

SATenstein: Automatically Building Local Search SAT Solvers From Components

by

Ashiqur Rahman KhudaBukhsh

A THESIS SUBMITTED IN PARTIAL FULFILMENT OF
THE REQUIREMENTS FOR THE DEGREE OF

Master of Science

in

The Faculty of Graduate Studies

(Computer Science)

The University of British Columbia

(Vancouver)

October 2009

© Ashiqur Rahman KhudaBukhsh 2009

Abstract

Designing high-performance solvers for computationally hard problems is a difficult and often time-consuming task. It is often the case that a new solver is created by augmenting an existing algorithm with a mechanism found in a different algorithm or by combining components from different algorithms. In this work, we demonstrate that this task can be automated in the context of stochastic local search (SLS) solvers for the propositional satisfiability problem (SAT). We first introduce a generalized, highly parameterized solver framework, dubbed **SATenstein**, that includes components drawn from or inspired by existing high-performance SLS algorithms for SAT. In **SATenstein**, we exposed several design elements in the form of parameters that control both the selection and the behavior of components. We also exposed some parameters that were hard-coded into the implementations of the algorithms we studied. By setting these parameters, **SATenstein** can be instantiated as a huge number of different solvers, including many known high-performance solvers and trillions of solvers never studied before. We used an automated algorithm configuration procedure to find instantiations of **SATenstein** that perform well on several well-known, challenging distributions of SAT instances. Overall, we consistently obtained significant improvements over the previous best-performing SLS algorithms, despite expending minimal manual effort.

Contents

Abstract	ii
Contents	iii
List of Tables	v
List of Figures	viii
Acknowledgements	ix
Dedication	x
1 Introduction	1
2 Background and Related Work	4
2.1 The Satisfiability Problem	4
2.2 Related Work on Automatic Algorithm Configuration, Portfolios, and Algorithm Synthesis	5
2.2.1 Automatic Algorithm Configuration	6
2.2.2 Parameter Tuning and Algorithm Configuration	7
2.2.3 Algorithm Selection and Portfolios	7
2.2.4 Further Related Work	9
2.3 Local Search for SAT	9
2.3.1 Different Classes of SLS Algorithms for SAT	9
2.3.2 Recent Trends in SLS Algorithms	14
2.3.3 UBCSAT	15
3 SATenstein-LS	16
3.1 Design	16
3.2 Implementation and Validation	22
4 Experimental Setup	25
4.1 Instance Distributions	25
4.2 Tuning Scenario and PAR	26
4.3 Solvers Used for Performance Comparison	28
4.4 Execution Environment	30

5 Results	31
5.1 Comparison with Challengers	31
5.2 Comparison with a Hypothetical Oracle	36
5.3 Comparison with Complete Solvers	36
5.4 Heterogeneous Mixtures of Instance Distributions	39
5.5 Configurations Found	40
6 Conclusions and Future Work	45
Bibliography	47
A SATenstein-LS User Manual	54
A.1 Introduction	54
A.2 How to Run SATenstein-LS	54
A.3 Configurable Parameters	54
A.4 Example Parameter Configurations	55
A.4.1 Novelty+	55
A.4.2 gNovelty+	56
A.4.3 SAPS	56

List of Tables

2.1	<i>Different search diversification strategies found in Novelty variants.</i>	12
2.2	<i>Different strategies used in G^2WSAT components.</i>	14
3.1	<i>SATenstein-LS components.</i>	18
3.2	<i>Categorical parameters of SATenstein-LS. Unless otherwise mentioned, for more than one parameters, the conditional operator used in the the ‘Active When’ column is AND.</i>	19
3.3	<i>Integer parameters of SATenstein-LS. Unless otherwise mentioned, for more than one parameters, the conditional operator used in the the ‘Active When’ column is AND.</i>	20
3.4	<i>Continuous parameters of SATenstein-LS. Unless otherwise mentioned, for more than one parameters, the conditional operator used in the the ‘Active When’ column is AND.</i>	20
3.5	<i>Design choices for selectFromPromisingList() and corresponding values of selectPromVariable.</i>	21
3.6	<i>List of heuristics chosen by the parameter heuristic and dependent parameters.</i>	21
4.1	<i>Our six benchmark distributions.</i>	26
4.2	<i>Integer parameters of SATenstein-LS and the values considered during ParamILS tuning. For each parameter, the known defaults are highlighted in bold. For parameters that are first introduced in SATenstein-LS, the default values are indicated with an underline.</i>	27
4.3	<i>Values considered for continuous parameters of SATenstein-LS during ParamILS tuning. For each parameter, the known defaults are highlighted in bold. For parameters that are first introduced in SATenstein-LS, the default values are indicated with an underline.</i>	28
4.4	<i>Our eleven challenger algorithms.</i>	29
4.5	<i>Complete solvers we compared against.</i>	29

5.1	Performance summary of <i>SATenstein-LS</i> and the 11 challengers. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm i on distribution j as $a/b/c$, where a (top) is the the penalized average runtime; b (middle) is the median of the median runtimes over all instances (not defined if fewer than half of the median runs failed to find a solution within the cutoff time); c (bottom) is the percentage of instances solved (i.e., having median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold, and additionally the best-scoring challenger(s) are indicated with an underline.	32
5.2	Percentage of instances on which <i>SATenstein-LS</i> achieved better median runtime than each of the 11 challengers. Medians were taken over 10 runs on each instance with a cutoff time of 600 CPU seconds per run. When both <i>SATenstein-LS</i> and challenger solved a given instance and achieved indistinguishable median runtimes, we counted that instance as 0.5 for <i>SATenstein-LS</i> and 0.5 for the challenger. . .	34
5.3	Performance summary of <i>SATenstein-LS</i> and the complete solvers. Every complete solver was run once (<i>SATenstein-LS</i> was run 10 times) on each instance with a per-run cutoff of 600 CPU seconds. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm i on distribution j as $a/b/c$, where a (top) is the the penalized average runtime; b (middle) is the median runtime over all instances (for <i>SATenstein-LS</i> , it is the median of the median runtimes over all instances); c (bottom) is the percentage of instances solved. The best-scoring algorithm(s) in each row are indicated in bold, and additionally the best-scoring complete solver(s) are indicated with an underline. An empty cell, $\langle i, j \rangle$, means we did not run complete solver i on the test set of distribution j . . .	38
5.4	Description of heterogenous distributions we used. In each case, we sampled uniformly at random from the component homogeneous distributions.	39
5.5	Performance summary of <i>SATenstein-LS</i> and the best challenger. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set penalized average runtime (PAR) of algorithm i on distribution j	39
5.6	Performance summary of <i>SATenstein-LS</i> obtained for each category (Handmade, Random, and Industrial) and the Big-mix distribution on a per-distribution basis. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set penalized average runtime (PAR) of algorithm i on distribution j	40
5.7	<i>SATenstein-LS</i> configurations.	40

5.8	Distance between <i>SATenstein-LS</i> [<i>D</i>] and each challenger. Each cell $\langle i, j \rangle$ summarizes the distance between challenger <i>j</i> and <i>SATenstein-LS</i> [<i>i</i>] as <i>a/b/c</i> , where <i>a</i> (top) is the level 0 distance; <i>b</i> (middle) is the level 1 distance; <i>c</i> (bottom) is the level 2 distance. For each distribution, the closest algorithm distances(s) are highlighted in bold.	43
5.9	<i>SATenstein-LS</i> parameter configuration found for each distribution.	44
A.1	Name mapping between <i>SATenstein-LS</i> parameters discussed so far and <i>SATenstein-LS</i> parameters exposed on the command line.	55

List of Figures

3.1	<i>Dependencies between SATenstein-LS parameters.</i>	18
3.2	<i>Quantile-quantile plots of the run-length distributions for SATenstein-LS vs. UBCSAT (Tompkins and Hoos, 2004) implementations of various well-known SLS solvers. Each implementation was run 5000 times on instance bw_large.a with a per-run cutoff time of 600 CPU seconds. The p-value of the performed KS-test is given in the caption of each subfigure.</i>	23
3.3	<i>Quantile-quantile plots of the run-length distributions for SATenstein-LS vs. SAT Competition, 2005 submission of VW (Prestwich, 2005) and SAT Competition, 2007 submission of gNovelty⁺ (Pham et al., 2008). Each implementation is run 5000 times on benchmark instance bw_large.a with a per-run cutoff of 600 CPU seconds. The p-value of the performed KS-test is given in the caption of each subfigure.</i>	24
5.1	<i>Scatter plot of median runtimes of SATenstein-LS[R3SAT] vs PAWS on the test set for R3SAT. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run.</i>	34
5.2	<i>Penalized average runtime for our SATenstein-LS solver and the top three challengers for each distribution, as a function of cutoff, based on 10 runs on each test-set instance.</i>	35
5.3	<i>Scatter plot of median runtimes of SATenstein-LS vs Oracle on the test set for each distribution.</i>	37

Acknowledgements

I express my gratitude to my supervisors, Dr. Kevin Leyton-Brown and Dr. Holger H. Hoos. I thank them for giving me exciting work, for always finding time to discuss it with me. In our work and in other projects, they demonstrated how one does good research by doing it themselves. Their help, advice, and example is much appreciated, and has, I believe, enhanced my scholastic abilities considerably.

I am grateful to Dr. Jim Little for taking the time to be my second reader.

I would like to thank Lin Xu, co-author of our IJCAI 2009 paper. Lin's guidance in designing experiments and generating problem instances were very valuable. His inputs to my thesis have been very useful. Thanks to Dave Tompkins, who got me started with UBCSAT, a crucial part of working on **SATenstein-LS**, and Dave Thompson, for his help with my IJCAI presentation. Thanks to Frank Hutter and Chris Fawcett for their help with ParamILS, as well as for clearing up concepts in empirical algorithmics. Thanks also to the participants in the EARG reading-group for many stimulating discussions.

I would also like to thank CaraBeth and Hoyt for proofreading my thesis.

My stay at UBC was fun-filled and full of extra-curricular activities. I would like to thank all members of our band Algo-Rhythm–Anthony, Baharak, Elan, Emtiyaz, CaraBeth, Lloyd, and Pooja—for so many wonderful moments. I also cherish my experience with a UBC Bengali students' drama group–Anirban, Angshul, Avik, Dipanjan, Debojit, Madhuja and Sridarshini. Thanks to my friend Lucas and roommate Chaitanya for making the campus life an experience to remember. I would also like to thank Ayan, Imon, Rajat, Sudipto, and Sayantan, my friends from India who were always there when I needed them.

My heartfelt thanks go to my manager, Scott M. Roberts, whose encouragement made my life at Microsoft a wonderful experience. He was very kind to allow me to attend the IJCAI conference and present this work, knowing that how important it was for me and how hard I worked on this. My acknowledgements would not be complete without thanking Jian, my friend and officemate at Microsoft.

Finally, I would like to thank my parents, my sisters and close family members for motivating and supporting me with my studies.

Dedication

To my cute little friends — Limko, Bolon, and Simba.

Chapter 1

Introduction

In the classic novel *Frankenstein; or, The Modern Prometheus* by Mary Shelley, a brilliant scientist, Victor Frankenstein was obsessed with the idea of creating a perfect human being by combining scavenged human body parts. Our approach in this work is based on the same general idea: we scavenge components from existing high-performance algorithms for a given problem and combine them to build new high-performance algorithms. Our idea finds support from the fact that many new solvers are created by augmenting an existing algorithm with a mechanism found in a different algorithm (see, e.g., Hoos, 2002; Li *et al.*, 2007b) or by combining components of different algorithms (see, e.g., Pham *et al.*, 2008). Unlike Victor Frankenstein’s creation, our algorithm is built using an automated construction process that enables us to optimize performance for specific types of tasks (i.e., problem instances) with minimal human effort.¹

Traditionally, high-performance heuristic algorithms are designed in an iterative, manual process in which most of the design choices are fixed during development time, leaving only a small number of parameters exposed to the user. In many cases, such unexposed parameters are not clearly visible even at the source code level (e.g., they may be hard-coded as numerical constants). In contrast, we propose a new approach to heuristic algorithm design in which the designer fixes as few design choices as possible at development time, instead exposing a huge number of design choices in the form of parameters. This approach removes from the algorithm designer the burden of making early design decisions without knowing how different algorithm components will interact on problem distributions of interest. Instead, it encourages the designer to consider several alternative designs drawn from known solvers as well as novel mechanisms for a given algorithm component. Of course, such flexible, highly parameterized algorithms must be instantiated to solve actual problems. With the availability of advanced automated parameter configurators and cheap computational resources, for a given distribution, it becomes practical to find good parameter configurations from such huge parameter spaces by using automated parameter configurators (see, e.g., Hutter *et al.*, 2007b; Birattari *et al.*, 2002; Chiarandini *et al.*, 2008) becomes practical. For a more general treatise of this approach, see also Hoos, 2008.

Our approach can be seen as a part of the growing trend towards automating the process of constructing high-performance algorithms. Hard combinato-

¹This also distinguishes our approach from work by Montes de Oca *et al.*, 2007, which used the Frankenstein metaphor in the context of manually combining algorithm components drawn from existing high-performance particle swarm optimisation procedures.

rial problems arise in numerous real-life applications. The demand for efficient solvers for such problems is ever increasing. With computational resources getting cheaper day by day, it becomes possible to reduce human effort by utilizing cheap computing power to automate the algorithm-building process. Examples of work along these lines are discussed in Chapter 2.

Although our general idea is not specifically tailored to a particular domain, in this work we address the challenge of constructing stochastic local search (SLS) algorithms for the propositional satisfiability problem (SAT): an \mathcal{NP} -complete problem of great interest for scientists and industry people alike. Improving the state of the art in this domain is not an easy task. Indeed, a substantial amount of research and engineering effort has been expended in building SLS algorithms for SAT since the late 1980s (see, e.g., Selman *et al.*, 1992; Hoos, 2002; Pham *et al.*, 2008). SLS-based solvers show a continuing trend of dominance over other classes of SAT solvers on random satisfiable instances. Since 2003, they won all medals in the random satisfiable category of the SAT Competition (SAT Competition, 2009). The **SATzilla** solvers that won the SAT+UNSAT tracks of two of the three categories (**handmade** and **random**) in the 2007 SAT Competition also make use of SLS procedures. Overall, three new SLS-based solvers emerged as winners in the random satisfiable category and new SLS-based solvers have been introduced roughly every year for the past 16 years. This highlights the research community’s continuing interest in this area and indicates the scope for further performance improvement. In some cases, the performance improvements achieved by these solvers are known to be the result of a design process during which a large number of candidate designs were tested (see, e.g., McAllester *et al.*, 1997); indeed, it is our belief that this is usually the case.

We have leveraged this rich body of work on SLS algorithms for SAT to design our generalized algorithm **SATenstein-LS** (KhudaBukhsh *et al.*, 2009). This algorithm draws mechanisms from two dozen existing high-performance SLS SAT solvers and also incorporates many novel strategies. Our design space contains a total of 4.82×10^{12} candidate solvers, including both most of the state-of-the-art SLS SAT solvers known from the literature and a vast number of novel designs that have not been studied before.

Another important element is the algorithm configurator we used. In principle, any powerful algorithm configuration tool could be used to find **SATenstein** configurations that perform well on problem distributions of interest. In our case, we used an off-the-shelf algorithm configuration procedure, **ParamILS** (Hutter *et al.*, 2007b, 2008). **ParamILS** has been successfully used to configure a diverse range of highly parameterized applications (see, e.g., **CPLEX** (Hutter *et al.*, 2008); **Spear** (Hutter *et al.*, 2007a); Chiarandini *et al.*, 2008).

We configured our generalized design, **SATenstein-LS**, on six well-known SAT distributions ranging from hard random 3-SAT instances to SAT-encoded factoring and software verification problems. We evaluated the performance of the new solvers thus obtained by comparing them against 11 state-of-the-art SLS-based solvers and 2 state-of-the-art complete solvers. Our results show that our new solvers outperformed existing high-performance SLS solvers in

all six distributions. Among these six distributions, our new solvers improved the state of the art for four. On the remaining two distributions — in particular the SAT-encoded software verification instances — our new solvers still emerged as the best-performing SLS-based solvers and considerably narrowed the performance gap between SLS-based algorithms and DPLL-based solvers. We analyzed the configurations of these new solvers and related them to existing high-performance solvers in various ways. We defined a distance measure between two **SATenstein-LS** configurations and computed the distance between our new solvers and known high-performance SLS solvers. Based on this measure, we found that many of our new solvers were considerably different from existing high-performance solvers.

The remainder of this thesis is organized as follows. After a discussion of related work (Chapter 2), we describe the design and implementation of **SATenstein-LS** (Chapter 3). This is followed by a description of the setup we used for empirically evaluating **SATenstein-LS** (Chapter 4) and a presentation and discussion of the results from our experiments (Chapter 5). We end with some general conclusions and an outlook on possible future work (Chapter 6).

Chapter 2

Background and Related Work

In this chapter, we first formally define the propositional satisfiability problem (SAT) and discuss why SAT is an important problem domain. We also mention the state of the art techniques used in SAT solving. Next, we discuss various lines of research that are related to our work on **SATenstein**. We then highlight some important aspects of four broad categories of stochastic local search (SLS) SAT solvers. We finally conclude with outlining recent research trends in SLS algorithm development.

2.1 The Satisfiability Problem

The propositional satisfiability problem (SAT) asks, for a given propositional formula F , whether there exists a complete assignment of truth values to the variables of F under which F evaluates to true (see, e.g., Biere *et al.*, 2009). Such complete assignments are also known as models. F is called satisfiable if there exists at least one such model. Otherwise, the formula is called unsatisfiable. A propositional satisfiability problem instance is usually represented as a conjunction of disjunctions, where each disjunction has one or more arguments that may either be variables or the negation of variables. Such arguments are known as literals. Each disjunction is called a clause. This syntactic form of representing a propositional satisfiability problem instance is known as conjunctive normal form. Thus the goal for a SAT solver is to find a variable assignment that satisfies all clauses or to prove that no such assignment exists.

