Dynamic Local Search for SAT

Design, Insights and Analysis

by

David Andrew Douglas Tompkins

B.Sc. (Computer Science), The University of Western Ontario, 1994

B.E.Sc. (Electrical Engineering), The University of Western Ontario, 1996

M.A.Sc. (Electrical Engineering), The University of British Columbia, 2000

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

Doctor of Philosophy

in

THE FACULTY OF GRADUATE STUDIES
(Computer Science)

The University Of British Columbia
(Vancouver)

October 2010

Abstract

In Boolean logic, a formula is satisfiable if a variable assignment exists that will make the formula equivalent to true, and the propositional satisfiability problem (SAT) is to determine if a given formula is satisfiable. SAT is one of the most fundamental problems in computer science, and since many relevant combinatorial problems can be encoded into SAT, it is of substantial theoretical and practical interest. A popular and successful approach to solving combinatorial problems such as SAT is Stochastic Local Search (SLS). In this dissertation we focus on SLS algorithms for SAT, which can find satisfying variable assignments effectively, but cannot determine if no satisfying variable assignment exists.

Our primary goal is to advance the state-of-the-art for SLS algorithms for SAT. We accomplish this goal explicitly by developing new SLS algorithms that outperform the existing algorithms on interesting benchmark problems, and implicitly by advancing the understanding of current algorithms and introducing tools for developing new algorithms. The prevalent theme of our work is Dynamic Local Search (DLS), where DLS algorithms use their search history to dynamically change their behaviour.

The cornerstone of this dissertation is UBCSAT, a software framework we developed for efficiently implementing and empirically evaluating SLS algorithms for SAT. We present the SCALING AND PROBABILISTIC SMOOTHING (SAPS) algorithm, which is amongst the state-of-the-art SLS algorithms for SAT. We provide an in-depth study of a class of DLS algorithms, analyze their performance and significantly advance the understanding of their behaviour. We also advance the understanding of the role of random decisions in SLS algorithms, by providing an empirical analysis on both the quality and quantity of random decisions. The capstone of this dissertation is a new conceptual model for representing and designing SLS algorithms for SAT. We introduce a new software design architecture that implements our model and is specifically designed to leverage recent tools to automate many of the tedious aspects of algorithm design. We demonstrate that by following our new algorithm design approach, we have achieved significant improvements over previous state-of-the-art SLS algorithms for SAT on encodings of software verification benchmark instances.

Table of Contents

List of Tables

List of Figures

List of Algorithms

Here we provide a list of all algorithms referenced in this dissertation, expanding abbreviated names and acronyms. We include in this list algorithm families and algorithm frameworks that are not specific algorithms *per se*. For algorithms that are described in detail in this dissertation, we provide the relevant section and page number below. In Appendix A, we provide a more detailed index of these algorithms, along with any additional implementation details, citations and a complete list of page numbers where the algorithm is referenced.

List of Instance Sets

All instances and instance sets we use in our experiments are described in Appendix B, and we provide some general background on SAT instances in Section 2.2.

Acronyms

AIS	All-Interval Series
ANSI	American National Standards Institute
BETA	Bioinformatics, Empirical & Theoretical Algorithmics
BLP	Binary Linear Program
BTA	Breaking Ties by Age
BTR	Breaking Ties Randomly
CNF	Conjunctive Normal Form
CPD	Clause Penalty value Distribution
CPU	Central Processing Unit
CSP	Constraint Satisfaction Problem
DIMACS	The Center for Discrete Mathematics and Theoretical Computer Science
DLS	Dynamic Local Search
DLS-CP	Dynamic Local Search with Clause Penalties
DPLL	Davis-Putnam-Logemann-Loveland
GCC	GNU Compiler Collection
GNU	GNU's Not Unix!
IDE	Integrated Development Environment
LFG	Lagged Fibonacci Generator
LCG	Linear Congruential Generator

MAX-SAT	Maximum Satisfiability Problem
MT	Mersenne Twister
NIST	National Institute of Standards and Technology
OOP	Object-Oriented Programming
PAC	Probabilistically Approximate Complete
PRNG	Pseudo-Random Number Generator
RAM	Random Access Memory
RLD	Run-Length Distribution
RNG	Random Number Generator
RTD	Run-Time Distribution
SAT	Propositional Satisfiability Problem
SKC	Selman-Kautz-Cohen
SLS	Stochastic Local Search
SQD	Solution Quality Distribution
SQT	Solution Quality over Time
TRNG	True Random Number Generator
UBC	University of British Columbia
VE	Variable Expression
VSM	Variable-Selection Mechanism
QRTD	Qualified Run-Time Distribution

Glossary

$\top$	true
$\bot$	false
$\vee$	Boolean OR operator (disjunction). $(x \vee y)$ is false if and only if x and y are both false.
$\wedge$	Boolean AND operator (conjunction). $(x \wedge y)$ is true if and only if x and y are both true.
$\neg$	Boolean NOT operator. $\neg x$ is true if and only if x is false.
a priori	from what comes before
a.k.a.	also known as
algorithm variant	An *algorithm variant* is a modified version of an existing algorithm. The variant may be a slight modification of the algorithm or be the same algorithm with different implementation details.
$\|C\|$	number of clauses in a formula
$\|C_\top\|$	number of currently satisfied clauses
$\|C_\bot\|$	number of currently unsatisfied clauses
$\|c_j\|$	clause length: number of literals (variables) in clause c_j
CPU time	The *CPU time* is the amount of computer processor time required to complete a single task in a multi-tasking computing environment, as opposed to the *wall clock time* that measures the total time elapsed.
cutoff	The *cutoff* specifies the number of *search steps* at which a *run* is terminated.

c_v	The *coefficient of variation* (c_v) is a measure of the variance in a distribution and is the standard deviation divided by the mean.
de facto	*by the fact*
domination	Algorithm A *dominates* algorithm B on an instance set c if for all instances of c a configuration of A exists that, on average, can solve the instance faster than any configuration of B.
e.g.	*exempli gratia* (for example)
et al.	*et alii* (and others)
evaluation function	An *evaluation function* maps a variable assignment for a formula to numerical a value (see Section 2.3). The *intrinsic* evaluation function is the number of unsatisfied clauses in the formula for the given variable assignment.
$\|F\|$	number of literals in a formula (*i.e.*, the size of a the formula)
flip	A *flip* is when the value of the variable is negated, or changed from true to false or vice-versa. We often use the term *flip* to correspond to a *search step* of an SLS algorithm.
i.e.	*id est* (that is)
improving step	An *improving step* is a *search step* that results in a decrease in the evaluation function.
Kolmogorov-Smirnov distance	The *Kolmogorov-Smirnov distance* is a measure of distance between two distributions of sampled data.
n.a.	*not applicable*
ncv_i	number of clauses a variable v_i appears in
scl_i	sum of the clause lengths for all clauses in which a variable v_i appears
Pearson correlation coefficient (r)	The *Pearson correlation coefficient* (r) is a measure of correlation between two sets of values.
per se	*in itself*

q_α	q_α corresponds to the α-quantile of a distribution. In this dissertation, we use quantiles to extract specific run-lengths from a distribution of run-lengths. For example, the $q_{0.50}$ run-length corresponds to the median run-length.
restart	A *restart* occurs within a *run* and is a re-initialization of the algorithm.
run	An independent *run* of an algorithm on an instance is a sequence of *search steps* that terminates either successfully when a solution has been found or unsuccessfully when a *cutoff* or *timeout* occurs. To distinguish from other notations that are common in the literature, a *restart* does not terminate a run, and several restarts can occur within a single run.
run-length	The *run-length* is the number of *search steps* that occurred in a *run*.
run-time	The *run-time* is the amount of *CPU time* that occurred during a *run*.
search step	A *search step* is one iteration of an SLS algorithm, and typically corresponds to a single variable *flip*.
sideways step	A *sideways step* is a *search step* that results in no change in the evaluation function.
s.f.	The *speedup factor* (*s.f.*) is the ratio of two performance measures.
timeout	The *timeout* specifies the number of elapsed seconds *(CPU time)* at which a *run* is terminated.
$\|V\|$	number of variables in a formula
vs	*versus*
wall clock time	The *wall clock time* is the conventional measure of time. In a multitasking computing environment it is the total amount of time elapsed for a task to complete, even though the *CPU time* may be much smaller.
worsening step	A *worsening step* is a *search step* that results in an increase in the evaluation function.

