other set contains more than 15 clauses.

Blocked Clause Elimination (BCE): [JBH10] During the computation of BVE, for each clause in S_x and $S_{\overline{x}}$ the number of resolvents is counted. This is needed for the computation of the bound. If a clause does not produce any resolvent, this clause is called *blocked* and can be removed, even if no variable elimination is performed [JBH10]. This technique can be disabled with the parameter *noBce*.

Bounded Variable Addition (BVA): [MHB12] Let x be a new variable that does not occur in F. Let $S \in F$ be a set of clauses, such that there exist two sets of clauses S_x and $S_{\overline{x}}$, and $S_x \otimes S_{\overline{x}} = S$. Opposite to BVE, BVA replaces the set of clauses S with $S_x \cup S_{\overline{x}}$, if $|S| > |S_x| + |S_{\overline{x}}|$. The current implementation in *Coprocessor3* is only able to find simple patterns, where the clauses in S have the form $\{l, D_i\}, \{l', D_i\}$, where D_i is a conjunction of literals and $i > 2$, so that the final set of clauses contains less clauses. The new variable will be added within the clause $x \rightarrow l \wedge l'$. If a set S is found and $i > 4$, another clause $l \wedge l' \rightarrow x$ is added, and further, all clauses C with $\overline{l}, \overline{l'} \in C$ are replaced with $(C \setminus \{\overline{l}, \overline{l'}\}) \cup \{\overline{x}\}$.

Failed Literal Detection (PROBE): [LA97b, Fre95, Ber01, LMS03] Contrary to the depth first search of CDCL solvers, probing (*a.k.a. failed literal detection*) performs a breadth first search to check whether certain literal l assignments leads to a conflict by unit propagation. If a conflict is found, the clause $\{\overline{l}\}$ can be added to the formula and propagated. Furthermore, conflict analysis with the first Unique Implication

Point (UIP) scheme as in the CDCL algorithms could be applied [MMZ$^+$01]. Since probing searches on the top level, also all UIPs could be collected and added to the formula. Furthermore, *double look-ahead* can be used [HvM09]. With the immediate implications of l and $\bar{l}$, *equivalent literals* and *necessary assignments* can be detected [Ber01, HvM09].

Covered Clause Elimination (CCE): [HJB10b] There exist several techniques that allow the addition of literals to a clause. CCE in *Coprocessor3* allows *hidden literal addition* [HJB10a], *asymmetric literal addition* [HJB10a] and *covered literal addition* [HJB10b], which are described in the literature. During the computation of hidden literal addition, or asymmetric literal addition, the procedure can also find failed literals. The final clause C' can either be a tautological clause, or C' can be blocked resulting in a removal of the clause from F according to BCE. Since the computation of C' is very expensive, CCE is only applied to clauses that have a length at least 40 % of the maximum clause length encountered in the formula.

Hidden Tautology Elimination (HTE): [HJB10a] When restricting the computation of the hidden literal addition to binary clauses only, a clause C can still become a tautology. These clauses are removed by HTE.

Equivalent Literal Substitution (ELS): [Gel05] A set of literals that contains equivalent literals can be computed based on the binary implication graph of the binary clauses in the formula [Gel05]. All literals in such a set are replaced with one representative literal of that class.

Unhiding (Unhide): [HJB11] HTE and STR can be approximated based on sampling the binary implication graph of the formula [HJB11]. During sampling the graph, failed literals can be detected. To improve the quality of the approximation, multiple randomized samplings (*iterations*) can be generated. Furthermore, one can choose whether HTE (*UHTE*) or STR (*UHLE*) should be applied based on the sampling.

Ternary Resolution (3RES): [BS92, LA97a, Bac02, BW03] Resolving two ternary or binary clauses or combinations of these can result in another ternary clause, or even in a binary clause. If this is the case, the clause is added to the formula. This technique can be applied before search. The newly created clause can even subsume other clauses in the formula, which can then be removed by SUB.

Add Binary Resolvents (ADD2): [WS02] For SLS solvers, it has been reported that adding redundant binary clauses to the formula can speed up the search of SLS solver. Thus, redundant binary clauses can be generated by performing probing on a literal

l, and then a clause $\{\bar{l}, l'\}$ can be added, if this clause is not already present in the formula, and the literal l' is propagated after assuming l. Since the possible number of these clauses can be very large, the number of added clauses is strongly restricted.

Dense: After performing several simplification techniques, the index of the variables in the formula is not continuous any more. By removing the index gaps, solvers will be able to have a more compact representation of the problem, which results in a more cache friendly and thus faster execution.

Implementation details The standard preprocessor *SatELite* is not processing instances with more than six million clauses. However, we think that applying PPTs on these large instances could also yield speed ups for the solver. Therefore, instead of applying a clause limit for the whole preprocessor, *CP3* provides limits for each technique in terms of the number of considered clauses. In this way, the overall preprocessing time can be controlled, and at the same time the process can be reproduced on any computing system, because it is independent of time. *CP3* also supports parallel BVE that was not used in our experiments.

We think that this kind of limitation is more robust when it comes to the utility to time ratio. In some cases in the SC12, solvers using *SatELite* as a PP timed out because of the very long preprocessing time of *SatELite*.

6.4 Preprocessing Techniques Analysis for SLS Solvers

6.4.1 Single PPT Analysis

To answer our first question, we have measured the utility of each PPT individually, keeping the parameters of each PPT close to the specifications widely used in the literature. The number of steps each PPT is allowed to perform is bounded by a constant, so that search steps can be also performed.

Figure 6.1 presents the result of the evaluation of each individual PPT as a cactus plot. Along with the configuration with the individual PPTs, we also evaluated *Sparrow* without a PP and prepended with *SatELite*. Most of the PPTs are not able to boost the performance of *Sparrow*, performing similar or even worse than *Sparrow*. BVE is the best performing PPT, decreasing the run time of *Sparrow* over the complete set, and allowing it to solve 11 instances more (*Sparrow* solved 209). Independent of the used PPT, the run time of *CP3* was in most cases around five seconds and seldom above ten.

When using *SatELite*, the run time distribution is completely different: up to 300 seconds, this configuration can solve less instances than any other configuration, but later on, it solves 240 instances, while being able to solve challenging instances faster

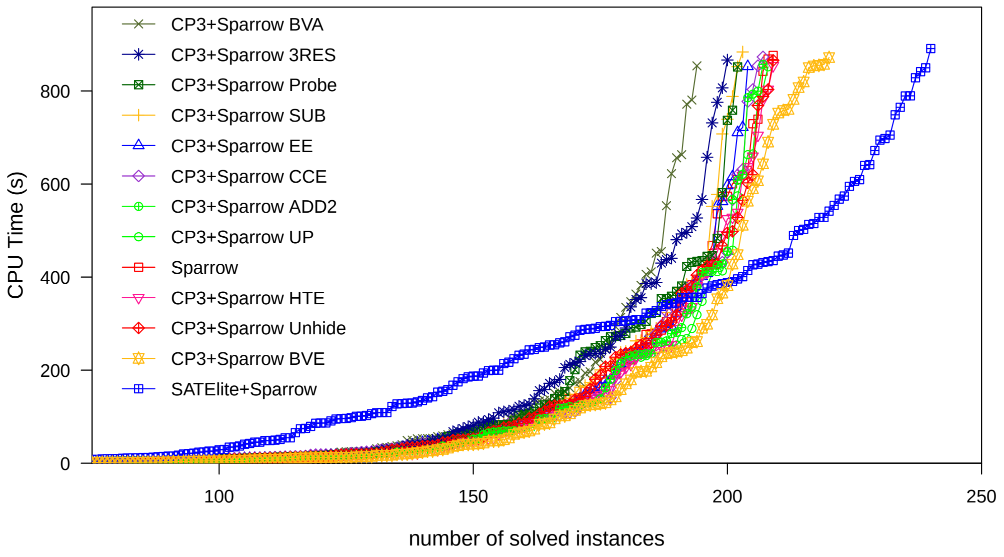


Figure 6.1: Number of solved instances (x-axis) by *Sparrow* alone and when prepended with the different PPTs individually and with *SatELite* on the set of satisfiable hard combinatorial instances from International SAT Challenge 2012 (SC12). The y-axis represents the run time of the solvers measured in seconds.

than by using any other PPT. Compared to our single PPT configurations, *SatELite* is performing a combination of PPTs until completion, which motivates the analysis of combinations of PPTs.

6.4.2 Combined PPTs Analysis

To answer question 2, we have parametrized all PPTs, allowing to turn each technique on and off. When turned on, also other PPT specific parameters can be configured. The PPTs execution order is fixed[3] for all experiments: UP, 3RES, SUB, STR, ELS, Unhide, HTE, PROBE, BVE, BVA, CCE, ADD2 and finally DENSE.

The parametrized version of *CP3* appended with *Sparrow* (which had fixed parameters) was then optimized with a parallel model based automatic algorithm configurator that is a parallel version of the SMAC configurator presented in [HHLB11]. The configurator is implemented in the EDACC framework [BDG$^+$11]. We have performed five configuration experiments with a configuration budget of $2 \cdot 10^6$ seconds and a cutoff of 450 seconds per job optimizing the Penalized Average Run time (PAR)

[3]Allowing the configurator to additionally alter the execution order of the PPT's could have yielded better results, but would have also resulted in a much larger configuration space.

statistic with penalization factor ten (PAR10), which is the sum of the run time of all solved instances plus the number of unsolved instances penalized with ten times the timeout.

Optimizing the PAR10 statistic is almost equivalent to optimizing the number of solved instances, because unsolved instances result in very high penalization. To select the instances for the configuration process, we have first computed for each instance the number of simplification steps performed by the PPTs in the single PPT analysis. From the set of instances, with at least one performed simplification step, we have randomly selected 150 instances.

When searching for the best PPT combination and parametrization, the configurator used only around 100 instances on average before reaching its configuration budget. The best performing configurations from our five configuration experiments were then evaluated on the complete set of available instances. We report the results only for the best combined configuration, though some other configurations performed also relatively good and had similar settings. The best combination of PPTs and parameters found by our configurator had the following settings:

- ELS
- BVA with a very limited step count of 60000
- BVE reducing the number of clauses
- CCE, using asymmetric literal addition only on clauses whose size is larger than 40 % of the maximum clause in the formula, with a high limit (20000000)
- Using binary and ternary clauses for clause vivification[4] on clauses whose size is larger than 40 % of the maximum clause in the formula with a small limit (40000)
- ADD2: When performing probing with binary clauses, per probing literal, add 10 % of the propagation queue length as redundant binary clauses
- Unhide without UHTE and only a single iteration, which can be seen as failed literal detection approximation
- Dense

From Figure 6.2 we can see that the best combined PPT technique denoted in the plot wit *CP3+Sparrow combined* is able to significantly improve *Sparrow*, solving a total of 250 instances from the set, more than the best single engine SLS solver from the SC12.

6.4.3 Extended Single PPT Analysis

The results of the first two experiments raised up the third question, which we tried to answer by further extending the parametrization of the best performing single PPT,

[4]The vivification process, described in [PHS08], is the removal of redundant clause (*i.e.* clauses that can be inferred from the rest of the problem).

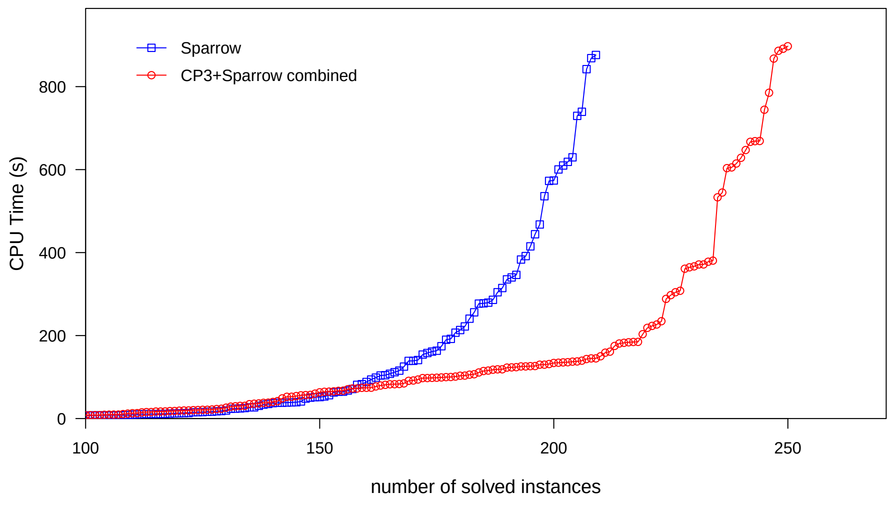


Figure 6.2: Number of solved instances (x-axis) by *Sparrow* alone and when prepended with the best combined PPT on the set of satisfiable hard combinatorial instances from SC12.

namely BVE, and optimizing it in a separate experiment with an optimization budget of $4 \cdot 10^6$ seconds. BVE was extended and parametrized further than proposed in the literature as described next. First, we loosened the limitation of BVE $|S| \leq |S_x| + |S_{\overline{x}}|$ by replacing it with $|S| + b_l \leq |S_x| + |S_{\overline{x}}|$, where b_l is a local limit that has to be met per step (default configuration $b_l = 0$). By setting $b_l > 0$ we allow each variable elimination step to increase the number of clauses in the formula up to b_l additional clauses.

As the number of clauses can now increase within each BVE step, we introduce a limit b_u for the total number of additional clauses. Once this limit has been exceeded, BVE steps will be rejected if they entail an increase in the number of clauses. For an unlimited increase of the number of clauses, this upper bound can be disabled (b_u is enabled by default). We also allow the elimination to take place when $|S|_l \leq |S_x|_l + |S_{\overline{x}}|_l$, where $|\cdot|_l$ measures the number of literals (parameter *red_lits*).

Finally, we have extended BVE with several new heuristics to control which variable is chosen for the next elimination. The variables are stored in a heap that can be sorted according to the number of occurrences and the ratio between positive and negative occurrences, which we will shortly denote simply by polarity ratio:

0. minimum occurrences

1. maximum occurrences

2. random order
3. smallest polarity ratio (in case of ties: sort according to 0)
4. smallest polarity ratio (in case of ties: sort according to 1)
5. largest polarity ratio (in case of ties: sort according to 0)
6. largest polarity ratio (in case of ties: sort according to 1)
7. sort according to 0 (in case of ties: smallest polarity ratio)
8. sort according to 1 (in case of ties: smallest polarity ratio)
9. sort according to 0 (in case of ties: largest polarity ratio)
10. sort according to 1 (in case of ties: largest polarity ratio)

The ratio between positive and negative occurrences has a high impact on the scoring functions used by SLS solvers, and for this reason we have decided to add so many orders that contain this polarity ratio. Suppose we have a formula F, which contains a variable x with $numOcc(x) >> numOcc(\overline{x})$ and that all satisfying assignments of F contains $\overline{x}$, then an SLS solver will probably have a hard time to set the value of x correctly because the scoring function of x will point in the wrong direction.

After performing the optimization of this new extended BVE method, the results were quite surprising:

- no gate detection[5]
- use only SUB but no STR
- sort variables according to the maximum occurrence
- allow the formula to grow per step $b_l = 10$
- allow a total growth of the formula up to $b_u = 1000$ clauses

Consequently, the preprocessed formulas can get larger in terms of clauses, but variables with large occurrences will be eliminated first. This type of BVE configuration is almost contrary to the standard configuration used for CDCL solvers.

In Figure 6.3 we can see that this type of configuration denoted in the plot with CP3+Sparrow ext bve is able to outperform the combined PPT configuration by 11 instances solving a total of 261 instances, which is far more than any single engine solver that participated in SC12. Note that the combined PPT configuration used BVE in a standard manner.

[5]Further details about gate detection can be found in [EB05].

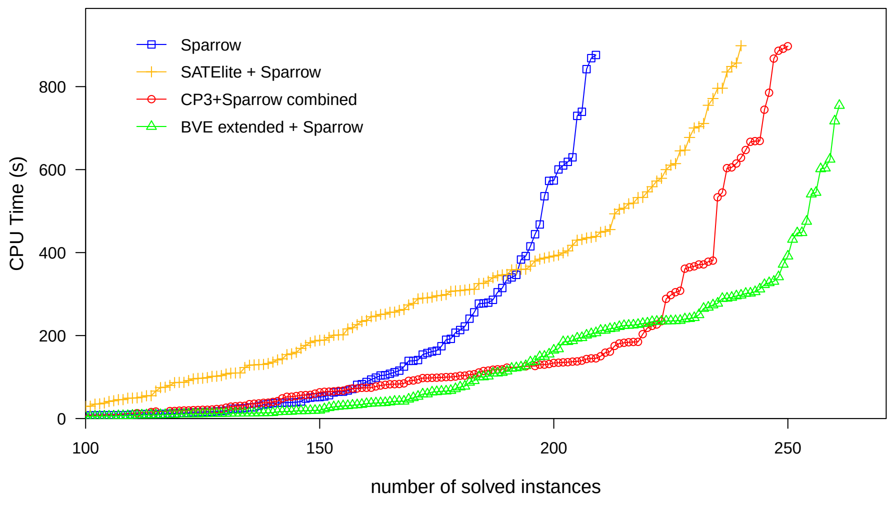


Figure 6.3: Number of solved instances (x-axis) by *Sparrow* alone and when prepended with the best combined PPT, the best BVE configuration and when prepended with *SatELite*, on the set of satisfiable hard combinatorial instances from SC12.

The performance increase is due to two classes of instances present in our evaluation set, the *fsf* and the *prime* instances, which *Sparrow* is barely able to solve without preprocessing. When prepended with the BVE PPT configuration, *Sparrow* solves all these instances quite easily, having run times lower than 300 seconds. None of these instances is solved only by preprocessing.

6.4.4 Applicability to other SLS Solvers

Our fourth question is answered by prepending the best found *CP3* configurations from the previous experiments to another SLS solver and evaluate it on our scenario. This gives us a clue how solver dependent the optimization process in the previous experiments was.

We have combined the best performing *CP3* configuration from our combined and extended single analysis with another SLS solver, namely *sattime2012* [LL12], which was the best performing single engine solver for the hard combinatorial SC12 instance set. The solver *sattime2012* already has an incorporated preprocessor, which performs failed literal probing and unit propagation.

In Figure 6.4 we can see that the performance of *sattime2012* can also be increased by using the combined and the extended BVE configuration, as it was the case for *Sparrow*. The gain in terms of number of solved instances is not as large as it is for *Sparrow*, though is still significant, and decreases the run time of *sattime2012* on a

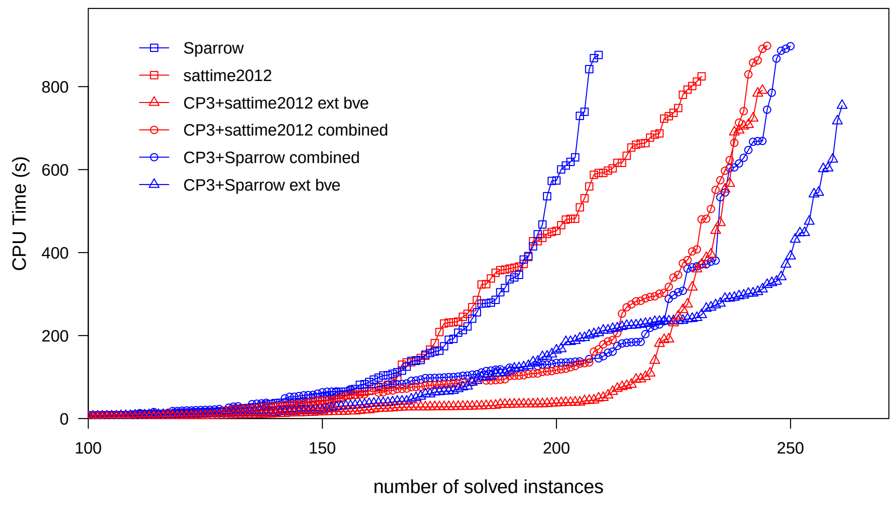


Figure 6.4: Number of solved instances (x-axis) by *Sparrow* and *sattime2012* alone and when prepended with the best combined PPT and the best BVE configuration on the set of satisfiable hard combinatorial instances from SC12.

large set of instances considerably (more than 200 instances can be solved in less than 50 seconds). The difference is significant at a significance value of $p = 0.05$ when tested with the non-parametric Kolmogorow-Smirnow two-sample test.

The solver *sattime2012* can not benefit that much from the BVE configuration because the preprocessing is not able to make those instances solvable which *sattime2012* cannot solve alone, namely the *VanDerWärden* instances. Still, it is worth using another preprocessor before running the internal preprocessor of *sattime2012*.

Software and Hardware. All experiments were performed on the bwGRiD clusters (see page 123 for more details). Experiments were conducted with EDACC, a platform that distributes solver execution on clusters [BDG+11]. The code of *Sparrow* is an improved version of the code used in [BF10].