Many of the hard combinatorial problems that arise in practical scenarios belong to the complexity class of so-called \mathcal{NP} -complete problems (see, e.g., Cook, 1971; Applegate *et al.*, 2006; Biere *et al.*, 1999; Fraenkel, 1993; Pop *et al.*, 2002). It is widely believed that there does not exist any deterministic, polynomial time solution to any of these \mathcal{NP} -complete problems. SAT is one of the most-studied \mathcal{NP} -complete problems. In fact, there are conferences¹ and journals² solely dedicated to the satisfiability problem. SAT is also important because \mathcal{NP} -complete problems from other classes can be encoded as SAT problem instances

¹For example: Proceedings of the International Conferences on Theory and Applications of Satisfiability Testing.

²For example: Journal on Satisfiability, Boolean Modeling and Computation (see, <http://www.isa.ewi.tudelft.nl/Jsat/>).

and effectively solved by using a SAT solver. For example, using this approach, problems arising in applications such as planning (e.g., Kautz and Selman, 1996, 1999), scheduling (Crawford and Baker, 1994), graph-coloring (Gelder, 2002), bounded model checking (Biere *et al.*, 1999), and formal verification (Stephan *et al.*, 1992) have been effectively solved.

Over the last few decades, a considerable amount of research and engineering effort has been expended to improve the state of the art in SAT solving. Present high-performance SAT solvers include tree-search algorithms (see, e.g., Sörensson and Eén, 2007; Heule and Maaren, 2007a; Biere, 2008; Dubois and Dequen, 2001), local search algorithms (see, e.g., Hutter *et al.*, 2002; Hoos, 2002; Pham and Anbulagan, 2007; Li and Huang, 2005; Prestwich, 2005; Pham *et al.*, 2008; Li *et al.*, 2007b) and resolution-based preprocessors (see, e.g., Davis and Putnam, 1960; Subbarayan and Pradhan, 2005; Dechter and Rish, 1994; Bacchus and Winter, 2003; Bacchus, 2002a,b). Every year, competitions are held (namely the SAT Competition and the SAT Race) where new state-of-the-art solvers emerge. The trend of continuing performance improvement in the solvers that take part in SAT competitions suggests that there is scope for yet further performance improvement.

2.2 Related Work on Automatic Algorithm Configuration, Portfolios, and Algorithm Synthesis

There is a large body of literature in AI and related areas that deals with automated methods for building effective algorithms. This includes work on automatic algorithm configuration (see, e.g., Gratch and Dejong, 1992; Minton, 1993), algorithm selection (see, e.g., Guerri and Milano, 2004; Carchrae and Beck, 2005; Xu *et al.*, 2008), algorithm portfolios (see, e.g., Gomes and Selman, 2001; Gagliolo and Schmidhuber, 2006), and, to some extent, genetic programming (see, e.g., Fukunaga, 2002, 2004; Oltean, 2005) and algorithm synthesis (see, e.g., Westfold and Smith, 2001; Monette *et al.*, 2009; Gaspero and Schaerf, 2007). The goal of minimizing human effort in algorithm design and implementation addresses different areas of human involvement and tries to achieve its goal in different ways. On one hand, there is a line of work on automatic algorithm configuration aimed at coming up with new high-performance solvers, often custom-tailored for a given problem distribution. On the other hand, there is a line of research on algorithm portfolios and algorithm selection that tries to make the most out of existing solvers without creating a new solver. For a more detailed discussion of automated algorithm design, see also Hoos, 2008.

In broad strokes, our approach can be differentiated from existing work in three key ways. First, we keep the two phases in our approach completely separate and independent from each other. In the first phase, we specify the combinatorial design space. In the second, we search the space for a good parameter configuration. Keeping these two phases separate allows us to enrich

the design space by adding newer design alternatives without worrying how that would affect the second phase of our approach. Also, we can plug in any advanced configurator without affecting the specified combinatorial design space. Second, we use an automated algorithm configuration tool for searching the design space. Every time we get a new problem distribution, this tool can be used to find a new algorithm configuration. This minimizes human involvement in the algorithm design process, since otherwise a considerable amount of human development time can be spent in re-engineering an old algorithm to make it perform well on a new distribution. Third, experimental results show that we have been able to improve the state of the art in four of the six widely studied problem distributions of propositional satisfiability problem (SAT) (KhudaBukhsh *et al.*, 2009). Detailed discussion of our results are presented in Chapter 5.

2.2.1 Automatic Algorithm Configuration

In the past, a lot of interesting work has been done on automatic algorithm configuration. For example, Minton (1993) used meta-level theories to produce distribution-specific versions of generic heuristics, and then found the most useful combination of these heuristics by evaluating their performance on a small set of test instances. Our work is different from Minton’s approach in the following ways. First, instead of producing distribution-specific versions of candidate heuristics for each distribution, we lay out a generalized, highly parameterized framework, which remains the same for every distribution, by exposing design choices as parameters. Second, our work is very different in terms of scale; we explore a space of 4.82×10^{12} candidate solvers, whereas Minton considered at most 100 possible heuristics. Finally, our results indicate that we improved the state of the art for most of the distributions we considered, whereas in work by Minton, (1993), the performance of the resulting algorithms was comparable with that of algorithms designed by a skilled programmer, but not an algorithm expert.

Gratch and Dejong (1992) presented a system that starts with a STRIPS-like planner, and augments it by incrementally adding search control rules. In **SATenstein**, we do not augment an existing solver; rather, our goal is to design a method for automatically building new solvers by combining components from various existing solvers.

Fukunaga’s (2002; 2004) genetic programming approach, **CLASS** (**C**omposite heuristic **L**earning **A**lgorithm for **S**AT **S**earch), is relevant to our own work in that it is motivated by a similar goal: the automated construction of local search heuristics for SAT. His approach is called In this approach, the considered design space is potentially unbounded, whereas in our approach, we explore a finite design space of 4.82×10^{12} candidate solvers. In order to construct variable selection mechanisms for a generic SLS algorithm, Fukunaga only considered GSAT-based and WalkSAT-based SLS algorithms developed up to the year 2000. Additionally, candidate variable selection mechanisms were evaluated mostly on Random-3-SAT instances and graph coloring instances with at most 250 variables. In contrast, we considered several mechanisms, including

variable selection in SLS algorithms up to the year 2007. Our target problems were much bigger, as they considered SAT instances with up to 998 variables and 9623 clauses, and also included instances that are hard for local search algorithms, such as software verification problems (Clarke *et al.*, 2004) and SAT encoded factoring problems (Uchida and Watanabe, 1999). The two steps in our approach - the specification of possible design choices and the automatic construction of algorithms for a given distribution - are kept separate from each other. The off-the-shelf configurator we used can be easily replaced by any other algorithm configurator without affecting our first step. On the other hand, the genetic programming procedure used by Fukunaga is tailored specifically to searching the design space considered in his study. Finally, while Fukunaga’s approach could in principle be used to obtain high-performance solvers for specific types of SAT instances, to our knowledge this potential was never realized. The best automatically-constructed solvers obtained by Fukunaga only achieved a performance level similar to that of the best WalkSAT variants available in 2000 on moderately-sized SAT instances. In contrast, our new **SATenstein-LS** solvers performed substantially better than current state-of-the-art SLS-based SAT solvers on a broad range of challenging, modern SAT instances.

2.2.2 Parameter Tuning and Algorithm Configuration

In order to allow flexibility and to give good performance over a broad range of instances, many recent high-performance algorithms expose configurable parameters (see, e.g., CPLEX, SAPS (Hutter *et al.*, 2002)). Finding a good parameter configuration is often the most critical task for obtaining good performance on a given problem instance or problem distribution. There are several techniques found in the literature that attempt to solve the problem of algorithm configuration. The two most prominent examples are F-race (Birattari *et al.*, 2002; Balaprakash *et al.*, 2007) and ParamILS (Hutter *et al.*, 2007b; Hutter *et al.*). F-race is a racing algorithm that is used for designing meta-heuristics. In this approach, a non-parametric statistical test is used to iteratively filter out configurations that are significantly worse than others. Such filtering continues until a cutoff time is reached and only a small number of good configurations are left. ParamILS (Hutter *et al.*, 2007b; Hutter *et al.*) is a parameter configuration tool based on iterated local search (ILS). ParamILS has been successfully used in configuring a diverse set of applications (see, e.g., Hutter *et al.*, 2007a; Chiarandini *et al.*, 2008). The main disadvantage of F-Race over ParamILS is its low scalability in comparison with ParamILS. Since **SATenstein-LS** has a vast design space, we chose ParamILS as our automated algorithm configurator.

2.2.3 Algorithm Selection and Portfolios

There are several other techniques aimed at making the most out of the existing high-performance solvers. Examples include algorithm selection and algorithm portfolios. For a given problem instance or problem distribution, often we have to solve an “algorithm selection problem” (Rice, 1976): which algorithm(s)

should be run in order to minimize some performance objective, such as expected runtime? Different machine learning techniques are applied to solve the algorithm selection problem (see, e.g., Leyton-Brown *et al.*, 2002; Guerri and Milano, 2004; Carchrae and Beck, 2005; Xu *et al.*, 2008). In Leyton-Brown *et al.*, (2002), statistical regression techniques are applied to identify the connection between instance-related features and its predicted hardness. In a similar vein, Guerri and Milano (2004) used a decision tree-based technique to identify the connection between the structure of a problem instance and algorithm performance. Although our problem can be reformulated as an algorithm selection problem, it is obvious that such techniques are infeasible for selecting from a set of 4.82×10^{12} candidate solvers. However, there exists a conceptual relationship between our work and algorithm selection approaches. For example, **SATzilla** (Nudelman *et al.*, 2004a; Xu *et al.*, 2008) performed per-instance algorithm selection based on the algorithms' predicted performance. Both **SATzilla** and **SATenstein-LS** leverage existing SAT solvers to achieve better performance (also, both are named after movie monsters!). However, **SATzilla** and **SATenstein-LS** are different in the following ways. First, **SATzilla** chooses from a relatively small set of known algorithms, which includes non-SLS-based algorithms. In contrast, **SATenstein-LS** can instantiate trillions of SLS solvers, including a vast number of solvers never studied before. Second, in order to perform algorithm selection, **SATzilla** uses empirical hardness models (see, e.g., Leyton-Brown *et al.*, 2002, 2009; Nudelman *et al.*, 2004b) to predict runtimes for each algorithm in the portfolio on a given SAT instance, based on instance-related features. Using this approach, creating such performance-predicting models for **SATenstein-LS** is infeasible, and by design, **SATenstein-LS** should be used on a per-distribution basis. On the other hand, the approaches for **SATzilla** and **SATenstein-LS** can be seen as complementary, as **SATenstein-LS** solvers can be used in a **SATzilla** portfolio. Indeed, the latest version of **SATzilla**, **SATzilla2009_R**, which performed extremely well in the random category in the 2009 SAT Competition (winning a gold medal in the random SAT+UNSAT category and placing 4th in the random SAT category), makes use of several solvers constructed using the **SATenstein-LS** framework (Xu *et al.*, 2009).

It has often been observed that a significant improvement in performance can be obtained by running multiple algorithms in parallel. Also, for stochastic algorithms, running multiple copies of the same algorithm can be beneficial. Algorithm portfolios aim to achieve performance improvement by running multiple algorithms concurrently (Gomes and Selman, 2001). In general, all or a subset of the algorithms in a portfolio are allocated CPU shares in a multi-tasking environment or run concurrently on a multi-processor (or multi-core) computing platform. Extensions of this general idea include the dynamic reallocation of CPU share between the component algorithms (Gagliolo and Schmidhuber, 2006) and communication between component algorithms (Huberman *et al.*, 1997).

2.2.4 Further Related Work

The Frankenstein metaphor of algorithm design has also been used in work by Montes de Oca *et al.*, 2007, where a Particle Swarm Optimization (PSO) algorithm is created by combining algorithm components drawn from existing high-performance PSO algorithms. In this approach, specific designs for each of the components are hand-picked by the algorithm designer. In contrast, we specify a combinatorial design space from which we use an automated algorithm configurator to find a good design for a given problem distribution. Clearly, Frankenstein’s PSO can be seen an example of manual algorithm design, whereas, our goal is to automate the algorithm-building process.

Existing work on algorithm synthesis is mostly focused on automatically generating algorithms that satisfy a given formal specification or that solve a specific problem from a large and diverse domain (see, e.g., Westfold and Smith, 2001; Monette *et al.*, 2009; Gaspero and Schaerf, 2007). In contrast, our work is focused on finding an efficient solver from amongst a huge space of candidate solvers that are all guaranteed to be correct by construction. Clearly, these two approaches are focused on solving different types of problems.

2.3 Local Search for SAT

The state-of-the-art algorithms for solving certain classes of SAT instances include Stochastic Local Search (SLS) algorithms. For a given SAT problem instance, the objective function, which an SLS SAT solver attempts to minimize, is the number of unsatisfied clauses. In most cases, the score of a variable x with respect to an assignment A , $\text{score}_A(x)$, is defined as the decrease of the value of the objective function when x is flipped (flipping a variable means changing its truth value). A typical SLS algorithm for SAT consists of an initialization phase and a search phase. In the initialization phase, a complete assignment of truth values to all variables is given. At each step in the search phase, the truth value of a heuristically-determined variable is changed. Since each step essentially means flipping a variable, the terms flip and step are often used interchangeably. Sometimes, we may have a null flip where no variable is flipped. At each search step, the set of variables that are considered for flipping is called the neighborhood. The search process is terminated when either a satisfying assignment is found or a given bound on the runtime or run-length is reached or exceeded. Almost all SLS algorithms for SAT are by nature incomplete, i.e., they cannot prove unsatisfiability.

2.3.1 Different Classes of SLS Algorithms for SAT

Existing SLS-based SAT solvers can be grouped into four broad categories: GSAT-based algorithms (Selman *et al.*, 1992), WalkSAT-based algorithms (Selman *et al.*, 1994), dynamic local search algorithms (Hutter *et al.*, 2002; Thornton *et al.*, 2004), and G²WSAT variants (Li and Huang, 2005). The following subsections describe the major features of each of these categories.

Procedure GSAT(*CNF formula* F , maxTries , maxSteps)

Input: CNF formula F , positive integers maxTries and MaxSteps
Output: model of F or ‘no solution found’

```

for try := 1 to maxTries do
   $a :=$  randomly chosen assignment of the variable in formula  $F$  ;
  for step := 1 to maxSteps do
    if  $a$  satisfies  $F$  then
       $\perp$  return  $a$  ;
     $x :=$  randomly selected variable flipping that minimizes the
    number of unsatisfied clause ;
     $a := a$  with  $x$  flipped;
  return ‘no solution found’

```

GSAT

GSAT (Selman *et al.*, 1992) was one of the earliest SLS SAT solvers. As shown in Procedure **GSAT**, at each step, **GSAT** computes the score of each variable using a scoring function, then flips the variable with the best score.³ To describe the scoring function of **GSAT**, we need to define the *MakeCount* and *BreakCount* of a variable. The *MakeCount* of a variable with respect to an assignment A is the number of previously-unsatisfied clauses that will be satisfied if the variable is flipped. Similarly, the *BreakCount* of a variable with respect to an assignment A is the number of previously-satisfied clauses that will be unsatisfied if the variable is flipped. The scoring function of **GSAT** is *MakeCount* - *BreakCount*.

Variants of **GSAT** introduced many techniques that were later used by other SLS solvers. For example, **GWSAT** (Selman and Kautz, 1993) performs a conflict-directed random walk step with probability w_p . Otherwise, it performs a regular **GSAT** step. Conflict-directed random walk is an example of a search diversification strategy that was later used by many SLS solvers. In an SLS search step, for a given scoring function, multiple variables can have the same score. **GSAT** breaks such ties randomly. **HSAT** (Gent and Walsh, 1993) introduced a new tie-breaking scheme in which ties are broken in favor of the least-recently-flipped variable. In subsequent SLS solvers, breaking ties randomly and breaking in the favor of the least-recently-flipped variable were prominent tie-breaking schemes. **GSAT** now has only historical importance, as there is a substantial performance gap between **GSAT** and recent state-of-the-art SLS solvers.

WalkSAT Architecture

The major difference between WalkSAT algorithms and GSAT algorithms is the neighborhood they consider. For GSAT algorithms, the neighborhood is the full set of variables in the formula. For a WalkSAT algorithm, the neighborhood consists of variables belonging to a selected unsatisfied clause. As shown in

³The pseudocodes of **GSAT** and **WalkSAT** are taken from Hoos and Stützle, 2005. All our pseudocodes follow the same notations as in Hoos and Stützle, 2005.

Procedure WalkSAT(*CNF formula F, maxTries, maxSteps*)

Input: CNF formula F , positive integers $maxTries$ and $MaxSteps$, heuristic function slc

Output: model of F or ‘no solution found’

```

for try := 1 to maxTries do
  a := randomly chosen assignment of the variable in formula  $F$  ;
  for step := 1 to maxSteps do
    if a satisfies  $F$  then
       $\perp$  return a ;
    c := randomly selected clause unsatisfied under a ;
    x := variable selected from c according to heuristic function  $slc$ ;
    a := a with x flipped;
  return ‘no solution found’

```

Procedure **WalkSAT**, at each search step, a WalkSAT algorithm picks an unsatisfied clause (the most popular way is picking an unsatisfied clause uniformly at random), and flips a variable (depending on some heuristic) from that clause. **WalkSAT/SKC** (Selman *et al.*, 1994) was one of the earliest algorithms belonging to WalkSAT architecture. A notable difference between **WalkSAT/SKC** and many other SLS algorithms, including **GSAT**, is that **WalkSAT/SKC** uses *BreakCount* as its scoring function.

Novelty (McAllester *et al.*, 1997) and its several variants are among the most prominent WalkSAT algorithms. **Novelty** uses the same scoring function as **GSAT** and also takes the age of a variable into account (the number of flips performed since it was last flipped). More precisely, **Novelty** picks a random unsatisfied clause and computes the variables with highest and second-highest scores. Ties are broken in favor of the least-recently-flipped variable. If the variable with the highest score is not the most-recently-flipped variable within the clause, then it is deterministically selected for flipping. Otherwise, it is selected with probability $(1 - p)$ where p is a parameter called the *noise setting* (with probability p , the second-best variable is selected). **Novelty** represents an interesting advancement in the research of SLS solvers. However, it often gets stuck in local minima of the objective functions since the selection is restricted only to the two best variables. To overcome this problem, **Novelty** was augmented with a probabilistic conflict-directed random walk (Hoos, 1999). The resulting algorithm, **Novelty⁺**, showed much better performance than **Novelty** on many of the benchmark instances. Table 2.1 summarizes different search diversification strategies found in **Novelty** variants. Other variants include **R-*Novelty***, **R-*Novelty⁺***, **Novelty+p** (**Novelty** with lookahead), and **RANOV** (Pham and Anbulagan, 2007) (**Novelty⁺** with a preprocessor).