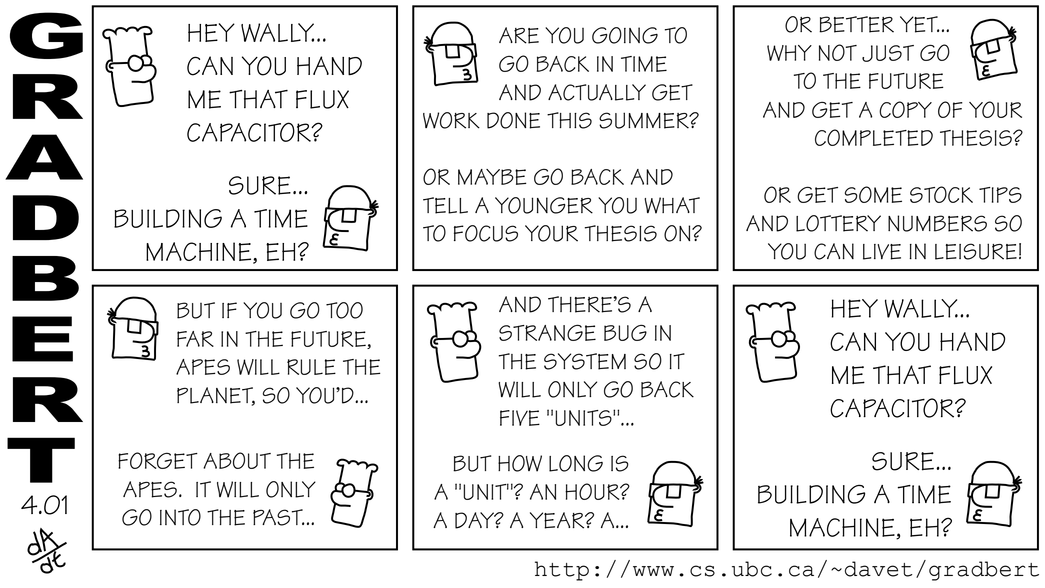
Preface

It's been such a long time, I think I should be goin'.
Time doesn't wait for me, it keeps on rollin'.
— Boston. "Foreplay / Long Time"

This dissertation contains work that spans eight years. Every graduate student has short periods of inactivity, and I had my fair share; however, I have had some exceptionally long breaks as well. I have taken two separate formal year-long leaves of absence in addition to an informal two year hiatus. The primary reasons for my absences were personal and health related, and in the following Acknowledgments Section, I thank the individuals that helped me through those times.

The core concepts and experimental results in each of the non-perfunctory chapters correspond to work originally presented in anonymously peer-reviewed conference proceedings, with the exception of our work in Chapter 5, which is presented here for the first time and has not yet been submitted for publication. In this dissertation we re-present the experimental results from our publications with expanded and updated text. We endeavored to strike a balance between retaining the essential elements of our prior work and presenting this entire dissertation with a cohesive narrative. We present this dissertation in a logical progression of concepts and ideas, rather than in chronological order.

In chronological order, the work in Chapter 4 was completed in 2002 and then published at the 2002 Constraint Programming Conference [61]. In 2004, we presented and published work [112] at the 2004 Artificial Intelligence and Mathematics Symposium; the two concepts from that work became the genesis for Chapter 5 (warped landscapes) and Chapter 6 (random decisions). The work in Chapter 3 was originally presented at the 2004 SAT Conference and published in the formal proceedings in 2005 [113]. In 2006, we revisited and extended our preliminary work on randomization from 2004 [112] and published new work (Chapter 6) that won the best paper award at the 2006 Canadian Conference on Artificial Intelligence [114]. The experiments in Chapter 5, which extended the warped landscapes from 2004 paper [112], were conducted in 2006 and early 2007. Finally, our most recent work in Chapter 7 was presented and published at the 2010 SAT conference [115].


GRADBERT
4.01
HEY WALLY... CAN YOU HAND ME THAT FLUX CAPACITOR?
SURE... BUILDING A TIME MACHINE, EH?
ARE YOU GOING TO GO BACK IN TIME AND ACTUALLY GET WORK DONE THIS SUMMER?
OR MAYBE GO BACK AND TELL A YOUNGER YOU WHAT TO FOCUS YOUR THESIS ON?
OR BETTER YET... WHY NOT JUST GO TO THE FUTURE AND GET A COPY OF YOUR COMPLETED THESIS?
OR GET SOME STOCK TIPS AND LOTTERY NUMBERS SO YOU CAN LIVE IN LEISURE!
BUT IF YOU GO TOO FAR IN THE FUTURE, APES WILL RULE THE PLANET, SO YOU'D...
FORGET ABOUT THE APES. IT WILL ONLY GO INTO THE PAST...
AND THERE'S A STRANGE BUG IN THE SYSTEM SO IT WILL ONLY GO BACK FIVE "UNITS"...
BUT HOW LONG IS A "UNIT"? AN HOUR? A DAY? A YEAR? A...
HEY WALLY... CAN YOU HAND ME THAT FLUX CAPACITOR?
SURE... BUILDING A TIME MACHINE, EH?
http://www.cs.ubc.ca/~davet/gradbert

Acknowledgments

How do I feel by the end of the day?
Are you sad because you're on your own?
No, I get by with a little help from my friends.
— The Beatles. "A Little Help from My Friends"

First and foremost, I thank Holger Hoos for his guidance, patience, understanding and his unlimited supply of carrots and sticks. Acting at times as my very own Severus Snape, Holger always had my best interests at heart, even though I may not have always appreciated it at the time. There is no doubt in my mind that I am a better researcher, and a better person, because of him.

I thank my PhD supervisory committee members, Alan Hu and Will Evans, who have seen me through a tremendous roller-coaster ride and have endured an extraordinary number of extraordinary circumstances. Former committee member Lee Iverson was also very supportive and a source of good advice. I also thank my external examiner Steve Prestwich, as well as my university examiners David Kirkpatrick and David Mitchell for taking the time to read this dissertation and provide valuable feedback. I appreciate the assistance provided by graduate program administrator Joyce Poon and graduate advisors Jim Little and Laks Lakshmanan.

I am thankful for the funding provided by the Natural Sciences and Engineering Research Council of Canada through my PGS-B grant and Holger's discovery grant. I am also thankful for the use of computing resources provided by WestGrid and Compute/Calcul Canada, as well as the arrow cluster provided by Kevin Leyton-Brown. The departmental technical staff are often unappreciated, and I thank them for their hard work, especially Dave Brent.

I thank the keen geeks who provided me with early feedback: Frank Hutter and Ani Sinha. My gratitude especially goes out to my non-geeky proofreaders: Season Johnson and Mallora Rayner.

To the fellow students in the BETA lab, past and present, you have always been encouraging and inspiring. I can't imagine a better place than a theory lab to discuss the finer points of LOST and Harry Potter. Special thanks to Frank Hutter for his help from cradle (SAPS) to grave (PARAMILS), Kevin Smyth for his help with UBCSAT, Chris Fawcett for his help with PARAMILS and Lin Xu for his help with various technical experimental procedures. I have fond memories of fellow labmates Alena Shmygelska, Ashique KhudaBukhsh, Baharak Rastegari, Chris Nell, Dan Tulpan, Jeff Sember, Hosna Jabbari, James Styles, Massih Korvash, Mirela Andronescu, Sanja Rogic, Sohrab Shah and Viann Chan. Steph Durocher and Chris Thachuk especially stand out as individuals who have made life in the BETA lab more fun.