6.5 CP3+Sparrow

For the International SAT Competition 2013 (SC13), we have submitted the best BVE configuration of *CP3* prepended to *Sparrow*. The set of hard combinatorial problems from SC13 contained 150 satisfiable instances. None of the SLS solvers can solve more than 80 of these, which results in a success rate of around 50%. This is somehow surprising when compared to the success rate of SLS solvers in the SC12 where these solvers solved around 80% of the satisfiable instances. From this, we can conclude that the

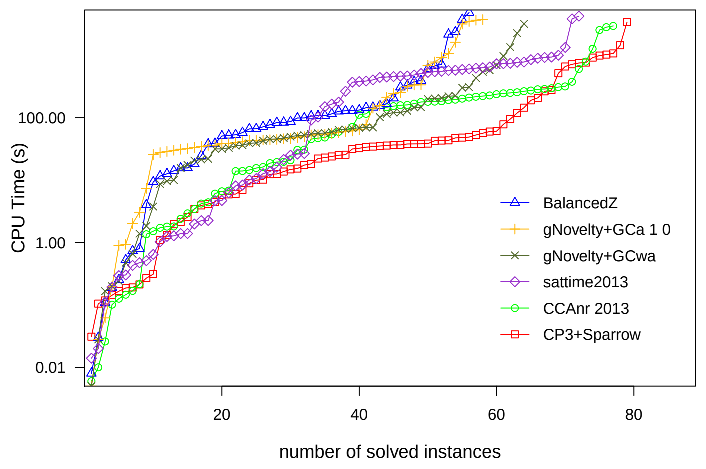


Figure 6.5: Number of solved instances (x-axis) by *Sparrow* when prepended with the best BVE configuration along with other SLS solvers on the set of satisfiable hard combinatorial instances from SC13. The run time (y-axis) is represented on a log-scale.

hard combinatorial problems used in SC13 are favoring CDCL solvers. Consequently, we restrict the presentation of the results to SLS solvers.

The results of SLS solvers that have also participated in the competition can be seen in Figure 6.5. *CP3+Sparrow* is the best performing SLS solver with overall good run times. Note that the run time of SLS solvers is lower than 100 seconds on most problems, which might motivate the analysis of restarts for this class of instances. The diversity of results between the different SLS solvers is rather low, because the number of solved instances by all SLS solvers is 82, which is only three instances more than *CP3+Sparrow* solved. The set of used instances contains around 70% new instances, which were not used in any previous competition, making this set of instances a reliable validation set for our results.

6.6 SparrowToRiss

Several observations made during the evaluation of *CP3* motivated the development of a hybrid solver, which we named *SparrowToRiss*, as it is using *Sparrow* (as an SLS solver) and *Riss* [BBHJ13, p. 72] (as a CDCL solver), which is a variant of *glucose* [AS09]. The key observations are the following:

1. The run time of SLS solvers is relatively low (less than one thousand seconds) and longer run time will not help to solve more problems.

2. There are hard combinatorial instances (satisfiable and unsatisfiable ones) that are not in reach of SLS solvers, though they can be easily solved by CDCL solvers.

As a consequence of these observations we have developed a hybrid solver called *SparrowToRiss* that has the following solving work flow that stops as soon the problem is solved:

1. Preprocess the instance with *CP3* configured with the extended BVE configuration.
2. Run *Sparrow* for $5 \cdot 10^8$ flips; if problem not solved, pass the last used assignment in chronological order to the CDCL solver (*i.e.* oldest variable first).
3. Preprocess the original instance with *CP3* with a configuration optimized for unsatisfiable hard combinatorial problems.
4. Run *Riss* on the preprocessed instances while initializing the phase savings according to the information provided by the SLS.

We have also tried to pass the last assignment ordered according to the activity of the variables (*i.e.* most often flipped variable first), and then initialize the activity of variables in the CDCL solver, though this method is less effective.

We have evaluated *SparrowToRiss* on the hard combinatorial problems from SC12 with and without information flow. The results can be seen in Figure 6.6 where *Sparrow+Riss* denotes the solver without information flow. There is no clear dominance between the solvers. Passing the information can help to solve some instances within timeout, but at the same time it can also hinder the solver to solve other problems (see the scatter plot on the left side). The benefit of this information flow is marginal, as only two more instances can be additionally solved (see cactus plot). However this difference might be decisive in a competition.

Within the SC13 category Hard Combinatorial SAT+UNSAT, *SparrowToRiss* was able to solve 191 instances and was ranked no. 12, solving 17 instances less than the winner. In the category Hard Combinatorial SAT it was able to solve 124 of the 150 instances and ranked first. It is worth noting that the following three best performing solvers followed a similar hybrid approach, though with other SLS and CDCL solvers and other limitations.

6.7 Related Work

Preprocessing for SLS solvers In general, formula simplifications are analyzed when new techniques are presented (*e.g.*, [JBH10, MHB12, HJB10a, HJB10b, HJB11]). The utility of each of these techniques on its own has not been compared yet, and also not

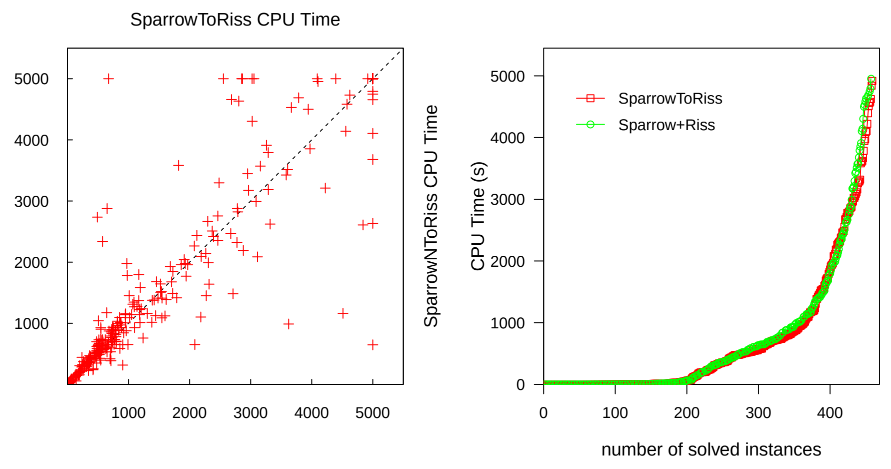


Figure 6.6: Scatter plot (left side) and cactus plot (right side) between *SparrowToRiss* and *Sparrow+Riss* on the set of hard combinatorial problems from SC12 with a cutoff time of 5000 seconds. Points on the 5000 line represent unsolved instances within the scatter plot.

thoroughly analyzed for SLS solvers. The utility of components of CDCL SAT solvers has been analyzed in [KSMS11], though without the analysis of complex combinations nor of PPTs. Furthermore, proposed PPTs are usually not well parametrized, although their implementation can offer many degrees of freedom in form of heuristics or limitations.

Formula simplifications have been combined with SLS solvers already in [WS02], where redundant binary clauses are added to a formula to help an SLS solver simulate unit propagation. Pham analyzed in 2005 in his Ph.D. thesis [Pha06] several preprocessors in combination with several SLS solvers on random and crafted instances coming from the SATLIB benchmarks[6]. He uses the existing preprocessors as black boxes and does not count the simplification time as solving time, neither does he analyze combinations of different PPs. Every PP is used with the default setting (which is probably rather optimal for DPLL solvers and not for SLS solvers).

In [LL12], the SLS solver *sattime2012* is combined with unit propagation and failed literal detection, to improve its performance on crafted instances. In [Gab12], the effect of Covered Clause Elimination (CCE) [HJB10b] techniques with respect to the performance of SLS solvers on crafted instances has been analyzed, showing that this technique family does not improve the performance, which is in line with the results we have obtained.

[6] `www.satlib.org`

Tuning of Preprocessors To the best of our knowledge, automated configuration of PPTs has not been considered so far – however, the power of automated configuration has been demonstrated on a wide range of optimization scenarios, including CDCL and SLS SAT solvers [HBHH07, Hut09]. The parameters of *Sparrow* itself have been optimized using this technique [BFTH11].

6.8 Conclusion and Future Work

In this chapter, we have analyzed the utility of Preprocessing Techniques (PPTs) for SLS solvers on hard combinatorial problems by means of single and combined PPT analysis using automatic algorithm configuration procedures. We showed that the performance of SLS solvers can be drastically improved by using appropriate PPTs (which in our case was BVE) and that the configuration of PPTs strongly differs from configurations used in the literature for CDCL solvers. The best performing PPT configuration for *Sparrow* can be seen as a totally nonstandard PPT for CDCL solvers. The utility of the best found PPT configuration keeps its validity even when exchanging the solver, allowing to increase also the performance of the *sattime2012* solver. Overall PPT and solver combinations found in this work achieve new state-of-the-art performance for SLS solvers on hard combinatorial problems.

We propose two different directions of research motivated by this work. The first one is the improvement of the overall solving performance, by extending the parametrization analysis to the solver parameters, execution order of PPTs and even including the PPTs as inprocessing steps (which is the most challenging but also the most promising area of research). This has been partially done within the Configurable SAT Solver Competition 2013 (CSSC13) where several solvers had parameters that allowed the configuration of the PPT and the solver.

The second direction would be to analyze the structural changes performed by the PPTs on the instances and analyze why they have a positive effect on the solving time of the solvers. In this way, new PPTs could be developed that try to further improve these beneficial changes.

7 Choosing Probability Distributions for Stochastic Local Search and the Role of Make versus Break

Stochastic local search solvers for SAT made a large progress with the introduction of probability distributions like the ones used by the SC11 winners *Sparrow2011* and *EagleUp*. These solvers, though, used a relatively complex decision heuristic where probability distributions played a marginal role.

In this chapter, we analyze a pure and simple probability distribution based solver named *probSAT*, which is probably one of the simplest SLS solvers ever presented. We analyze different functions for the probability distribution for selecting the next flip variable with respect to the performance of the solver. Further we also analyze the role of *make* and *break* within the definition of these probability distributions, and show that the general definition of the *score* improvement by flipping a variable as *make* minus *break* is questionable. By empirical evaluations, we show that the performance of our new algorithm exceeds that of the SAT Competition winners by one order of magnitude.

This chapter contains work published in [BS12, BS13], which was extended in several directions.

7.1 Introduction

Stochastic Local Search (SLS) solvers operate on complete assignments and try to find a solution by flipping variables according to a given heuristic. Most SLS solvers are based on the following scheme: Initially, a random assignment is chosen. If the formula is satisfied by the assignment, the solution is found. If not, a variable is chosen according to a (possibly probabilistic) variable selection heuristic, which is called *pickVar*. The heuristics mostly depend on some score, which is based on the number of satisfied/unsatisfied clauses, as well as other aspects like the "age" of variables. It was believed that capable flip heuristics should be designed in a very sophisticated way to obtain a really efficient solver. We show in the following that it pays off to "come back to the roots", since a very elementary and (as we think) elegant design principle for the *pickVar* heuristic just based on probability distributions performs extraordinary well.

It is especially popular (and successful) to pick the flip variable from an unsatisfied clause. This is called *focused* local search in [Pap91] (see Chapter 2 for more details). In each round, the selected variable is flipped and the process starts over again until a solution is eventually found.

Most important for the flip heuristic seems to be the *score* of an assignment, *i.e.* the number of satisfied clauses. Considering the process of flipping one variable, we get the *relative score change* produced by a candidate variable for flipping as: (*score after flipping* minus *score before flipping*) which is equal to *make* minus *break* (see Chapter 2 for more details).

Most of the SLS solvers so far, if not all, follow the strategy that whenever the score improves by flipping a certain variable from an unsatisfied clause, they will indeed flip this variable without referring to probabilistic decisions. Only if no improvement is possible, as it is the case in local minima or if there is a tie, a probabilistic strategy is performed. The winner of the SC11 category random SAT, *Sparrow*, mainly follows this strategy, though when it comes to a probabilistic strategy it uses a probability distribution function [BF10] (see Chapter 5). The probability distribution in *Sparrow* is defined as an exponential function over the *score* value. In this chapter, we analyze several simple SLS solvers that use only probability distributions within their search.

Before describing our new algorithm, we provide a short description of its development process. Starting with the solver *Sparrow*, we tried to understand the importance of its components by disabling one component at a time, and performing a configuration of the parameters of the remaining components' parameters with automated configuration tools. First we removed the clause weighting scheme from *Sparrow*. After configuration, we obtained parameter settings with similar performance to the original code. As a next step, we removed the gradient based steps. For this we had to adapt the probability distribution, because in *Sparrow* only variables with negative score were taken into consideration within the probability distribution step. Variables with a positive score would have already been picked by the gradient step. We introduced a new parameter for the positive score variables and then optimized the new parameters. We were again able to find appropriate parameter settings for the solver to match the performance of the old solver. By refining the definition of the probability distribution, we ended up with the solver that will be presented next.

7.2 The New Algorithm Paradigm

We propose a new class of solvers here, called *probSAT*, which base their probability distributions for selecting the next flip variable solely on the *make* and *break* values, though not necessarily on the value of the *score* = *make* − *break*, as it was the case in *Sparrow*. Our experiments indicate that the influence of *make* should be kept rather weak – it is even reasonable to ignore *make* completely, like in implementations of *WalkSAT*. The role of *make* and *break* in these SLS-type algorithms should be seen in a new light. The new type of algorithm presented here can also be applied for general constraint satisfaction problems and works as follows:

Algorithm 6: ProbSAT

```
   Input  : Formula $F$, $maxTries$, $maxFlips$
   Output: satisfying assignment $\alpha$ or UNKNOWN
1  for $i = 1$ to $maxTries$ do
2      $\alpha \leftarrow$ randomly generated assignment
3      for $j = 1$ to $maxFlips$ do
4          if ($\alpha$ is model for $F$) then
5              return $\alpha$
6          $C_u \leftarrow$ randomly selected unsatisfiable clause
7          for $x$ in $C_u$ do
8              compute $f(x, \alpha)$
9          $var \leftarrow$ random variable $x$ according to probability $\frac{f(x,\alpha)}{\sum_{z \in C_u} f(z,\alpha)}$
10         flip($var$)
11 return UNKNOWN;
```

The idea is that the function f should give a high value to variable x, if flipping x seems to be advantageous, and a low value otherwise. Using f, the probability distribution for the potential flip variables is calculated. The flip probability for x is proportional to $f(x, \alpha)$. Setting f as a constant function leads in the k-SAT case to the probabilities $(\frac{1}{k}, \ldots, \frac{1}{k})$ morphing the *probSAT* algorithm to the random walk algorithm that is theoretically analyzed in [Sch99]. In all our experiments with various functions f we made f depend only on *break*(x, α) and possibly on *make*(x, α), but no other properties of x and α. In the following, we analyze experimentally the effect of several functions to be plugged in for f. The way probabilities are computed from the f values is similar to the *Softmax* activation function used in reinforcement learning (see page 51).

7.2.1 An Exponential Function

First, we considered an exponential decay 2-parameter function:

$$f(x, \alpha) = \frac{(c_m)^{make(x,\alpha)}}{(c_b)^{break(x,\alpha)}}$$

The parameters of the function are c_b and c_m. The exponential functions used here (think of $c^x = e^{\frac{1}{T}x}$) remind of the way the Metropolis Focused Search algorithm presented in [SAO05] selects a variable. We call this the *exp-algorithm*. The separation into the two base constants c_m and c_b will allow us to find out whether there is a different influence of the *make* and the *break* value – and there is one, indeed.

It seems reasonable to try to maximize *make* and to minimize *break*. Therefore, we expect $c_m > 1$ and $c_b > 1$ to be suitable choices for these parameters. Actually, one might expect that c_m should be identical to c_b such that the above formula simplifies to $c^{make-break} = c^{score}$ for an appropriate parameter c.

To get a picture on how the performance of the solver varies for different values of c_m and c_b, we performed a uniform sampling of $c_b \in [1.0, 4.0]$ and of $c_m \in [0.1, 2.0]$ for this exponential function and of $c_m \in [-1.0, 1.0]$ for the polynomial function (see below). Then we run the solver with the different parameter settings on a set of randomly generated 3-SAT instances with 1000 variables at a clause to variable ratio of 4.26. The cutoff limit was set to 10 seconds. As a performance measure we use PAR10: penalized average run time, where a timeout of the solver is penalized with 10·(cutoff limit). A parameter setting where the solver is not able to solve anything has a PAR10 value of 100 in our case.

In the case of 3-SAT, a good choice of the parameters is $c_b > 1$ (as expected) and $c_m < 1$ (totally unexpected), for example, $c_b = 3.6$ and $c_m = 0.5$ (see Figure 1 left upper diagram and the survey in Table 7.2) with small variation depending on the considered set of benchmarks. In the interval $c_m \in [0.3, 1.8]$ the optimal choice of parameters can be described by the hyperbola-like function $(c_b - 1.3) \cdot c_m = 1.1$. Almost optimal results were also obtained if c_m is set to 1 (and c_b to 2.5), see Figure 1, both upper diagrams. In other words, the value of *make* is not taken into account in this case.

As mentioned, it turns out that the influence of *make* is rather weak, therefore it is reasonable and still leads to considerable algorithms if we ignore the *make* value completely– also because the implementation is simpler and has less overhead – and consider the one-parameter function:

$$f(x, \alpha) = (c_b)^{-break(x,\alpha)}$$

We call this the *break-only-exp-algorithm.*

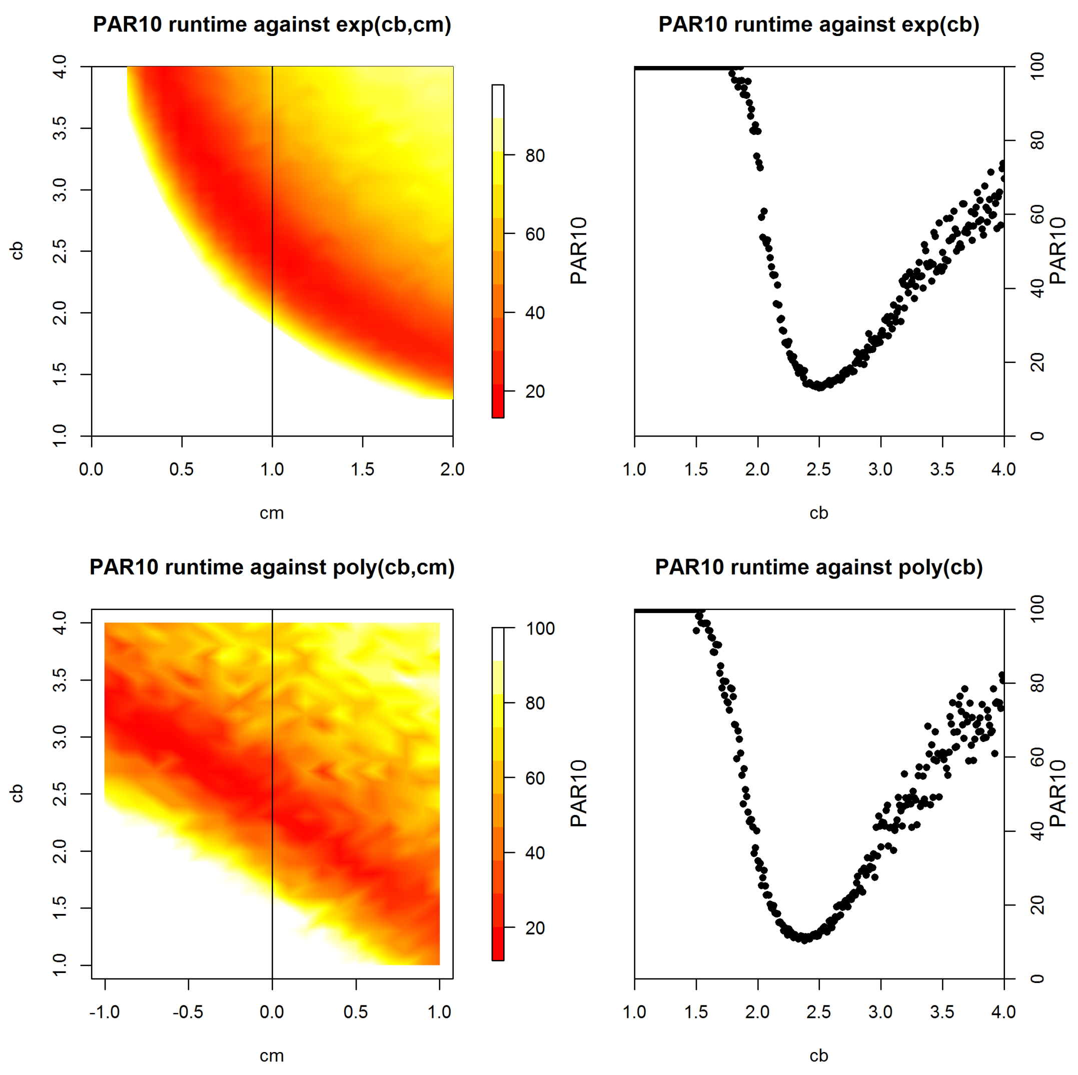


Figure 7.1: The left plots show the performance of different combinations of c_b and c_m for the exponential (upper left corner) and the polynomial (lower left corner) functions. The darker the area the better the run time of the solver with that particular parameter settings. The right plots show the performance variation if we ignore the *make* values (corresponding to the cut in the left plots), by setting $c_m = 1$ for the exponential function, and $c_m = 0$ for the polynomial function.