The performance of the new algorithm **Novelty⁺**, although more robust than **Novelty**, still depends a lot on a suitable *noise* setting. In **adaptNovelty⁺** (Hoos, 2002), a reactive mechanism that adaptively changes the *noise* parameter was first introduced. This reactive mechanism is extended to many SLS solvers (see,

Novelty variant	Search diversification strategy
Novelty ⁺ (Hoos, 1999)	flips a randomly selected variable
Novelty++ (Li and Huang, 2005)	flips the least recently flipped variable
Novelty++' (Li <i>et al.</i> , 2007a)	randomly flips a variable excluding the <i>best</i> and the <i>second-best</i> variable

Table 2.1: Different search diversification strategies found in Novelty variants.

Procedure	Dynamic Local Search(<i>CNF formula</i> F , maxTries , maxSteps)
------------------	--

Input: CNF formula F , positive integers maxTries and MaxSteps
Output: model of F or 'no solution found'

```

for try := 1 to maxTries do
  a := randomly chosen assignment of the variable in formula  $F$  ;
  initialize clause-weight for each clause ;
  for step := 1 to maxSteps do
    if a satisfies  $F$  then
       $\perp$  return a ;
    x := variable selected using a scoring function that considers
    clause-weights;
    a := a with x flipped;
    update clause-weight for each clause ;
  return 'no solution found'
```

e.g., Li *et al.*, 2007b) yielding improved performance.

The idea of considering flip history is exploited in various ways in different SLS solvers. These include considering the age of a variable (e.g., in Novelty), tabu search (e.g., in WalkSAT/TABU (McAllester *et al.*, 1997)) and flipping the least-recently-flipped variable (e.g., in Novelty++). VW (Prestwich, 2005), one of the most recent WalkSAT algorithms, considers flip counts (the total number of times a variable is flipped).

Dynamic Local Search Algorithms

The most prominent feature of dynamic local search (DLS) algorithms is the use of “clause penalties” or “clause weights”. The general structure of a DLS algorithm is presented in Figure 3. At each step, the clause penalty of an unsatisfied clause is increased (this increase can be additive (Thornton *et al.*, 2004) or multiplicative (Hutter *et al.*, 2002)). In this way, information about the *difficulty* of solving a given clause is recorded in its associated clause penalty. In order to prevent an unbounded increase in weights and to keep the information about the *difficulty* of solving a given clause recent, occasional *smoothing* steps are performed to reduce them. The scoring function is the sum of the clause penalties of all unsatisfied clauses. For prominent DLS solvers like SAPS,

Procedure G²WSAT(*CNF formula* F, *maxTries*, *maxSteps*)

Input: CNF formula F , positive integers $maxTries$ and $MaxSteps$, deterministic heuristic function dhf and **Novelty** variant

Output: model of F or 'no solution found'

```

for try := 1 to maxTries do
  a := randomly chosen assignment of the variable in formula  $F$  ;
  put decreasing variable(s) in promising list ;
  for step := 1 to maxSteps do
    if a satisfies F then
      ⊥ return a ;
    if promising list is empty then
      c := randomly selected clause unsatisfied under a ;
      x := variable selected from c according to Novelty variant;
    else
      ⊥ x := variable selected from promising list according to  $dhf$  ;
      a := a with x flipped ;
      update promising list ;
  return 'no solution found'

```

RSAPS (Hutter *et al.*, 2002), and PAWS (Thornton *et al.*, 2004), the neighborhood consists of variables that appear in at least one unsatisfied clause.

G²WSAT Variants

G²WSAT (Li and Huang, 2005) can be viewed as a combination of the GSAT and WalkSAT architectures. Similar to GSAT, G²WSAT has a deterministic greedy component that looks at a large number of variables belonging to a *promising list* data structure that contains *promising decreasing variables*. As shown in Procedure G²WSAT, if the list has at least one *promising decreasing variable*, G²WSAT deterministically selects the variable with the best score for flipping. Ties are broken in favor of the least-recently-flipped variable. If the list is empty, G²WSAT executes its stochastic component, a **Novelty** variant that belongs to the WalkSAT architecture.

The definition of a *promising decreasing variable* is somewhat technical; skipping this would not affect understanding the rest of the content in this thesis. However, we present the definition for the sake of completeness. A variable x is said to be *decreasing* with respect to an assignment A if $score_A(x) > 0$ (the scoring function is $MakeCount - BreakCount$). A *promising decreasing variable* is defined as follows:

1. For the initial random assignment A , all *decreasing* variables with respect to A are *promising*.
2. Let x and y be two different variables where x is not *decreasing* with respect to A . If, after y is flipped, x becomes *decreasing* with respect to

the new assignment, then x is a *promising decreasing variable* with respect to the new assignment.

3. As long as a *promising decreasing variable* is *decreasing*, it remains *promising* with respect to subsequent assignments in local search.

Table 2.2 summarizes the different ways the two components of G^2WSAT can vary. Apart from G^2WSAT (Li and Huang, 2005), all G^2WSAT variants use the reactive mechanism found in $adaptNovelty^+$ (Hoos, 1999). $gNovelty_+$ (Pham *et al.*, 2008), the winner of 2007 SAT Competition in the random satisfiable category also uses clause penalties and smoothing found in dynamic local search algorithms (Thornton *et al.*, 2004).

G^2WSAT component	Different strategies
Heuristic selection	$Novelty^+$ (Li <i>et al.</i> , 2007b) $Novelty^{++}$ (Li and Huang, 2005) $Novelty^{++'}$ (Li and Huang, 2005) $Novelty^{+p}$ (Li <i>et al.</i> , 2007b)
Variable selection from <i>promising list</i>	Select the variable with highest score, breaking ties in favor of the least-recently-flipped variable (Li and Huang, 2005). Select the least recently flipped variable (Li <i>et al.</i> , 2007b).

Table 2.2: *Different strategies used in G^2WSAT components.*

2.3.2 Recent Trends in SLS Algorithms

Recent developments on SLS algorithms show an increasing trend towards creating new algorithms by combining components of different algorithms, albeit manually. For example, the augmentation of an existing algorithm with mechanisms from another's can be observed in $adaptG^2WSAT$ (Li *et al.*, 2007b). There, the adaptive noise mechanism, first introduced in $Novelty^+$ (Hoos, 2002), was successfully combined with an existing algorithm G^2WSAT (Li and Huang, 2005), resulting in an improved algorithm. $gNovelty^+$ (Pham *et al.*, 2008), the most recent SAT Competition winner in the random satisfiable category, can be viewed as a manually-designed algorithm that combines components from G^2WSAT and $Ranov$ (Pham and Anbulagan, 2007). The design of $gNovelty^+$ was largely guided by the fact that G^2WSAT and $Ranov$ tend to do well on different types of random- k -SAT instances. The recent high-performance algorithm $Hybrid$ (Wei *et al.*, 2008) combines the strengths of two different algorithms: VW (Prestwich, 2005) and $adaptG^2WSAT_p$ (Li *et al.*, 2007b). Such successes indicate that the performance of SLS solvers can be improved by combining algorithm components.

Without an automated approach, we believe that this type of design will not be able to reach its full potential, since there can be many possible combinations, and exhaustively searching all of them is typically infeasible. Also, manual design is largely guided by an algorithm practitioner’s expertise and intuition. Relying on such intuition may lead to the neglect of unintuitive combinations that perform well in practice.

2.3.3 UBCSAT

UBCSAT (Tompkins and Hoos, 2004) is an SLS solver implementation and experimentation environment for SAT. It has already been used to implement many existing high-performance SLS algorithms from the literature. (e.g., SAPS Hutter *et al.*, 2002, `adaptG2WSAT+` Li *et al.*, 2007b). These implementations generally match or exceed the efficiency of implementations by the original authors. UBCSAT implementations have therefore been widely used as reference implementations (see, e.g., Prestwich, 2005; Kroc *et al.*, 2009) for many well-known local search algorithms. UBCSAT provides a rich interface that includes numerous statistical and reporting features facilitating empirical analysis of SLS algorithms.

Many existing SLS algorithms for SAT share common components and data structures. The general design of UBCSAT allows for the reuse and extension of such common components and mechanisms. This made UBCSAT a suitable environment for the implementation of `SATenstein-LS`. All the algorithm components and mechanisms used in `SATenstein-LS` were already implemented in UBCSAT.⁴ However, many of these components were never used within a single algorithm and were optimized in a way that their coexistence inside a single algorithm might lead to erroneous behavior. Because of this, creating `SATenstein-LS` was a major software development task that required thorough validation of each of the underlying mechanisms and components used. At a conceptual level, UBCSAT and `SATenstein-LS` are quite different. UBCSAT has a set of well-known solvers implemented in it in a stand-alone fashion; UBCSAT does not create new solvers by combining these solvers’ components. `SATenstein-LS`, on the other hand, is a highly parameterized SLS solver that draws components from several well-known SLS algorithms. `SATenstein-LS` can be configured as trillions of SLS solvers that have never been studied before. In combination with an automatic algorithm configurator, `SATenstein-LS` is indeed a framework for automatically finding new SLS solvers.

⁴During the implementation phase of `SATenstein-LS`, we included several recent high-performance algorithms in the UBSAT implementation suite (e.g., `VW`, `G2WSAT`, `adaptG2WSAT+`, and `adaptG2WSATp`). In UBCSAT versions prior to 1.1, many of the components used in these algorithms were not implemented.

Chapter 3

SATenstein-LS

In this chapter, we describe the design of **SATenstein-LS**, a highly parameterized, stochastic local search (SLS) SAT solver. **SATenstein-LS** draws components from several high-performance SLS SAT solvers as well as uses various novel mechanisms. We first present a high-level pseudocode of **SATenstein-LS** and explain the functionalities of different building blocks used in our design. We then present all the parameters that we expose to the end user to configure **SATenstein-LS**. We conclude with the results of our experiments that we conducted to confirm that different mechanisms of SLS solvers have been correctly implemented in **SATenstein-LS**.

3.1 Design

As mentioned already, almost all SLS algorithms for SAT can be categorized into four broad categories: GSAT-based algorithms, WalkSAT-based algorithms, dynamic local search algorithms and G²WSAT variants. Since none of the state-of-the-art SLS solvers is GSAT-based, **SATenstein-LS** has been constructed by drawing its components from algorithms belonging to the three remaining broad categories.

Procedure **SATenstein-LS** presents very high-level pseudocode of **SATenstein-LS**. The pseudocode is divided into five building blocks, B_1 – B_5 . Any **SATenstein-LS** instantiation has the following general structure:

1. Optionally execute B_1 , which performs search diversification.
2. Execute either B_2 , B_3 or B_4 , which represent G²WSAT-based, WalkSAT-based and dynamic local search algorithms, respectively.
3. Optionally execute B_5 , to update data structures such as **promising list**, clause penalties, dynamically adaptable parameters or tabu attributes.

Each of our building blocks is composed of one or more components (listed in Table 3.1). Some of the components are shared across different building blocks. Each component is configurable by one or more parameters. Out of 41 total parameters, **SATenstein-LS** has 16 continuous parameters (listed in Table 3.4), 19 categorical parameters (listed in Table 3.2), and 6 integer parameters (listed in Table 3.3). All these parameters are exposed on the command line and tuned by using our automatic configurator. Many of these parameters conditionally depend on other parameters. A high-level representation of such dependencies is

Procedure SATenstein-LS(CNF formula ϕ)

Input: CNF formula ϕ ; real number *cutoff*;
 booleans *performDiversification*, *singleClauseAsNeighbor*, *usePromisingList*

Output: Satisfying variable assignment

Start with random Assignment A;
 Initialise parameters;

while *runtime* < *cutoff* **do**

if A satisfies ϕ **then**

└ return A;

VarFlipped \leftarrow FALSE;

if *performDiversification* **then**

B1 **if** *within probability diversificationProbability()* **then**

B1 └ Execute B_1 ;

if *Not VarFlipped* **then**

if *usePromisingList* **then**

B2 **if** *promisingList is nonempty* **then**

B2 └ $y \leftarrow$ *selectFromPromisingList()* ;

else

B2 └ $c \leftarrow$ *selectClause()*;

B2 └ $y \leftarrow$ *selectHeuristic(c)* ;

else if *singleClauseAsNeighbor* **then**

B3 └ $c \leftarrow$ *selectClause()*;

B3 └ $y \leftarrow$ *selectHeuristic(c)* ;

else

B4 └ $sety \leftarrow$ *selectSet()*;

B4 └ $y \leftarrow$ *tieBreaking(sety)*;

flip y ;

B5 └ *update()*;

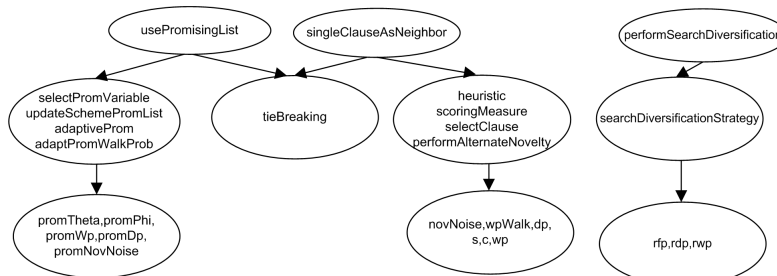
shown in Figure 3.1. The total number of valid SATenstein-LS instantiations is about five trillions. This of course depends on the number of discrete values we consider for each of our continuous parameters (see Table 4.3) and the number of values we consider for each of our integer parameters (see Table 4.2). With a finer grid, the number of valid SATenstein-LS instantiations will increase.

We now give a high-level description of each of the building blocks. B_1 is constructed using the *SelectClause()*, *DiversificationStrategy()* and *DiversificationProbability()* components. *SelectClause()* is configured by one categorical parameter and, depending on its value, either selects an unsatisfied clause uniformly at random or selects a clause with probability proportional to its clause penalty (Tompkins and Hoos, 2004). Component *diversificationStrategy()* can be configured by a categorical parameter to do any of the following with probability *diversificationProbability()*: flip the least recently flipped variable (Li and Huang, 2005), flip the least frequently flipped variable (Prestwich, 2005), flip the variable with minimum variable weight (Prestwich, 2005), or flip a randomly selected variable (Hoos, 2002).

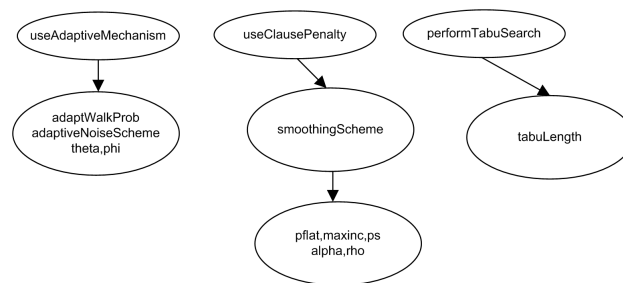
Block B_2 instantiates G^2 WSAT-based algorithms that use a data structure *promising list* (for detailed definition, see Chapter 2.3.1) that keeps track of a set of variables considered for being flipped. In the literature on G^2 WSAT,

Component	Block	params	poss. values	Based on
diversificationStrategy()	1	1	4	Pham et al., 2008 Hoos, 2002 Prestwich, 2005 Li and Huang, 2005
SelectClause()	1, 2, 3	1	2	Tompkins and Hoos (2004)
diversificationProbability()	1	4	24	Pham et al., 2008 Hoos, 2002 Prestwich, 2005 Li and Huang, 2005
selectFromPromisingList()	2	4	≥ 4341	Li and Huang, 2005 Li et al., 2007b Li et al., 2007a Pham et al., 2008
selectHeuristic()	2, 3	9	≥ 1220800	Hoos, 2002 Li and Huang, 2005 Li et al., 2007b Li et al., 2007a Prestwich, 2005 Selman et al., 1994 McAllester et al., 1997
selectSet()	4	9	≥ 111408	Hutter et al., 2002 Thornton et al., 2004
tiebreaking()	4	1	4	Hoos, 2002 Li and Huang, 2005 Prestwich, 2005
update()	5	12	≥ 73728	Hoos, 2002 Li et al., 2007b Hutter et al., 2002 Thornton et al., 2004 Pham et al., 2008 Li and Huang, 2005

Table 3.1: SATenstein-LS components.



(a) Base-level parameters select building blocks.



(b) Base-level parameters specify update mechanisms.

Figure 3.1: Dependencies between SATenstein-LS parameters.

Parameter	Active When	Domain	Description
performSearchDiversification	Base level parameter	{0,1}	If true, block B ₁ is performed
usePromisingList	Base level parameter	{0,1}	If true, block B ₂ is performed
singleClauseAsNeighbor	Base level parameter	{0,1}	If true, block B ₃ is performed else, block B ₄ is performed
selectPromVariable	usePromisingList = 1	{1,...,11}	See Table 3.5
heuristic	singleClauseAsNeighbor = 1	{1,...,13}	See Table 3.6
performAlternateNovelty	singleClauseAsNeighbor = 1	{0,1}	If true, performs Novelty variant with 'flat move'.
useAdaptiveMechanism	Base level parameter	{0,1}	If true, uses adaptive mechanisms.
adaptivenoisescheme	useAdaptiveMechanism = 1 usePromisingList = 1	{1,2}	Specifies adaptive noise mechanisms.
adaptWalkProb	useAdaptiveMechanism = 1	{0,1}	If true, walk probability or diversification probability of a heuristic is adaptively tuned.
performTabuSearch	Base level parameter	{0,1}	If true, tabu variables are not considered for flipping.
useClausePenalty	Base level parameter	{0,1}	If true, clause penalties are computed.
selectClause	singleClauseAsNeighbor = 1	{1,2}	1 selects an UNSAT clause uniformly at random. 2 selects an UNSAT clause with a probability proportional to its clause penalty.
searchDiversificationStrategy	performSearchDiversification = 1	{1,2,3,4}	1 randomly selects a variable from an UNSAT clause. 2 selects the least-recently-flipped variable from an UNSAT clause. 3 selects the least-frequently-flipped variable from an UNSAT clause. 4 selects the variable with least VW2 weight from an UNSAT clause.
adaptiveProm	usePromisingList = 1	{0,1}	If true, performs adaptive versions of Novelty variants to select variable from promising list.
adaptpromwalkprob	usePromisingList = 1 adaptiveProm = 1	{0,1}	If true, walk probability or diversification probability of Novelty variants used on promising list is adaptively tuned.
scoringMeasure	usePromisingList = 0 singleClauseAsNeighbor = 0	{1,2,3}	Specifies the scoring measure. 1 uses MakeCount - BreakCount 2 uses MakeCount 3 uses -BreakCount
tieBreaking	usePromisingList = 1 selectPromVariable ∈ {1, 4, 5}	{1,2,3,4}	1 breaks ties randomly. 2 breaks ties in favor of the least-recently-flipped variable. 3 breaks tie in favor of the least-frequently-flipped variable. 4 breaks tie in favor of the variable with least VW2 score.
updateSchemePromList	usePromisingList = 1	{1,2,3}	1 and 2 follow G^2WSAT . 3 follows $gNovelty+$.
smoothingScheme	useClausePenalty = 1	{1,2}	When singleClauseAsNeighbor = 1 : 1 performs smoothing for only random 3-SAT instances with 0.4 fixed smoothing probability. 2 performs smoothing for all instances. When singleClauseAsNeighbor = 0 : 1 performs SAPS-like smoothing. 2 performs PAWS-like smoothing.