I cannot thank my parents enough (see the following dedication), and I thank my sister Christina, her husband Barry and all of the other members of my family for being so loving and supportive. Jason Garber is my *de facto* brother, and has always been a positive influence in my life. Mike and Mallora Rayner qualify as family, and will always have a special place in my heart. I thank Season's family (Debora, Rick, Reo, Lani, Drew and Heather) for making me feel welcome and providing me with a sense of family while being so far from home.

To my fellow board members (Jerry, Keith, Scott and Walsh) and the extended motley crew (Ben, Corky, DOB, Jamie, and Tim), I hope we're still all friends when we're old and grey (too late).

There have been a lot of people who have made my life in Vancouver, and at UBC, so enjoyable. During the final stages of my PhD I have been a little reclusive, and I look forward to reconnecting with many of you. I want to thank the friends I made at the Alma Mater Society (AMS), the Graduate Student Society (GSS), Koerner's Pub, The Blenheim house, Urban Rec Beach Volleyball and the Vancouver Martini Tours. I also want to give a special 'shout out' to DJ Chiclet and MC Velvet K from DiscoTronic. Jon Freedman believed in me and went "all in"; it ended up changing my life and I will always be grateful. Jack and Erik helped keep me grounded in the "real world", although neither of them live there themselves. I will always cherish my six feet under group (April, Becky, Carrie and Desiree) and fellow one-day-vacationers Genni and Allie. Fellow cinemaphile Dory Boyer was always encouraging, and was calling me Dr. Tompkins long before it was *en vogue*. Suresh Pillai deserves a very special mention, and has been the source of probably too many good times.

During the construction of this dissertation, I consumed copious amounts of Coca-Cola Zero while listening to the Rhythm Radio series from Promo Only Canada.

And finally, my partner, Season. I cannot begin to enumerate all the ways that you have helped me through these last two years and through the final stages of this dissertation. I love you.

Dedication

When I was young it seemed that life was so wonderful…
a miracle, oh it was beautiful, magical.
And all the birds in the trees, well they'd be singing so happily…
oh joyfully, oh playfully watching me.
But then they sent me away to teach me how to be sensible…
logical, oh responsible, practical.
And then they showed me a world where I could be so dependable…
oh clinical, oh intellectual, cynical.
— Supertramp. "The Logical Song"

This is dedicated to my father Douglas Norman for buying our family a TRS-80 Color Computer…
…and to my mother Mary Jane for giving me a book of logic puzzles.

Both of these events occurred at approximately the same time and were clearly formative in my development.

Statement of Collaboration

O Fortuna... velut luna... statu variabilis, semper crescis... aut decrescis.
— Carl Orff. "O Fortuna"

All of the work presented here was conducted under the supervision of, and in collaboration with, Holger Hoos. Most of the work presented was originally published with Holger as a co-author, and much of the original text from those publications has survived the transition to this dissertation. As a result of this collaboration and co-authorship, many of the sentences and paragraphs have been edited by, re-written by or authored by Holger. If the reader encounters a particularly well-written, concise and technically dense sentence, there is a high probability that Holger was involved in its construction.

In Chapter 4, we present the SCALING AND PROBABILISTIC SMOOTHING (SAPS) algorithm. This was the result of a course project Frank Hutter and I completed in Holger's graduate-level course in Stochastic Search Algorithms; we subsequently published this work at the 2002 Constraint Programming conference [61]. The initial concept and design for the SAPS algorithm and the behaviour discussion in Section 4.3 was primarily my work, although Frank was certainly a collaborator. In the same publication we also presented the REACTIVE SAPS (RSAPS) algorithm, which was predominately Frank's design, and as a result I have excluded the RSAPS portion of the paper and generally avoided the use of RSAPS in the experiments throughout this dissertation. Since the paper was a collaborative effort, and we had both made significant contributions, it was decided that our two names would appear alphabetically on the paper. Frank helped conduct the experiments and find the parameter settings for the algorithms presented in Table 4.1 and Table 4.2. In an interesting twist of fate, it was this frustrating process of hand-tuning algorithm parameters that later inspired Frank to develop PARAMILS [59], which is used in our Chapter 7 experiments.

Chapter 1

Introduction

I've got something to tell ya…
I've got news for you…
— The Vengaboys. "We Like To Party"

In this chapter, we motivate our dissertation and identify our major contributions in Section 1.1. Then, in Section 1.2, we provide a brief outline of our dissertation. For those readers not familiar with our field of study, we provide background information in Chapter 2.

1.1 Motivation

The basic axioms of logic that form the backbone of the Propositional Satisfiability Problem (SAT) have been known for over a millennium. Over a century ago, George Boole formalized logic and gave us an algebra. Almost four decades ago, Stephen Cook established SAT as one of the most fundamental problems in computer science [18]. Despite its long history, the study of solving SAT is still a prominent and active area of research that spans many disciplines. We were drawn to SAT because of its purity, elegance and versatility. Many relevant combinatorial problems can be encoded into SAT, so by solving SAT we effectively solve a multitude of problems. In the forward of the recently published *Handbook of Satisfiability*, Turing Award winner Edmund Clarke states "clearly, efficient SAT solving is a key technology for 21st century computer science" [13].

In this dissertation, we focus on Stochastic Local Search (SLS) methods of solving SAT. SLS algorithms are an exciting and active area of research, and there are problem instances in numerous domains (including SAT) where SLS algorithms are the most effective approach for finding a solution. As with SAT, we are attracted to the SLS paradigm because of its simplicity and flexibility, and we believe it closely resembles the way problems are solved in nature. Often a straightforward SLS algorithm that can be described in a sentence or two can exhibit sophisticated behaviour and solve large and hard problems surprisingly effectively and robustly [55]. In addition, SLS techniques developed for solving SAT can often be easily and effectively transferred to other problem domains.

Our primary goal was to advance the state-of-the-art for SLS-based SAT solving. We accomplished this goal explicitly by developing new SLS algorithms that outperform the current state-of-the-art SLS-based SAT solvers on interesting benchmark problems, and implicitly by advancing the understanding of current SAT solvers and introducing development tools for the next generation of SAT solvers. More specifically, our contributions are as follows:

1. We developed UBCSAT, a framework for efficiently implementing and empirically evaluating SLS algorithms. UBCSAT is the cornerstone of our dissertation, and the framework from which we conducted most of our experiments (Chapter 3).

2. We created the SCALING AND PROBABILISTIC SMOOTHING (SAPS) algorithm, which belongs to the class of SLS algorithms for SAT we refer to as Dynamic Local Search with Clause Penalties (DLS-CP). We demonstrated that SAPS dominates the performance of its DLS-CP predecessor, the EXPONENTIATED SUBGRADIENT (ESG) algorithm [97], and is amongst the state-of-the-art SLS algorithms for SAT (Chapter 4).

3. We provided an in-depth study of DLS-CP algorithms, advancing our understanding of their behaviour. We discovered that there are interesting examples of instances where DLS-CP algorithms can identify problem clauses that can be weighted to make solving the instance easier, but we demonstrated that this behaviour is rare and not how DLS-CP algorithms solve most instances in practice. We concluded that typically only the very short-term memory of DLS-CP algorithms is useful, and primarily for escaping local minima (Chapter 5).

4. We studied the role of random decisions in SLS algorithms, and performed an empirical analysis on both the quality and quantity of random decisions. We concluded that SLS algorithms are very robust with respect to the quality of their randomness source, and that widely available Pseudo-Random Number Generators (PRNGs) are of sufficient quality for implementing SLS algorithms. We presented evidence that even highly randomized SLS algorithms can be derandomized in a straightforward manner without significantly changing their behaviour. We observed an interesting phenomenon where, by making only one or two changes in their initial variable assignment, derandomized algorithms can exhibit the same full variability in run-time observed for randomized algorithms. (Chapter 6).