7.2.2 A Polynomial Function

Our experiments showed that the exponential decay in probability with growing *break* value might be too strong in the case of 3-SAT. The above mentioned formulas have an exponential decay in probability comparing different *break* values. The relative decay is the same if we compare $break = 0$ with $break = 1$, and if we compare, say, $break = 5$ with $break = 6$. A "smoother" function for high values is a polynomial decay function. This led us to consider the following 2-parameter function ($\epsilon = 1$ if not otherwise mentioned):

$$f(x, \alpha) = \frac{(make(x, \alpha))^{c_m}}{(\epsilon + break(x, \alpha))^{c_b}}$$

We call this the *poly-algorithm.* The best parameters in case of 3-SAT proved to be $c_m = -0.8$ (notice the minus sign!) and $c_b = 3.1$ (see Figure 7.1, lower part). In the interval $c_m \in [-1.0, 1.0]$, the optimal choice of parameters can be described by the linear function $c_b + 0.9c_m = 2.3$. Without harm one can set $c_m = 0$, and then take $c_b = 2.3$, and thus ignore the *make* value completely.

Ignoring the *make* value (*i.e.* setting $c_m = 0$) results in the function

$$f(x, \alpha) = (\epsilon + break(x, \alpha))^{-c_b}$$

We call this the *break-only-poly-algorithm.*

Some Remarks As mentioned above, in both the exp- and the poly-algorithm, it was possible to ignore the *make* value completely (by setting $c_m = 1$ in the exp-algorithm, or by setting $c_m = 0$ in the poly-algorithm). This corresponds to the vertical lines in the left diagrams of Figure 7.1.

If we use the *make* values within the functions, then the optimal choice is to set $c_m = 0.5$ and $c_b = 3.6$ for the exp-algorithm (and similarly for the poly-algorithm.) We have $\frac{0.5^{make}}{3.6^{break}} \approx 3.6^{-(break+make/2)}$. This can be interpreted as follows: instead of the usual $score = make - break$ a better score measure is $-(break + make/2)$.

The value of c_b determines the greediness of the algorithm. We concentrate on c_b in this discussion since it seems to be the more important parameter. The higher the value of c_b, the more greedy is the algorithm. A low value of c_b (in the extreme, $c_b = 1$ in the exp-algorithm) changes the algorithm to a random walk algorithm with flip probabilities $(\frac{1}{k}, \dots, \frac{1}{k})$ like the one considered in [Sch99]. Examining Figure 7.3, almost a phase-transition can be observed. If c_b falls under some critical value, like 2.0, the expected run time increases tremendously. Turning towards the other side of the scale, increasing the value of c_b (*i.e.* making the algorithm more greedy) also degrades the performance, though not with such an abrupt rise of the run time as in the other case. These observations have also been made in [KSS10] and in [Hoo02].

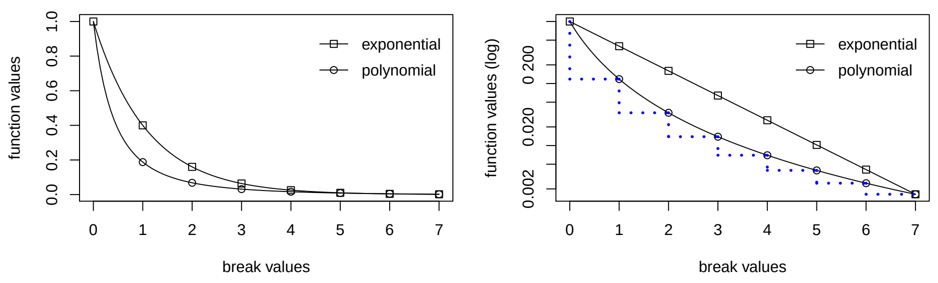


Figure 7.2: Comparison of the exponential function $(2.5)^{-break(x,\alpha)}$ and the polynomial function $(1 + break(x, \alpha))^{-2.38}$ on a linear scale (left plot) and on a logarithmic scale (right plot). The c_b values are the ones determined to be optimal for the 3sat1k instance set. The dotted line in the right plot represents the step function of the polynomial function.

7.2.3 Comparison of the functions

To visualize the difference between the polynomial and exponential functions in case of the *break-only* algorithms, we have plotted the two functions in Figure 7.2 for optimal c_b values (for the 3sat1k instances) and typical *break* values, which occur in 3-SAT problems (*i.e.* $break(x, \alpha) \in [0 \ldots 7]$). On a linear scale (Figure 7.2 left side), the difference between these two functions is notable only for $break \in \{1, 2, 3\}$. For the other *break* values the functions produce similar values.

The relative decay in function values for two consequent *break* values (*e.g.*, $break =$ 3 and $break = 4$) is constant in case of the exponential function (see right plot, which has a logarithmic y-axis). In case of the polynomial function, the function decay is getting smaller with increasing break values (see dotted step function in the right lower part of the plot). Consequently the polynomial function looses its differentiation potential with increasing break values.

For randomly generated k-SAT problems, we can compute the expected range of the *break* values by approximating the number of occurrences of a literal within the problem. A randomly generated k-SAT problem contains $k \cdot m$ literals, where the number of clauses is $m = r \cdot n$. There can be only $2 \cdot n$ distinct literals in the formula. Consequently, the average number of occurrences of a literal is:

$$\frac{k \cdot m}{2n} = \frac{k \cdot r \cdot n}{2n} = \frac{k \cdot r}{2}$$

The *break* value of a variable $break(x, \alpha)$ is bounded by the number of occurrences of the literals of x, which is on average $\frac{k \cdot r}{2}$. Table 7.1 shows the maximum expected *break* values for different k values at ratios reported in Table 2.1. Note that these maximal values can only occur when a literal l is the only satisfiable literal in b_{max} clauses.

k	3	4	5	6	7
max. expected *break* $b_{max} = k \cdot r_k/2$	6.30	19.80	52.75	129.90	307.30

Table 7.1: Maximum expected *break* values for different k values.

However, this is very unlikely, because the average number of true literals within a clause in a randomly generated k-SAT problem is $k/2$.

7.3 Experimental Analysis of the Functions

To determine the performance of our probability distribution based solver, we have designed a wide variety of experiments. In the first part of our experiments we try to determine appropriate settings for the parameters c_b and c_m by means of automatic algorithm configuration procedures. In the second part we will compare our solver to other state-of-the-art solvers.

The Benchmark Problems All random instances used in our settings are uniform random k-SAT problems generated with different clause to variable ratios (see Chapter 2). The class of random 3-SAT problems is the best studied class of random problems and for this reason we have four different sets of 3-SAT instances.

1. 3sat1k[TBH11]: 10^3 variables at $r = 4.26$ (500 instances)
2. 3sat10k[TBH11]: 10^4 variables at $r = 4.2$ (500 instances)
3. 3satComp[1]: all large 3-SAT instances from the SAT Competition 2011 category random with variables range $2 \cdot 10^3 \ldots 5 \cdot 10^4$ at $r = 4.2$ (100 instances)
4. 3satExtreme: $10^5 \ldots 5 \cdot 10^5$ variables at $r = 4.2$ (180 instances)

The 5-SAT and 7-SAT problems used in our experiments come from [TBH11]: 5sat500 (500 variables at $r = 20$) and 7sat90 (90 variables at $r = 85$). The 3sat1k, 3sat10k, 5sat500 and 7sat90 instance classes are randomly split into two equal sized classes called train and test. The train set is used to determine parameters for c_b and c_m and the test set is used to report the performance. Further, we also include the set of satisfiable random and crafted instances from SC11.

Parameter Setting The problem that every solver designer is confronted with is the determination of appropriate parameters for its solver. We have avoided to accomplish this task by manual tuning, but instead have used an automatic procedure.

[1] `http://www.cril.univ-artois.fr/SAT11/bench/SAT11-Competition-SelectedBenchmarks.tar`

	3sat1k		3sat10k		5sat500		7sat90	
$exp(c_b, c_m)$	3.6	0.5	3.97	0.3	3.1	1.3	3.2	1.4
$poly(c_b, c_m)$	3.1	-0.8	2.86	-0.81	-		-	
$exp(c_b)$	2.50		2.33		3.6		4.4	
$poly(c_b)$	2.38		2.16		-		-	

Table 7.2: Each cell represents a setting for c_b and c_m dependent on the function used by the solver and the type of the problems. Parameter values close to these values have similar performance.

As our parameter search space is relatively small, we have opted to use a modified version of the iterated *F-Race* [BYBS10] configurator, which we have implemented in Java. The *F-Race* configurator starts with a set of randomly selected parameter configurations that are initially evaluated on a small set of instances. To determine if there is a significant difference of performance between the configurations, *F-Race* performs a Friedman test[2] (see Test 25 in [She07] for more details about the test) to check if there is a significant performance difference between the configurations. If there is no significant difference, the solvers are further evaluated and the test is applied again. The test is conducted every time the solvers have been run on an instance. If the test is positive, poor configurations (determined with a post-hoc test) are dropped, and only the remaining ones are further evaluated. The configurator ends when the number of solvers left in the race is less than 2 times the number of parameters, or if there are no more instances to evaluate on.

To determine appropriate values for c_b and c_m we run our modified version of *F-Race* on the training sets 3sat1k, 3sat10k, 5sat500 and 7sat90. The cutoff time for the solvers were set to 10 seconds for 3sat1k and to 100 seconds for the rest. The best parameter values returned by this procedure are listed in Table 7.2. The polynomial function had very poor performance in our preliminary tests and was not further analyzed with the configurator. This could be probably due to the weak differentiation potential of the polynomial function for increasing *break* values. The parameter settings for the 3sat1k problems were also included, because the preliminary analysis of the parameter search space was done on this class. The best parameter of the break-only-exp-algorithm for k-SAT can be roughly described by the formula $c_b = k^{0.8}$, thus $f(x, \alpha) = k^{-0.8 \cdot break(x,\alpha)}$.

For the 3sat10k instance set, the parameter space performance plots in Figure 7.3 looks similar to that of 3sat1k (Figure 7.1), though the area with good configurations is more narrow, which can be explained by the short cutoff limit of 100 seconds used for this class (SLS solvers from SC11 had an average run time of 180 seconds on this

[2]The Friedman test is a non-parametric test for complete block designs. It tests whether there is a difference between different treatments (measurements) with respect to some given summary statistics.

type of instances).

In case of 5sat500 and 7sat90, we have opted to analyze only the exponential function, because the polynomial function, other than in the 3-SAT case, exhibited poor performance on these sets. Figure 7.4 shows the parameter space performance plot for the 5sat500 and 7sat90 sets. When comparing these plots with those for 3-SAT, the area with good configurations is much larger. Or in other words, the performance of the solver is not very sensitive with respect to the values of the parameters. For the 7-SAT instances, the promising area seems to take almost half of the parameter space. The performance curve of the break-exp-only algorithm is also wider than that of 3-SAT, and in the case of 7-SAT no clear curve is recognizable.

7.4 Empirical Evaluation

In the second part of our experiments we compare the performance of our solvers to that of the SC11 winners and also to *WalkSAT*. An additional comparison to a survey propagation algorithm will show how successful our *probSAT* local search solver can get. We have submitted different *probSAT* implementations to all major SAT competitions, for which we will also present the results.

Soft- and Hardware The solver *probSAT* is implemented in C and is available online[3]. The selection of a random number according to the generated probability distribution has been described within the *Sparrow* solver (see Algorithm 5 on p. 53). This procedure can be improved by binary search if the size of the clause is large, as could be the case for structured problems.

Details about the used hardware and software can be found on page 123.

The competitors The *WalkSAT* solver is implemented within our own code basis. We use our own implementation and not the original code (version 48) provided by Henry Kautz[4], because our implementation is approximately 1.35 times faster.[5]

We have used version 1.4 of the survey propagation solver provided by Zecchina[6], which was changed to be DIMACS conform. For all other solvers we have used the binaries from SC11[7].

[3]http://www.uni-ulm.de/in/theo/m/balint.html
[4]`http://www.cs.rochester.edu/u/kautz/walksat/`
[5]The latest version 50 of *WalkSAT* has been improved by 20%, but was not available at the time we have performed the experiments
[6]`http://users.ictp.it/~zecchina/SP/`
[7]`http://www.cril.univ-artois.fr/SAT11/solvers/SAT2011-static-binaries.tar.gz`

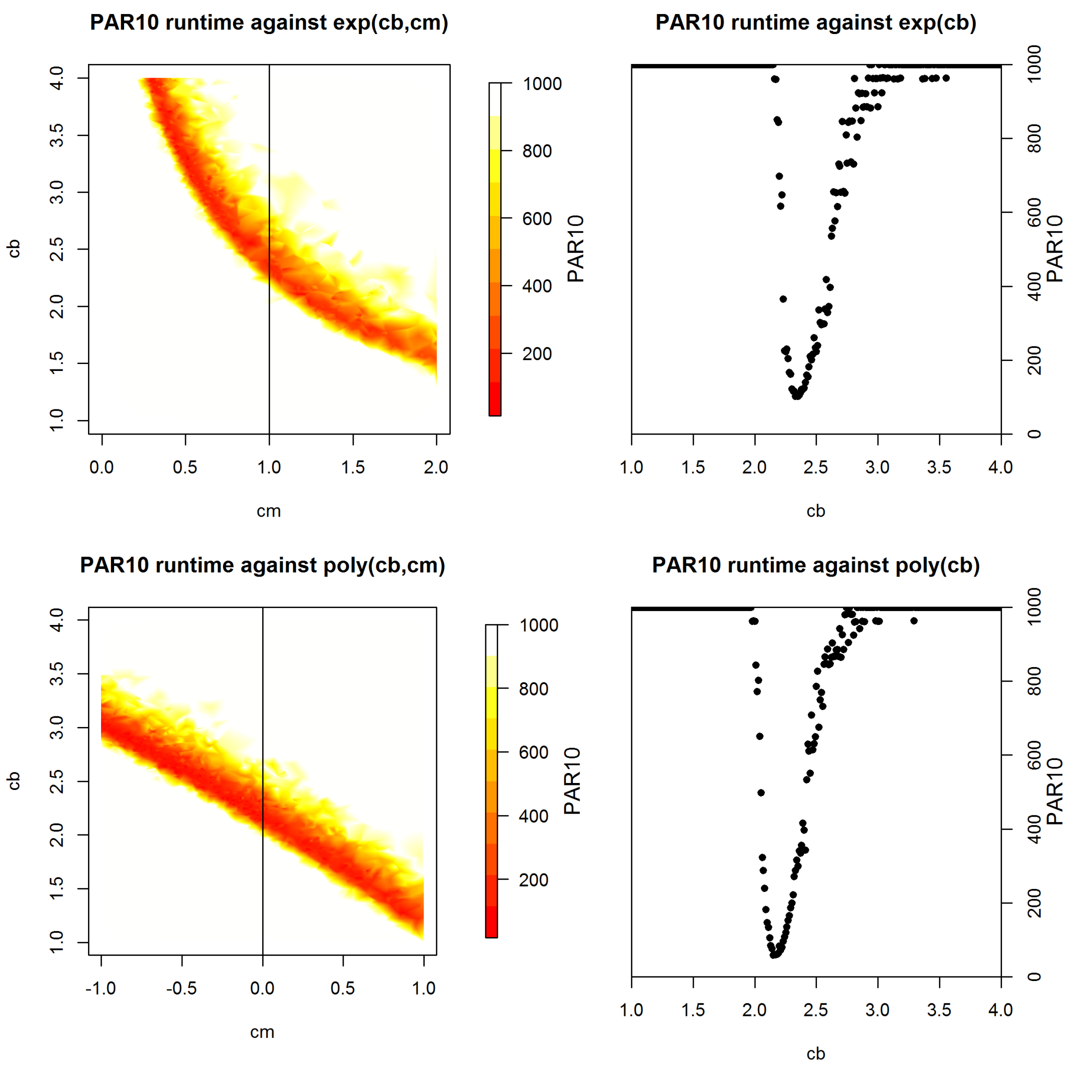


Figure 7.3: The run time of the solver using different functions and varying c_b and c_m on the 3sat10k instances set. The cut through the left plot represents the plot on the right side (*i.e.* $cm = 1$).

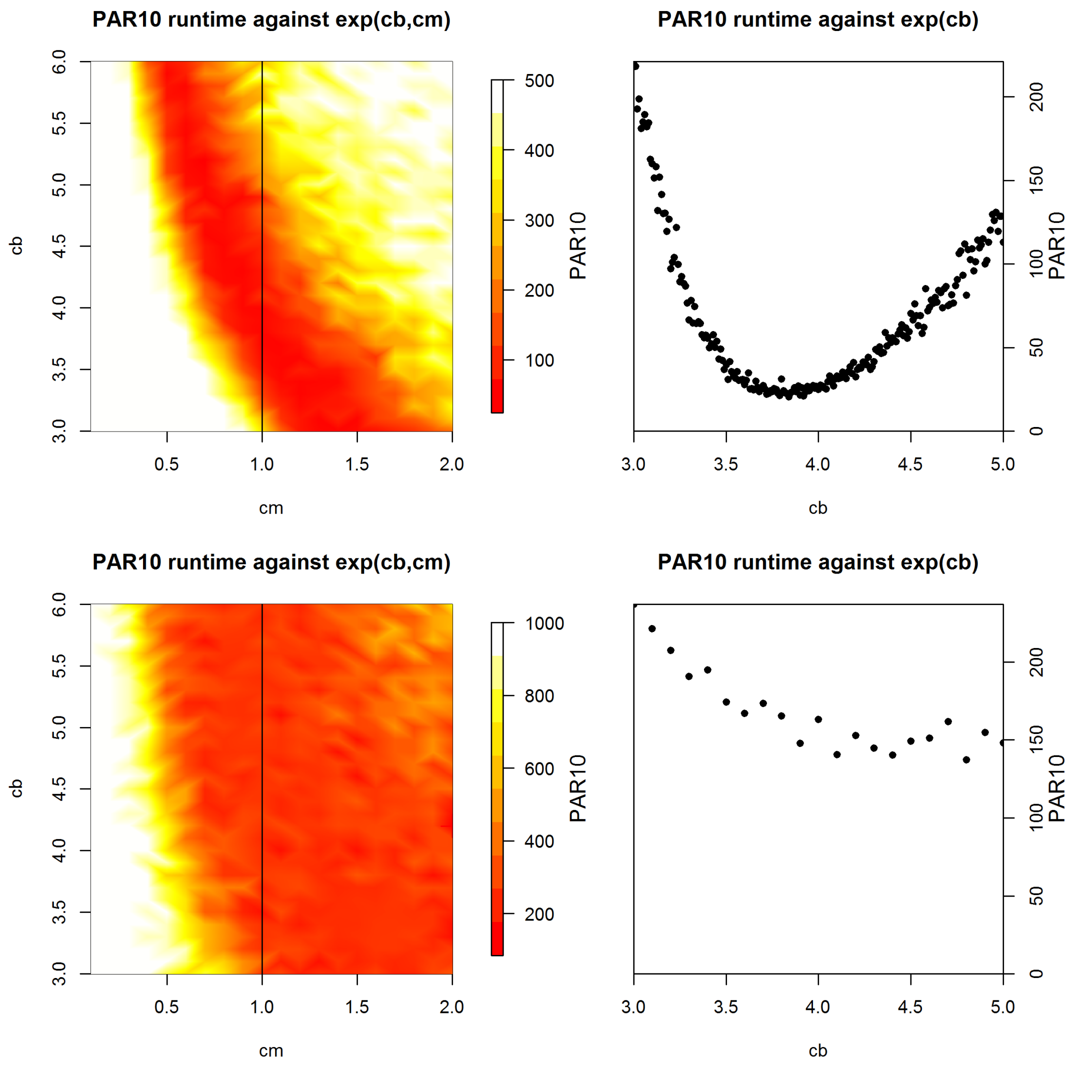


Figure 7.4: The run time of the exp-solvers with different functions and varying c_b and c_m on the 5sat500 instances at the top, and on the 7sat90 instances at the bottom.

Parameter Settings of Competitors *Sparrow* is highly tuned on our target set of instances and incorporates optimal settings for each set within its code. *WalkSAT* has only one single parameter, the noise probability p (see Section 3.3). In case of 3-SAT, we took the optimal values for $p = 0.567$ which have been established in an extensive analysis in [KSS10]. Since we did not find any settings for 5-SAT and 7-SAT problems, we run our modified version of *F-Race* to find suitable settings. For 5sat500 the configurator reported $p = 0.25$, and for 7sat90 $p = 0.1$. This result is very surprising. Such a low value for p means that *WalkSAT* will choose in almost all cases the best variable from an unsatisfied clause, thus being much more greedy than in the case of 3-SAT problems.