Table 3.2: Categorical parameters of SATenstein-LS. Unless otherwise mentioned, for more than one parameters, the conditional operator used in the the 'Active When' column is AND.

Parameter	Active When	Description
tabuLength	performTabuSearch = 1	Specifies tabu step-length.
phi	useAdaptiveMechanism = 1 singleClauseAsNeighbor = 1	Parameter for adaptively setting noise.
theta	useAdaptiveMechanism = 1 singleClauseAsNeighbor = 1	Parameter for adaptively setting noise.
promPhi	usePromisingList = 1 adaptiveProm = 1 selectPromVariable ∈ {7, 8, 9, 10, 11}	Parameter for adaptively setting noise.
promTheta	usePromisingList = 1 adaptiveProm = {1} selectPromVariable ∈ {7, 8, 9, 10, 11}	Parameter for adaptively setting noise.
maxinc	singleClauseAsNeighbor = 0 useClausePenalty = 1 smoothingScheme = 2	PAWS (Thornton et al., 2004) parameter for additive clause weighting.

Table 3.3: Integer parameters of SATenstein-LS. Unless otherwise mentioned, for more than one parameters, the conditional operator used in the the ‘Active When’ column is AND.

Parameter	Active When	Description
wp	singleClauseAsNeighbor = 1 heuristic ∈ {2, 6, 11} useAdaptiveMechanism = 0	Randomwalk probability for Novelty ⁺
dp	singleClauseAsNeighbor = 1 heuristic ∈ {3, 4, 12, 13} useAdaptiveMechanism = 0	Diversification probability for Novelty++ and Novelty++'
promDp	usePromisingList = 1 selectPromVariable ∈ {8, 10} adaptiveProm = 0	Diversification probability for Novelty variants used to select variable from promising list
novNoise	singleClauseAsNeighbor = 1 heuristic ∈ {1, 2, 3, 4, 5, 6, 10, 11, 12, 13} useAdaptiveMechanism = 0	Noise parameter for all Novelty variants
wpWalk	singleClauseAsNeighbor = 1 heuristic ∈ {7, 9} useAdaptiveMechanism = 0	Noise parameter for WalkSAT and VW1
promWp	usePromisingList = 1 selectPromVariable ∈ {9, 11}	Randomwalk probability for Novelty variants used to select variable from promising list
promNovNoise	usePromisingList = 1 selectPromVariable ∈ 7, 8, 9, 10, 11	Noise parameter for all Novelty variants used to select variable from promising list
alpha	singleClauseAsNeighbor = 0 useClausePenalty = 1 smoothingScheme = 1	Parameter for SAPS
rho	singleClauseAsNeighbor = 0 useClausePenalty = 1 smoothingScheme = 1	Parameter for SAPS
ps	useClausePenalty = 1	Smoothing parameter for SAPS, RSAPS, and gNovelty ⁺
s	singleClauseAsNeighbor = 1 heuristic = 8 useAdaptiveMechanism = 0	VW parameter for smoothing
c	singleClauseAsNeighbor = 1 heuristic = 8 useAdaptiveMechanism = 0	VW parameter for smoothing
rdp	performSearchDiversification = 1 searchDiversificationStrategy ∈ {2, 3}	Parameter for search diversification
rfp	performSearchDiversification = 1 searchDiversificationStrategy = 4	Parameter for search diversification
rwp	performSearchDiversification = 1 searchDiversificationStrategy = 1	Parameter for search diversification
pflat	singleClauseAsNeighbor = 0 useClausePenalty = 1 smoothingScheme = 2	Parameter for PAWS that controls “flat-moves”

Table 3.4: Continuous parameters of SATenstein-LS. Unless otherwise mentioned, for more than one parameters, the conditional operator used in the the ‘Active When’ column is AND.

Param. Value	Design choice	Based on
1	If freebie exists, use tieBreaking(); else, select uniformly at random	(Selman et al., 1994)
2	Variable with best score	(Li and Huang, 2005)
3	Least-recently-flipped variable	(Li et al., 2007b)
4	Variable with best VW1 score	(Prestwich, 2005)
5	Variable with best VW2 score	(Prestwich, 2005)
6	Variable selected uniformly at random	(Hoos, 1999)
7	Variable selection from Novelty	(McAllester et al., 1997)
8	Variable selection from Novelty++	(Li and Huang, 2005)
9	Variable selection from Novelty ⁺	(Hoos, 1999)
10	Variable selection from Novelty++'	(Li et al., 2007a)
11	Variable selection from Novelty+p	(Li et al., 2007a)

Table 3.5: Design choices for `selectFromPromisingList()` and corresponding values of `selectPromVariable`.

there are two strategies for selecting a variable from *promising list*: choosing the variable with the highest score (Li and Huang, 2005) or choosing the least recently flipped variable (Li et al., 2007b). We added nine novel strategies based on variable selection heuristics from other solvers. These, to the best of our knowledge, have never been used before in the context of *promising variable* selection for G²WSAT-based algorithms. For example, in previous work, variable selection mechanisms used in Novelty variants are only applied to variables of unsatisfiable clauses, not to *promising lists*. Table 3.5 lists the eleven possible strategies for *SelectFromPromisingList*.

Param. Value	Selected Heuristic	Dependent Parameters
1	Novelty (McAllester et al., 1997)	novnoise
2	Novelty ⁺ (Hoos, 2002)	novnoise, wp
3	Novelty++ (Li and Huang, 2005)	novnoise, dp
4	Novelty++' (Li et al., 2007a)	novnoise, dp
5	R-Novelty (McAllester et al., 1997)	novnoise
6	R-Novelty ⁺ (Hoos, 2002)	novnoise, wp
7	VW1 (Prestwich, 2005)	wpwalk
8	VW2 (Prestwich, 2005)	s, c, wp
9	WalkSAT-SKC (Selman et al., 1994)	wpwalk
10	Noveltyp (Li et al., 2007a)	novnoise
11	Novelty ⁺ p (Li et al., 2007a)	novnoise, wp
12	Novelty++p (Li et al., 2007a)	novnoise, dp
13	Novelty++'p (Li et al., 2007a)	novnoise, dp

Table 3.6: List of heuristics chosen by the parameter heuristic and dependent parameters.

If *promising list* is empty, B_2 behaves exactly like B_3 , which instantiates WalkSAT-based algorithms. As already described in B_1 , component *SelectClause()* is used to select an unsatisfiable clause c . The *SelectHeuristic()* component selects a variable from c for flipping. Depending on a categorical parameter, *SelectHeuristic()* can behave like any of the thirteen well-known WalkSAT-based heuristics that include Novelty variants, VW1 and VW2. Table 3.6 lists these heuristics and related continuous parameters. We also extend the Novelty vari-

ants with an optional “flat move” mechanism as found in the selection strategy in `gNovelty+` (Thornton *et al.*, 2004; Pham *et al.*, 2008).

Block B_4 instantiates dynamic local search algorithms. The `selectSet()` component considers the set of variables that occur in any unsatisfied clause. It associates with each such variable v a score, which depends on the *clause weights* of each clause that changes satisfiability status when v is flipped. These clause weights reflect the perceived importance of satisfying each clause. For example, weights might increase the longer a clause has been unsatisfied, and decrease afterwards (Hutter *et al.*, 2002; Thornton *et al.*, 2004). After scoring the variables, `selectSet()` returns all variables with maximal score. Our implementation of this component incorporates three different scoring functions, including those due to McAllester *et al.*, 1997, Selman *et al.*, 1994, and a novel, greedier variant that only considers the number of previously unsatisfied clauses that are satisfied by a variable flip. The `tieBreaking()` component selects a variable from the maximum-scoring set according to the same strategies used by the `diversificationStrategy()` component.

Block B_5 updates underlying data structures after a variable has been flipped. Performing these updates in an efficient manner is a core issue in optimizing the performance of SLS algorithms. Because the SATenstein-LS framework supports the combination of mechanisms from many different SLS algorithms, each depending on different data structures, the implementation of the `update()` function was technically quite challenging.

3.2 Implementation and Validation

As already mentioned, SATenstein-LS is built on top of UBCSAT (Tompkins and Hoos, 2004), a well-known SLS solver development framework. UBCSAT has a novel trigger-based architecture that facilitates the reuse of existing mechanisms. While designing and implementing SATenstein-LS, we not only studied existing SLS algorithms, as presented in the literature, but we also looked at the SAT Competition submissions of such algorithms. We found that the pseudocode of `VW2` according to Preswitch (2005) is different from its SAT Competition 2005 submission, which includes a reactive mechanism. We included both versions in SATenstein-LS’s implementation. We also found that in the SAT Competition implementation of `gNovelty+`, `Novelty` uses a PAWS-like (Thornton *et al.*, 2004) “flat move” mechanism. We implemented this alternate version of `Novelty` in SATenstein-LS and exposed a categorical parameter to choose between the two implementations. While looking at the various source codes, we found that certain data structures are implemented in different ways. For example, different `G2WSAT` variants implement the update scheme of *promising list* in different ways. We include all such different update schemes in SATenstein-LS and declared parameter `updateSchemePromList`.

In order to support a vast number of possible heuristics and combinations of data structures, the implementation of SATenstein-LS is fairly complicated and required the validation of individual components. For that, we compared the

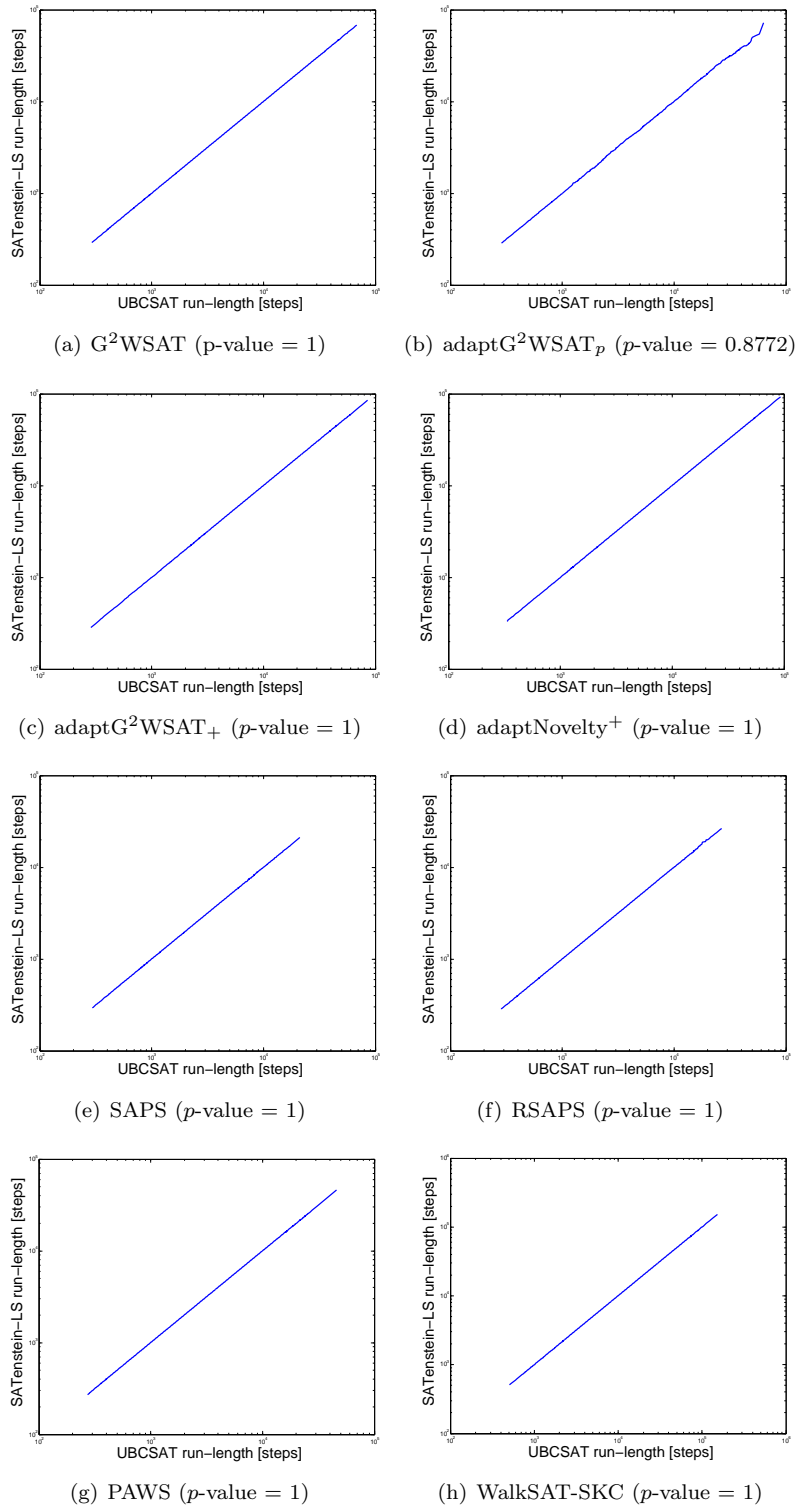
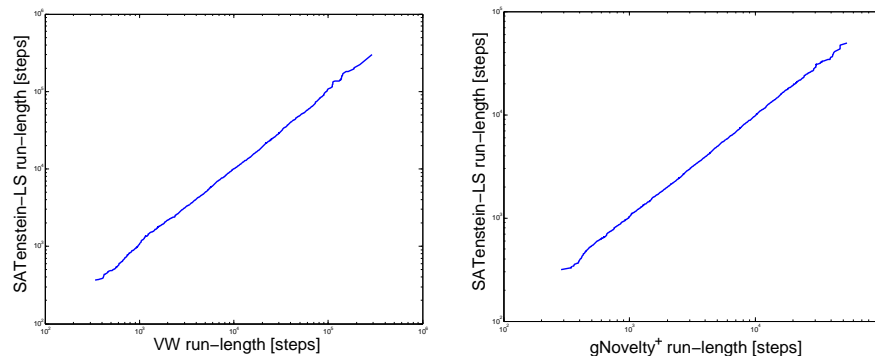


Figure 3.2: Quantile-quantile plots of the run-length distributions for *SATenstein-LS* vs. *UBCSAT* (Tompkins and Hoos, 2004) implementations of various well-known SLS solvers. Each implementation was run 5000 times on instance *bw_large.a* with a per-run cutoff time of 600 CPU seconds. The p -value of the performed KS-test is given in the caption of each subfigure.

performance of ten well-known algorithms implemented in UBCSAT with their SATenstein-LS equivalents. We carefully chose these ten algorithms to cover all three major classes of SLS algorithm and all the prominent mechanisms used in SLS solvers. Since SATenstein-LS and UBCSAT use the same random number generation routine, for a given instance-seed pair, UBCSAT and SATenstein-LS mostly show identical flip performance. However, the SATenstein-LS equivalents of some algorithms (e.g., `adaptG2WSATp`) do not follow algorithm execution path identical to their respective UBCSAT implementations; the flip performance of these algorithms is not exactly identical. Figure 3.2 shows the quantile-quantile plots of the run-length distributions of UBCSAT implementation and SATenstein-LS instantiation of several well-known algorithms. It demonstrates that SATenstein-LS correctly reproduces the behavior of these known algorithms, and that the underlying mechanisms have been correctly implemented in SATenstein-LS.

As mentioned earlier, we implemented the 2005 SAT Competition submission of VW in SATenstein-LS, which is different from Prestwich, 2005 in its use of reactive mechanisms. Also, the 2007 SAT Competition winner `gNovelty+` has not been implemented in UBCSAT, version 1.1. For these two algorithms, we compared the performance with their original SAT Competition submissions (shown in Figure 3.3).



(a) VW SAT Competition 2005 submission (p -value = 0.2392) (b) `gNovelty+` SAT Competitions 2007 submission (p -value = 0.6609)

Figure 3.3: *Quantile-quantile plots of the run-length distributions for SATenstein-LS vs. SAT Competition, 2005 submission of VW (Prestwich, 2005) and SAT Competition, 2007 submission of gNovelty⁺ (Pham et al., 2008). Each implementation is run 5000 times on benchmark instance `bw.large.a` with a per-run cutoff of 600 CPU seconds. The p -value of the performed KS-test is given in the caption of each subfigure.*

Chapter 4

Experimental Setup

In order to study the effectiveness of our proposed approach for algorithm design, we configured `SATenstein-LS` on various problem distributions and compared the performance of the obtained `SATenstein-LS` solvers against that of several existing high-performance solvers on the test set for each distribution. This chapter presents the problem distributions, tuning configurations and machine specifications used in our experiments, describes the solvers we selected for comparison, and explains their selection.

4.1 Instance Distributions

We considered six sets of well-known benchmark instances for SAT (see Table 4.1). These six distributions can be roughly categorized into three broad categories of SAT instances namely, industrial (CBMC(SE), FAC), handmade (QCP, SW-GCP), and random (R3SAT, HGEN). Because SLS algorithms are unable to prove unsatisfiability, we constructed our benchmark sets to include only satisfiable instances. The generators for HGEN (Hirsch, 2002) and FAC (Uchida and Watanabe, 1999) distributions generate only satisfiable instances. For the remaining distributions, we filtered out unsatisfiable instances using complete solvers as described below.