5. We introduced a new conceptual model for representing and designing new SLS algorithms with variable expressions. We developed the DESIGN ARCHITECTURE FOR VARIABLE EXPRESSIONS (DAVE), an extension of UBCSAT that implements our model. DAVE was

designed to leverage the use of recent automated algorithm configuration tools for the automated development of new algorithms. DAVE is the capstone of our dissertation, and we demonstrated that by following our new algorithm design approach, we achieved significant improvements over previous state-of-the-art SLS-based SAT solvers on software verification benchmark instances (Chapter 7).

Throughout this dissertation, the prevalent theme is the study of how SLS algorithms for SAT incorporate elements of history into their search. We refer to algorithms that incorporate history into their search as Dynamic Local Search (DLS) algorithms, as they *dynamically* change their behaviour throughout the search. We revisit this theme throughout this dissertation and discuss it further in Chapter 8.

1.2 Overview

The remainder of this dissertation is structured as follows. First, in Chapter 2, we provide a general background on topics that are pervasive throughout this dissertation; for topics that are more relevant to a single chapter we present that background material in the chapter as needed. Next, in Chapter 3, we motivate and introduce our highly successful UBCSAT software. Then, in Chapter 4, we present our SAPS algorithm, and introduce the basic concepts behind Chapter 5 and Chapter 6. Next, in Chapter 5, we conduct a comprehensive study of DLS-CP algorithms, providing many insights into their behaviour. Then, in Chapter 6, we present our award-winning work on the impact of random decisions in SLS algorithms. Next, in Chapter 7, we present our most recent work on our new conceptual model and its highly flexible software implementation (DAVE). Finally, in Chapter 8, we summarize our contributions and propose several areas where our work can be extended.

Chapter 2

Background

I can't get no satisfaction…
'Cause I try, and I try, and I try, and I try.
— The Rolling Stones. "Satisfaction"

In this chapter, we provide background information and work related to this dissertation as a whole. In the subsequent chapters, we provide additional background material that is more specifically related to the content of those chapters. The remainder of this chapter is structured as follows. First, in Section 2.1, we introduce the propositional satisfiability problem (SAT), which is the focus of this dissertation. Next, in Section 2.2, we discuss the sources of SAT instances. Then, in Section 2.3, we describe general approaches to solving SAT, including Stochastic Local Search (SLS). Next, in Section 2.4, we introduce the technical notations we used throughout this dissertation to describe SLS algorithms. In Section 2.5, we outline some of the more popular SLS algorithms for solving SAT. Finally, in Section 2.6, we describe Dynamic Local Search (DLS) approaches to solving SAT.

2.1 The Propositional Satisfiability Problem

In Boolean logic, a formula is satisfiable if, and only if, a variable assignment exists that will make the formula equivalent to true, and the propositional satisfiability problem (SAT) is to determine whether a given formula is satisfiable. The only goal of this *decision* variant of the problem is to determine if a variable assignment exists, whereas the goal of the *model-finding* variant of the problem is to also find such a variable assignment if it does exist. A variable assignment that satisfies a formula is called a *solution*. The process of finding a solution of a given formula is referred to as *solving* the formula, and an algorithm designed to find solutions of a given formula is referred to as a SAT *solver*.

The propositional satisfiability problem only requires that the formula be well-formed, but in practice it is convenient to require that the formula be in Conjunctive Normal Form (CNF). A formula in CNF is a conjunction of *clauses*, where clauses are disjunctions of *literals*, and a literal is either positive (a variable) or negative (the negation of a variable). An example of a formula in CNF is:

$$F = (x_1 \vee x_2) \wedge (\neg x_1 \vee x_2) \wedge (x_1 \vee \neg x_2) \wedge (\neg x_3 \vee x_4) \wedge (\neg x_3 \vee x_5) \wedge (\neg x_1 \vee \neg x_2 \vee x_3). \tag{2.1}$$

The SAT formula in Equation 2.1 has five variables and six clauses, where the first clause $(x_1 \vee x_2)$ has two positive literals. If, for a particular variable assignment, a literal is equivalent to true then the literal is *satisfied*. If at least one of the literals in a clause is satisfied, then the clause is satisfied, and if all clauses in the formula are satisfied then the formula is satisfied and a solution has been found. The formula in Equation 2.1 has a single solution, in which all variables are assigned the value true. Throughout this dissertation, when we refer to SAT, we are referring to the model-finding variant of the propositional satisfiability problem with formulae in CNF. An *instance* in the SAT domain is a Boolean formula in CNF.

If the maximum clause length in a formula is k, the formula is referred to as a k-SAT formula. It has been established that 2-SAT formulae can be solved in linear time [6], while the problem of solving 3-SAT (and higher) formulae belongs to a complexity class known as $\mathcal{NP}$-complete [18]. If any $\mathcal{NP}$-complete problem can be solved in polynomial time, then all problems in $\mathcal{NP}$ can also be solved in polynomial time. The question of whether or not an algorithm exists that can solve 3-SAT in guaranteed polynomial time remains an open problem, and is beyond the scope of this dissertation. Since SAT is both conceptually straightforward and in $\mathcal{NP}$-complete, it is of substantial theoretical and practical interest and plays an important role in many areas of computing.

The Maximum Satisfiability Problem (MAX-SAT) is the model-finding *optimization variant* of the propositional satisfiability problem, where the objective is to find a variable assignment that satisfies the maximal number of clauses. In weighted MAX-SAT, a *weight* is assigned to each clause, and the objective is to find a variable assignment that maximizes the sum of the satisfied clause weights. The focus of this dissertation is not on MAX-SAT, although we discuss it in Section 3.6 and Section 5.2.

2.2 Sources of SAT Instances

To empirically study SAT solvers, we need SAT instances to solve. Throughout this dissertation we often refer to specific instances or *sets* of instances we have used in our experiments, and we provide brief descriptions and references for those instances in Appendix B. In this section, we provide a general overview of SAT instances.

One of the reasons SAT is a popular area of research is that it is often straightforward to *encode* an instance from another $\mathcal{NP}$ problem domain into SAT [36, 89]. As an example, we will study an encoding of the All-Interval Series (AIS) problem, and we will refer back to this encoding in Section 5.2.

The AIS problem is of theoretical interest [104] and also has applications in music theory [50]. The ais-N problem is to arrange the integers $1 \ldots N$ such that the set of differences between adjacent values (the intervals) consists of all the numbers ranging from $1 \ldots (N-1)$. For this example, we consider ais10, for which one solution is 10 1 9 2 8 3 7 4 6 5. For this encoding, we use two types of variables and six types of clauses. First, we use $N \cdot N = 100$ Boolean variables $(x_{n,p})$ to indicate if the number n occupies position p. For the solution above, $x_{10,1}$ and $x_{9,3}$ are true, whereas $x_{9,1}$ is false. Next we use $(N-1) \cdot (N-1)$ variables $y_{i,p}$ to indicate if the interval i occurs between the numbers in positions p and $(p+1)$. For the solution above, $y_{9,1}$ is true. The first type of clause ensures that each number appears at most once; for example, the number 10 cannot occupy both positions 5 and 6 and the clause $(\neg x_{10,6} \vee \neg x_{10,5})$ ensures this is true. The second type of clause ensures that each number occupies at least one position. For example, the number 3 must occupy at least one position, so the clause $(x_{3,1} \vee x_{3,2} \vee \ldots \vee x_{3,10})$ ensures this is true. Together, these two types of clauses ensure that each number appears once and only once. The next two types of clauses are nearly identical to the first two, and ensure that each *interval* also appears once and only once. The fifth type of clause is similar to the first, but instead ensures that each position is filled by no more than one number. Finally, the sixth type of clause maps the number positions to the intervals; for example, if the first number is 10 and the second is 1, the first interval will be 9 and the clause $(\neg x_{10,1} \vee \neg x_{1,2} \vee y_{9,1})$ ensures this is true. In total, this encoding of the ais10 instance has 181 variables and 3 151 clauses. As with most encodings into SAT, there are several different methods available to perform this encoding [89], and there are publications that study this particular encoding [1, 38].