The survey propagation solver was evaluated with the default settings reported in [BMZ05] (fixing 5% of the variables per step).

7.4.1 Random SAT

We have evaluated our solvers and the competitors on the test set of the instance sets 3sat10k, 5sat500 and 7sat90 (note that the training set was used only for finding optimal parameters for the solvers). The parameter settings for c_b and c_m are those from Table 7.2 (in case of 3-SAT we always used the parameters for 3sat10k). The results of the evaluations are listed in Table 7.3.

On the 3-SAT instances, the polynomial function yields the overall best performance. On the 3-SAT competition set, all of our solver variants exhibited the most stable performance, solving all problems within cutoff time. The survey propagation solver has problems with the 3sat10k and the 3satComp problems (probably because of the relatively small number of variables). The performance of the break-only-poly-solver remains surprisingly high even on the 3satExtreme set where the number of variables reaches $5 \cdot 10^5$ (ten times larger than that from the SAT Competition 2011). From the class of SLS solvers, it exhibits the best performance on this set and is only approximately two times slower than survey propagation. Note that a value of $c_b = 2.165$ for the break-only-poly solver further improved the run time of the solver by approximately 30% on the 3satExtreme set.

On the 5-SAT instances the exponential break-only-exp solver yields the best performance, exceeding even the performance of *Sparrow*, which was the best solver for 5-SAT within the SC11. On the 7-SAT instances though, the performance of our solvers is relatively poor. We observed a very strong variance of the run times on this set and it was relatively hard for the configurator to cope with such high variances.

Overall the performance of our simple probability based solvers reaches state-of-the-art performance, and can solve problems that where solved only by survey propagation solvers.

	3sat10k	3satComp	3satExtreme	5sat500	7sat90
$exp(c_b, c_m)$	46.6 (998)	93.84 **(500)**	-	12.49 **(1000)**	201.68 (974)
$poly(c_b, c_m)$	46.65 996	76.81 **(500)**	-	-	-
$exp(c_b)$	53.02 (997)	126.59 **(500)**	-	**7.84** **(1000)**	134.06 (984)
$poly(c_b)$	**22.80** **(1000)**	**54.37** **(500)**	1121.34 **(180)**	-	-
Sparrow	199.78 (973)	498.05 (498)	47419 (10)	9.52 **(1000)**	**14.94** **(1000)**
WalkSAT	61.74 (995)	172.21 (499)	1751.77 (178)	14.71 **(1000)**	69.34 (994)
sp 1.4	3146.17 (116)	18515.79 (63)	**599.01** **(180)**	5856 (6)	6000 (0)

Table 7.3: **Evaluation results:** Each cell represents the PAR10 run time and the number of successful runs for the solvers on the given instance set. Results are highlighted if the solver succeeded in solving all instances within the cutoff time, or if it has the best PAR10 run time. Missing results indicate extremly low performance of the solver on the benchmark set. Cutoff times are 600 seconds for 3sat1k, 5sat500 and 7sat90, and 5000 seconds for the rest.

Scaling Behavior Experiments show that the survey propagation algorithm scales linearly with n on formulas generated near the threshold ratio. The same seems to hold for *WalkSAT* with optimal noise, like the results in [KSS10] shows. The 3satExtreme instance set contains very large instances with varying $n \in \{10^5 \dots 5 \cdot 10^5\}$. To analyze the scaling behavior of *probSAT* in the break-only-poly variant, we have computed for each run the number of flips per variable performed by the solver until a solution was found, which remains constant at about $2 \cdot 10^3$ independent of n. The same holds for *WalkSAT*, though *WalkSAT* seems to have a slightly larger run time variance.

7.4.2 SC11 Random SAT

We compiled an adaptive version of *probSAT* and of *WalkSAT*, which first checks the size of the clauses (*i.e.* k) and then sets the parameters accordingly (like *Sparrow2011* does). We ran these solvers on the complete satisfiable instances set from the SC11 random category along with all other competition winning solvers from this category: *Sparrow2011*, *sattime2011* and *EagleUp*. Cutoff time was set to 5000 seconds. We report only the results on the large set, because the medium set was completely solved

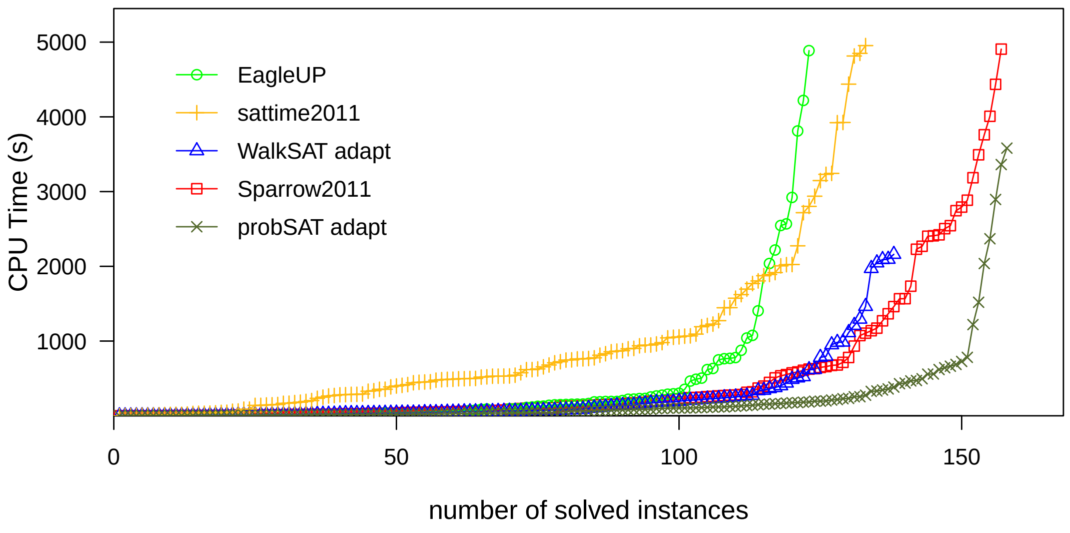


Figure 7.5: Results on the "large" set of the SAT Competition 2011 random instances.

by all solvers and the solvers had a median run time under one second. As can be seen from the results of the cactus plot in Figure 7.5, the adaptive version of *probSAT* would have been able to win the competition. Interestingly, the adaptive version of *WalkSAT* would have ranked third.

7.4.3 SC11 Crafted SAT

We also run the different solvers on the satisfiable instances from the crafted set of SC11 (with a cutoff time of 5000 seconds). The results are listed in Table 7.4. We included the results of the best three complete solvers from the crafted category. *probSAT* and *WalkSAT* were first evaluated with their different parameter configurations for (3,5,7-SAT). The 7-SAT break-only configuration performed best for both solvers, solving 81 in case of *probSAT* respectively 101 instances in case of *WalkSAT*. The performance of *WalkSAT* could not be improved by varying the noise setting. *probSAT* though exhibited better performance when increasing $c_b = 7$ and by switching to the polynomial break-only scheme, being then able to solve 93 instances. With such a high c_b value (very greedy), the probability of getting stuck in local minima is very high. By adding a static restart strategy after $2 \cdot 10^4$ flips per variable, *probSAT* was able to solve 99 instances (as listed in the Table 7.4).

The high greediness level needed for *WalkSAT* and *probSAT* to solve the crafted instances indicates that this instances might be more similar to the 7-SAT instances (generally to higher k-SAT). A confirmation of this conjecture is that *Sparrow* with fixed parameters for 7-SAT instances could solve 103 instances vs. 104 in the default

	sattime	Sparrow	WalkSAT	probSAT	MPhaseSAT (complete)	clasp (complete)
Crafted	**107**	104	101	99	93	81
Crafted pre.	86	97	**101**	95	98	80

Table 7.4: Each cell reports the number of solved instances within the cutoff time (5000 seconds). The first line shows the results on the original instances and the second one on the preprocessed instances.

setting (which adapts the parameters according to the maximum clause length found in the problem). We suppose that improving SLS solvers for random instances with large clause length would also yield improvements for non random instances.

To check whether the performance of SLS solvers can be improved by preprocessing the instances first, we run the preprocessor of *lingeling* [Bie11], which incorporates all main preprocessing techniques, to simplify the instances. The results show the contrary of what would have been expected (see Table 7.4). None of the SLS solvers could benefit from the preprocessing step on these instances, solving equal or less instances. These results motivated the analysis of preprocessing techniques in more detail, which was performed in Chapter 6.

7.4.4 SC12 Random SAT

We have submitted the *probSAT* solver (the adaptive version) to the SC12 random satisfiable category. The results of the best performing solvers are shown as a cactus plot in Figure 7.6. The *probSAT* solver was the second best solver on these instances, being outperformed only by *CCASat*.

While the difference between the two leading solvers *CCASat* and *probSAT* and all the other competitors is remarkable, the difference between *probSAT* and *CCASat* is not that large.

7.4.5 SC13 Random SAT

We have also submitted an improved version of *probSAT* to the SAT Competition 2013 to the Random Satisfiable category. The implementation of *probSAT* (the version submitted to SC13) was improved with respect to parameters, data structures and work flow. The parameters of *probSAT* have been set according to the values in Table 7.5 (where k is the size of the longest clause found in the problem during parsing). Note that we decided to ignore the *make* value completely in this implementation. These parameter values have been determined in different configuration experiments.

All array data structures where ended by a sentinel[8] (*i.e.* the last element in the array is the stop value; in our case we have used 0). All for-loops have been changed into

[8]We thank Armin Biere for this suggestion.

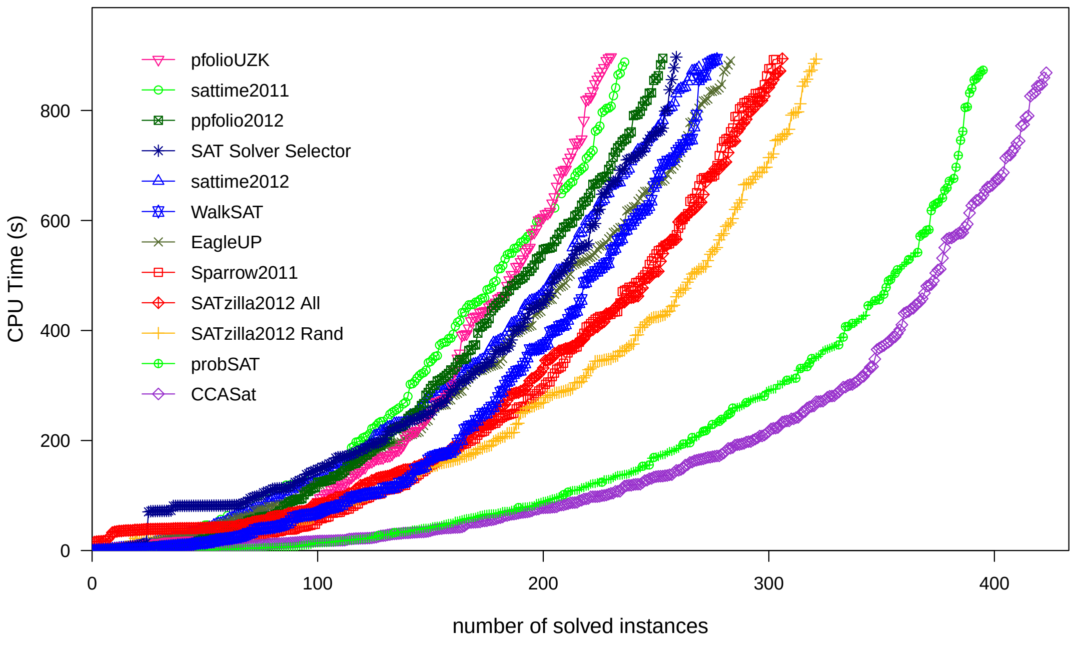


Figure 7.6: Results of the top solvers from SC12 random track.

k	fct	cb	ϵ
3	poly	2.06	0.9
4	exp	2.85	-
5	exp	3.7	-
6	exp	5.1	-
≥ 7	exp	5.4	-

Table 7.5: Parameter settings for *probSAT* used at the SC13.

while-loops that have no counter but only a sentinel check. As most of the operations performed by SLS solvers are loops over some small sized arrays, this optimization turns out to improve the performance of the solver between 10%-25% (dependent on the instances).

Compared to the original version of *probSAT*, the version submitted to the competition is not selecting an unsatisfied clause randomly, but will approximately iterate through the set of unsatisfied clauses with the flip counter (*i.e.* instead of *clauseindex=rand() modulo numUnsatClauses* we use *clauseindex=flipCounter modulo numUnsatClauses*). This small change seems to improve in some cases the stagnation

	Solver	total CPU time	average CPU time	median CPU time	average speed up
1	probSAT sc13 (nc)	4356.0729	17.4243	7.886	2.01
2	probSAT sc13	4696.9674	18.7879	8.499	1.86
3	probSAT sc12(2)	7632.1326	30.5285	10.695	1.15
4	probSAT sc12(1)	8781.8255	35.1273	12.489	-

Table 7.6: Performance comparison between the different version of *probSAT*. The last column shows the average speed up when compared to the results of line four, the basic version.

behavior of the solver giving it an additional boost.[9]

To measure the isolated effect of the different changes we have performed a small experiment on the training set of 3sat10k instances. We start with the version that was submitted to the SC12 with new parameters (sc12(1)), then we add the code optimizations (sc12(2)) and finally we remove the random selection of a false clause (sc13). A further version was added to this evaluation that does not cache the *break* values, but recomputes them before selecting a variable (more details about this can be found in Section 7.6). This version is denoted with (nc) and was analyzed after the competition. The results of the evaluation are listed in Table 7.6.

The code optimizations yield an average speed up of 15%, while removal of random clause selection is improving performance by around 70%. If the *break* values are being computed from scratch for the variables taken into consideration for flipping (also called non caching of *break* values), we achieve a twofold speed up compared to the original version with new parameters. See Section 7.6 for more details about this implementation.

The *probSAT* sc13 solver was submitted to SC13[10]. The results of the best performing solvers submitted to SC13 can be seen as a cactus plot in Figure 7.7. The *probSAT* solver is able to outperform all its competitors. The instances used in SC13 contained randomly generated instances on the phase transition point for $k = 3, \dots, 7$, and also a small set of huge instances (in terms of number of variables). The last were intended to test robustness of solvers. The *probSAT* solver turns out to be a very robust solver, solving many of the huge instances (18 out of 36). From the set of phase transition instances, *probSAT* solved 81 out of 109 that were solved by any other solver. Altogether, this shows that the solving approach (and the parameter settings) used by *probSAT* have an overall stable performance.

[9]This might also be the case for the *WalkSAT* solver.

[10]The code was compiled with the Intel®Compiler 12.0 with the following parameters: *-O3 -xhost -static -unroll-aggressive -opt-prefetch -fast*

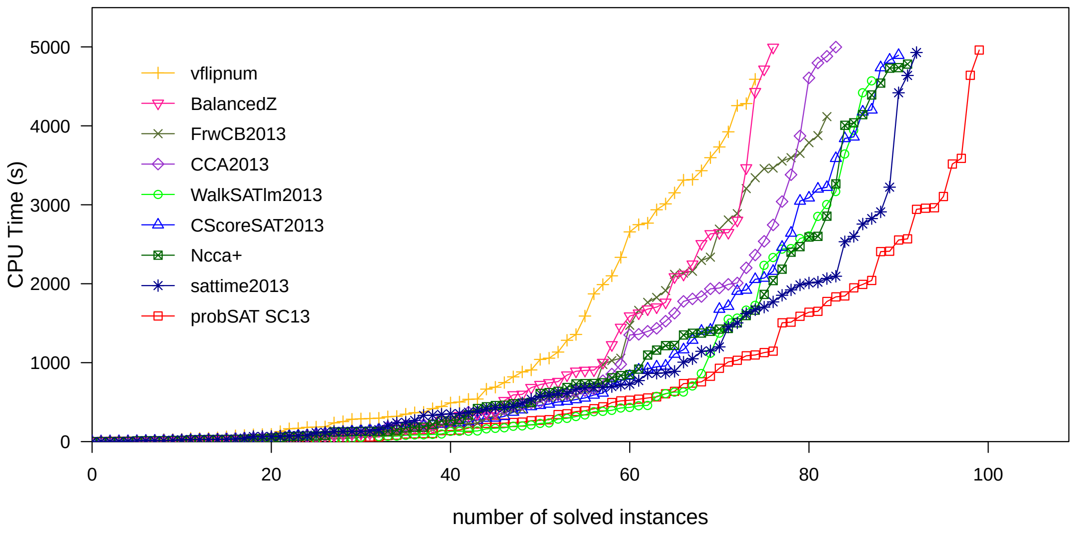


Figure 7.7: Results of the top solvers on the SC13 random satisfiable instances.

7.5 Comparison with WalkSAT

In principle, *WalkSAT* [MSK97] (see Section 3.3) also uses a specific pattern of probabilities for flipping one of the variables within a non-satisfied clause. Nevertheless, the probability distribution does not depend on a single continuous function f as in our algorithms described above, but uses non-continuous if-then-else decisions as described in [MSK97].

In Table 7.7 we compare the flipping probabilities in *WalkSAT* (using the noise value $p = 0.567$, which is the optimal value for 3-SAT problems [KSS10]) with the break-only-poly-algorithm (with $c_b = 2.06$ and $\epsilon = 0.9$, the current parametrization of *probSAT* for 3-SAT problems) using several examples of *break* value combinations that can occur within a 3-CNF clause.

Within a run of *probSAT* on a randomly generated problem ($n = 10000,\ r = 4.2$), we count how often each break combination occurs (first column). Interestingly, the combinations containing a *break* value of one in combination with low values occur very often, altogether in more than 85% of the steps.

The probabilities of *WalkSAT* and *probSAT* differ in many cases significantly, though in the most often occurring case, namely *break* $= 1\ 1\ 2$, they are very similar. While *WalkSAT* exhibits only five different probability patterns, *probSAT* is differentiating between each individual combination, producing each time a unique probability distribution.

Further, *probSAT* has the PAC property (see Chapter 3 for further details). In

%	*break*			*WalkSAT*			*probSAT*		
< 0.001	0	0	0	0.333	0.333	0.333	0.333	0.333	0.333
0.003	0	0	1	0.500	0.500	**0.000**	0.452	0.452	**0.097**
0.001	0	0	2	0.500	0.500	0.000	0.479	0.479	0.043
< 0.001	0	0	3	0.500	0.500	0.000	0.488	0.488	0.024
< 0.001	0	0	4	0.500	0.500	0.000	0.492	0.492	0.015
< 0.001	0	0	5	0.500	0.500	0.000	0.495	0.495	0.010
0.024	0	1	1	**1.000**	**0.000**	**0.000**	**0.700**	**0.150**	**0.150**
0.029	0	1	2	**1.000**	**0.000**	**0.000**	**0.767**	**0.164**	**0.069**
0.017	0	1	3	**1.000**	**0.000**	0.000	**0.792**	**0.170**	0.039
0.007	0	1	4	**1.000**	**0.000**	0.000	**0.803**	**0.172**	0.024
0.002	0	1	5	**1.000**	**0.000**	0.000	**0.810**	**0.174**	0.017
0.006	0	2	2	**1.000**	**0.000**	**0.000**	**0.848**	**0.076**	**0.076**
0.007	0	2	3	**1.000**	**0.000**	0.000	**0.878**	**0.079**	0.043
0.003	0	2	4	**1.000**	**0.000**	0.000	**0.893**	**0.080**	0.027
0.001	0	2	5	**1.000**	**0.000**	0.000	**0.900**	**0.081**	0.019
0.002	0	3	3	**1.000**	0.000	0.000	**0.911**	0.044	0.044
0.002	0	3	4	**1.000**	0.000	0.000	**0.927**	0.045	0.028
< 0.001	0	3	5	**1.000**	0.000	0.000	**0.935**	0.046	0.019
< 0.001	0	4	4	**1.000**	0.000	0.000	**0.943**	0.029	0.029
< 0.001	0	4	5	1.000	0.000	0.000	0.951	0.029	0.020
< 0.001	0	5	5	1.000	0.000	0.000	0.960	0.020	0.020
0.129	1	1	1	0.333	0.333	0.333	0.333	0.333	0.333
0.222	1	1	2	0.406	0.406	0.189	0.413	0.413	0.173
0.121	1	1	3	0.406	0.406	**0.189**	0.449	0.449	**0.102**
0.052	1	1	4	**0.406**	**0.406**	**0.189**	**0.467**	**0.467**	**0.066**
0.018	1	1	5	**0.406**	**0.406**	**0.189**	**0.477**	**0.477**	**0.046**
0.095	1	2	2	**0.622**	0.189	0.189	**0.544**	0.228	0.228
0.099	1	2	3	0.622	**0.189**	**0.189**	0.608	**0.254**	**0.138**
0.043	1	2	4	0.622	**0.189**	**0.189**	0.641	**0.268**	**0.091**
0.015	1	2	5	0.622	**0.189**	**0.189**	0.660	**0.276**	**0.064**
0.026	1	3	3	**0.622**	0.189	0.189	**0.687**	0.156	0.156
0.023	1	3	4	**0.622**	0.189	**0.189**	**0.730**	0.166	**0.104**
0.008	1	3	5	**0.622**	0.189	**0.189**	**0.755**	0.172	**0.073**
0.005	1	4	4	**0.622**	**0.189**	**0.189**	**0.779**	**0.111**	**0.111**
0.004	1	4	5	**0.622**	**0.189**	**0.189**	**0.807**	**0.115**	**0.078**
< 0.001	1	5	5	**0.622**	**0.189**	**0.189**	**0.838**	**0.081**	**0.081**
0.004	2	2	2	0.333	0.333	0.333	0.333	0.333	0.333
0.004	2	2	3	0.406	0.406	0.189	0.393	0.393	0.214
0.002	2	2	4	0.406	0.406	0.189	0.427	0.427	0.145
< 0.001	2	2	5	0.406	0.406	**0.189**	0.448	0.448	**0.104**
0.001	2	3	3	**0.622**	**0.189**	**0.189**	**0.479**	**0.260**	**0.260**
< 0.001	2	3	4	**0.622**	**0.189**	0.189	**0.531**	**0.289**	0.180
< 0.001	2	3	5	**0.622**	**0.189**	**0.189**	**0.563**	**0.306**	**0.130**

Table 7.7: Comparison of the flip probabilities of *WalkSAT* ($p = 0.567$) and *probSAT* (polynomial function $c_b = 2.06$ $\epsilon = 0.9$). The first column shows the percentage of steps where *probSAT* (when executed on a randomly generated 3-SAT problem with $n = 10000$, $r = 4.2$) encountered the *break* values combinations from the next column (*break*). The next columns (*WalkSAT* and *probSAT*) show the flipping probabilities for the variables if they have the *break* values from the previous column. Probabilities that differ more than 0.05% between the solvers are bold. Rows containing *break* combinations that occur in more than 0.02% of the search steps are highlighted proportional to the occurrence value.

each step, every variable has a probability greater than zero to be picked for flipping. This is though not the case for *WalkSAT*. A variable occurring in a clause where another variable has a score of zero can not be chosen for flipping in *WalkSAT*. When Donald Knuth analyzed the *WalkSAT* algorithm for his book "The Art of Computer Programming" Volume 4 Section 7.2.2.2, he asked Brian Cohen, the author of *WalkSAT*, for an example where *WalkSAT* might loop forever. The example is provided in the solution of exercise 185 of Section 7.2.2.2[11].