For HGEN, we generated 5000 instances with the number of variables randomly selected from 250 to 400 (Hirsch, 2002). From these 5000 instances, we randomly chose 1000 to form a training set, and another 1000 to form a test set of the same size. For the FAC distribution, we randomly selected 2000 instances from 7140 instances generated (Uchida and Watanabe, 1999) and divided these randomly into a training and test set containing 1000 instances each. The range of the prime numbers we used was from 3000 to 4000.

For QCP, we generated 30,620 instances around the solubility phase transition with a satisfiable/unsatisfiable ratio of 58.7/41.3. For our QCP instance generator, we used the same parameter values given by Gomes and Selman, 1997.¹ We confirmed the satisfiability status of each of the generated instances using several complete solvers. From the satisfiable instances, we randomly sampled 2000 instances and divided them randomly into a training set and test set, each containing 1000 instances. For SW-GCP, we generated 20,000 instances (Gent *et al.*, 1999) with a satisfiable/unsatisfiable ratio of 55.9/44.1

¹ $h \times order^{1.55}$ holes, where h was randomly selected from [1,2,2,2] and $order$ was randomly selected from [10,40].

Distribution	Description
QCP	SAT-encoded quasi-group completion problems (Gomes and Selman, 1997)
SW-GCP	SAT-encoded small-world graph-colouring problems (Gent <i>et al.</i> , 1999)
R3SAT	uniform-random 3-SAT instances (Simon, 2002)
HGEN	random instances generated by generator HGEN2 (Hirsch, 2002)
FAC	SAT-encoded factoring problems (Uchida and Watanabe, 1999)
CBMC(SE)	SAT-encoded bounded model checking problems (Clarke <i>et al.</i> , 2004), preprocessed by SatELite (Eén and Biere, 2005)

Table 4.1: *Our six benchmark distributions.*

and confirmed the satisfiability status of each of the generated instances using several complete solvers. We used the same parameter values given by Gent *et al.*, 1999. The ring lattice size was randomly selected from [100,400] with ten connected nearest neighbors. We set the chromatic number to six and used a rewiring probability of 2^{-7} . From the 20,000 SW-GCP instances thus generated, we randomly sampled 2000 satisfiable instances and divided randomly into training and test sets of 1000 instances each. For R3SAT, we generated a set of 1000 instances with 600 variables and a clauses-to-variables ratio of 4.26 (instances generated with 4.26 clauses-to-variable ratio are around the solubility phase transition and are considered to be hard to solve). We identified 521 satisfiable instances by running `March_p1` (Heule and Maaren, 2007a), each of our 11 high-performance algorithms (see Table 4.4) and `Kcnfs-04` (Dubois and Dequen, 2001) with cutoffs of 3600, 600 and 36000 seconds respectively.² From these 521 instances, we randomly chose 250 to form a training set and another 250 to form a test set of the same size. Using the CBMC generator (Clarke *et al.*, 2004), we created 611 SAT-encoded software verification instances based on a binary search algorithm with different array sizes and loop unwinding numbers. We preprocessed these instances using SatELite (Eén and Biere, 2005). Of the instances thus obtained, we filtered out 7 unsatisfiable instances using the complete solver `Minisat2.0` (Sörensson and Eén, 2007) and confirmed all the others as satisfiable. The 604 satisfiable instances thus obtained were randomly divided into a training set of 303 instances and a test set of 301 instances.

4.2 Tuning Scenario and PAR

In order to perform automatic algorithm configuration using the `SATenstein-LS` framework, we first had to quantify performance using an objective function.

²Our complete solvers were able to rule out only 56 of the remaining 479 instances as unsatisfiable.

We chose to focus on mean runtime. However, efficient algorithm configurators cut off some runs before they terminate, making the mean ill-defined. Thus, following Hutter *et al.*, (2007a), we define the penalized average runtime (PAR) of a set of N runs with a k -second cutoff as the mean runtime over all N runs, with capped runs counted as $10 \cdot k$ seconds.

Parameter	Values considered
tabuLength	1, 3, 5, 7, 10 , 15, 20
phi	3, 4, 5 , 6, 7, 8, 9, 10
theta	3, 4, 5 , 6 , 7, 8, 9, 10
promPhi	3, 4, <u>5</u> , 6, 7, 8, 9, 10
promTheta	3, 4, 5, <u>6</u> , 7, 8, 9, 10
maxinc	5, 10 , 15, 20

Table 4.2: Integer parameters of *SATenstein-LS* and the values considered during *ParamILS* tuning. For each parameter, the known defaults are highlighted in bold. For parameters that are first introduced in *SATenstein-LS*, the default values are indicated with an underline.

For our algorithm configurator, we chose the FocusedILS procedure from the *ParamILS* framework, version 2.2 (Hutter *et al.*, 2008, 2007b), because it is the only approach of which we are aware that is able to operate effectively on extremely large, discrete parameter spaces. We set k to five seconds, and allotted 2 days to each run of FocusedILS. FocusedILS takes as input a parameterized algorithm (target algorithm), a specification of the domains for all the parameters, a set of training instances and an evaluation metric. FocusedILS outputs a parameter configuration of the target algorithm that approximately minimizes the evaluation metric. Since *ParamILS* cannot operate on continuous parameters, each continuous parameter is discretized between 3 and 16 values that seemed reasonable to us (see Table 4.3), always including the known defaults. Except for few parameters (e.g., s, c) for which we used the same discrete domain as mentioned in the paper from the original author (Prestwich, 2005), we mostly selected values using a regular grid and a range that seemed reasonable to us. For integer parameters, we specified 4 to 10 values for each of them, always including the known defaults (see Table 4.2). Categorical parameters and their respective domains are listed in Table 3.2. The total size of the space is 4.82×10^{12} . Since the performance of FocusedILS can vary significantly depending on the order in which instances appear in the training set, we ran FocusedILS 10 times on the training set, using different, randomly determined instance orderings for each run. From the 10 parameter configurations obtained from FocusedILS for each instance distribution D , we selected the parameter configuration with the best penalized average runtime on the training set. We then evaluated this configuration on the test set. For a given distribution D , we refer to the corresponding instantiation of *SATenstein-LS* as *SATenstein-LS*[D].

Parameter	Discrete Values Considered
wp	0, 0.01 , 0.03, 0.04, 0.05, 0.06, 0.07, 0.1, 0.15, 0.20
dp	0.01, 0.03, 0.05 , 0.07, 0.1, 0.15, 0.20
promDp	0.01, 0.03, <u>0.05</u> , 0.07, 0.1, 0.15, 0.20
novNoise	0.1, 0.2, 0.3, 0.4, 0.5 , 0.6, 0.7, 0.8
wpWalk	0.1, 0.2, 0.3, 0.4, 0.5 , 0.6, 0.7, 0.8
promWp	<u>0.01</u> , 0.03, 0.05, 0.07, 0.1, 0.15, 0.20
promNovNoise	0.1, 0.2, 0.3, 0.4, <u>0.5</u> , 0.6, 0.7, 0.8
alpha	1.01, 1.066, 1.126, 1.189, 1.3 , 1.256, 1.326, 1.4
rho	0, 0.17, 0.333, 0.5, 0.666, 0.8 , 0.83, 1
s	0.1 , 0.01, 0.001
c	0.1, 0.01 , 0.001, 0.0001, 0.00001, 0.000001
rdp	0.01, 0.03, 0.05 , 0.07, 0.1, 0.15
rfp	0.01, 0.03, <u>0.05</u> , 0.07, 0.1, 0.15
rwp	0.01 , 0.03, 0.05, 0.07, 0.1, 0.15
pflat	0.05, 0.10, 0.15 , 0.20
ps	0, 0.033, 0.05 , 0.066, 0.1, 0.133, 0.166, 0.2, 0.3, 0.4 , 0.5, 0.6, 0.7, 0.8, 0.9, 1.0

Table 4.3: Values considered for continuous parameters of *SATenstein-LS* during *ParamILS* tuning. For each parameter, the known defaults are highlighted in bold. For parameters that are first introduced in *SATenstein-LS*, the default values are indicated with an underline.

4.3 Solvers Used for Performance Comparison

For each distribution D , we compared the performance of *SATenstein-LS*[D] against that of 11 high-performance SLS-based SAT solvers on the test set. We included every SLS algorithm that won a medal in any category of the SAT Competition in the last five years (SAT Competition, 2009).³ Although dynamic local search (DLS) algorithms did not win any SAT Competition medals, we also included three prominent, high-performing DLS algorithms for the following reasons. First, some of them represented the state of the art when introduced (e.g., *SAPS* (Hutter *et al.*, 2002)) and still offer competitive performance on many instances. Secondly, techniques used in these algorithms are incorporated in other recent high-performance SLS algorithms. For example, the additive clause weighting scheme used in *PAWS* is also used in the 2007 SAT Competition winner *gNovelty*⁺ (Pham *et al.*, 2008). Finally, all the medal-winning SLS algorithms won medals in the random satisfiable category. In our experiments, we have also used problem distributions from the other two categories (hand-made, industrial) where these solvers may not represent the state-of-the-art SLS

³Although we cite 2009 SAT Competition winners at some places, our experiments were conducted before the 2009 SAT Competitions took place. Hence, we compare solvers up to 2007.

solvers. Overall, we selected what we believe to be the most representative high-performance solvers from each SLS solver category. We call these algorithms *challengers* and list them in Table 4.4.

SATenstein-LS can be instantiated such that it emulates all 11 of these challenger algorithms, except for preprocessing components used by three of the challengers (**Ranov**, **G2** and **AG20**). However, in some cases the original implementations of these algorithms are more efficient (on our data, by at most about a factor of two). This is because **SATenstein-LS**'s generality rules out some data structure optimizations. Thus, we based all of our experimental comparisons on the original algorithm implementations, as submitted to the respective SAT Competitions. The exceptions are **PAWS**, to which the **UBCSAT** implementation is almost identical to the original in terms of runtime, **SAPS**, **RSAPS**, and **ANOV**, for which the **UBCSAT** implementations are those used in the competitions. All of our comparisons on the test set are based on running each solver 10 times, with a per-run cutoff of 600 seconds.

Algorithm	Short Name	Reason for Inclusion
Ranov	Ranov	gold 2005 SAT Competition (random)
G ² WSAT	G2	silver 2005 SAT Competition (random)
VW	VW	bronze 2005 SAT Competition (random)
gNovelty ⁺	GNOV	gold 2007 SAT Competition (random)
adaptG ² WSAT ₀	AG20	silver 2007 SAT Competition (random)
adaptG ² WSAT ₊	AG2+	bronze 2007 SAT Competition (random)
adaptNovelty ⁺	ANOV	gold 2004 SAT Competition (random)
adaptG ² WSAT _p	AG2p	performance comparable to adaptG ² WSAT ₊ , G ² WSAT, and Ranov; see (Li et al., 2007a)
SAPS	SAPS	prominent DLS algorithm
RSAPS	RSAPS	prominent DLS algorithm
PAWS	PAWS	prominent DLS algorithm

Table 4.4: Our eleven challenger algorithms.

Category	Solver	Reason for Inclusion
Industrial (CBMC(SE) and FAC)	Picosat (Biere, 2008, 2007) Minisat2.0 (Sörensson and Eén, 2007)	gold, silver 2007 SAT Competition (industrial) bronze, silver 2007 SAT Competition (industrial)
Handmade (QCP and SW-GCP)	Minisat2.0 (Sörensson and Eén, 2007) March_p1 (Heule and Maaren, 2007a)	bronze, silver 2007 SAT Competition (handmade) Improved, bug-free version of March_ks (Heule and Maaren, 2007b), winner of gold in 2007 SAT Competition (handmade)
Random (HGEN and R3SAT)	Kcnfs_04 (Dubois and Dequen, 2001) March_p1 (Heule and Maaren, 2007a)	silver 2007 SAT Competition (random) Improved, bug-free version of March_ks (Heule and Maaren, 2007b), winner of silver in 2007 SAT Competition (random)

Table 4.5: Complete solvers we compared against.

Our goal was to improve the state of the art in SAT solving. Thus, although **SATenstein-LS** can be only instantiated as SLS solvers, we also compared its performance to that of complete solvers (listed in Table 4.5) that are state of

the art for industrial distributions. Unlike SLS solvers, the complete solvers are deterministic. Thus, on every instance in each distribution, we ran each complete solver once with a per-run cutoff of 600 seconds.

4.4 Execution Environment

We carried out our experiments on a cluster of 55 dual 3.2GHz Intel Xeon PCs with 2MB cache and 2GB RAM, running OpenSUSE Linux 10.1. Our computer cluster was managed by a distributed resource manager, Sun Grid Engine software (version 6.0). Runtimes for all algorithms (including FocusedILS) were measured as CPU time on these reference machines. Each run of any solver only used one CPU.

Chapter 5

Results

In this chapter, we present the results of performance comparisons of **SATenstein-LS** against our 11 challenger SLS solvers (listed in Table 4.4), two state-of-the-art complete solvers (see Table 4.5), and a hypothetical **Oracle** that always picks an SLS solver with minimum medium runtime on a per-instance basis. Although in our tuning experiments we optimized **SATenstein-LS** for Penalized Average Runtime (PAR), we also examine its performance in terms of other performance metrics such as median-of-median runtime and percentage of instances solved within cutoff time. We describe each **SATenstein-LS** solver found in terms of the challengers that it most closely resembles. We also present a distance metric between two **SATenstein-LS** configurations and use it to compute the distance between each **SATenstein-LS** solver found and the best **SATenstein-LS** equivalent within the parameter space of each challenger.

5.1 Comparison with Challengers

In every one of our six benchmark distributions, we were able to find a **SATenstein-LS** configuration that outperformed all 11 challengers. Our results are summarized in Table 5.1.

In terms of penalized average runtime, the performance metric we explicitly optimized using ParamILS (evaluated using a different cutoff of 600 seconds during testing), our **SATenstein-LS** solvers achieved better performance than every challenger on every distribution. For QCP, HGEN, and CBMC(SE), **SATenstein-LS** achieved a PAR that was orders of magnitude better than the respective best challengers. For SW-GCP, R3SAT, and FAC there was substantial, but less dramatic improvement. The modest improvement in R3SAT was not very surprising; R3SAT is a well-known SAT distribution on which SLS solvers for the past 17 years have been evaluated and optimized. Conversely, on a new benchmark distribution, CBMC(SE), where DPLL solvers are considered to represent the state of the art, **SATenstein-LS** solvers performed markedly better than every challenger and narrowed the huge performance gap between DPLL solvers and SLS solvers on this distribution. We were surprised to see the amount of improvement we obtained for HGEN, a hard random SAT distribution very similar to R3SAT, and QCP, a widely-known SAT distribution. We noticed that in HGEN, some older solvers such as SAPS and PAWS performed much better than recent SAT Competition medal winners such as GNOV and AG20. Also, for QCP, a relatively older algorithm, ANOV was the best

Solvers	QCP	SW-GCP	R3SAT	HGEN	FAC	CBMC(SE)
SATenstein-LS[D] (KhudaBukhsh et al., 2009)	0.13 0.01 100%	0.03 0.03 100%	1.51 0.14 100%	0.03 0.02 100%	12.22 8.03 100%	5.59 0.02 100%
GNOV (Pham et al., 2008)	422.33 0.03 92.7%	0.24 0.09 100%	10.93 0.14 100%	52.87 0.73 99.4%	5912.99 N/A 0%	2238.7 0.75 61.13%
AG20 (Li et al., 2007a)	1051 0.03 80.5%	0.62 0.12 100%	2.37 0.14 100%	139.33 0.57 98%	3607.4 N/A 30.2%	2170.67 0.67 61.13%
AG20 (Li et al., 2007b)	1080.29 0.03 80.3%	0.45 0.08 100%	3.3 0.16 100%	138.84 0.61 97.8%	1456.4 237.50 84.2%	2161.59 0.77 61.13%
RANOV (Pham and Anbulagan, 2007)	76.22 0.1 98.7%	0.15 0.12 100%	14.14 0.32 100%	156.96 0.95 97.7%	942.26 155.77 92.1%	1231.01 1231.01 79.73%
G2 (Li and Huang, 2005)	2952.56 361.21 80.3%	4103.27 N/A 100%	5.32 0.13 100%	110.02 0.61 97.8%	5944.6 N/A 84.2%	2150.31 0.68 61.13%
VW (Prestwich, 2005)	1025.9 2952.56 82.2%	159.67 40.96 98.9%	9.53 0.75 100%	177.9 3.23 97.5%	3411.93 N/A 31.7%	385.73 0.27 92.69%
ANOV (Hoos, 2002)	28.3 0.01 99.6%	0.06 0.03 100%	12.4 0.21 100%	147.53 0.76 97.6%	3258.66 N/A 37.2%	2081.94 5.81 61.79%
AG2p (Li et al., 2007b)	1104.42 0.02 79.4%	0.45 0.07 100%	2.38 0.13 100%	107.4 0.49 98.4%	1989.91 315.48 72.5%	2282.37 3.18 61.13%
SAPS (Hutter et al., 2002)	1256.2 0.03 79.2%	3872.08 N/A 33.2%	22.81 1.80 100%	48.31 3.00 99.5%	17.39 11.60 100%	613.15 0.04 90.03%
RSAPS (Hutter et al., 2002)	1265.37 0.04 78.4%	5646.39 N/A 5%	14.81 2.13 100%	38.51 2.44 99.7%	19.39 12.88 100%	794.93 0.03 85.38%
PAWS (Thornton et al., 2004)	1144.2 0.02 80.8%	4568.59 N/A 22.1%	2.4 0.12 100%	73.27 0.96 99.2%	26.51 12.62 99.9%	1717.79 19.99 68.77%

Table 5.1: Performance summary of SATenstein-LS and the 11 challengers. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm i on distribution j as $a/b/c$, where a (top) is the penalized average runtime; b (middle) is the median of the median runtimes over all instances (not defined if fewer than half of the median runs failed to find a solution within the cutoff time); c (bottom) is the percentage of instances solved (i.e., having median runtime $<$ cutoff). The best-scoring algorithm(s) in each column are indicated in bold, and additionally the best-scoring challenger(s) are indicated with an underline.

challenger. These observations led us to believe that the strong performance of **SATenstein-LS** was partly due to the fact that the past seven years of SLS SAT solver development did not take these types of distributions into account, nor yielded across-the-board improvements.