To empirically compare the performance of different algorithms, we rely on *benchmark* instances, which are widely available and known to other researchers. Two popular sources of instances are the SATLIB library (hosted at [138]) [54] and the quasi-annual SAT Competition (hosted at [137]). SAT benchmark instances have been typically grouped into three categories: *random*, *application* and *crafted* (*e.g.*, these are the three main categories in the most recent competition) where crafted and application instances are also loosely referred to as *structured* instances. Each of the three categories of instances have advantages and disadvantages for empirical study.

Random instances are generated using a randomized source such as a Pseudo-Random Number Generator (PRNG). The most common and well-studied class of random instances is known as the unforced random (uf) instances, where for each clause three unique variables are selected uniformly at random and then randomly negated. The hardness of a uf instance depends on the clause to variable ratio: instances with too few clauses are trivially easy, and instances with too many clauses are unsatisfiable. There is a so-called phase transition between these two extremes where the hardest problems are obtained [15, 82]. All of the uf instances we use in experiments are situated in this phase transition. Random instances are scalable, easy to generate, well studied and can have large variation in hardness. One possible drawback of experimenting exclusively with random instances is that the results may not generalize to instances encountered in practical applications.

Application instances (*a.k.a.* industrial instances) arise from scenarios where the source of the encoding is a so-called *real world* problem, such as a software verification or scheduling. Encodings of these problems are typically very structured and can have clauses similar in structure to those in the ais encoding example above. In Chapter 7, our focus is on the industrial software verification sets cbmc and swv. Whereas it is reasonably straightforward to generate large random instance sets with a desired property (such as hardness), it is more difficult to obtain application instance sets of the same size and quality.

The ais example above is a crafted instance, where crafted instances (*a.k.a.* handmade instances) are constructed to solve a particular class of problem, often with the purpose to obtain instances with a specific structural property. Crafted instances are often encodings from other domains. Examples of such crafted instances include the qg and flat instances. Crafted instances may include some source of randomness, typically introduced in the original domain (before encoding), so the resulting SAT instance has elements of both the encoding structure and the original randomness. Crafted instances are often designed to expose weaknesses in SAT solvers.

2.3 Methods of Solving SAT

Before we introduce Stochastic Local Search (SLS), we first describe the most popular alternative approach to solving SAT, a constructive backtracking method referred to as a DPLL-based approach (named for the authors Davis, Putnam, Logemann and Loveland) [20, 21]. DPLL-based solvers can be described as recursive procedures, where at each level in the recursion a variable from the formula is selected and assigned a value of either true or false. Once a variable is assigned a value, a new *simplified* formula can be generated by removing the resulting satisfied clauses and unsatisfied literals. Additional simplification methods are typically used, including unit propagation. Unit propagation occurs when a clause has only a single literal remaining, causing the corresponding variable to be assigned the value that satisfies the literal and the clause. During simplification, additional variables can be assigned values, causing further simplification; once simplification is exhausted, there are three possible outcomes for the new simplified formula. If the new formula is empty (*i.e.*, all the clauses are true), a solution has been found and the procedure terminates. If the new formula contains an empty clause (*i.e.*, all literals are false), a contradiction has been found. Otherwise, if there is neither a solution nor a contradiction, the new formula is tested for satisfiability by recursively calling the DPLL-based procedure. If the new formula is unsatisfiable or there was a contradiction found, the truth value of the variable is negated and a new simplified formula is generated and tested. If both variable assignments generate a formula that is unsatisfiable or contain a contradiction, the given formula is unsatisfiable and the procedure backtracks to the previous recursion level or terminates the procedure if it is at the highest recursion level (*i.e.*, the original formula is unsatisfiable).

In this dissertation, we do not focus on DPLL-based approaches, but we make occasional reference to the approach for comparisons. A large body of research has been developed to improve DPLL-based solvers, including: methods to select the variables and assignment at each recursion, methods for non-chronological backtracking to different recursion levels, methods of learning new clauses as the search progresses to aid in the search, randomization techniques and restart strategies [13]. DPLL-based algorithms are the most effective methods currently known for solving large structured instances, especially application instances. Two examples of DPLL-based solvers that performed well in the 2009 SAT Competition [137] were PRECOSAT [133] and GLUCOSE [8].

The approach we focus on is known as Stochastic Local Search (SLS), where an SLS algorithm starts with a complete variable assignment (instead of constructing a solution variable-by-variable as in a DPLL-based approach). The complete variable assignment is known as a *candidate solution*. The fundamental approach of SLS algorithms is to iteratively generate and evaluate candidate solutions until a solution is found. The largest distinction between two SLS algorithms is their method of selecting the next candidate solution at each iteration. By definition and in practice, random decisions are an essential ingredient of most SLS algorithms, and we empirically analyze the role of randomness in these algorithms in Chapter 6. SLS-based algorithms are the most effective methods currently known for solving satisfiable random instances [137] as well as many crafted problems (*e.g.*, graph colouring instances [70]).

In Figure 2.1, we provide pseudo-code for a typical SLS algorithm for SAT. The algorithm starts by *initializing* the search, which includes determining an initial, complete assignment of truth values for all variables in the given SAT formula. Then, in each *search step*, unless a *restart* condition has been met, a set of variables is selected to have their truth values changed from true to false or vice versa. Each change of a single variable's truth value is called a *variable flip*. Almost all SLS algorithms perform exactly one variable flip in each search step, but there are cases in which a given SLS algorithm may flip no variables in a given search step (a so-called *null-flip*), or several variables at once (a *multi-flip*). The search process is terminated when a *termination condition* is met; this is typically the case when either a solution (satisfying assignment) has been found or when a given bound on the execution time, often measured in search steps or CPU time, has been reached or exceeded. To overcome or avoid search stagnation, many SLS algorithms for SAT make use of a *restart mechanism* that re-initializes the search process whenever a *restart condition* is met. While restart mechanisms are crucial for the performance of some SLS algorithms for SAT, they have been found to be ineffective in other cases [55: p. 225], and in this dissertation we do not focus on restart conditions. The entire search process from start to finish in Figure 2.1 is a *run* of an algorithm on an instance. Throughout the run, we maintain a count of how many search steps have occurred. The total number of steps that occurred in the run is the *run-length*, and the amount of CPU time that

```
Algorithm TypicalSLS
Input: SAT formula
Output: solution or no solution found

initialize search
while not solution found and not termination conditions met do
    if restart conditions met then
        re-initialize search
    else
        select the variable(s) to flip
        flip the variable(s)
    end
end
if solution found then
    return solution
else
    return no solution found
end
```

Figure 2.1: Typical SLS algorithm for SAT. Included in the search initialization is the assignment of truth values to all variables in the SAT formula.

elapsed is the *run-time*.

SLS algorithms traverse what is known as a *search landscape* defined by a neighbourhood graph and an evaluation function. The *neighbourhood graph* is a graph whose vertices are all possible candidate solutions and whose edges are defined by a neighbourhood relation. For a formula with $|V|$ variables, there are $2^{|V|}$ possible candidate solutions (or *points*) in the search space. Most SLS algorithms for SAT use the so-called one-flip neighbourhood relation (*i.e.*, at each step at most one variable is flipped) under which candidate solutions are direct neighbours if, and only if, they differ only in the truth value assigned to one variable. In Figure 2.2, we present the one-flip neighbourhood graph for the instance described in Equation 2.1.