7.6 Implementation Variations

In previous sections, we have compared the solvers based on run time. As a consequence, the efficiency of the implementation plays a crucial role and the best available implementation should be taken for comparison. Another possible comparison measure is the number of flips the solver needs to perform to find a solution. From a practical point of view this is not optimal. The number of flips per second (denoted with $flips/sec$) is a key performance measure of SLS solvers when it comes to compare algorithm implementations or two different similar algorithms. In this section, we address this issue by comparing two different implementations of *probSAT* and *WalkSAT* on a set of very large 3-SAT problems.

All efficient implementations of SLS solvers are computing scores of variables from scratch only within the initialization phase. During search, the scores are updated incrementally. This is possible because only the scores of variables that are in the neighborhood of the flipped variable can change. This method is also known as *caching* (the scores of the variables are being cached) in [HS05, p. 273] or *incremental* approach in [Fuk04].

An alternative method would be to compute the score of variables on the fly before taking them into consideration for flipping. This method is called *non-caching* or *non-incremental* approach. In case of *probSAT* or *WalkSAT*, only the score of variables from one single clause have to be computed as opposed to other solvers where all variables from all unsatisfied clauses are taken into consideration for flipping.

We have implemented two different versions of *probSAT* and *WalkSAT* within the same code basis (*i.e.* the solvers are identical with exception of the *pickVar* method), one that uses caching and one that does not. We have evaluated the four different solvers on a set of 100 randomly generated 3-SAT problems with 10^6 variables and a ratio of 4.2. Results of these solvers along with the evaluation of the currently best known *WalkSAT* implementation (within *UBCSAT*) are shown in Figure 7.8.

Within a time limit of $1.5 \cdot 10^4$ seconds only the variants without caching are able to solve all problems. The implementation with caching solved only 72 (*probSAT*)

[11]A draft of Section 7.2.2.2 is available online at: `http://www-cs-faculty.stanford.edu/~uno/fasc6a.ps.gz`

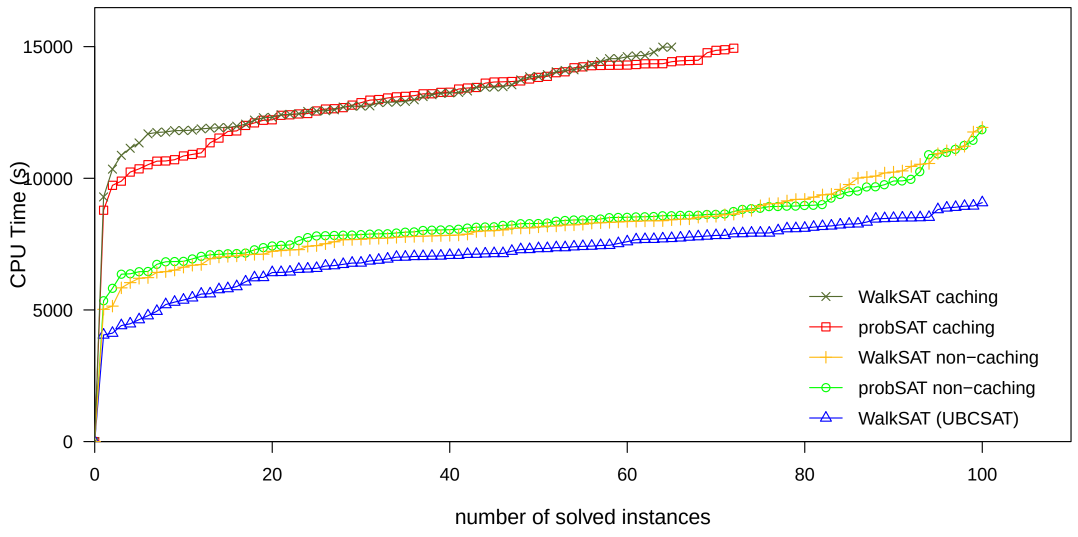


Figure 7.8: Comparison of the different implementation variants of *probSAT* and *WalkSAT* (within the same code basis), with and without caching of the break values. We also evaluate the best known *WalkSAT* implementation (non-caching) from *UBCSAT* as a reference.

respectively 65 (*WalkSAT*) problems. Note that all solvers started with the same seed (*i.e.* they perform search on exactly the same search trajectory). The difference between the different implementations in terms of performance can be explained by different numbers of $flips/sec$. While the version with caching performs around $1.4 \cdot 10^5$ $flips/sec$, the version without caching is able to perform around $2.2 \cdot 10^5$ $flips/sec$. Similar observations occur in [Tom10, p. 27] and in [Fuk04].

The advantage of non-caching decreases with increasing k (for randomly generated k-SAT problems) and becomes even a disadvantage for 5-SAT problems and upwards. As a consequence the latest version of *probSAT* uses non-caching for 3-SAT problems and caching for other types of problems. The results obtained by the best known implementation of *WalkSAT* (*UBCSAT* framework) is another example for how the performance of an algorithm can be improved by clever implementation.

The currently best version of *probSAT* is the hybrid version described in Chapter 4.5, where we have used *probSAT* together with the *picosat* solver to construct the *hybridPP* solver. By using the hybridization approach we could further improve the performance of *probSAT*.

7.7 Conclusion and Future Work

We introduced a simple algorithmic design principle for an SLS solver, which does its job without complex heuristics and "tricks". Our approach relies on the concept of probability distribution and focused search, although being flexible enough to allow plugging in various functions f that guide search. Using this concept, we were able to discover an asymmetry regarding the influence of *break* and *make* values: the *break* value is the more important one; one can even do without the *make* value completely. We have systematically used an automatic configurator to find the best parameters and to visualize the mutual dependency and impact of the parameters.

Within a wide range of empirical evaluations, we showed that *probSAT* reaches state-of-the-art performance on randomly generated k-SAT problems, establishing new performance marks for randomly generated problems. We also provided a detailed description of several possibilities of improvement and a detailed comparison with the *WalkSAT* solver.

The simplicity of *probSAT* allows the plugging of different functions and of different variable properties, which in our opinion can be a fruitful direction of research and should be further analyzed.

Finally, a theoretical analysis of the Markov chain convergence and speed of convergence underlying this algorithm would be most desirable, extending the results in [Sch99].

Acknowledgments We thank the BWGrid [bhg10] project for providing the computational resources. This project was funded by the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 302/9-1. We thank Daniel Diepold and Simon Gerber for implementing the F-race configurator, and providing different analysis tools within the EDACC framework. Thank goes also to Andreas Fröhlich for fruitful discussions on this topic and Armin Biere for helpful suggestions regarding code optimizations.

8 An advanced Platform for the Experiment Design, Administration and Analysis of Algorithms

After the author of this thesis worked on the *hybridGM* solver (presented in Chapter 4), and spent more time on the evaluation, collection and analysis of results than on implementation of the solver, it became clear that it is worth to design a framework that automates all these tedious tasks and especially allows the simple usage of distributed computers. The framework is called Experiment Design and Administration for Computer Cluster (EDACC). The main motivation of EDACC was to design a system that distributes jobs over arbitrary computing systems (like large clusters) and is independent of the job queuing system of the computing system. After the initial design sketches have been finalized, it was clear that the project is by far too large to be implemented by a single person. Consequently, the project was split in small sub-projects that were assigned to students interested in the topic of software design and distributed computing. After finalization of the sub-projects, different new ideas and possible improvements gave rise to new sub-projects. In this way, EDACC evolved over three years to a versatile system for the empirical evaluation and analysis of algorithms.

The work presented in this chapter is mainly based on work published in [BGKR10] and [BDG+11].

8.1 Introduction

Many problems that come from practical applications or from theory are known to be very hard to solve. The SAT problem is only one example. This means that the time for solving these problems increases exponentially with the size of the input. The class

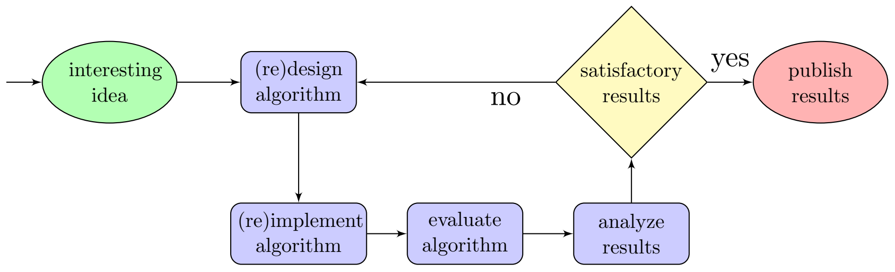


Figure 8.1: A typical work flow for the development, evaluation and analysis of empirical algorithms

of $\mathcal{NP}$-complete problems is probably the most well known class of such problems. Formerly, proving that a problem was $\mathcal{NP}$-complete meant that the design of a practical algorithm for this problem would be useless, because of the estimated exponential time needed by the algorithm to solve the problem. The situation changed drastically with the development of heuristics, meta-heuristics and approximation algorithms for hard combinatorial problems. The size of the problems that can be solved by these kinds of algorithms has increased continuously over the years. For example, the *probSAT* solver from Chapter 7 is orders of magnitude faster than the *hybridGM* solver presented in Chapter 4.

This progress can be seen as the result of a paradigm change from “algorithms are fast if they have a theoretical good upper bound for their run time” to “algorithms are fast if they are fast in practical experiments”. This does not mean that theoretical results are unimportant, but rather that the design of algorithms for hard combinatorial problems has become oriented towards practical methodologies.

With this paradigm change, methodologies have also changed considerably. A theoretical analysis of heuristics is not possible in most cases, and has been replaced by an empirical evaluation like the ones used in engineering. Most development of empirical algorithms now follows an engineering scheme sketched out in Figure 8.1.

With the use of new methodologies, new problems arise. After the design and implementation phase, the algorithm has to be tested and evaluated, which in most cases is a very time consuming task. The first problem that an algorithm designer encounters is the collection and selection of instances on which the solver will be evaluated. Dependent on the set of instances chosen for the evaluation, a parameter configuration for the algorithm has to be chosen. This problem can be solved often by automated procedures like *ParamILS* [HHLBS09], *SMAC* [HHLB11] or *GGA* [AST09], although preliminary tests are performed with manually configured parameters.

Having instances and parameters for evaluation, the user still has to choose a computing system. A multi-core computer, a cluster, or even a grid can speed up computations considerably. However at the same time the problem of equally distributing the work-load arises. In most cases it is solved by some home brewed scripts. After finishing the computation, the results have to be gathered from the computing systems, and important information has to be extracted from the output by some parsing procedures. To examine to what extent the results are satisfactory, statistical tests have to be performed on the collected data. Comparing the performance of two or more algorithms demands further elaborated statistics.

The processes of evaluation and analysis are seldom reproducible between different researchers, because of the complexity of the process and the lack of common methods and of documentation. This is probably the reason why most of the communities working on empirical algorithms periodically organize international competitions, where researches are encouraged to submit their latest algorithms. The purpose of these competitions is to provide the same evaluation and analysis environment for all algorithms. Unfortunately, the underlying evaluation system of competitions consists of scripts and databases that are not freely available.

The EDACC system tries to overcome most of these problems. EDACC is capable of managing solvers with their parameters, instances, creating experiments, running them on arbitrary computing systems ranging from multi-core computers to large scale grids, collecting the results and post-processing them. It also offers the possibility to export and import experiments between different research groups. Advanced methods for automatically extracting and archiving information from the results (or from the instances) are also provided in EDACC. The extracted information can be analyzed with a large variety of statistical tests and descriptive statistics, which are provided by different components of EDACC. To make the organization and execution of competitions as simple as possible, EDACC also provides a competition mode that follows a widely accepted scheme in the field of computational logic.

8.2 EDACC - Overview of the Components

Notations Before describing the main components, some entities that will be used through the rest of this chapter are defined. A *solver* is an implementation of an algorithm that works on some input and has an output. The behavior of a solver is controlled by some *parameters*. A solver together with a fixed set of parameters is called a *solver configuration*. The input to a solver is called an *instance*. Any information that can be computed from an instance is called an *instance property*. A *computing system* is defined as a single computer, a computer cluster, or a grid on which the solvers are executed. When running a solver on a computing system, *computational limits* can be imposed (*e.g.*, maximum computation time or maximum memory). If S is a set of solver configurations, I a set of instances, then we can define an experiment noted

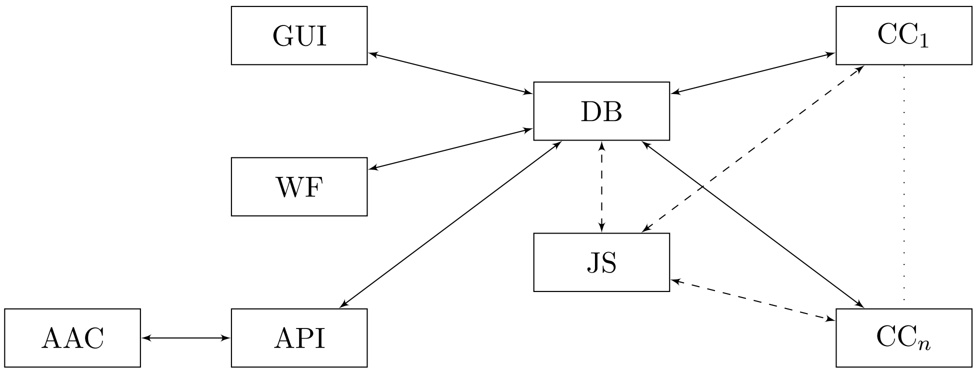


Figure 8.2: The components of EDACC and their interactions. The dotted line between client CC$_1$ and CC$_n$ represents the presence of n clients. Dashed lines represent optional communication between the components. The minimal set of components is the GUI, DB and the computation clients (CC).

with E as:

$$E \subset (S \times I)$$

A single element of an experiment is called a *job*. The computation of a job produces a *result*. Any information that is computed from a result is called a *result property*, while any type of information computed from an instance is called an *instance property* (also called *feature* in the AI community).

The main components of EDACC are the Database (DB), the Graphical User Interface (GUI) and the computation client (CC), which is also called only client. These are the top three components in Figure 8.2.

The DB is the central component of EDACC, because it is used by all other components for information retrieval and communication. The DB is responsible for storing and archiving all the information about solvers, instances, experiments, computing systems and their relations.

The GUI is the main control system of EDACC, and is split in two operation modes: *DB management* and *experiment management*. The first mode provides all necessary DB operations to create, remove, update and delete (CRUD) solvers, parameters, instances and other necessary auxiliary programs. It also allows the computation of instance properties (*e.g.*, the number of variables in an instance). The experiment mode provides the necessary procedures for the generation of experiments and their jobs. At the same time, it also provides monitoring possibilities for the computation clients and for the experiments. Within the experiment mode the user can post-process the results by extracting result properties (*e.g.*, number of flips performed by the solver), which are then stored in the DB and can be used for statistical tests.

The clients are responsible for the execution of the jobs that were created in the experiment mode. One client manages one single machine with arbitrary many cores. The client fetches jobs from the DB, starts them and monitors the resource usage through an external watcher program. The client is configured to run with *runsolver* [Rou11]. After jobs finalized their execution, results are copied back to the DB and new jobs are fetched for processing.

These three components are sufficient for most of the experiments performed for the evaluation of solvers. Further details about these components and their features are provided in [BGKR10, BDG+11].

8.3 Job Server

All jobs of EDACC are stored in the relational database from where they are fetched by the clients and then marked as being processed. Every query for an unprocessed job has to have as a result a unique job ID. Thus, within the execution of an experiment, the DB represents a job queue for the client. Relational DB's can emulate queues, but with a relative low efficiency. The major reason for this is that the job table in the database has to be locked for update operations, which occur every time a job is marked as being processed (*i.e.* an element has been extracted from the queue). Here we could use a non-SQL database, which is much more faster, but we would loose many features of relational databases needed for EDACC.

As a performance measure for the queue, we use the number of elements it can provide per second (*i.e.* jobs/second). By using the DB as queuing system, the locking mechanism of the DB is becoming the limiting factor, and rates higher than 20 jobs/second are not possible on a desktop computer. This means that if more than 20 jobs per second are requested, computational resources available to the clients will start to idle. Note that if the average runtime of jobs is 100 seconds, EDACC can manage around 2000 cores. The same problem was also addressed within the BOINC project [AKW05].

One of the key motivations of EDACC was to avoid idling of free available resources as much as possible. This motivated the development of the *job server* (JS).

The job server implements fast queues in memory. The queue is periodically synchronized with the DB by means of a single large SQL operation, which can be performed very fast when compared to many small SQL operations (*i.e.* job requests). As can be seen in Figure 8.2, the clients can connect to the job server for retrieving a unique job, and then connect to the DB to retrieve the necessary information about the job. In this way, the number of jobs per second that can be provided to the clients increases, allowing up to 100 jobs/second, which is sufficient to execute EDACC experiments on large scale grids with 10000 cores (when the average job run time is 100 seconds). The task server of the BOINC project, which also implements a queue in memory, can also provide up to 100 jobs/second [AKW05]. To further increase the performance of the queuing system, the job server could use non-blocking queues implementations.

8.4 Parallel Automated Algorithm Configuration

During the development of SLS solvers for the SAT problem, we observed that the performance of all developed solvers is highly dependent on their parameter settings. It is not sufficient to design a fancy solver and then hope that it will reach high performance. Appropriate parameters have to be determined to make it a state-of-the-art solver. This parameter dependency motivated the integration of Automated Algorithm Configuration (AAC) tools into EDACC.

The problem of automated algorithm configuration has been described detailed in two Ph.D. thesis, one by Birattari [Bir04], and one by Hutter [Hut09]. Several algorithm configurators have been proposed in the last years: *ParamILS* [HHS07], *GGA* [AST09], *F-Race* [BYBS10], *SMAC* [HHLB11] and *dSMAC* [HHLB12].

All the presented configurators except *dSMAC* are sequential or low parallel (can use only the cores of one single machine in case of *GGA*). To be able to make use of the high parallel execution ability of EDACC, we partially re-engineered several AAC algorithms and developed parallel versions of these in a unified configurator framework.

To make this task as easy as possible, we first designed an API for configurators that provides all necessary functionality, allowing developers to concentrate on the configuration algorithm. Before describing the API and the algorithm configuration framework provided by EDACC, we will first give a description and motivation of the algorithm configuration problem.

8.4.1 The Algorithm Configuration Problem

After the development of an (parametrized) algorithm, we want to solve the following problem: Given a set of instances I, we want to know how to set the parameters of the algorithm such that its performance on I is maximized. An alternative (possibly easier) problem is the question whether there exists a parameter setting such that a given performance value (*e.g.*, reached by competing algorithms) can be reached or even exceeded.