We also evaluated the performance of **SATenstein-LS** solvers using other performance metrics like median-of-median runtime and percentage of solved instances. If a solver finishes most of the runs on most instances, the capped runs will not affect its median-of-median performance. Thus the advantage of this metric is that it removes the dependence on the penalty for timed-out runs in used in the calculation of PAR. Table 5.1 shows that although our **SATenstein-LS** solvers are obtained by optimizing for PAR, **SATenstein-LS** solvers outperformed every challenger in every distribution except for R3SAT, in which the challengers achieved slightly better performance than **SATenstein-LS**. Finally, we measured the percentage of instances on which the median runtime was below the cutoff used for capping runs. According to this measure, **SATenstein-LS** either equaled or beat the best challenger, since it solved 100% of the instances in every benchmark distribution. In contrast, only 3 challengers managed to solve more than 40% of the instances in every distribution. Overall, **SATenstein-LS** solvers did well on these measures that were not explicitly optimized.

The relative performance of the challengers varied significantly across different distributions. For example, the three dynamic local search solvers (**SAPS**, **PAWS**, and **RSAPS**) performed substantially better than the other challengers on factoring instances (FAC). However, on SW-GCP, their relative performance is weak. Similarly, **GNOV** (SAT Competition 2007 winner in the random satisfiable category) performed very poorly in our two industrial benchmark distributions, CBMC(SE) and FAC, but solved SW-GCP and HGEN instances quite efficiently.¹ This suggests that different distributions are most efficiently solved by rather different solvers. Our automatic algorithm construction process found solvers that outperformed every challenger on each distribution. This result clearly demonstrates that the vast design space of candidate solvers built from combining components gleaned from existing high-performance algorithms contains better solvers than those previously known, and that the automatic exploration of this huge space can effectively find such improved designs. Of course, solvers optimized for individual benchmark distributions could in principle be combined using an instance-based algorithm selection technique like SATzilla (Nudelman *et al.*, 2004a; Xu *et al.*, 2008), yielding even stronger performance.

So far, we have discussed performance metrics that describe aggregate performance over the entire test set. One might wonder whether a possible reason for the strong performance of **SATenstein-LS** was that its efficiency in solving a few instances compensating for poor performance on other instances. Table 5.2 summarizes the compared performance of each **SATenstein-LS** solver with each

¹Interestingly, on both types of random instances we considered, **GNOV** failed to outperform some of the older solvers, in particular, **PAWS** and **RSAPS**.

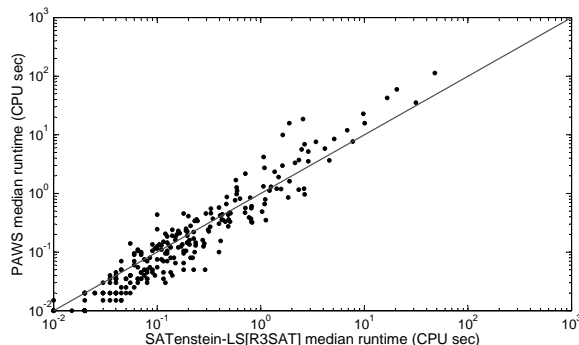


Figure 5.1: Scatter plot of median runtimes of *SATenstein-LS[R3SAT]* vs *PAWS* on the test set for *R3SAT*. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run.

challenger on a per-instance basis, and shows that this is typically not the case. Except for *R3SAT*, *SATenstein-LS* solvers outperformed the respective best challengers for each distribution. On *R3SAT*, *PAWS* was the challenger that outperformed *SATenstein-LS* most frequently (62%). Figure 5.1 shows that the performance of *PAWS* and *SATenstein-LS* were highly correlated, and that, while instances that are easy for both algorithms tended to be solved faster by *PAWS*, *SATenstein-LS* performed better on harder instances. We observe the same qualitative trend for other challengers on *R3SAT*.

Challengers	QCP	SW-GCP	R3SAT	HGEN	FAC	CBMC(SE)
GNOV	99	100	55	100	100	100
AG20	87	98	55	100	100	100
AG20	87	97	64	96	100	100
RANOV	100	100	96	99	100	100
G2	91	100	44	100	100	100
VW	92	100	100	100	100	100
ANOV	83	63	72	99	100	100
AG2p	81	93	55	100	100	100
SAPS	82	100	99	100	73	80
RSAPS	79	100	99	100	78	75
PAWS	79	100	38	100	80	100

Table 5.2: Percentage of instances on which *SATenstein-LS* achieved better median runtime than each of the 11 challengers. Medians were taken over 10 runs on each instance with a cutoff time of 600 CPU seconds per run. When both *SATenstein-LS* and challenger solved a given instance and achieved indistinguishable median runtimes, we counted that instance as 0.5 for *SATenstein-LS* and 0.5 for the challenger.

Our penalized average runtime measure depends on the choice of test cutoff, which sets the penalty. For example, if an algorithm solves all the instances

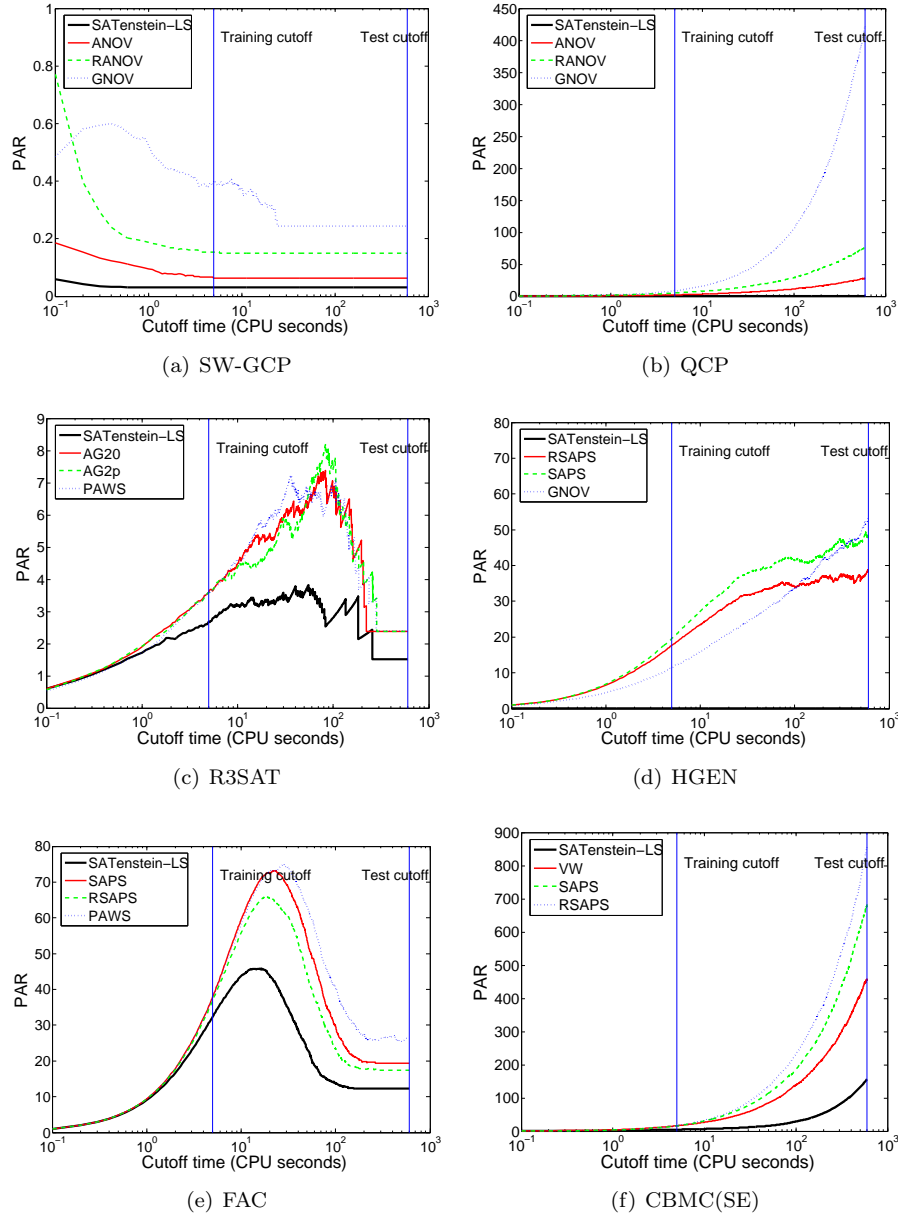


Figure 5.2: Penalized average runtime for our `SATenstein-LS` solver and the top three challengers for each distribution, as a function of cutoff, based on 10 runs on each test-set instance.

but does not solve many of them quickly, then it will score well if the cutoff is large, but it will have weaker PAR score for lower cutoffs. We found it reassuring that this problem did not arise in the **SATenstein-LS** solvers we considered. In Figure 5.2, we plot the PAR score of the **SATenstein-LS** solvers and the top three challengers in each distribution. We see that, while the choice of cutoff increases, the PAR increases (since the penalty for not solving an instance increases) and then decreases (since for solved instances, the penalty gets replaced by the actual runtime). Once all instances on a given test set are solved, PAR remains constant. Figure 5.2 shows that regardless of the choice of cutoff, **SATenstein-LS** always outperformed the top three challengers in every distribution (the only exception being in R3SAT, where **AG20** had a marginally better PAR score than **SATenstein-LS** within a very small range of cutoff times).

5.2 Comparison with a Hypothetical Oracle

We have already demonstrated that, when compared against an individual challenger, our **SATenstein-LS** solvers achieved better performance than every challenger in every benchmark distribution. Figure 5.3 explores the potential of **SATenstein-LS** solvers further by comparing them against a hypothetical **Oracle** that picks the challenger with best runtime on a per-instance basis without any overhead. Note that such an **Oracle** does not exist in the real world.

Figure 5.3 clearly shows that the performance of **SATenstein-LS**[D] was comparable with that of **Oracle**[D] in every distribution, though slightly worse in R3SAT. For QCP, HGEN, and CBMC(SE), **SATenstein-LS**[D] performed much better than the respective **Oracle** solvers. Figure 5.3 also shows a general qualitative trend — on most distributions, in particular for SW-GCP, QCP, and CBMC(SE), the performance advantage of **SATenstein-LS** over the **Oracle** seems to increase with instance hardness. For a given distribution D , the performance of **Oracle**[D] is the upper bound of performance achieved by any algorithm-selection-based method using a given set of algorithms. Note that **SATenstein-LS** is not an algorithm selection approach. Our results suggest that our **SATenstein-LS** solvers found designs that can be better than a per-instance algorithm selection approach and that our new solvers are very likely to perform favorably against any algorithm portfolio constructed by using the challengers.

5.3 Comparison with Complete Solvers

Table 5.3 summarizes the comparison of performance results between **SATenstein-LS** solvers and four complete solvers (two for each distribution). On four of our six benchmark distributions, **SATenstein-LS** solvers comprehensively outperformed the complete solvers. For the two industrial distributions we considered,

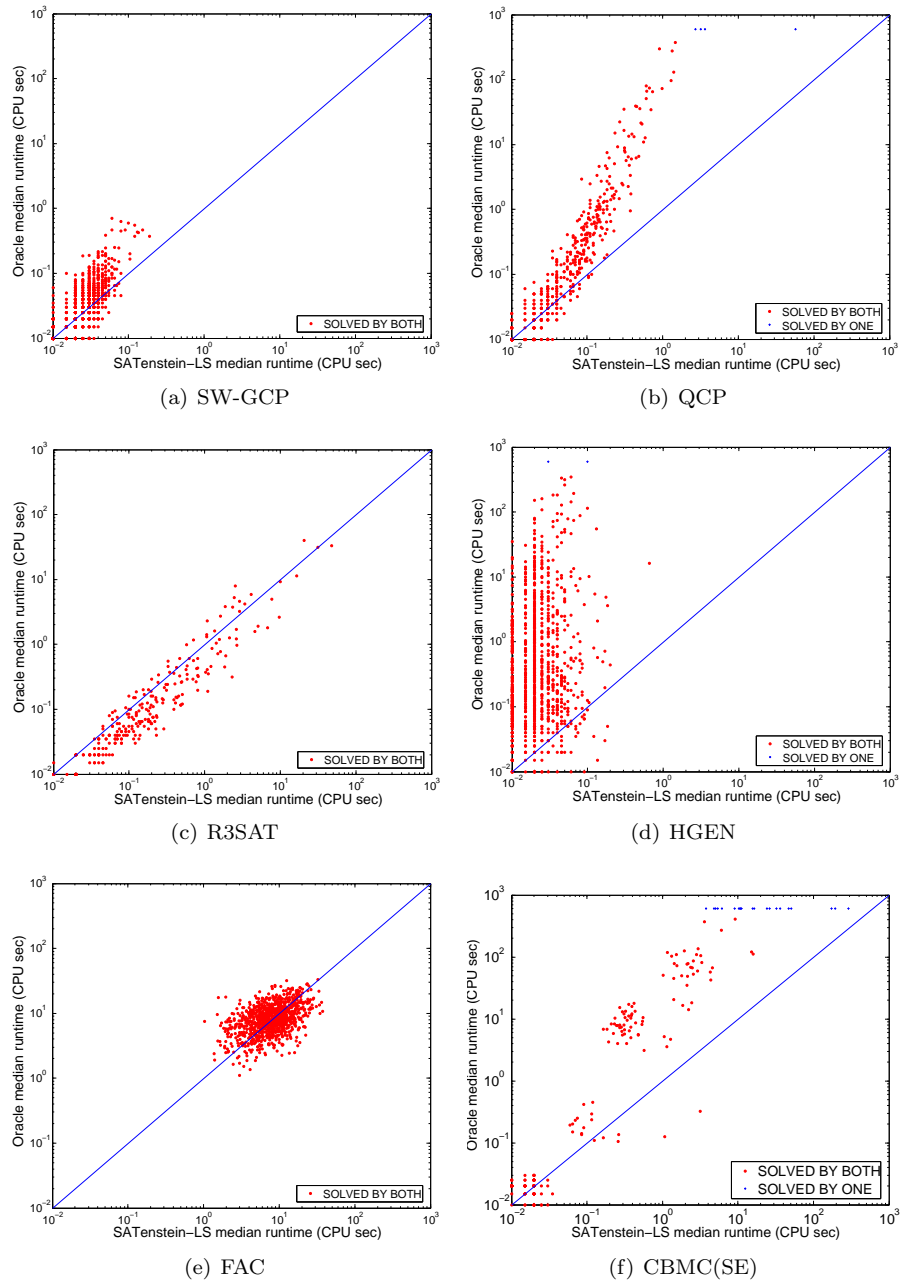


Figure 5.3: Scatter plot of median runtimes of *SATenstein-LS* vs *Oracle* on the test set for each distribution.

Solvers	QCP	SW-GCP	R3SAT	HGEN	FAC	CBMC(SE)
SATenstein-LS [D]	0.13 0.01 100%	0.03 0.03 100%	1.51 0.14 100%	0.03 0.02 100%	12.22 8.03 100%	5.59 0.02 100%
Minisat2.0	<u>35.05</u> <u>0.02</u> 99.5%	<u>2.17</u> <u>0.9</u> 100%			0.03 0.02 100%	0.23 0.03 100%
Picosat					0.02 0.02 100%	0.03 0.01 100%
March_pl	120.29 0.2 98.1%	253.99 1.12 95.8%	<u>3543.01</u> N/A 42%	<u>2763.41</u> 400.78 55.2%		
Kcnfs_04			4905.60 N/A 18.8%	3108.77 N/A 49.5%		

Table 5.3: Performance summary of SATenstein-LS and the complete solvers. Every complete solver was run once (SATenstein-LS was run 10 times) on each instance with a per-run cutoff of 600 CPU seconds. Each cell $\langle i, j \rangle$ summarizes the test-set performance of algorithm i on distribution j as $a/b/c$, where a (top) is the penalized average runtime; b (middle) is the median runtime over all instances (for SATenstein-LS, it is the median of the median runtimes over all instances); c (bottom) is the percentage of instances solved. The best-scoring algorithm(s) in each row are indicated in bold, and additionally the best-scoring complete solver(s) are indicated with an underline. An empty cell, $\langle i, j \rangle$, means we did not run complete solver i on the test set of distribution j .

we found the performance of the complete solvers to be much better than that of the SATenstein-LS solvers and any known local search solver. The strong results on our four benchmark distributions did not come as a surprise, since we primarily chose to compare against the complete solvers because they are known to perform much better than SLS solvers on industrial instances. The success of DPLL solvers on industrial instances is partly attributed to their ability to take advantage of instance structure (e.g., clause learning). With current technology, we find that local search solvers cannot compete with state-of-the-art DPLL solvers on industrial instances. However, SATenstein-LS solvers have made significant progress in closing the gap. For example, for CBMC(SE), complete solvers were five orders of magnitude better than the previously known best SLS solver, VW. SATenstein-LS reduced the performance gap to three orders of magnitude. We also obtained some modest improvements (a factor of 1.42) for the FAC distribution.

5.4 Heterogeneous Mixtures of Instance Distributions

So far, we have seen that for homogeneous distributions, our **SATenstein-LS** solvers performed extremely well. To test our approach on heterogeneous distributions, specifically, we configured **SATenstein-LS** on four distributions made up of instances from two or more homogeneous distributions (see Table 5.4). In order to reduce the effect of having different numbers of training instances, for each heterogeneous distribution, we constructed the training set in the following way. We first select a component distribution uniformly at random. From the component distribution, we then select a problem instance uniformly at random. We continue this process till we gather 2000 training instances. For the test set, we combine all the instances from the component homogeneous distributions. We used the same tuning scenario as for our homogeneous distributions and evaluated the performance of our newly obtained solvers in the same way (see Chapter 4).

Heterogenous Distribution	Component Distributions
Random	R3SAT and HGEN
Handmade	QCP and SW-GCP
Industrial	FAC and CBMC(SE)
Big-mix	R3SAT, HGEN, QCP, FAC, SW-GCP and CBMC(SE)

Table 5.4: Description of heterogeneous distributions we used. In each case, we sampled uniformly at random from the component homogeneous distributions.

Dist	SATenstein-LS[D]	Best Challenger
Handmade	0.11	ANOV : 14.18
Random	19.71	RSAPS : 33.77
Industrial	16.21	SAPS : 155.23
Big-mix	253	RANOV : 340.73

Table 5.5: Performance summary of **SATenstein-LS** and the best challenger. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set penalized average runtime (PAR) of algorithm i on distribution j .