An *evaluation function* maps a candidate solution to a numerical value. SLS algorithms for SAT use a wide variety of evaluation functions, but the most obvious and common evaluation function, which we shall refer to as the *intrinsic* evaluation function, is the number of unsatisfied clauses for the given variable assignment. In other words, a solution to the formula has a intrinsic evaluation function value of zero, and a common goal of SAT solvers is to minimize the intrinsic evaluation function. In the context of the search landscape, the value of the evaluation function intuitively corresponds to the *height* of each point in the landscape. We often refer to a search step in the context of the change in height that occurs during the step; the terms *improving step*, *sideways step* and *worsening step* correspond to steps that have respectively lower, equal, or greater evaluation

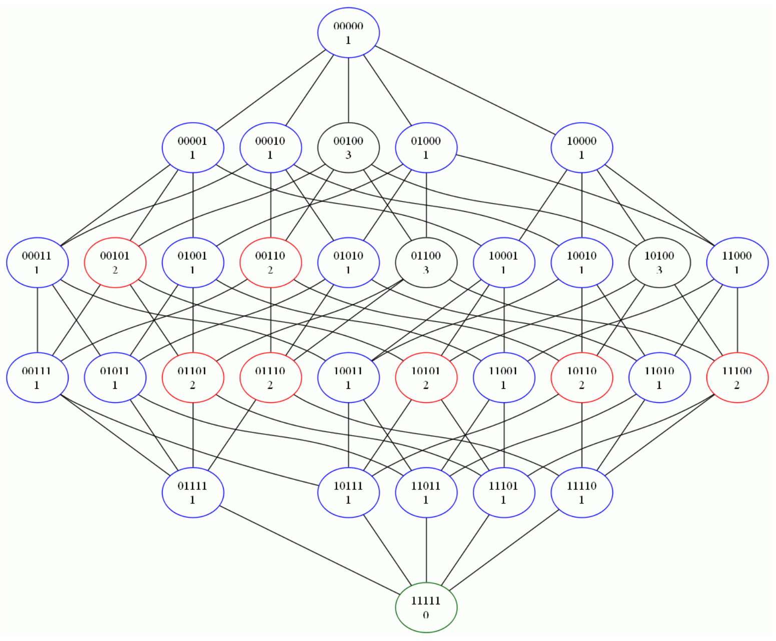


Figure 2.2: SAT neighbourhood graph. The neighbourhood graph for the instance described in Equation 2.1. Each node corresponds to a unique variable assignment (*i.e.*, the node labeled "00001" corresponds to the assignment with all variables equal to false except for x_5). In addition to the variable assignment, each node contains the number of false clauses under that assignment. With five variables there are $2^5 = 32$ nodes in the graph, with each node connected to five neighbours that differ in exactly one variable's assigned value.

function values. When there are no improving steps possible from a location, the location is a *local minimum*, and if there are only worsening steps possible, then the location is a *strict* local minimum. In Figure 2.3, we reproduce the neighbourhood graph of Figure 2.2 as a search landscape, arranging the variable assignments by the value of the intrinsic evaluation function value for that assignment (height).

Now that we have briefly described both SLS algorithms and DPLL-based algorithms for SAT, we can see one of the main distinctions between the two is that DPLL-based algorithms are considered *complete* (*i.e.*, they can determine if an instance is unsatisfiable), whereas a typical SLS algorithm cannot. However, in practice, this theoretical distinction is not as significant as it may appear because SLS-based solvers can be rendered complete with a straightforward extension [24], and DPLL-based solvers that cannot solve an instance within a specified amount of time are essentially incomplete. Another distinction between the two approaches is that for satisfiable instances there is a known worst-case bound on the time required for DPLL-based solvers to find a solution (*i.e.*, $O(2^{|V|})$), whereas no such bound exists for SLS-based solvers, and for some SLS-based

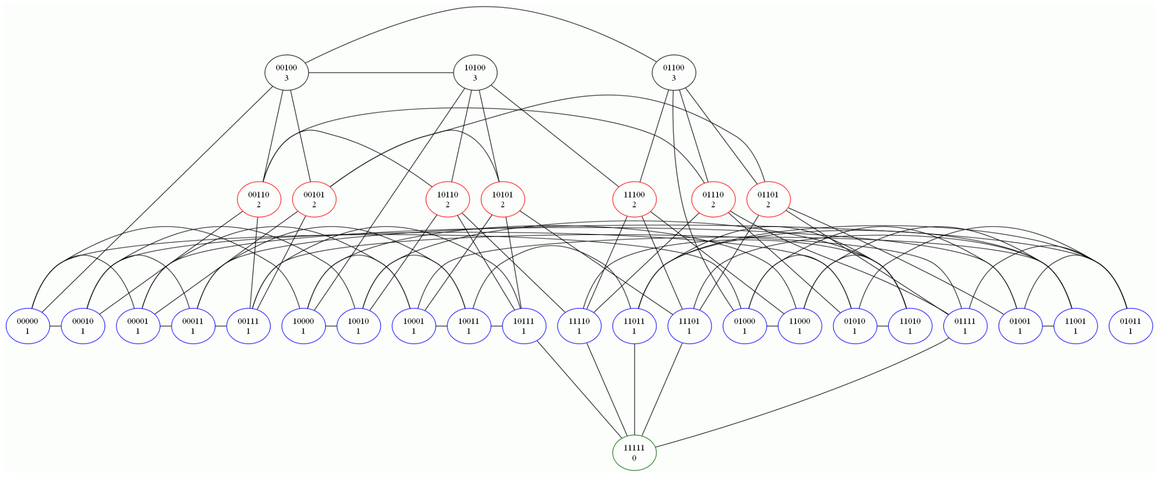


Figure 2.3: SAT search landscape. The neighbourhood graph from Figure 2.2 with the nodes in the graph (variable assignments) arranged by height to correspond to the solution quality (number of false clauses) of the assignment.

solvers there is no guarantee that a solution will be found. However, many SLS algorithms are Probabilistically Approximate Complete (PAC) and will solve a satisfiable instance with arbitrarily high probability when allowed to run long enough [55: p. 153]. There are numerous other distinctions between DPLL-based algorithms and SLS algorithms that we do not explore here, with advantages and disadvantages to each approach [31].

2.4 Modelling SLS Algorithms for SAT

Throughout this dissertation we use our own pseudo-code syntax for describing and representing SLS algorithms. Our goal is to be clear and concise, while providing enough technical detail for practical analysis and discussion. Our approach will be familiar and easy to understand for individuals with a computer science or programming background. We will describe the features of our syntax as they are introduced.

We use *objects*, similar in nature to classes in Object-Oriented Programming (OOP), where objects are simply a collection of *properties*. A property can be a single value, another object, or a list (array) of objects. In OOP parlance, a property encompasses both data fields (members) and methods that return a value. We will not be concerned with OOP-related issues of distinguishing between an object and an instance of an object, an object and a reference (or pointer) to an object, or if a property is a data field or a method.

In Figure 2.4, we describe the minimal set of objects required by an SLS algorithm to represent a SAT formula. We can see that a formula object has two properties: one is a list of clause objects

Object formula

---

property : Clauses[ ] **is** list of clause objects
All clauses that appear in the formula.

property : Variables[ ] **is** list of variable objects
All variables that appear in the formula.

Object variable

---

property : value **is** Boolean
Current (run-time) assignment for the variable.

Object clause

---

property : Literals[ ] **is** list of literal objects
All literals that appear in the clause.

Object literal

---

property : var **is** a variable object
The corresponding variable of the literal.

property : negated **is** Boolean
If the variable in the literal is negated ($\neg x$) this property is true.

Figure 2.4: Representation of a CNF SAT formula for SLS algorithms.

and the other a list of variable objects. The clause object has a property that is the list of literal objects for the clause, where the literal object has a var and a negated property. During execution, an SLS algorithm dynamically assigns Boolean values to the variables, which are stored in the value property of the variable object. The objects and properties in Figure 2.4 represent the minimal set of objects and properties required by an SLS algorithm. In practice, it is convenient for algorithms to define and add their own properties.