To specify the algorithm configuration problem, we will use the following notations. We define a solver $\mathcal{S}$ as the implementation of an algorithm $\mathcal{A}$. A solver has parameters that control its behavior, which can be of different types like real, integer or categorical. Parameters and their values can also be conditioned by other parameters and their values. All valid combinations of parameter values for one solver $\mathcal{S}$ is called the *parameter search space* $\mathcal{P}$. One point p from this search space is called a *parameter configuration* or *solver configuration* S_p. A set of instances is noted with $\mathcal{I}$. Evaluating a solver configuration of a deterministic solver on one instance (a *job*) produces a *result* that has a cost c, which can be the run time (in the case of decisions problems) or some quality measure of the produced solution (in case of optimization problems). In case of stochastic solvers, a job is additionally identified by a seed, therefore a job is generally identified by an instance-seed pair.

Definition 8.4.1 (Algorithm Configuration Problem)**.** Given a solver $\mathcal{S}$, a specification of its parameter search space $\mathcal{P}$, a set of instances $\mathcal{I}$, a cost metric c and a statistic function γ over c, the algorithm configuration problem consists in finding a parameter configuration p from $\mathcal{P}$ that minimizes γ, *i.e.*:

$$p_{opt} \in \operatorname{argmin} \gamma(c(S_p(I)))$$

The AAC problem is very similar to black-box optimization[1], where a function has to be optimized without knowing its explicit form. The function can only be evaluated at arbitrary points and the evaluation value is the only information available for the optimization procedure.

While in black-box optimization the search domain is typically $\mathbb{R}^n$, in AAC the search domain can have very complex structures with mixed types of variables (parameters) and with forbidden subspaces. To cope with all this type of complex parameter search spaces, we have developed the parameter search space graph specification.

8.4.2 Parameter Graph

In [AST09] the configurator *GGA* used for the first time AND/OR trees to represent the parameter search space. AND/OR trees have originally been proposed in [MD05] for the compact representations of constrained search spaces. We have extended the concept of AND/OR trees to acyclic graphs. This allows an OR node to have multiple incoming edges from AND nodes, which allows us to represent multiple conditioning of parameters (*e.g.*, parameter p_i can only be active when $p_j = x$ and $p_k = y$). More details about the parameter graph representation can be found in the user guide of EDACC.[2]

8.4.3 EDACC API

The API provides two major types of functionalities for the configurators, abstracting from the database structures of EDACC. The first one is related to the management of jobs, while the second one is related to querying the parameter graph.

[1]Black-box optimization is concerned with the optimization of a black-box functions (*i.e.* the semantic of the function is not known and can not be used to guide the optimization process, as it is the case for SAT problems; only evaluations of the function are possible).

[2]`https://github.com/EDACC/edacc_user_guide/blob/master/user_guide.pdf`

Algorithm 7: EDACC AAC Main Workflow

```
   Input  : configuration experiment
   Output: best found parameter configuration
1  initialize race and search procedure
2  while computation budget not exceeded do
3      expansion ← racing.computeOptimalExpansion()
4      $\vec{P}$ ← search.generateNewConfigs(expansion)
5      racing.startEvaluation($\vec{P}$)
6      update information about jobs from DB
7      $\vec{P_f}$ ← set of complete evaluated parameter configurations
8      racing.notify($\vec{P_f}$)
9  return best found parameter configuration
```

The API provides all necessary procedures to manage solver configurations and their jobs. The most important functionalities are:

- create/remove/update solver configurations
- check for user specified solver configurations
- add/remove/stop jobs of a given solver configuration with a given priority
- check status of jobs

The API provides procedures to query the parameter graph typically encountered in local search and genetic configurator algorithms.

- retrieve random parameter configurations
- retrieve different types/sizes of neighborhoods of a given parameter configuration
- perform different types of crossovers of two parameter configurations
- perform mutations on a given parameter configuration

8.4.4 EDACC Configurator Framework

To make the configurator framework as general as possible, and to be able to incorporate all possible types of configurators into our framework, we have split our framework into three parts: main procedure (loop), search procedure and racing procedure. The main procedure is sketched out in Algorithm 7.

The search procedure is responsible for generating new solver configurations (see line 4 in Algorithm 7), thus performing the search in the search space spanned by the parameter graph. The racing procedure determines how a solver configuration is evaluated (line 5 *i.e.* how many jobs will be allocated to each solver configuration). Containing the management of the jobs, the racing procedure is also responsible for the management of available resources, which should be exploited as much as possible to guarantee a good parallelization of the configurator. The main procedure is responsible for maintaining the data synchronized with the database (line 6), and for performing the communication between the search and racing procedures.

Almost all major configurators (*e.g.*, *ParamILS*, *SMAC*, *F-Race*) and variations of these have been reimplemented in our framework and used in different research projects (see Chapter 5, 6, 7). The advantage of the separation of the search and racing procedure is that new configurators can be designed by using existing search or racing procedures or even by creating new combinations of these. The parallel version of the configurators are able to reach performance similar to their sequential counterparts and sometimes even exceed it (see [Die12] for more details about some experimental evaluations).

8.5 Web Frontend

The Web Frontend (WF) of EDACC was designed with the goal to leverage the publication of experiment results online, also enabling to share results with collaborators. The WF also enables the analysis of the results by means of descriptive statistics and statistical analysis. Since the WF provides most of the functionalities needed for the organization of competitions, it was further extended to a full competition system.

8.5.1 Analysis and Statistical Tools

The information extraction mechanism provided in EDACC can supply a wide variety of information about an experiment [BDG+11]. This information can be analyzed with the WF by means of descriptive statistics and statistical tests. These tests can be used to measure the performance of algorithms, to show correlations between some properties of the results, or to simply provide a graphical representation of the results, enabling the user to directly analyze the results without having to export the data and post-process it with external tools.

The information that can be used for analysis is stored in the DB within instance properties and result properties. We differentiate between two scenarios in which analysis is performed: analysis of a single solver or comparison of two or more solvers. We also have to differentiate between single runs or multiple runs of a solver on the same instance. If multiple runs are available, the information used for statistics can be chosen by the user from median, mean, all runs or only a single specified run.

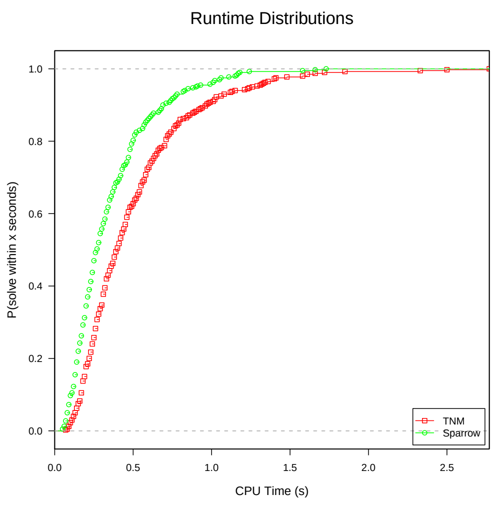


Figure 8.3: Comparison of the run time distribution of two solvers.

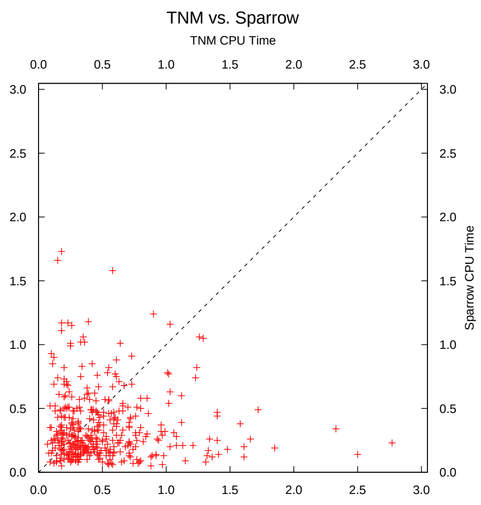


Figure 8.4: Scatter plot to compare the run time of two solvers.

A distribution plot (see Figure 8.3) and a nonparametric kernel density estimation is provided for the analysis of the results of a single solver on an arbitrary instance by means of an arbitrary result property. To analyze the results of a solver on all instances (or a selection), the user can use scatter plots. The compared information can be some instance property with a result property, like for example number of variables vs. CPU time, or two result properties, like memory-usage vs CPU-time. Beside scatter plots, we also compute the Spearman rank correlation coefficient and the Pearson product-moment correlation coefficient.

A scatter plot (see fig. 8.4 for a run time comparison) together with the two mentioned correlation tests is provided for the comparison of two solvers by means of an arbitrary result property. EDACC also provides two non-parametric tests: the Kolmogorow-Smirnow two-sample test and the Mann-Whitney-U test (Wilcoxon Rank Sum test).

A well founded comparison of the performance of two solvers can also be done with the help of a probabilistic domination test by means of an arbitrary result property (*e.g.*, CPU time, flips, number of branches). Within this test instances are split into three categories. The first category contains the instances on which the first solver probabilistically dominates the second one. The second one contains the instances on which the second solver probabilistically dominates the first one and the third category contains the instances on which no probabilistic domination can be found due to the crossing of the distributions.

Analyzing one result property for one or more solvers can be done with a box plot

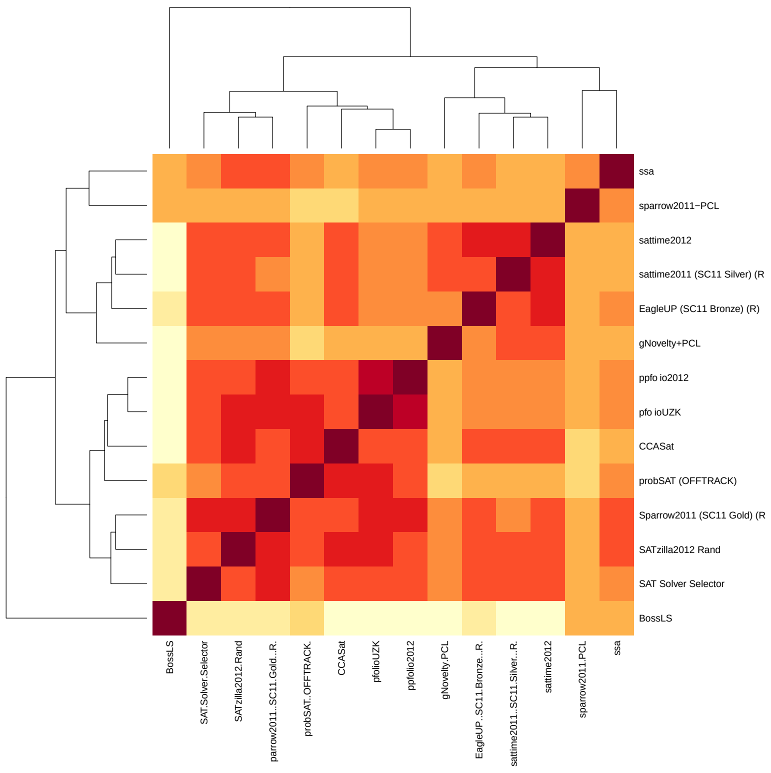


Figure 8.5: A heat map plot of the clustered Spearman correlation matrix of the result of the random satisfiable category from SC12. The darker the area, the more correlated the solvers are with respect to their run time.

or with a cactus plot as in Figure 7.7 on page 95 (number of solved instances within a given amount of result property). To analyze the similarities of the results of a set of solvers, the WF can plot a clustered Spearman correlation matrix as shown in Figure 8.5.

The WF can also compute the minimal set of solvers needed to solve all problems within an experiment. The minimum set of solvers problem is actually a minimum hitting set problem, which is encoded as a MAX-SAT problem and solved with a MAX-SAT solver.

Finally EDACC can export the generated plots in several file formats, including vector graphics. To support third-party analysis tools, results can also be exported to the widespread csv-format.

8.5.2 Competition Mode

Solver competitions can be an incentive for researchers to implement new ideas, to improve existing solvers, and spark interest in the field. Recurring competitions can show the progress in the development of solvers by comparing new solvers with reference solvers from previous competitions. They can also help to identify challenging instances for state-of-the-art solvers. The results of such a competition can be used by researchers to identify the strengths and weaknesses of solvers and instances, and to guide further research.

There are several competitions in the field of algorithms for logic problems, for example the series of *SAT Competition*[3] [BS04], the *SMT-COMP*[4] [BdMS05] or *CASC* [Sut10].

Running such competitions is an organizational challenge, and brings the inevitable need for tools to make it possible to run dozens of solvers on a large set of instances in a multi-computer environment, and retrieve and process the results for competition purposes. The competitions mentioned above do have such internal tools and web interfaces, but to our knowledge, they are not publicly available. To make the organization of competitions possible to everybody (who has the computational resources), we extended EDACC to provide all required functionalities for the organization of competitions.

We first started by analyzing the existing competition systems to find out their commonalities, and to identify interesting or missing features.

From an abstract point of view most competitions have:

1. static web pages to provide information about rules and the course of events
2. user administration to control the access to the results
3. an execution system to run solvers and manage the results
4. dynamic web pages to present the results

As necessary, interesting or missing features we have identified:

1. plausibility and verifiability of the steps taken in all competition phases, by providing participants real-time access to all relevant information
2. the results have to be reproducible, which means all required information (*e.g.*, starting command, seeds, input files, output files) should be easily accessible through a web interface
3. various forms of presentation of the results with cross linking and filtering

[3]`www.satcompetition.org`
[4]`www.smtcomp.org`

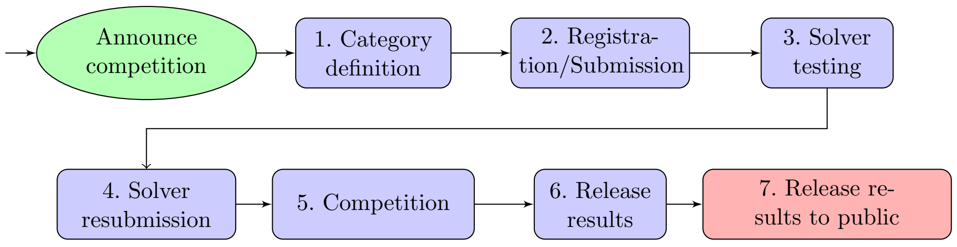


Figure 8.6: The typical phases of a competition in the field of logic computing.

4. different graphical presentations of the results, including interactive elements such as clickable points in plots that lead to detailed information
5. all graphical presentations can be exported both as image and as numerical data
6. descriptive statistics and statistical tests for analysis of the results
7. clean encapsulation of the ranking system enabling easy implementation of new ranking systems

We have extended the WF of EDACC to provide all these features. Further we have added a phase system (see Figure 8.6) to specify the course of events during a competition. Each phase also defines access control to the various information.

Next we are going to describe the organization of a competition with the EDACC WF by describing each phase and pointing out the interesting features that are provided. The access control to different kinds of information (*e.g.*, individual results, all results, statistics, etc.) can be configured by the organizers for each phase individually, according to their competition policies. Through the description of the phases an exemplary access control is given.

Category definition In the first phase, the organizers of the competition define the competition categories (which actually can be seen as sub-competitions). A category is defined by the instances it will contain, and should give the competitors an orientation where to submit their solvers. In EDACC, each category will be represented by an experiment. In this phase, competitors have access only to general information, rules and the schedule. The WF provides containers for these static web pages.

Registration and Submission In the second phase, competitors are requested to create an account for the web interface. After login they can submit their solvers (*i.e.* source code or binary), which are directly saved within the DB. They have to provide detailed information about their solvers, like the parameters and the competition category where the solver should participate. Instances can also be submitted, which are then available to the organizers. During this phase, competitors have no access to other competitors' solvers nor instances.

Solver testing This phase is used to ensure that the submitted solvers are able to run on the computing system of the competition. Within the EDACC GUI, organizers create test experiments, corresponding to each of the competition categories. Each solver will be tested in all categories it was submitted to. The experiments are then executed on the competition computing system with the help of the client. Competitors have the possibility to monitor their solvers through the web frontend (results of other solvers are not visible) in real time. From this phase on, results are accessible in several forms[5]:

1. by solver configuration: the results for all instances computed by a solver configuration
2. by instance: the results of all solver configurations
3. by solver configuration and instance (if multiple runs are allowed): multiple jobs of each solver configuration on an instance are accumulated and some descriptive statistics like the minimum, maximum, median and mean run time displayed
4. single result: the result of a single job, including the output of solver, launcher, watcher, verifier and also all result properties that were computed for this result

Solver resubmission During this phase, competitors have the opportunity to submit solver updates if bugs or compatibility issues with the computing system occurred. Automated scripts will rerun the tests.

Competition Similar to the testing phase, in the competition phase organizers create experiments based on the competition categories and choose the solvers and instances for each experiment. This task is accomplished easily with the help of the GUI. The experiments are run on the computing system and competitors have the possibility to monitor the results of their own solvers online (and of others if configured so by the organizers).

Results release In the release phase, competitors gain access to the results of all competing solvers. Before making the results available to the public, a ranking has to be calculated. The ranking can either be calculated dynamically by the web application or

[5]An example for the results of a competition can be found at `http://edacc4.informatik.uni-ulm.de/`

simply displayed after a manual calculation. We have implemented a simple, exemplary ranking using the number of correct results and breaking ties by the accumulated CPU time. Other rankings like the careful ranking [VG11] or rankings based on the PAR10 measure are also implemented.

Also available in this phase is the complete spectrum of descriptive statistics and statistical tests described in Section 8.5.1. For pointing out interesting results or correlations, the organizers have the possibility to extract instance or result properties within the GUI, and make them available within the web frontend.

Final release In a last phase, instances, results and possibly solver source codes and binaries are made publicly available on the web interface without requiring registration.

8.6 Implementation Details

The EDACC DB requires a user-account on a MySQL 5.1 database with read and write access. The location of the DB has no importance, but to reach high performance it should be on the same computer as the WF. The GUI, API and configurator framework of EDACC is written in Java and is independent of the operating system of the computer. It needs only the Java virtual machine version 6. For the statistical evaluation, the R[6] programming language should also be installed on the computer.

The compute client consists of three sub programs: the launcher, the watcher and the verifier. The launcher builds a DB-connection and is responsible for fetching the jobs and all necessary files, providing them to the watcher. The launcher is written in C and was tested on unix-like systems. The watcher, a replaceable component in EDACC, is the runsolver tool of Olivier Roussel [Rou11]. The verifier is problem dependent and has to be provided by the user. If the results of the solver can be trusted (*e.g.*, the solver contains a verifier procedure) the verifier can be omitted. The job server is implemented in C++.

The web interface for the competition mode is implemented as a Python WSGI (Web Server Gateway Interface) application. The application uses a web framework and several open source libraries, which are available on most platforms. All competition specific data like user accounts, instance types and the phase of the competition are stored in the central DB. To generate plots and calculate statistics it uses an interface library to the statistical computing language R.

The code of EDACC components is open source and is released under the MIT License (except the watcher, which has a GPLv3 license). The code is available at the project site: `http://sourceforge.net/projects/edacc/` or at `https://github.com/EDACC`.

[6] `http://www.r-project.org/`

8.7 Related Work

We are not aware of the existence of an experimentation system for empirical algorithms that provides all the functionalities of EDACC within the same platform and is freely available. Though, parts of EDACC's functionalities are provided by different systems or tools.

The EDACC system is highly inspired by the *SatEx* system that was first presented in [SC01] and further developed and used for one decade for the execution of the SAT competitions. The *SatEx* system is mainly a web based platform for the presentation of SAT solver results. The web client of EDACC provides similar functionalities as *SatEx*, extending them with statistical tools for analysis and visualization of the results.

GridTPT [BCDF12] supports the testing of SMT solvers and their distribution on computer clusters, having a master/slave architecture. The system is also able to parse information from the output and present some statistics as scatter plots.

The SMT Competition [BdMS05] systems have several tools similar to our WF, however they lack the possibility to perform advanced analysis of the results and are not freely available nor portable to other computing systems.

The distributed system implemented in EDACC is similar to that of the BOINC project [And04], which is also based on a central MySQL database and computation clients. Our implementation, though, is much more lightweight but not as versatile.

The high throughput computing system Condor [TTL05] is also providing a general purpose distributed queuing and execution system similar to that of EDACC.

8.8 Conclusion

In this work we have introduced EDACC, a platform for the design, administration and analysis of experiments on empirical algorithms. EDACC consists of three major components, the database, a graphical user interface and a compute client. Other auxiliary components are the web frontend, the job server, the API and the automated algorithm configurator.

We succeed to create a system that can scale up to large scale grids, allowing researchers to tremendously speed up the empirical evaluations of algorithms of any kind. This has been achieved by using the job server, which implements a queuing system in memory.

The EDACC system provides a parallel algorithm configuration framework that allows the optimization of parameters and also the easy design of new parallel algorithm configurators.