Table 5.5 summarizes the performance of **SATenstein-LS** solvers and the respective best challengers in each of our four mixed distributions. For Handmade and Industrial, **SATenstein-LS** solvers performed much better than their respective best challengers: we obtained, respectively, two and one order(s) of magnitude performance improvements. For Random, we obtained a considerable improvement by a factor of 1.71. We also obtained a 34% overall improvement over the best challenger, RANOV.

We also looked at how these new solvers performed on individual homogeneous distributions (see Table 5.6). We found that there was a performance trade-off between HGEN and R3SAT instances; none of the `SATenstein-LS[Random]` and `SATenstein-LS[Big-mix]` could do well on both distributions. We also found that the high PAR score of `SATenstein-LS[Big-mix]` was mainly due to its poor performance in QCP and CBMC(SE).

Dist	SATenstein-LS[Category]	SATenstein-LS[Big-mix]
QCP	0.17	915.81
SW-GCP	0.07	0.27
R3SAT	97.38	1.55
HGEN	0.29	31.96
FAC	17.35	13.27
CBMC(SE)	12.43	640.82

Table 5.6: Performance summary of `SATenstein-LS` obtained for each category (Handmade, Random, and Industrial) and the Big-mix distribution on a per-distribution basis. Every algorithm was run 10 times on each instance with a cutoff of 600 CPU seconds per run. Each cell $\langle i, j \rangle$ summarizes the test-set penalized average runtime (PAR) of algorithm i on distribution j .

Distribution	Using Block	Most Similar to
QCP	1,3,5	VW, R- <code>Novelty</code> ⁺ , AG20
SW-GCP	1,3	G2, <code>Novelty</code> ⁺⁺
HGEN	1,3,5	G2, VW, ANOV
R3SAT	4, 5	RSAPS with a different tie-breaking mechanism
FAC	4,5	SAPS with a different way of scoring variables
CBMC(SE)	1,4,5	GNOV and RSAPS
Random	1,2,5	AG2+ and VW
Handmade	2, 5	G2, that uses <code>Novelty</code> ^{++'} for promising variable selection
Industrial	4, 5	SAPS with a different tie-breaking mechanism
Big-mix	4, 5	SAPS with a different way of scoring variables

Table 5.7: `SATenstein-LS` configurations.

5.5 Configurations Found

To better understand the `SATenstein-LS[D]` solvers, we compared them with the `SATenstein-LS` instantiations corresponding to each challenger (shown in Table 5.7). `SATenstein-LS[QCP]` uses blocks 1, 3, and 5 and an adaptive parameter update mechanism similar to that in AG2+. Recall that block 1 is used for performing search diversification and block 5 is used to update data structures, tabu attributes and clause penalties. In block 3, which is used to instantiate a solver belonging to WalkSAT architecture, `selectHeuristic` is based on R-`Novelty`⁺, and in block 1, `diversification` flips the variable with minimum vari-

able weight as in VW1 (Prestwich, 2005). `SATenstein-LS[SW-GCP]` uses blocks 1 and 3, resembling `Novelty++` as used within `G2`. `SATenstein-LS[R3SAT]` uses blocks 4 and 5; it is closest to `RSAPS`, but uses a different tie-breaking mechanism. Recall that block 4 is used to instantiate dynamic local search algorithms. `SATenstein-LS[HGEN]` uses blocks 1, 3, and 5. In block 1 it is similar to `G2` and in block 3 is closest to `VW`. In block 5 it uses the same adaptive parameter update mechanism as `ANOV`. `SATenstein-LS[FAC]` uses blocks 4 and 5; its instantiation closely resembles that of `SAPS`, but differs in the way the score of a variable is computed. `SATenstein-LS[CBMC(SE)]` uses blocks 1, 4, and 5, drawing on elements of `GNOV` and `RSAPS`. `SATenstein-LS[Random]` uses blocks 1, 2 and 5. It is closest to `VW` and `AG2+`. It selects the *promising variable* with least `VW2` score and uses `VW1` as its stochastic component. In the update block, it uses reactive mechanisms and also employs a tabu-length of 1. `SATenstein-LS[Handmade]` uses blocks 2 and 5. It is closest to `G2`, but uses a different stochastic component (`R-Novelty+`). It uses `Novelty++'`, one of our novel mechanisms for *promising variable* selection. `SATenstein-LS[Industrial]` uses blocks 4 and 5; it is closest to `SAPS`, but uses a different tie-breaking mechanism. `SATenstein-LS[Big-mix]` uses blocks 4 and 5; it is closest to `SAPS`, but differs in the way the score of a variable is computed. It also considers tabu status in its update mechanism.

Interestingly, only two of the ten configurations we found uses block 2 that uses *promising list*, the block used by all the recent SAT Competition winners. This indicates that many interesting designs that could compete with existing high-performance solvers still remain unexplored in SLS design space. Also, we find that some of the `SATenstein-LS` configurations (e.g., `SATenstein-LS[R3SAT]`, `SATenstein-LS[FAC]`, and `SATenstein-LS[Industrial]`) are indeed very close to existing SLS solvers (e.g., `SAPS` and `RSAPS`) yet perform much better than the algorithms they closely resemble. This shows how even subtle variations in design may lead to substantial performance improvements. This also underscores the importance of an automated approach, since manually finding such subtle changes is very difficult. The full parameter configurations of these algorithms can be found in Table 5.9.

To understand how different our solvers are from existing solvers, we define a new distance metric that measures the degree of similarity between a `SATenstein-LS` configuration (denoted as B) and a challenger. We define our distance function in the following way. First, we find the equivalent `SATenstein-LS` configuration for the challenger. Let this configuration be denoted as A . Then we arrange all active parameters of A in a hierarchical fashion. Level 0 parameters are not conditioned on any other parameters. Any level i parameter is a parameter that is conditioned on the values of one or more level $i-1$ parameters. Next, for each level, we compare all the active parameters in A with those in B . The distance for a given level is the percentage of unmatched active parameters of A . A 100% mismatch at a given level indicates 100% mismatch at subsequent higher levels. The distance between A and B is expressed by a vector $[a_0, a_1, \dots, a_j]$, where $(j + 1)$ is the total number of levels present in the hierarchy of all active parameters of A and a_k denotes the level k distance between A and B .

Table 5.8 presents the distance between the `SATenstein-LS` solvers and the

challengers. For a given distribution D , the closest challenger to `SATenstein-LS[D]` is the challenger that has the minimum level 0 distance. Ties are broken in favor of the challenger with the lesser level 1 distance. Similarly, any remaining ties are broken in favor of the challenger with the lesser level 2 distance. We find that our distance metric reasonably matches our design-level descriptions of the `SATenstein-LS` solvers. For example, for R3SAT, FAC, SW-GCP, Industrial, and Big-mix, the design-level descriptions of the new solvers completely matched with our computed distances. For some distributions there were ties between two or more algorithms. For example, the distances between `SATenstein-LS[Handmade]` and `AG20`, `AG2p`, and `AG2+` were identical. This is expected because `AG20`, `AG2p`, and `AG2+` are very similar algorithms and differ by just one parameter (heuristic). For a given level, we treat all parameters equally when we compute the distance. However, our design-level description is often guided by our perceived importance of certain parameters over others. Because of this, our design-level description did not match with our distance metric for some distributions. For example, `SATenstein-LS[QCP]` is closest to `VW` according to the design-level description. However, according to our distance metric, `AG20`, `AG2p`, and `AG2+` are the closest challengers.

Distribution	GNOV	AG20	AG2+	G2	VW	ANOV	AG2p	SAPS	RSAPS	PAWS
QCP	33 55 0	16 37 0	16 37 0	33 25 0	16 50 0	16 60 0	16 37 0	66 0 0	50 0 0	66 0 0
SW-GCP	50 11 0	33 12 0	33 12 0	16 25 0	33 50 0	33 20 0	33 12 0	50 0 0	66 0 0	50 0 0
R3SAT	50 0 0	50 50 0	50 50 0	66 0 0	33 0 0	33 20 0	50 50 0	16 33 75	0 33 100	16 66 0
HGEN	33 44 0	16 50 0	16 50 0	33 25 0	16 50 0	16 80 0	16 50 0	66 0 0	50 0 0	66 0 0
FAC	66 0 0	66 0 0	66 0 0	50 0 0	50 0 0	50 0 0	66 0 0	0 33 100	16 33 100	0 66 0
CBMC(SE)	33 0 0	50 50 0	50 50 0	66 0 0	50 0 0	50 20 0	50 50 0	33 0 75	16 0 100	33 33 0
Industrial	66 0 0	66 0 0	66 0 0	50 0 0	50 0 0	50 0 0	66 0 0	0 33 100	16 33 100	0 66 0
Handmade	50 33 0	16 25 0	16 25 0	0 50 0	33 50 0	33 20 0	16 25 0	50 0 0	66 0 0	50 0 0
Random	33 77 0	16 62 0	16 62 0	33 50 0	50 50 0	50 60 0	16 62 0	100 0 0	83 0 0	100 0 0
Big-mix	83 0 0	83 0 0	83 0 0	66 0 0	66 0 0	66 0 0	83 0 0	16 33 100	33 33 100	16 66 0

Table 5.8: Distance between *SATenstein-LS[D]* and each challenger. Each cell $\langle i, j \rangle$ summarizes the distance between challenger j and *SATenstein-LS[i]* as $a/b/c$, where a (top) is the level 0 distance; b (middle) is the level 1 distance; c (bottom) is the level 2 distance. For each distribution, the closest algorithm distance(s) are highlighted in bold.

Distribution	Parameter Configuration
QCP	singleClauseAsNeighbor = 1, performSearchDiversification = 1, searchDiversificationStrategy = 4, useAdaptiveMechanism = 0, useClausePenalty = 0, usePromisingList = 0, heuristic=5, performTabuSearch = 0, scoringMeasure = 1, rfp = 0.07, selectClause = 1
SW-GCP	singleClauseAsNeighbor = 1, performSearchDiversification = 1, searchDiversificationStrategy = 3, useAdaptiveMechanism = 0, useClausePenalty = 0, usePromisingList = 0, heuristic = 1, performTabuSearch = 0, novNoise = 0.1 , rdp=0.01, selectClause = 1
R3SAT	singleClauseAsNeighbor = 0, performSearchDiversification = 0, useAdaptiveMechanism = 1, usePromisingList = 0, performTabuSearch = 0, useClausePenalty = 1, selectClause = 1, tieBreaking = 2, scoringMeasure = 1, alpha = 1.126, wp = 0.04, smoothingScheme = 1, rho=0.17,
HGEN	singleClauseAsNeighbor = 1, performSearchDiversification = 1, searchDiversificationStrategy = 3, useAdaptiveMechanism = 1, performTabuSearch = 0, usePromisingList = 0, adaptiveProm = 0, adaptWalkProb = 0, phi=10, selectClause = 1, heuristic = 7, rdp = 0.15, theta = 3
FAC	singleClauseAsNeighbor = 0, performSearchDiversification = 0, useAdaptiveMechanism = 0, usePromisingList = 0, useClausePenalty = 1, performTabuSearch = 0, scoringMeasure = 3, selectClause = 1, wp = 0, alpha = 1.126, smoothingScheme = 1, tieBreaking = 1, ps = 0.033
CBMC(SE)	singleClauseAsNeighbor = 0, performSearchDiversification = 1, useAdaptiveMechanism = 1, searchDiversificationStrategy = 1, usePromisingList = 0, performTabuSearch = 0, useClausePenalty = 1, smoothingScheme = 1, selectClause = 1, tieBreaking = 1, wp = 0.01, rwp=0.01, scoringMeasure = 1, rho = 1, alpha = 1.126,
Random	singleClauseAsNeighbor = 1, performSearchDiversification = 1, useAdaptiveMechanism = 1, searchDiversificationStrategy = 4, usePromisingList = 1, performTabuSearch = 1, heuristic = 7, useClausePenalty = 0, updateSchemePromList = 1, theta = 4, selectPromVariable = 5, adaptiveNoiseScheme = 2, rfp = 0.07, adaptWalkProb = 0, tabuLength = 3, tieBreaking = 3, phi = 7, selectClause = 1
Handmade	singleClauseAsNeighbor = 1, performSearchDiversification = 0, useAdaptiveMechanism = 0, usePromisingList = 1, heuristic = 6, useClausePenalty = 0, performTabuSearch 0, novNoise = 0.1, selectPromVariable = 10, updateSchemePromList = 2, promNovNoise = 0.6, selectClause = 1, adaptiveProm = 0, promDp = 0.01
Industrial	singleClauseAsNeighbor = 0, performSearchDiversification = 0, useAdaptiveMechanism = 0, usePromisingList = 0, alpha = 1.126, performTabuSearch = 0, smoothingScheme = 1, tieBreaking = 3, useClausePenalty = 1, scoringMeasure = 1, ps = 0, rho = 0.83, wp = 0.06
Big-mix	singleClauseAsNeighbor = 0, performSearchDiversification = 0, useAdaptiveMechanism = 0, performTabuSearch = 1, useClausePenalty = 1, usePromisingList = 0, alpha = 1.126, tabuLength = 1, smoothingScheme = 1, scoringMeasure = 3, tieBreaking = 1, ps = 0.033, rho = 0.666, wp = 0.04

Table 5.9: *SATenstein-LS* parameter configuration found for each distribution.

Chapter 6

Conclusions and Future Work

In this work, we have proposed a new way of designing heuristic algorithms that is based on (1) a framework that can flexibly combine components drawn from existing high-performance solvers, and (2) a powerful algorithm configuration tool for finding instantiations that perform well on given sets of instances. We demonstrated the effectiveness of our approach by automatically constructing high-performance stochastic local search solvers for SAT. We presented empirical evidence that our automatically constructed SAT solvers outperformed existing state-of-the-art solvers in several widely studied distributions of SAT instances. Source code and documentation for our **SATenstein-LS** framework are available online at <http://www.cs.ubc.ca/labs/beta/Projects/SATenstein>.

We believe that **SATenstein-LS** is a useful piece of software. Its flexibility allows the user to configure it as a broad range of SLS solvers. No previous SLS solver has a design space as vast as that of **SATenstein-LS**. Also, it inherits all the reporting features that UBCSAT provides. For these reasons, **SATenstein-LS** can be very useful in studying the behavior of different SLS algorithms. Secondly, **SATenstein-LS** can benefit the development of SLS solvers. Instead of trying out new ideas from scratch, one can extend it with new ideas and then, configure the modified **SATenstein-LS** using an automated algorithm configurator. An inclusion of the new strategy in the configurations thus found will suggest that the new strategy has promise. Also, the configurations will provide insights into designs that best complement the new strategy. Finally, **SATenstein-LS** can be used to evaluate the performance of a new SLS solver. We can easily create a wrapper-solver that either instantiates a **SATenstein-LS** solver or behaves like the new solver, depending on a categorical parameter. Then, we can configure the wrapper-solver on different distributions. For a given distribution, the instantiation of the wrapper-solver as the new solver essentially means that the configurator chose the new solver over trillions of solvers that can be instantiated using **SATenstein-LS**. In this way, we can easily compare the performance of a new SLS solver against that of a rich space of solvers.

Although we have already demonstrated strong results that improve the state of the art in solving several types of SAT instances, our framework can be further improved in the following ways. First, **SATenstein-LS** can be extended by including an optional preprocessing block and exposing parameters that specify

the combination of preprocessors. Second, the current design of **SATenstein-LS** does not allow combinations of components from the G^2 WSAT architecture and dynamic local search algorithms. None of the high-performance algorithms we studied combined components from these two categories. However, our results show that dynamic local search algorithms and G^2 WSAT-based algorithms tend to perform well in different distributions. Thus extending **SATenstein-LS** to allow such combinations may lead to finding algorithms with more robust performance across different distributions. Third, at present, our categorical parameters take integer values and can specify only one option among the available ones. **SATenstein-LS** can be extended to allow probabilistic choice among multiple options. In similar vein to the ideas behind **GSAT** with random walk (Selman and Kautz, 1993) and behind the more general **GLSM** model of **SLS** algorithms (Hoos and Stützle, 2005), we can extend the idea of assigning non-zero execution-probabilities to the use of **SATenstein-LS** building blocks 2, 3, and 4. This increases the size of the already huge design space. As a result, we may be able to find algorithms demonstrating even better performance than our present **SATenstein-LS** solvers. Finally, other interesting lines of work concerned with enhancing the framework include exploring the combination of **SATenstein-LS** solvers trained on various types of SAT instances by means of an algorithm selection approach (see, e.g., Xu *et al.*, 2008), and the investigation of algorithm configuration procedures other than **ParamILS** in the context of our approach.

Like Dr. Frankenstein, we find our creations somewhat monstrous, recognizing that our **SATenstein-LS** solvers do not embody the most elegant designs. We have tried to relate **SATenstein-LS** solvers to known algorithms by presenting a novel distance metric based on hierarchically arranged parameters. We have also described **SATenstein-LS** solvers in terms of known algorithm components. Nevertheless, additional work is required to gain a detailed understanding of how the **SATenstein-LS** solvers relate to previously known SAT algorithms and to explore which parameters are crucial for their superior performance.

The tragic figure of Dr. Frankenstein from Mary Shelley’s novel was so haunted by the his own creation that he forever abandoned the ambition of creating a ‘perfect’ human being. Unlike him, we feel encouraged to unleash not only our new solvers, but also the full power of our automated solver-building process, onto other classes of SAT benchmarks. We also believe that the general approach behind **SATenstein-LS** is equally applicable to non-SLS-based solvers and to other combinatorial problems, and that our work can be fruitfully extended to other algorithm/problem combinations.