Some property values can be calculated and are the result of a sequence of instructions, as demonstrated in Figure 2.5, where we add a satisfied property for clause and formula objects. We use a function-like syntax that returns a value to describe how calculated properties are determined. As can be seen in Figure 2.5, we use the notation object.property to refer to the property of a specific object. We ignore the reference to the formula object where it is obvious (*i.e.*, Clauses[ ] is really formula.Clauses[ ]).

We will now introduce the simplest SLS algorithm for SAT, which we refer to as UNIFORM

Object clause **Property** satisfied **is** Boolean
This property is true if the clause is currently satisfied.

```
foreach literal l in Literals[ ] do
    if l.var.value xor l.negated then return true
end
return false
```

Object formula **Property** satisfied **is** Boolean
This property is true if the formula is currently satisfied.

```
foreach clause c in Clauses[ ] do
    if not c.satisfied then return false
end
return true
```

Figure 2.5: The satisfied **property of** clause **and** formula **objects.**

RANDOM WALK (URWALK), although it is known by other names. Almost all SLS algorithms for SAT (including URWALK) initialize the variables by assigning each variable either true or false uniformly at random. At each search step, the URWALK algorithm simply selects a variable uniformly at random and flips it. To describe URWALK, we use three *procedures*, where a procedure in our syntax is simply a sequence of instructions, a concept similar in nature to subroutines or functions that return no value. In our syntax, we are not concerned with the inputs and outputs of procedures. All objects and properties are globally available to all procedures, and procedures can create global properties to share information with other procedures. The three procedures required to implement the URWALK algorithm are presented in Figure 2.6, along with pseudo-code for URWALK. Note that in this figure, and for the remainder of this dissertation, we have simplified our pseudo-code representation of an algorithm by omitting restarts and early termination criteria.

Because of their stochastic nature, the number of search steps (run-length) required by an SLS algorithm to find a solution of an instance is a random variable with a distribution that is specific to the algorithm and the instance. The URWALK algorithm is straightforward enough that we could construct a mathematical model to predict the run-length distribution, but for the majority of the algorithms in this dissertation, it is infeasible to analytically derive such a model. To determine the run-length distribution of an algorithm on an instance, we need to measure the distribution empirically by performing multiple runs of the algorithm on the instance. In Figure 2.7, we present a Run-Length Distribution (RLD) of 10 000 runs of URWALK on the formula from Equation 2.1. The distribution in Figure 2.7 closely matches an *exponential* distribution, which is common for well-behaved SLS algorithms [55: p. 189]. All empirically measured RLDs are discrete. This is

```
Procedure InitializeVariables
foreach variable v in Variables[ ] do
    v.value := select from {true,false } uniformly at random
end

Procedure PickVariableURWALK
flipVariable := select variable from Variables[ ] uniformly at random

Procedure FlipVariable
flipVariable.value := not flipVariable.value

Algorithm URWALK
Input: formula
Output: solution
InitializeVariables
while not formula.satisfied do
    PickVariableURWALK
    FlipVariable
end
return solution (stored in Variables[ ].value)
```

Figure 2.6: The URWALK algorithm.

especially visible in Figure 2.7 because the instance is very small, it can be solved in very few search steps. In the figure we can observe evidence for the PAC property of the URWALK algorithm, where the probability of success approaches one as time increases.

In general, we measure the *median* of the run-length distribution as the single statistic to measure the performance of the algorithm on the instance. The median run-length corresponds to the $q_{0.50}$ quantile of the distribution, and we occasionally refer to other quantiles. In addition to the median run-length, we frequently report the median run-time, which requires an execution environment for context. In Appendix C, we provide details of all of our experimental execution environments and procedures, and we refer to that appendix when describing experiments.

Throughout this dissertation, we often need to refer to the size of the formula or the time complexity of a procedure with respect to the size of the formula. The symbols we use are listed in Figure 2.8.

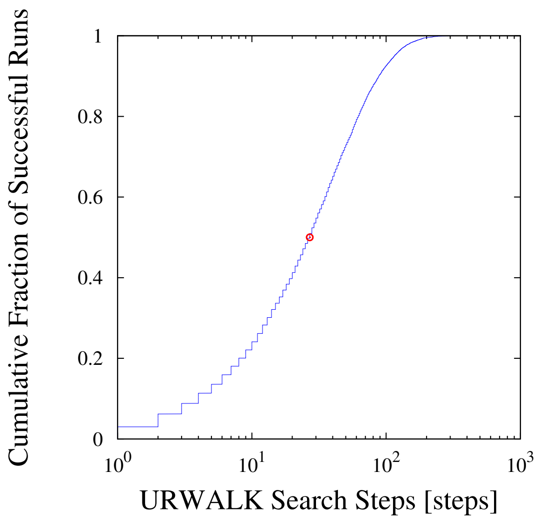


Figure 2.7: Empirical measurement of URWALK performance. Run-length distribution from 10 000 runs of URWALK on the formula from Equation 2.1. The median run-length is highlighted (27 search steps).

$\lvert C\rvert$	number of clauses in a formula
$\lvert C_\top\rvert$	number of currently satisfied clauses
$\lvert C_\bot\rvert$	number of currently unsatisfied clauses
$\lvert c_j\rvert$	clause length: number of literals (variables) in clause c_j
$\lvert V\rvert$	number of variables in a formula
$\lvert V_\bot\rvert$	number of variables that currently appear in false clauses
ncv_i	number of clauses a variable v_i appears in $ncv_i := \lvert\{c_j : v_i \in c_j\}\rvert$
scl_i	sum of the clause lengths for all clauses in which a variable v_i appears $scl_i := \sum_{c_j : v_i \in c_j} \lvert c_j\rvert$
$\lvert F\rvert$	number of literals in a formula (*i.e.*, the size of a the formula)

Figure 2.8: Symbols for representing the size of a SAT formula.

Object variable

---

property: make **is** Integer
Number of clauses that become satisfied if this variable is flipped.

property: break **is** Integer
Number of clauses that become unsatisfied if this variable is flipped.

property: score **is** Integer
*Net change in the intrinsic evaluation function if this variable is flipped (*break - make*).*

Procedure `PickVariableGSAT`

```
flipVariable := select variable from Variables[ ] with minimum score (BTR)
```

Figure 2.9: The score variable property. BTR is short for Breaking Ties Randomly.

2.5 Existing SLS Algorithms for SAT

In this section, we briefly describe some of the more prominent SLS algorithms for SAT we refer to throughout this dissertation. In Appendix A, we provide an index listing all of the algorithms we reference, along with citations and additional implementation details.

The GREEDY SEARCH FOR SAT (GSAT) algorithm by Selman *et al.*, one of the oldest SLS algorithms for SAT, is arguably the most well-known [100]. The GSAT algorithm adds a new score property to the variable objects, which we define in terms of two other variable properties: make and break. We describe the three new properties in Figure 2.9. As we discussed in Section 2.3 with search landscapes, flipping a variable with a negative, zero or positive score corresponds to an improving, sideways or worsening step. Once the score property is introduced, the GSAT algorithm is rather straightforward; at each search step, GSAT simply selects the variable with the minimum score to flip, Breaking Ties Randomly (BTR). In Figure 2.9, we also include the `PickVariableGSAT` procedure. Note that aside from the `PickVariable` procedure, the GSAT algorithm is identical to the URWALK algorithm described in Figure 2.6. Unlike the URWALK algorithm, the GSAT algorithm is not PAC, and can become stuck in regions of the search space known as *traps*. For the simplest example of a trap, consider two local minima in a search space that each have no possible improving steps, and only one sideways step possible that leads to the other location. If the GSAT algorithm were to fall into such a trap, it would simply alternate between the two local minima [49]. For this reason, GSAT requires restarts to solve most interesting instances. The original GSAT algorithm has a *maxFlips* parameter, where periodic restarts occur every *maxFlips* steps.