With the help of the web frontend, the organization of competitions becomes an easier task (at least from a technical point of view). The plethora of tools provided by the web frontend gives the participants of the competition the possibility to thoroughly analyze the results. The EDACC competition system was successfully used to perform the SAT Competitions in 2012[7] and 2013[8].

We think that researchers studying empirical algorithms can drastically speed up their experimental and analysis work by using EDACC as their experimental platform.

Acknowledgments We thank the bwGrid [bhg10] project for providing the test environment, and Borislav Junk and Raffael Bild for the first version of the execution client.

[7] `edacc2.informatik.uni-ulm.de`
[8] `edacc4.informatik.uni-ulm.de`

9 Conclusion and Future Work

Within this chapter, we provide a synthesis of the main ideas presented in this thesis and give several directions of future research inspired by our findings.

The satisfiability problem, one of the most prominent problems in computer science, occurs in a wide range of theoretical and especially practical applications. Any progress on the field of SAT solving is influencing the progress in other fields, like hardware and software verification, cryptography or the solving of mathematical problems. In the last two decades, much research was devoted to the improvement of SAT solving techniques based on the principle of Stochastic Local Search (SLS). Nevertheless, the inner working of SLS solvers is poorly understood and new analysis methods are required. Furthermore, SLS solving techniques still have a high improvement potential in general, but especially on structured SAT problems. The work presented in this thesis addresses these problems.

Our major goal was to find new ways to solve different types of SAT problems more efficiently, and to gain new insights about the inner working of modern SAT solvers. We reached our goals in several steps: by providing new analysis methods of the search behavior of SLS solvers, by designing new state-of-the-art SLS solvers, by analyzing preprocessing techniques suited for SLS solvers, by studying the essentials of SLS solvers and finally by providing a parallel experimentation framework for algorithms.

In Chapter 4 we started our work with the analysis of intensification and diversification of the search of the state-of-the-art SLS solver *gNovelty+*. While the latter was sufficient according to our analysis methods, the former indicated improvement potential. We analyzed the intensification of the search by constructing partial assignments (called also SSP) around local minima, and then analyzing the simplified formula with a complete SAT solver, which either finds a solution or proves unsatisfiability. This analysis approach entails a hybridization scheme for SLS and complete solvers, which we originally implemented in the hybrid solver *hybridGM*. We were able to show that *hybridGM* is outperforming its SLS component (*gNovelty+*) on randomly generated SAT problems.

Further, we also showed that the hybridization scheme can be successfully applied (improving the performance over the SLS component) to arbitrary SLS and complete solvers by implementing the *hybridGP* and *hybridPP* solvers that are based on modern state-of-the-art solvers. This demonstrates that our analysis and hybridization method can be applied to any new type of SLS solver that will be presented in the future, possibly yielding a further improvement of those methods.

We think that our hybridization scheme has improvement potential in several directions, especially considering the way SSPs are constructed and how they are sized. Another direction of research is the extension of the information exchange between the SLS and CDCL solver. The CDCL solver could compute the set of core variables (the set of variables that are needed to prove the unsatisfiability of the problem) and pass them to the SLS solver, which could use this set to further guide its flipping heuristic. In case of clause weighting solvers, this can be easily achieved by increasing the weights of the core variables.

As a next step we studied new heuristics that improve the intensification of the search directly within the SLS solver. In Chapter 5 we proposed a new heuristic for SLS solvers, which we have embedded in the *gNovelty+* solver, creating the solver *Sparrow*. The main idea of this heuristic is to compute a probability distribution for variables (taken in consideration for flipping) based on different properties of the variables and on parameters that control their influence. *Sparrow* selects the variables for flipping according to this probability distribution. By using appropriate values for these parameters, we show that *Sparrow* reaches state-of-the-art performance on randomly generated SAT problems, strongly outperforming all its competitors. The *Sparrow* solver was submitted to the SAT Competition 2011 where it was ranked first in random category satisfiable sequential and parallel, exceeding also the performance of the best parallel SLS solvers.

Motivated by the remarkable performance of *Sparrow* on randomly generated problems, several research directions regarding *Sparrow* have been further analyzed within this thesis. To understand which solving component of *Sparrow* is the most important one, the solver has been dismantled and reduced to a minimum, yielding the new SAT solver *probSAT*, which is described in Chapter 7. The second direction is the improvement of the *Sparrow* solver for hard combinatorial problems by means of preprocessing.

In Chapter 6 we performed an extended analysis of preprocessing methods with the goal to improve the performance of *Sparrow* in particular and of SLS solvers in general on hard combinatorial problems. We have analyzed the utility of all major available Preprocessing Techniques (PPTs) individually and also of combinations of these. To find the most appropriate PPT for SLS solvers, we have parametrized all PPTs (also allowing each technique to be turned on or off) and then used an automated algorithm configuration procedure to find the best combination and the best parametrization. Our results showed that it is possible to significantly improve the performance of the SLS solver *Sparrow* and of another SLS solver on hard combinatorial problems with appropriate parametrized PPTs. The performance mark reached by *Sparrow* on our benchmark sets significantly exceeded the performance of any other type of solvers, especially that of CDCL and look-ahead solvers. We think that our findings will motivate researchers to rethink the applicability of SLS solvers in combination with appropriate PPTs.

Our analysis has many degrees of freedom that have not been analyzed yet. In our opinion the most promising path is to optimize the parameters of the solver in combination with the parameters of the preprocessor or to incorporate inprocessing techniques within SLS solvers, while the latter is by far more complex.

With the aim of producing an effective and simple SLS solver, we have dismantled the *Sparrow* solver and reduced it to its essential heuristic, which turned out to be the probability based decision heuristic. Based on this finding, we proposed and analyzed in Chapter 7 a new solver named *probSAT*, which uses only the *make* and *break* values of variables to compute a probability distribution. With our simple solver design, we were also able to analyze the importance of these two properties, showing that the *make* property can be ignored, being inessential for reaching good performance. Within thoroughly experimental evaluations, we have shown that *probSAT* reaches state-of-the-art performance on a wide range of randomly generated problems.

The *probSAT* solver is probably the most simple and one of the best SLS solvers currently available. Despite its simplicity, it offers many degrees of freedom, namely exchanging the probability function or incorporating new types of information. We also think that the simplicity of *probSAT* will leverage the design of more efficient SLS implementations.

Within the last part of this thesis, we presented EDACC, a parallel framework for the design, execution and analysis of experiments with empirical algorithms. We presented a brief overview of the main components and their interactions, emphasizing the automated algorithm configuration framework, which is described in more detail, as it has been often used within our solver studies. We also presented the major functionalities of the web frontend, which provides a large variety of statistical analysis tools and enables the organization of competitions.

The plethora of tools provided by EDACC makes research on empirical algorithms a simpler task, allowing researchers to concentrate on their algorithms. The parallel automated algorithm configuration framework, one of the most powerful tools in EDACC, can be generally used for any type of black box optimization. Currently, several other research groups are actively using the system to analyze and optimize their algorithms. The EDACC system was successfully used to organize two major SAT Competitions in 2012 and in 2013, and is currently tested for an oncoming competition in the field of computational logic.

With our work we contributed in several ways to a deeper understanding and improvement of SLS solving techniques, enabling new promising directions for future research.

Technical Specification of the Execution Environment

All the empirical evaluations performed within this thesis were conducted on the clusters from the bwGRiD, which is a distributed grid located at different universities within the state of Baden-Württemberg (southern Germany). The resources used within the bwGRiD have a homogeneous hardware and software configuration allowing a distributed execution of experiments, without having to take into account differences between the clusters.

Hardware Specification The bwGRiD clusters nodes have two sockets with Intel Harpertown quad-core CPUs with 2.83 GHz and 16 GByte RAM. Each of these CPUs contains two dual-core dies. The exact topology (cores, cache levels and sizes and memory) of the CPUs is represented in Figure 9.1[1].

If not mentioned otherwise all cores of a node have been used for performing the experiments. Since two adjacent cores share the L2-cache, an increased number of cache misses can occur, resulting in a higher runtime (than when running only one solver per CPU). This slowdown affects all evaluated solvers approximately in the same manner, thus still allowing a fair comparison. Using less cores of a node to avoid cache sharing would result in a lower number of available resources and thus in a less significant comparison.

Software Specification The operating system of bwGRiD is Scientific Linux. The compilers used in this thesis are mainly *gcc* and *intel* in the versions that were available by the time the experiments have been performed. Solvers compared within one

[1]The CPU and cache topology of a machine can be displayed with the *lstopo* command provided within the *hwloc* package. See `http://www.open-mpi.de/projects/hwloc/` for more details.

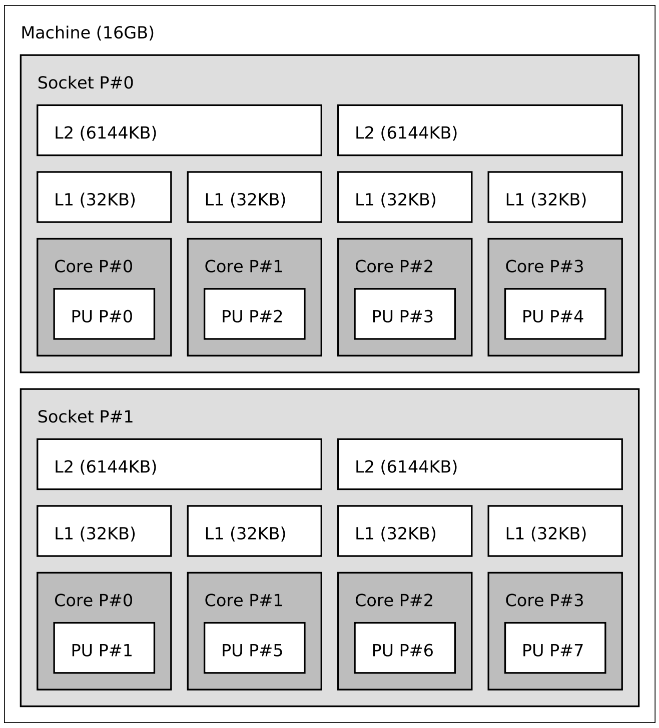


Figure 9.1: The CPU, cache and memory architecture of the computing nodes of the bwGRiD cluster used in all experiments performed in this thesis.

experiment have been compiled with the same compiler version, or a static binary has been used. If not mentioned otherwise, the experimental evaluations within this thesis has been performed with the EDACC system.

Acronyms

3RES	Ternary Resolution
AAC	Automated Algorithm Configuration
ADD2	Add Binary Resolvents
API	Application Programming Interface
BCE	Blocked Clause Elimination
BVA	Bounded Variable Addition
BVE	Bounded Variable Elimination
bwGRiD	Baden-Württemberg Grid
CCE	Covered Clause Elimination
CDCL	Conflict Driven Clause Learning
CNF	Conjunctive Normal Form
CSSC13	Configurable SAT Solver Competition 2013
DB	Database
DIMACS	The Center for Discrete Mathematics and Theoretical Computer Science
DLS	Dynamic Local Search
DPLL	Davis-Putnam-Logemann-Loveland

ELS	Equivalent Literal Substitution
EDACC	Experiment Design and Administration for Computer Cluster
GUI	Graphical User Interface
HTE	Hidden Tautology Elimination
PAC	Probabilistically Approximate Complete
PAR	Penalized Average Run time
PP	Preprocessor
PPT	Preprocessing Technique
PROBE	Failed Literal Detection
SAT	Propositional Satisfiability Problem
SC07	International SAT Competition 2007
SC09	International SAT Competition 2009
SC11	International SAT Competition 2011
SC12	International SAT Challenge 2012
SC13	International SAT Competition 2013
SLS	Stochastic Local Search
SSP	Search Space Partition
STR	Strengthening
SUB	Subsumption
UIP	Unique Implication Point
UP	Unit Propagation
WF	Web Frontend

Bibliography

[AGRY09] Stefan Andrei, Gheorghe Grigoras, Martin C. Rinard, and Roland H. C. Yap. A hierarchy of tractable subclasses for sat and counting sat problems. In Stephen M. Watt, Viorel Negru, Tetsuo Ida, Tudor Jebelean, Dana Petcu, and Daniela Zaharie, editors, *SYNASC*, pages 61–68. IEEE Computer Society, 2009.
Referenced in text: page(s) 5

[AJM04] Dimitris Achlioptas, Haixia Jia, and Cristopher Moore. Hiding satisfying assignments: two are better than one. In *Proceedings of AAAI04*, pages 131–136, 2004.
Referenced in text: page(s) 15, 18

[AKW05] David P. Anderson, Eric Korpela, and Rom Walton. High-performance task distribution for volunteer computing. In *Proceedings of the First International Conference on e-Science and Grid Computing*, E-SCIENCE '05, pages 196–203, Washington, DC, USA, 2005. IEEE Computer Society.
Referenced in text: page(s) 105, 106

[And04] David P. Anderson. Boinc: A system for public-resource computing and storage. In *Proceedings of the 5th IEEE/ACM International Workshop on Grid Computing*, GRID '04, pages 4–10, Washington, DC, USA, 2004. IEEE Computer Society.
Referenced in text: page(s) 116

[AS09] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 399–404,

2009.
Referenced in text: page(s) 73

[AST09] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP-09)*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157, 2009.
Referenced in text: page(s) 102, 106, 107

[Bac02] Fahiem Bacchus. Enhancing davis putnam with extended binary clause reasoning. In *Eighteenth national conference on Artificial intelligence*, pages 613–619, Menlo Park, CA, USA, 2002. American Association for Artificial Intelligence.
Referenced in text: page(s) 65

[BBD$^+$12] Adrian Balint, Anton Belov, Daniel Diepold, Simon Gerber, Matti Järvisalo, and Carsten Sinz, editors. *Proceedings of SAT Challenge 2012: Solver and Benchmark Descriptions*, volume B-2012-2 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2012. ISBN ISBN 978-952-10-8106-4.
Referenced in text: page(s) 4, 9, 10, 14

[BBHJ13] Adrian Balint, Anton Belov, Marijn Heuele, and Matti Järvisalo, editors. *Proceedings of SAT Competition 2013: Solver and Benchmark Descriptions*, volume B-2013-1 of *Department of Computer Science Series of Publications B*. University of Helsinki, 2013.
Referenced in text: page(s) 4, 11, 14, 73

[BBJS12] Adrian Balint, Anton Belov, Matti Järvisalo, and Carsten Sinz. Sat challenge 2012 random SAT track: Description of benchmark generation. *Proceedings of SAT Challenge 2012; Solver and Benchmark Description*, page 72, 2012.
Referenced in text: page(s) 58

[BCDF12] Thomas Bouton, Diego Caminha, David Déharbe, and Pascal Fontaine. Gridtpt: a distributed platform for theorem prover testing. In Renate A. Schmidt, Stephan Schulz, and Boris Konev, editors, *PAAR-2010*, volume 9 of *EPiC Series*, pages 33–39. EasyChair, 2012.
Referenced in text: page(s) 116

[BDG$^+$11] Adrian Balint, Daniel Diepold, Daniel Gall, Simon Gerber, Gregor Kapler, and Robert Retz. Edacc - an advanced platform for the experiment design, administration and analysis of empirical algorithms. In

Learning and Intelligent Optimization, volume 6683 of *Lecture Notes in Computer Science*, pages 586–599. Springer Berlin Heidelberg, 2011.
Referenced in text: page(s) 57, 67, 72, 101, 105, 109

[BdMS05] Clark W. Barrett, Leonardo Mendonça de Moura, and Aaron Stump. Smt-comp: Satisfiability modulo theories competition. In *CAV*, pages 20–23, 2005.
Referenced in text: page(s) 112, 116

[Ber01] Daniel Le Berre. Exploiting the real power of unit propagation lookahead. *Electronic Notes in Discrete Mathematics*, 9:59–80, 2001.
Referenced in text: page(s) 64, 65

[BF10] Adrian Balint and Andreas Fröhlich. *Improving Stochastic Local Search for SAT with a New Probability Distribution*, volume 6175, pages 10–15. Springer, 2010.
Referenced in text: page(s) 3, 49, 52, 72, 78

[BFTH11] Adrian Balint, Andreas Fröhlich, Dave A. D. Tompkins, and Holger H. Hoos. Sparrow2011. Solver description booklet SAT Competition 2011, 2011.
Referenced in text: page(s) 3, 49, 52, 76

[BGKR10] Adrian Balint, Daniel Gall, Gregor Kapler, and Robert Retz. Experiment design and administration for computer clusters for sat-solvers (edacc). *JSAT*, 7(2-3):77–82, 2010.
Referenced in text: page(s) 57, 101, 105

[BHG09] Adrian Balint, Michael Henn, and Oliver Gableske. A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem. In *SAT*, pages 284–297, 2009.
Referenced in text: page(s) 3, 31

[bhg10] bwGRiD (http://www.bw grid.de/). Member of the German D-Grid initiative, funded by the Ministry of Education and Research (Bundesministerium für Bildung und Forschung) and the Ministry for Science, Research and Arts Baden-Württemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg). Technical report, Universities of Baden-Württemberg, 2007-2010.
Referenced in text: page(s) 47, 99, 117

[BHL$^+$01] Wolfgang Barthel, Alexander K. Hartmann, Michele Leone, Federico Ricci-Tersenghi, Martin Weigt, and Riccardo Zecchina. Hiding solutions in random satisfiability problems: A statistical mechanics approach.

CoRR, cond-mat/0111153, 2001.
Referenced in text: page(s) 16

[BHvMW09] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications.* IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.
Referenced in text: page(s) 20, 21

[Bie05] Armin Biere. Resolve and expand. In *Proceedings of the 7th International Conference on Theory and Applications of Satisfiability Testing*, SAT'04, pages 59–70, Berlin, Heidelberg, 2005. Springer-Verlag.
Referenced in text: page(s) 64

[Bie08] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.
Referenced in text: page(s) 40

[Bie11] Armin Biere. Lingeling and friends at the SAT competition 2011. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria., 2011.
Referenced in text: page(s) 92

[Bir04] M. Birattari. *The Problem of Tuning Metaheuristics as Seen from a Machine Learning Perspective.* PhD thesis, Université Libre de Bruxelles, Brussels, Belgium, 2004.
Referenced in text: page(s) 106

[Blo70] Burton H. Bloom. Space/time trade-offs in hash coding with allowable errors. *Commun. ACM*, 13(7):422–426, July 1970.
Referenced in text: page(s) 32

[BM13a] Adrian Balint and Norbert Manthey. Boosting the performance of SLS and CDCL solvers by preprocessor tuning. Proccedings of EasyChair, 07 2013.
Referenced in text: page(s) 3, 49, 54, 61

[BM13b] Adrian Balint and Norbert Manthey. Sparrow+cp3 and sparrowtoriss. Proceeding of SAT Competition 2013: Solver and Benchmark Description, July 2013.
Referenced in text: page(s) 46

[BMZ02] Alfredo Braunstein, Marc Mézard, and Riccardo Zecchina. Survey propagation: an algorithm for satisfiability. *CoRR*, cs.CC/0212002, 2002.
Referenced in text: page(s) 22

[BMZ05] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Structures & Algorithms*, 27(2):201–226, 2005.
Referenced in text: page(s) 22, 89

[BS92] A. Billionnet and A. Sutter. An efficient algorithm for the 3 satisfiability problem. *Operation Research Letters*, 12:29–36, 1992.
Referenced in text: page(s) 65

[BS97] Roberto J. Bayardo Jr. and Robert C. Schrag. Using csp look-back techniques to solve real world SAT instances. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, 1997.
Referenced in text: page(s) 20

[BS04] Daniel Berre and Laurent Simon. The essentials of the SAT 2003 competition. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 452–467. Springer Berlin Heidelberg, 2004.
Referenced in text: page(s) 112

[BS12] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. *Proccedings of SAT2012*, 2012.
Referenced in text: page(s) 3, 41, 43, 58, 77

[BS13] Adrian Balint and Uwe Schöning. probsat. Proceeding of SAT Competition 2013: Solver and Benchmark Description, July 2013.
Referenced in text: page(s) 3, 41, 77

[BSBG02] Eli Ben-Sasson, Yonatan Bilu, and Danny Gutfreund. Finding a randomly planted assignment in a random 3cnf. Technical report, In preparation, 2002.
Referenced in text: page(s) 15

[BW03] Fahiem Bacchus and Jonathan Winter. Effective preprocessing with hyper-resolution and equality reduction. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 341–355, 2003.
Referenced in text: page(s) 65

[BYBS10] Mauro Birattari, Zhi Yuan, Prasanna Balaprakash, and Thomas Stützle. F-Race and iterated F-Race: An overview. In Thomas Bartz-Beielstein,

Marco Chiarandini, Luís Paquete, and Mike Preuss, editors, *Experimental Methods for the Analysis of Optimization Algorithms*, pages 311–336. Springer Berlin Heidelberg, 2010.
Referenced in text: page(s) 85, 106

[Coo71] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the third annual ACM symposium on Theory of computing*, STOC '71, pages 151–158, New York, NY, USA, 1971. ACM.
Referenced in text: page(s) 5

[Cra96] James M. Crawford. Solving satisfiability problems using a combination of systematic and local search. In *Rutgers University*, 1996.
Referenced in text: page(s) 45