Bibliography

- D. L. Applegate, R. E. Bixby, V. Chvátal, and W. J. Cook. *The Traveling Salesman Problem: A Computational Study*. Princeton University Press, 2006.
- F. Bacchus and J. 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)*, pages 341–355, 2003.
- F. Bacchus. Enhancing Davis Putnam with extended binary clause reasoning. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 613–619, 2002.
- F. Bacchus. Exploring the computational tradeoff of more reasoning and less searching. In *Proceedings of the Fifth International Conference on Theory and Applications of Satisfiability Testing (SAT'02)*, pages 7–16, 2002.
- P. Balaprakash, M. Birattari, and Thomas Stützle. Improvement strategies for the f-race algorithm: Sampling design and iterative refinement. In *Proceedings of the fourth International Workshop on Hybrid Metaheuristics (MH'07)*, pages 108–122, 2007.
- A. Biere, A. Cimatti, E. M. Clarke, M. Fujita, and Y. Zhu. Symbolic model checking using SAT procedures instead of BDDs. In *Proceedings of Design Automation Conference (DAC'99)*, pages 317–320, 1999.
- A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, editors. *Handbook of Satisfiability*. IOS Press, 2009.
- A. Biere. Picosat version 535. Solver description, SAT competition 2007. <http://www.satcompetition.org/2007/picosat.pdf>. Last accessed on Sept. 16, 2009, 2007.
- A. Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:75–97, 2008.
- M. Birattari, T. Stützle, L. Paquete, and K. Varrentrapp. A racing algorithm for configuring metaheuristics. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2002)*, pages 11–18, 2002.
- T. Carchrae and J. C. Beck. Applying machine learning to low knowledge control of optimization algorithms. *Computational Intelligence*, 21(4):373–387, 2005.

-
- M. Chiarandini, C. Fawcett, and H. H. Hoos. A modular multiphase heuristic solver for post enrollment course timetabling (extended abstract). In *Proceedings of the Seventh International Conference for the Practice and Theory of Automated Timetabling (PATAT'2008)*, 2008.
- E. Clarke, D. Kroening, and F. Lerda. A tool for checking ANSI-C programs. In *Proceedings of the Tenth International Conference on Tools and Algorithms for the Construction and Analysis of Systems (TACAS'2004)*, pages 168–176, 2004.
- S. 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. ACM, 1971.
- J. M. Crawford and A. B. Baker. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 1092–1097, 1994.
- M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM*, 7(1):201–215, 1960.
- R. Dechter and I. Rish. Directional resolution: The Davis-Putnam procedure, revisited. In *Proceedings of the Fourth International Conference on Principles of Knowledge Representation and Reasoning (KR'94)*, pages 134–145, 1994.
- O. Dubois and G. Dequen. A backbone-search heuristic for efficient solving of hard 3-SAT formulae. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI'01)*, pages 248–253, 2001. Last accessed on September 16, 2009.
- N. Eén and A. 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)*, pages 61–75, 2005.
- A. S. Fraenkel. Complexity of protein folding. *Bulletin of Mathematical Biology*, 55:1199–1210, 1993.
- A. S. Fukunaga. Automated discovery of composite SAT variable-selection heuristics. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 641–648, 2002.
- A. S. Fukunaga. Evolving local search heuristics for SAT using genetic programming. In *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO-2004)*, pages 483–494, 2004.
- M. Gagliolo and J. Schmidhuber. Learning dynamic algorithm portfolios. *Annals of Mathematics and Artificial Intelligence*, 47(3-4):295–328, 2006.

- L. D. Gaspero and A. Schaerf. Easysyn++: A tool for automatic synthesis of stochastic local search algorithms. In *Proceedings of the International Workshop on Engineering Stochastic Local Search Algorithms (SLS 2007)*, pages 177–181, 2007.
- A. V. Gelder. Another look at graph coloring via propositional satisfiability. In *Proceedings of Computational Symposium on Graph Coloring and Generalizations (COLOR-02)*, pages 48–54, 2002.
- I. P. Gent and T. Walsh. Towards an understanding of hill-climbing procedures for SAT. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI'93)*, pages 28–33, 1993.
- I. P. Gent, H. H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI'99)*, pages 654–660, 1999.
- C. P. Gomes and B. Selman. Problem structure in the presence of perturbations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 221–226, 1997.
- C. P. Gomes and B. Selman. Algorithm portfolios. *Artificial Intelligence*, 126(1-2):43–62, 2001.
- J. Gratch and G. Dejong. COMPOSER: A probabilistic solution to the utility problem in speed-up learning. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 235–240, 1992.
- A. Guerri and M. Milano. Learning techniques for automatic algorithm portfolio selection. In *Proceedings of the Sixteenth European Conference on Artificial Intelligence (ECAI-04)*, pages 475–479, 2004.
- M. Heule and H. V. Maaren. Improved version of march_ks. http://www.st.ewi.tudelft.nl/sat/Sources/stable/march_pl. Last accessed on Sept. 16, 2009., 2007.
- M. Heule and H. V. Maaren. march_ks. Solver description, SAT competition 2007. http://www.satcompetition.org/2007/march_ks.pdf. Last accessed on Sept. 16, 2009., 2007.
- E. A. Hirsch. Random generator hgen2 of satisfiable formulas in 3-CNF. <http://logic.pdmi.ras.ru/~hirsch/benchmarks/hgen2-1.01.tar.gz>. Last accessed on Sept. 16, 2009., 2002.
- H. H. Hoos and T. Stützle. *Stochastic local search: Foundations and applications*. Morgan Kaufmann, 2005.
- H. 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.

-
- H. H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the Eighteenth National Conference on Artificial Intelligence (AAAI'02)*, pages 655–660, 2002.
- Holger H. Hoos. Computer-aided design of high-performance algorithms. Technical report, University of British Columbia, Department of Computer Science, 2008. Last accessed on Sept. 16, 2009.
- B. A. Huberman, R. M. Lukose, and T. Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, January 3 1997.
- F. Hutter, Holger H. Hoos, K. Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research (JAIR)*. Accepted for publication.
- F. Hutter, D. A. D. Tompkins, and H. H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Eighth International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 233–248, 2002.
- F. Hutter, D. Babić, H. H. Hoos, and A. 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.
- F. Hutter, H. H. Hoos, and T. Stützle. Automatic algorithm configuration based on local search. In *Proceedings of the Twentysecond National Conference on Artificial Intelligence (AAAI'07)*, pages 1152–1157, 2007.
- F. Hutter, H. H. Hoos, T. Stützle, and K. Leyton-Brown. ParamILS version 2.2. <http://www.cs.ubc.ca/labs/beta/Projects/ParamILS>. Last accessed on Sept. 16, 2009., 2008.
- H. Kautz and B. Selman. Pushing the envelope: Planning, propositional logic, and stochastic search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI'96)*, pages 1194–1201, 1996.
- H. Kautz and B. Selman. Unifying SAT-based and graph-based planning. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI'99)*, pages 318–325, 1999.
- A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown. SATenstein: Automatically building local search SAT solvers from components. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 517–524, 2009.
- L. Kroc, A. Sabharwal, C. P. Gomes, and B. Selman. Integrating systematic and local search paradigms: A new strategy for maxSAT. In *Proceedings of the Twenty-first International Joint Conference on Artificial Intelligence (IJCAI'09)*, pages 544–551, 2009.

-
- K. Leyton-Brown, E. Nudelman, and Y. Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Eighth International Conference on Principles and Practice of Constraint Programming (CP'02)*, pages 556–572, 2002.
- Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Empirical hardness models: Methodology and a case study on combinatorial auctions. *Journal of the ACM*, 56(4), 2009.
- C. M. Li and W. 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)*, pages 158–172, 2005.
- C. M. Li, W. Wei, and H. Zhang. Combining adaptive noise and promising decreasing variables in local search for SAT. Solver description, SAT competition 2007, 2007.
- C. M. Li, W. X. Wei, and H. 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)*, pages 121–133, 2007.
- D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI'97)*, pages 321–326, 1997.
- S. Minton. An analytic learning system for specializing heuristics. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 922–929, 1993.
- J. Monette, Y. Deville, and P. V. Hentenryck. Aeon: Synthesizing scheduling algorithms from high-level models. In *Proceeding INFORMS (to appear)*, 2009.
- M. A. Montes de Oca, T. Stützle, M. Birattari, and M. Dorigo. Frankenstein's PSO: A composite particle swarm optimization algorithm. Technical report, IRIDIA, Institut de Recherches Interdisciplinaires, et de Développements en Intelligence Artificielle Université Libre de Bruxelles, 2007. Last accessed on Sept. 16, 2009.
- E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. H. Hoos. SATzilla: An algorithm portfolio for SAT. In *Seventh International Conference on Theory and Applications of Satisfiability Testing, SAT 2004 Competition: Solver Descriptions*, pages 13–14, 2004.
- E. Nudelman, K. Leyton-Brown, A. Devkar, Y. Shoham, and H. H. Hoos. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Tenth International Conference on Principles and Practice of Constraint Programming (CP'04)*, pages 438–452, 2004.

- M. Oltean. Evolving evolutionary algorithms using linear genetic programming. *Evolutionary Computation*, 13(3):387–410, 2005.
- D. N. Pham and Anbulagan. Resolution enhanced SLS solver: R+AdaptNovelty+. Solver description, SAT competition 2007, 2007.
- D. N. Pham, J. Thornton, C. Gretton, and A. Sattar. Combining adaptive and dynamic local search for satisfiability. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:149–172, 2008.
- M. Pop, S. L. Salzberg, and M. Shumway. Genome sequence assembly: Algorithms and issues. *Computer*, 35(7):47–54, 2002.
- S. Prestwich. Random walk with continuously smoothed variable weights. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT'05)*, pages 203–215, 2005.
- J. R. Rice. The algorithm selection problem. *Advances in Computers*, 15:65–118, 1976.
- SAT Competition. <http://www.satcompetition.org>, 2009. Last accessed on Sept. 16, 2009.
- B. Selman and H. A. Kautz. Domain-independent extensions to GSAT : Solving large structured variables. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI'93)*, pages 290–295, 1993.
- B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI'92)*, pages 440–446, 1992.
- B. Selman, H. A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI'94)*, pages 337–343, 1994.
- L. Simon. SAT competition random 3CNF generator. www.satcompetition.org/2003/TOOLBOX/genAlea.c. Last accessed on Sept. 16, 2009., 2002.
- N. Sörensson and N. Eén. Minisat2007. <http://www.cs.chalmers.se/Cs/Research/FormalMethods/MiniSat>. Last accessed on Sept. 16, 2009, 2007.
- P. R. Stephan, R. K. Brayton, and A. L. Sangiovanni-Vincentelli. Combinational test generation using satisfiability. Technical report, EECS Department, University of California, Berkeley, 1992. Last accessed on Sept. 16, 2009.
- S. Subbarayan and D. Pradhan. NiVER: Non-increasing variable elimination resolution for preprocessing SAT instances. *Lecture Notes in Computer Science, Springer*, 3542/2005:276–291, 2005.

-
- J. Thornton, D. N. Pham, S. Bain, and V. Ferreira. Additive versus multiplicative clause weighting for SAT. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI'04)*, pages 191–196, 2004.
- D. A. D. Tompkins and H. H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT. In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT'04)*, pages 37–46, 2004.
- T. Uchida and O. Watanabe. Hard SAT instance generation based on the factorization problem. <http://www.is.titech.ac.jp/~watanabe/gensat/a2/GenAll.tar.gz>, 1999.
- W. Wei, C. M. Li, and H. Zhang. A switching criterion for intensification and diversification in local search for SAT. *Journal on Satisfiability, Boolean Modeling and Computation (JSAT)*, 4:219–237, 2008.
- S. J. Westfold and D. R. Smith. Synthesis of efficient constraint-satisfaction programs. *The Knowledge Engineering Review*, 16(1):69–84, 2001.
- L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla: portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research (JAIR)*, 32:565–606, 2008.
- L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown. SATzilla2009: An automatic algorithm portfolio for SAT. Solver description, 2009 SAT Competition, 2009.

Appendix A

SATenstein-LS User Manual

A.1 Introduction

SATenstein-LS is built on top of the UBCSAT (Tompkins and Hoos, 2004) framework. It is highly recommended that the user go through the quick start guide¹ of UBCSAT to gain more familiarity with SATenstein-LS.

A.2 How to Run SATenstein-LS

SATenstein-LS uses the same interface as UBCSAT. It requires two key command-line parameters:

- an algorithm to use, specified by the [-alg] parameter, and
- a SAT instance file to solve, specified by the [-i] parameter

We run SATenstein-LS by using `satenstein` as the algorithm. So, for a SAT instance ‘sample.cnf’, the simplest execution command for SATenstein-LS is

```
> ubcsat -alg satenstein -i sample.cnf
```

Once you execute this command, in the output, along with the run result (e.g., number of flips, runtime), you will notice a list of parameters. The execution command you just performed ran SATenstein-LS with all the parameters set to their default values. You can find the default value of each parameter listed along with the parameter name. Similar to UBCSAT, the values for these parameters can be specified by [-*paramname* value]. For example, the execution command for running SATenstein-LS with the parameter *adaptive* set to 0 will be

```
> ubcsat -alg satenstein -i sample.cnf -adaptive 0
```

A.3 Configurable Parameters

Table 3.4 and 4.2 respectively present the continuous and integer parameters of SATenstein-LS, along with a short description of each parameter. The categorical parameters are summarized in Table 3.2. Table 4.3 and 3.3 respectively present the parameter values we considered in our experiments.

¹<http://www.satlib.org/ubcsat/quickstart.html>

The parameters that we expose on the command line have slightly different names from those we have used so far. Table A.1 shows the name mapping between SATenstein-LS parameters discussed so far and SATenstein-LS parameters exposed on the command line. Parameters that do not require any such mapping are omitted from this Table.

Parameter Name	Command-Line Name
singleClauseAsNeighbor	singleclause
usePromisingList	promisingList
useClausePenalty	clausepen
performTabuSearch	tabusearch
performSearchDiversification	performrandomwalk
useAdaptiveMechanism	adaptive
adaptWalkProb	adaptwalkprob
adaptiveProm	adaptiveprom
adaptPromWalkProb	adaptPromWalkProb
updateSchemePromList	updateschemepromlist
adaptiveNoiseScheme	adaptivenoisescheme
smoothingScheme	smoothingscheme
searchDiversificationStrategy	randomwalk
selectPromVariable	decreasingvariable
tieBreaking	tiebreaking
scoringMeasure	scoringmeasure
performAlternateNovelty	performalternatenovelty
selectClause	selectclause
promTheta	promtheta
promPhi	promPhi
promDp	promdp
promWp	promwp
promNovNoise	promnovnoise
novNoise	novnoise
wpWalk	wpwalk
psWalk	ps

Table A.1: Name mapping between SATenstein-LS parameters discussed so far and SATenstein-LS parameters exposed on the command line.

A.4 Example Parameter Configurations

Now we may configure some known algorithms with SATenstein-LS. Here we give brief outlines for the algorithms. For detailed description of the algorithms, please refer to the citepd papers.

A.4.1 Novelty+

Let us start with a simple algorithm, Novelty⁺ (Hoos, 1999). This algorithm uses block 3 (`-singleclause 1`) (see Section 3.1) and does not use *promising list* (`-promisinglist 0`). Table 3.6 shows that Novelty⁺ has two heuristic-specific parameters, *novnoise* and *wp*, and they have default values of 0.5 and 0.01, respectively. Table 3.6 also shows that *heuristic* needs to set to 2. Novelty⁺ does not use any adaptive mechanism (`-adaptive 0`).

Hence, the command to configure SATenstein-LS as Novelty⁺ is

```
> ubcsat -alg satenstein -i sample.cnf -selectclause 1 -promisinglist 0
-adaptive 0 -heuristic 2 -wp 0.01 -novnoise 0.5
```

You may want to have a look at the default values of these parameters; you will find that the default values for *selectclause*, *novnoise* and *wp* are the same as the ones used to run Novelty⁺. Hence, this command can be simplified as

```
> ubcsat -alg satenstein -i sample.cnf -promisinglist 0
-adaptive 0 -heuristic 2
```

Simple, isn't it? You may want to run adaptNovelty⁺ (Hoos, 2002). For that, all that you need to do is to change the value of *adaptive*.

The command to configure SATenstein-LS as adaptNovelty⁺ is:

```
> ubcsat -alg satenstein -i sample.cnf -promisinglist 0
-adaptive 1 -heuristic 2
```

A.4.2 gNovelty⁺

gNovelty⁺ (Pham *et al.*, 2008) uses block 2 (*-promisinglist 1*) (see Section 3.1), block 1 (*-performrandomwalk 1*), clause penalties (*-clausepen 1*), smoothing (*-smoothingScheme 1*), and adaptive mechanisms (*-adaptive 1 -adaptwalkprob 0*). It picks the variable with highest score from a non-empty promising list (*-decreasingvariable 2*). If the list is empty, it performs the Novelty heuristic (with “flat moves”) (*-heuristic 1 -performalternatenovelty 1*). In module 1, gNovelty⁺ randomly picks a variable (*-randomwalk 1*) from a randomly selected false clause (*-selectclause 1*) with a probability 0.01 (*-rwp 0.01*).

The command to configure SATenstein-LS as gNovelty⁺ is:

```
> ubcsat -alg satenstein -inst sample.cnf -selectclause 1
-performrandomwalk 1 -randomwalk 1 -rwp 0.01 -promisinglist 1
-decreasingvariable 2 -updateschemepromlist 3 -adaptive 1
-adaptwalkprob 0 -adaptivenoiseScheme 1 -heuristic 1 -smoothingScheme 1
-clausepen 1 -smoothingScheme 1
```

As explained earlier, we can simplify this command by eliminating parameters whose values are same as their respective default values.

A.4.3 SAPS

SAPS (Hutter *et al.*, 2002) uses block 4 (see Section 3.1) (*-promisinglist 0 -singleclause 0*) and does not use any adaptive mechanism (*-adaptive 0*). (Makecount - Breakcount) is used to score a variable (*-scoringmeasure 1*) and any remaining ties are broken randomly (*-tiebreaking 1*). It uses clause

penalties (`-clausepen 1`) and multiplicative smoothing scheme (`-smoothingscheme 1`).

Clearly, the command for running SAPS with the SAPS-specific parameters *alpha*, *rho*, and *wp* set to their respective default values, is

```
> ubcsat -alg satenstein -i sample.cnf -singleclause 0 -promisinglist 0
  -scoringmeasure 1 -tiebreaking 1 -clausepen 1 -smoothingscheme 1
  -adaptive 0
```

Since RSAPS (Hutter *et al.*, 2002) is SAPS with a reactive mechanism, by toggling the value of *adaptive* in the above command, we can configure SATenstein-LS as RSAPS.

PAWS uses exactly the same configuration as SAPS except for the smoothing scheme. By changing the value of *smoothingscheme* to 2 in the above command, we can configure SATenstein-LS as PAWS. The two PAWS specific parameters *maxinc* and *pflat* have default values of 10 and 0.15, respectively.