[CS12] Shaowei Cai and Kaile Su. Configuration checking with aspiration in local search for sat. In *Proceedings of AAAI-12*, 2012.
Referenced in text: page(s) 58

[Die12] Daniel Diepold. Model-based parallel automated algorithm configuration. Master's thesis, Ulm University, 2012.
Referenced in text: page(s) 109

[DIM93] DIMACS. Satisfiability suggested format. Technical report, DIMACS, 1993.
Referenced in text: page(s) 8

[DLL62] Martin Davis, George Logemann, and Donald Loveland. A machine program for theorem proving. *Communications of the ACM*, 5(7):394–397, 1962.
Referenced in text: page(s) 20

[DP60] Martin Davis and Hilary Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(3):201–215, 1960.
Referenced in text: page(s) 20, 64

[EB05] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75, 2005.
Referenced in text: page(s) 61, 63, 64, 70

[EPV08] Tobias Eibach, Enrico Pilz, and Gunnar Völkel. Attacking Bivium using SAT solvers. In *SAT*, pages 63–76, 2008.
Referenced in text: page(s) 6

[FF04] B. Ferris and Jon Froehlich. Walksat as an informed heuristic to DPLL in SAT solving. *Department of Computer Science, University of Washington, Seattle*, 2004 2004.
Referenced in text: page(s) 45, 46

[FH07] Lei Fang and Michael S. Hsiao. A new hybrid solution to boost SAT solver performance. In *Proceedings of the conference on Design, automation and test in Europe*, DATE '07, pages 1307–1313, San Jose, CA, USA, 2007. EDA Consortium.
Referenced in text: page(s) 45, 46

[FR04] Hai Fang and Wheeler Ruml. Complete local search for propositional satisfiability. In *Proceedings of the Ninteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 161–166, 2004.
Referenced in text: page(s) 46

[Fra96] Jeremy Frank. Weighting for Godot: Learning heuristics for GSAT. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 338–343, 1996.
Referenced in text: page(s) 28

[Fra97] Jeremy Frank. Learning short-term clause weights for GSAT. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 384–389, 1997.
Referenced in text: page(s) 28

[Fre95] Jon William Freeman. *Improvements To Propositional Satisfiability Search Algorithms.* PhD thesis, University of Pennsylvania Philadelphia, PA, USA, 1995.
Referenced in text: page(s) 64

[Fuk04] Alex Fukunaga. Efficient implementations of SAT local search. In *In Seventh Int'l Conf. on Theory and Applications of Satisfiability Testing (SAT2004), 2004 (this volume*, pages 287–292, 2004.
Referenced in text: page(s) 97, 98

[Gab09] Oliver Gableske. Towards the development of a hybrid SAT solver. Master's thesis, Ulm University - Institute of Theoretical Computer Science, 2009.
Referenced in text: page(s) 45, 46

[Gab12] Oliver Gableske. The effect of clause elimination on SLS for SAT. In *Pragmatics of SAT(POS'12)*, 2012.
Referenced in text: page(s) 62, 75

[Gab13] Oliver Gableske. On the interpolation between product-based message passing heuristics for SAT. In *SAT*, pages 293–308, 2013.
Referenced in text: page(s) 22

[Gel05] Allen Van Gelder. Toward leaner binary-clause reasoning in a satisfiability solver. *Ann. Math. Artif. Intell.*, 43(1):239–253, 2005.
Referenced in text: page(s) 65

[GH11] Oliver Gableske and Marijn J. H. Heule. Eagleup: solving random 3-sat using SLS with unit propagation. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 367–368, Berlin, Heidelberg, 2011. Springer-Verlag.
Referenced in text: page(s) 57

[HBHH07] Frank Hutter, Domagoj Babić, Holger H. Hoos, and Alan J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proceedings of the Seventh International Conference on Formal Methods in Computer-Aided Design (FMCAD-07)*, pages 27–34, 2007.
Referenced in text: page(s) 76

[HD04] William S. Havens and Bistra N. Dilkina. A hybrid schema for systematic local search. In Ahmed Y. Tawfik and Scott D. Goodwin, editors, *Advances in Artificial Intelligence*, volume 3060 of *Lecture Notes in Computer Science*, pages 248–260. Springer Berlin Heidelberg, 2004.
Referenced in text: page(s) 45

[HHLB11] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Sequential model-based optimization for general algorithm configuration. In *Proceedings of the 5th international conference on Learning and Intelligent Optimization*, LION'05, pages 507–523, Berlin, Heidelberg, 2011. Springer-Verlag.
Referenced in text: page(s) 67, 102, 106

[HHLB12] Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. Parallel algorithm configuration. In *LION*, pages 55–70, 2012.
Referenced in text: page(s) 106

[HHLBS09] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.
Referenced in text: page(s) 56, 102

[HHS07] Frank Hutter, Holger H. Hoos, and Thomas Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the Twenty-Second National Conference on Artificial Intelligence (AAAI-07)*, pages 1152–1157, 2007.
Referenced in text: page(s) 56, 106

[Hir00] Edward A. Hirsch. SAT local search algorithms: Worst-case study. *J. Autom. Reasoning*, 24(1/2):127–143, 2000.
Referenced in text: page(s) 17

[HJB10a] Marijn Heule, Matti Järvisalo, and Armin Biere. Clause elimination procedures for CNF formulas. In *Proceedings of the 17th international conference on Logic for programming, artificial intelligence, and reasoning*, pages 357–371. Springer-Verlag, 2010.
Referenced in text: page(s) 61, 63, 65, 74

[HJB10b] Marijn Heule, Matti Järvisalo, and Armin Biere. Covered clause elimination. *CoRR*, abs/1011.5202, 2010.
Referenced in text: page(s) 61, 63, 65, 74, 75

[HJB11] Marijn J. H. Heule, Matti Järvisalo, and Armin Biere. Efficient cnf simplification based on binary implication graphs. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 201–215, Berlin, Heidelberg, 2011. Springer-Verlag.
Referenced in text: page(s) 61, 63, 65, 74

[HLDV02] Djamal Habet, Chu Min Li, Laure Devendeville, and Michel Vasquez. A hybrid approach for SAT. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, CP '02, pages 172–184, London, UK, UK, 2002. Springer-Verlag.
Referenced in text: page(s) 46

[Hoo99] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 661–666, 1999.
Referenced in text: page(s) 25, 26

[Hoo02] Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the Eighteenth National Conference in Artificial Intelligence (AAAI-02)*, pages 655–660, 2002.
Referenced in text: page(s) 26, 29, 82

[HS05] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, 2005.
Referenced in text: page(s) 23, 97

[HTH02] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248, 2002.
Referenced in text: page(s) 28, 29

[Hut09] Frank Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University of British Columbia, 2009.
Referenced in text: page(s) 76, 106

[HvM09] Marijn Heule and Hans van Maaren. Look-ahead based SAT solvers. In Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, pages 155–184. IOS Press, 2009.
Referenced in text: page(s) 65

[HXHLB12] Frank Hutter, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Algorithm runtime prediction: The state of the art. *CoRR*, abs/1211.0906, 2012.
Referenced in text: page(s) 8

[JBH10] Matti Järvisalo, Armin Biere, and Marijn Heule. Blocked clause elimination. In *Proceedings of the 16th international conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS'10, pages 129–144, Berlin, Heidelberg, 2010. Springer-Verlag.
Referenced in text: page(s) 61, 63, 64, 74

[JL02] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. *Artif. Intell.*, 139(1):21–45, July 2002.
Referenced in text: page(s) 46

[JMS05] Haixia Jia, Cristopher Moore, and Doug Strain. Generating hard satisfiable formulas by hiding solutions deceptively. In *In AAAI*, pages 384–389. AAAI Press, 2005.
Referenced in text: page(s) 16, 18

[KS03] Henry Kautz and Bart Selman. Ten challenges redux: Recent progress in propositional reasoning and search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 1–18, 2003.
Referenced in text: page(s) 2

[KSMS11] Hadi Katebi, Karem A. Sakallah, and Joao P. Marques-Silva. Empirical study of the anatomy of modern SAT solvers. In *Proc. 14th Int. Conf. on Theory and Applications of Satisfiability Testing (SAT '11)*, volume 6695 of *LNCS*, pages 343–356. Springer, 2011.
Referenced in text: page(s) 75

[KSS10] Lukas Kroc, Ashish Sabharwal, and Bart Selman. An empirical study of optimal noise and runtime distributions in local search. In *SAT*, pages 346–351, 2010.
Referenced in text: page(s) 26, 82, 89, 90, 95

[LA97a] Chu Min Li and Anbulagan. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on Principles and Practice of Constraint Programming (CP-97)*, volume 1330 of *Lecture Notes in Computer Science*, pages 341–355, 1997.
Referenced in text: page(s) 65

[LA97b] Chu Min Li and Anbulagan Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th international joint conference on Artifical intelligence - Volume 1*, IJCAI'97, pages 366–371, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.
Referenced in text: page(s) 64

[LH05] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 158–172, 2005.
Referenced in text: page(s) 26, 27, 50, 59

[LJPJ02] Chu Min Li, Bernard Jurkowiak, Paul W. Purdom, and Jr. Integrating symmetry breaking into a dll procedure. In *In Fifth International Symposium on the Theory and Applications of Satisfiability Testing (SAT2002*, pages 149–155, 2002.
Referenced in text: page(s) 11

[LL12] Chu Min Li and Yu Li. Satisfying versus falsifying in local search for satisfiability. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, volume 7317 of *Lecture Notes in Computer Science*, pages 477–478. Springer Berlin Heidelberg, 2012.
Referenced in text: page(s) 57, 62, 71, 75

[LMS03] Inês Lynce and João Marques-Silva. Probing-based preprocessing techniques for propositional satisfiability. In *Proceedings of the 15th IEEE International Conference on Tools with Artificial Intelligence*, ICTAI '03, pages 105–110. IEEE Computer Society, 2003.
Referenced in text: page(s) 64

[LMS08] Florian Letombe and Joao Marques-Silva. Improvements to hybrid incremental SAT algorithms. In *International Conference on Theory and Applications of Satisfiability Testing*. Springer LNCS, May 2008. Event Dates: May 2008.
Referenced in text: page(s) 46

[LWZ07] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and look-ahead in local search for SAT. In *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT-07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 121–133, 2007.
Referenced in text: page(s) 26, 27

[Man12] Norbert Manthey. Coprocessor 2.0 - a flexible CNF simplifier - (tool presentation). In Alessandro Cimatti and Roberto Sebastiani, editors, *SAT*, volume 7317 of *Lecture Notes in Computer Science*, pages 436–441. Springer, 2012.
Referenced in text: page(s) 61

[MD05] Radu Marinescu and Rina Dechter. And/or branch-and-bound for graphical models. In *IJCAI*, pages 224–229, 2005.
Referenced in text: page(s) 107

[MHB12] Norbert Manthey, Marijn J. H. Heule, and Armin Biere. Automated reencoding of boolean formulas. In *Proceedings of Haifa Verification Conference 2012*, 2012.
Referenced in text: page(s) 61, 63, 64, 74

[MMZ$^+$01] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the Thirty-Eighth Design Automation Conference (DAC-01)*,

2001.
Referenced in text: page(s) 65

[MMZ06] Stephan Mertens, Marc Mézard, and Riccardo Zecchina. Threshold values of random k-SAT from the cavity method. *Random Struct. Algorithms*, 28(3):340–373, 2006.
Referenced in text: page(s) 13, 14, 22

[Mor93] Paul Morris. The breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference in Artificial Intelligence (AAAI-93)*, pages 40–45, 1993.
Referenced in text: page(s) 28

[MSG98] Bertrand Mazure, Lakhdar Saïs, and Éric Grégoire. Boosting complete techniques thanks to local search methods. *Annals of Mathematics and Artificial Intelligence*, 22(3-4):319–331, 1998.
Referenced in text: page(s) 45

[MSK97] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 321–326, 1997.
Referenced in text: page(s) 26, 59, 95

[MSL92] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In *Proceeding of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.
Referenced in text: page(s) 13, 14

[MSS99] João P. Marques-Silva and Karem A. Sakallah. GRASP: A search algorithm for propositional satisfiability. *IEEE Transactions on Computers*, 48(5):506–521, 1999.
Referenced in text: page(s) 20

[MT99] Patrick Mills and Edward Tsang. Guided local search applied to the satisfiability (SAT) problem. In *Proceedings of the Fifteenth National Conference of the Australian Society for Operations Research (ASOR-99)*, pages 872–883, 1999.
Referenced in text: page(s) 28

[MZ02] Marc Mézard and Riccardo Zecchina. Random K-satisfiability problem: From an analytic solution to an efficient algorithm. *Physical Review E*, 66(056126), 2002.
Referenced in text: page(s) 22

[Pap91] Christos H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS-91)*, pages 163–169, 1991.
Referenced in text: page(s) 25, 78

[Pha06] Duc Nghia Pham. *Modelling and Exploiting Structures in Solving Propositional Satisfiability Problems.* PhD thesis, Griffith University, Quennsland, Australia, 2006.
Referenced in text: page(s) 75

[PHS08] Cédric Piette, Youssef Hamadi, and Lakhdar Saïs. Vivifying propositional clausal formulae. In *Proceedings of the 2008 Conference on ECAI 2008: 18th European Conference on Artificial Intelligence*, pages 525–529, Amsterdam, The Netherlands, The Netherlands, 2008. IOS Press.
Referenced in text: page(s) 68

[PL06] Steven Prestwich and Inês Lynce. Local search for unsatisfiability. In *Proceedings of the Ninth International Conference on Theory and Applications of Satisfiability Testing (SAT-06)*, volume 4121 of *Lecture Notes in Computer Science*, pages 283–296, 2006.
Referenced in text: page(s) 21

[PTGS07] Duc Nghia Pham, John Thornton, Charles Gretton, and Abdul Sattar. Advances in local search for satisfiability. In *Proceedings of the Twentieth Australian Joint Conference on Artificial Intelligence (AI-07)*, volume 4830 of *Lecture Notes in Computer Science*, pages 213–222, 2007.
Referenced in text: page(s) 29, 40, 50

[PTGS08] Duc Nghia Pham, John Thornton, Charles Gretton, and Abdul Sattar. Combining adaptive and dynamic local search for satisfiability. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:149–172, 2008.
Referenced in text: page(s) 29, 32

[Rob65] J. A. Robinson. A machine-oriented logic based on the resolution principle. *Journal of the ACM*, 12(1):23–41, 1965.
Referenced in text: page(s) 63

[Rou11] Olivier Roussel. Controlling a solver execution with the runsolver tool. *JSAT*, 7(4):139–144, 2011.
Referenced in text: page(s) 105, 115

[SAO05] Sakari Seitz, Mikko Alava, and Pekka Orponen. Focused local search for random 3-satisfiability. *CoRR*, abs/cond-mat/0501707, 2005.
Referenced in text: page(s) 80

[SB98] Richard S. Sutton and Andrew G. Barto. *Reinforcement Learning: An Introduction*. MIT Press, 1998.
Referenced in text: page(s) 51

[SB08] Zbigniew Stachniak and Anton Belov. Speeding-up non-clausal local search for propositional satisfiability with clause learning. In *Proceedings of the 11th international conference on Theory and applications of satisfiability testing*, SAT'08, pages 257–270, Berlin, Heidelberg, 2008. Springer-Verlag.
Referenced in text: page(s) 15

[SC01] Laurent Simon and Philippe Chatalic. Satex: A web-based framework for SAT experimentation. *Electronic Notes in Discrete Mathematics*, 9:129–149, 2001.
Referenced in text: page(s) 116

[Sch99] Uwe Schöning. A probabilistic algorithm for k-SAT and constraint satisfaction problems. In *Proceedings of the Fourtieth Annual Symposium on Foundations of Computer Science (FOCS-99)*, pages 410–414, 1999.
Referenced in text: page(s) 1, 25, 79, 82, 99

[SD05] Carsten Sinz and Edda-Maria Dieringer. DPvis - a tool to visualize structured SAT instances. In *Proc. of the 8th Intl. Conf. on Theory and Applicationsof Satisfiability Testing (SAT 2004)*, pages 257–268, St. Andrews, Scotland, June 2005. Springer-Verlag.
Referenced in text: page(s) 9

[She07] David J. Sheskin. *Handbook of Parametric and Nonparametric Statistical Procedures*. Chapman & Hall/CRC, 4 edition, 2007.
Referenced in text: page(s) 85

[SKC94] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 337–343, 1994.
Referenced in text: page(s) 26, 28, 41

[SLM92] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.
Referenced in text: page(s) 27

[SP05] Sathiamoorthy Subbarayan and Dhiraj K. Pradhan. NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. In *Proceedings of the Seventh International Conference on The-*

ory and Applications of Satisfiability Testing (SAT-04), volume 3542 of *Lecture Notes in Computer Science*, pages 276–291, 2005.
Referenced in text: page(s) 64

[SS00] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. In *Proceedings of the Seventeenth National Conference in Artificial Intelligence (AAAI-00)*, pages 297–302, 2000.
Referenced in text: page(s) 28

[SSH01] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic Boolean programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 334–341, 2001.
Referenced in text: page(s) 28

[Sut10] Geoff Sutcliffe. The CADE-22 automated theorem proving system competition - CASC-22. *AI Commun.*, 23(1):47–59, January 2010.
Referenced in text: page(s) 112

[SW98] Yi Shang and Benjamin W. Wah. A discrete Lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–100, 1998.
Referenced in text: page(s) 28

[TBH11] Dave A.D. Tompkins, Adrian Balint, and Holger H. Hoos. Captain Jack: New variable selection heuristics in local search for SAT. In KaremA. Sakallah and Laurent Simon, editors, *Theory and Applications of Satisfiability Testing - SAT 2011*, volume 6695 of *Lecture Notes in Computer Science*, pages 302–316. Springer Berlin Heidelberg, 2011.
Referenced in text: page(s) 3, 24, 40, 41, 52, 56, 84

[TH04] Dave A. D. Tompkins and Holger H. Hoos. Ubcsat: An implementation and experimentation environment for sls algorithms for SAT & max-sat. In *SAT*, 2004.
Referenced in text: page(s) 56

[TH05] Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *Revised Selected Papers of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, volume 3542 of *Lecture Notes in Computer Science*, pages 306–320, 2005.
Referenced in text: page(s) 23

[Tho05] John Thornton. Clause weighting local search for SAT. *Journal of Automated Reasoning*, 35(1-3):97–142, 2005.
Referenced in text: page(s) 28, 29

[Tom10] David Andrew Douglas Tompkins. *Dynamic Local Search for SAT: Design, Insights and Analysis.* PhD thesis, University of British Columbia, October 2010.
Referenced in text: page(s) 28, 98

[TTL05] Douglas Thain, Todd Tannenbaum, and Miron Livny. Distributed computing in practice: the condor experience: Research articles. *Concurr. Comput. : Pract. Exper.*, 17(2-4):323–356, February 2005.
Referenced in text: page(s) 116

[Tur88] Jonathan S. Turner. Almost all k-colorable graphs are easy to color. *J. Algorithms*, 9(1):63–82, 1988.
Referenced in text: page(s) 6

[VG11] Allen Van Gelder. Careful ranking of multiple solvers with timeouts and ties. In *Proceedings of the 14th international conference on Theory and application of satisfiability testing*, SAT'11, pages 317–328, Berlin, Heidelberg, 2011. Springer-Verlag.
Referenced in text: page(s) 115

[WH06] Qingwei Wu and Michael S. Hsiao. A new simulation-based property checking algorithm based on partitioned alternative search space traversal. *IEEE Trans. Comput.*, 55(11):1325–1334, November 2006.
Referenced in text: page(s) 34

[WLZ08] Wanxia Wei, Chu Min Li, and Harry Zhang. A switching criterion for intensification and diversification in local search for SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:219–237, 2008.
Referenced in text: page(s) 27

[WS02] W. Wei and B. Selman. Accelerating random walks. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 216–232., 2002.
Referenced in text: page(s) 65, 75

[WW99] Zhe Wu and Benjamin W. Wah. Trap escaping strategies in discrete Lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In *Proceedings of the Sixteenth National Conference*

on Artificial Intelligence (AAAI-99), pages 673–678, 1999.
Referenced in text: page(s) 28

[WW00] Zhe Wu and Benjamin W. Wah. An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In *Proceedings of the Seventeenth National Conference in Artificial Intelligence (AAAI-00)*, pages 310–315, 2000.
Referenced in text: page(s) 28

[Yao82] Andrew C. Yao. Theory and application of trapdoor functions. In *Proceedings of the 23rd Annual Symposium on Foundations of Computer Science*, SFCS '82, pages 80–91, Washington, DC, USA, 1982. IEEE Computer Society.
Referenced in text: page(s) 18

[Zha04] Weixiong Zhang. Configuration landscape analysis and backbone guided local search.: Part i: Satisfiability and maximum satisfiability. *Artificial Intelligence*, 158(1):1 – 26, 2004.
Referenced in text: page(s) 33