To address the issue of GSAT's tendency to become stuck in traps, Selman and Kautz developed the GSAT WITH RANDOM WALK (GWSAT) algorithm [98]. The GWSAT algorithm adds a random walk step to GSAT, where for each search step with some probability *wp*, a variable is selected uniformly at random to be flipped, otherwise a regular GSAT search step occurs. The GWSAT algorithm uses a unique approach, where the random walk does not select the variable uniformly at random from all variables, but rather only from those variables that currently appear in false clauses. This method was inspired by the theoretical results of the CONFLICT-DIRECTED RANDOM WALK (CRWALK) algorithm [85].

In the CRWALK algorithm (*a.k.a.* Papadimitriou's Algorithm), at each search step a currently unsatisfied clause is selected uniformly at random, then one of the variables that appears in that clause is selected uniformly at random to be flipped. Papadimitriou demonstrated that CRWALK can solve an arbitrary satisfiable 2-SAT instance in $O(|V|^2)$ steps with a probability arbitrarily close to one [85]. SCHÖNING'S ALGORITHM is very closely related to CRWALK, and adds a periodic restart every $(3 \cdot |V|)$ steps. SCHÖNING'S ALGORITHM was shown to be able to solve an arbitrary satisfiable 3-SAT instance in $O(1.334^{|V|})$ expected steps [94]. Iwama and Tamaki extended SCHÖNING'S ALGORITHM to improve this bound to $O(1.324^{|V|})$ [64].

After GSAT, perhaps the next most well-known SLS algorithm for solving SAT is WALKSAT/SKC, named for its authors: Selman, Kautz and Cohen [99]. WALKSAT/SKC was inspired by the CRWALK algorithm in that it starts each search step by selecting an unsatisfied clause uniformly at random. In general, we refer to this strategy as the WALKSAT strategy, and algorithms that adopt this strategy are in the WALKSAT *family* of algorithms. After the clause is selected, the WALKSAT/SKC algorithm checks if any of the variables in the selected clause have a break property equal to zero; if any such freebie variables exist, one is selected to be flipped uniformly at random. If no freebies exist, then with probability *wp*, one of the variables in the selected clause is flipped uniformly at random. Otherwise, the variable with the lowest break property is flipped, breaking ties uniformly at random.

The NOVELTY algorithm by McAllester *et al.* [80] is from the WALKSAT family. NOVELTY uses the age property of a variable, which we define in Figure 2.11 along with a lastChange property, where the age of a variable is the number of search steps since the variable was last flipped. NOVELTY determines the two variables in the clause with the minimal score as in GSAT, breaking ties in favour of the variables with the larger age. The NOVELTY algorithm is *novel* in its approach when the best variable is also the youngest variable (with the smallest age property), in which case, with a given probability (which we call *noveltyNoise*), it will select the second best variable. The variable selection procedure of NOVELTY is shown in Figure 2.11.

Hoos showed that NOVELTY is not PAC by demonstrating that for some formulae, such as the one in Equation 2.1, under certain initial conditions, NOVELTY will never reach a solution [49]. Hoos introduced the NOVELTY$^+$ algorithm that added a second random walk parameter *wp*, where at each step with probability *wp* a variable from the selected clause is flipped uniformly at random. In practice, the value of *wp* can be very small and a value of 0.01 is common. After demonstrating

Object variable

```
property: freebie is Boolean
          true if (break = 0), otherwise false
```

Procedure `PickVariableSKC`

```
selectedClause := select currently unsatisfied clause uniformly at random
if exists variable in selectedClause.Vars[ ] with freebie = true then
  | flipVariable := select variable from selectedClause.Vars[ ] with freebie = true (BTR)
else
  | with probability wp
  |   | flipVariable := select variable from selectedClause.Vars[ ] uniformly at random
  | otherwise
  |   | flipVariable := select variable from selectedClause.Vars[ ] with min. break (BTR)
  | end
end
```

Figure 2.10: Variable selection in the WALKSAT/SKC algorithm.

Object variable

```
property: lastChange is Integer
          The search step count at which this variable was most recently changed
property: age is Integer
          The current search step counter minus lastChange
```

Procedure `PickVariableNovelty`

```
selectedClause := select currently unsatisfied clause uniformly at random
bestVariable := select variable from selectedClause.Vars[ ] with minimum score (BTA)
secondBestVariable := next best variable (using the same criteria)
youngestAge := minimum age from selectedClause.Vars[ ].age
flipVariable := bestVariable
if bestVariable.age = youngestAge then
  | with probability noveltyNoise
  |   | flipVariable := secondBestVariable
  | end
end
```

Figure 2.11: Variable selection in the NOVELTY algorithm. BTA is short for Breaking Ties by Age (*i.e.*, in favour of the variable with the largest age property, otherwise ties are broken uniformly at random).

that the performance of the NOVELTY$^+$ algorithm was very sensitive to the *noveltyNoise* parameter, Hoos developed the ADAPTIVE NOVELTY$^+$ algorithm that adjusts *noveltyNoise* automatically as the search progresses [51]. This *adaptive* scheme was based on the progress of the algorithm and increased and decreased *noveltyNoise* accordingly (see [51] for further details). Li and Huang introduced the NOVELTY^{++} algorithm where, instead of a walk probability *wp* to flip a random variable, a diversification probability *dp* is used to flip the oldest variable in the selected unsatisfied clause.

The NOVELTY algorithm uses the age property in its variable selection. Another popular method for considering the age of a variable is known as a *tabu* strategy [41]. There is extensive literature on tabu search strategies in other domains [42], but in this dissertation we are only concerned with *simple tabu search*. In simple tabu search, after a variable is flipped it is then considered tabu and cannot be flipped until the age property of the variable is greater than the *tabuTenure*, a parameter of the algorithm. Several SLS algorithms for SAT use a simple tabu strategy, including the WALKSAT/TABU [80] algorithm, which we discuss in Section 3.2.

The GSAT WITH HISTORY (HSAT) algorithm was the first algorithm to use the flips property (*a.k.a.* flipCount) in its variable selection, where the flips property is simply the number of times the variable has been flipped [39]. HSAT is nearly identical to GSAT, but uses the flips property as a tie-breaker. The HSAT WITH RANDOM WALK (HWSAT) algorithm added a random walk mechanism similar to the one we described in NOVELTY$^+$ [40].

The GRADIENT-BASED GREEDY WALKSAT (G^{2}WSAT) algorithm by Li and Huang can be seen as a hybrid between the GSAT and WALKSAT approaches and introduced a new promising variable property [76]. The promising property is Boolean, and is true if the variable has a negative score property, unless the negative score was a result of flipping the variable when it had a non-negative score. If a non-promising variable's score is negative after being flipped, the variable can only become promising if its score becomes non-negative and then negative again as a result of flipping other variables. If promising variables exist, then G^{2}WSAT is greedy like GSAT and selects the promising variable with the lowest score (breaking ties by age), otherwise G^{2}WSAT behaves like a WALKSAT algorithm. In the G^{2}WSAT algorithm, the WALKSAT algorithm selected is NOVELTY^{++}, although there are several different G^{2}WSAT variants that use different WALKSAT algorithms.

We must note that some of the SLS algorithms for SAT we have introduced in this section (*e.g.*, GSAT and CRWALK) have very poor performance in practice and were introduced to provide background information. Of the algorithms we introduced, the two most prominent state-of-the-art algorithms are ADAPTIVE NOVELTY$^+$ and G^{2}WSAT.

2.6 Dynamic Local Search for SAT

Before we discuss Dynamic Local Search (DLS) algorithms in general, we first describe a class of DLS algorithms we refer to as Dynamic Local Search with Clause Penalties (DLS-CP), which will

Object clause

---

property : penalty **is** Real
Dynamic penalty (or weight) assigned to this clause.

Object variable

---

property : penMake **is** Real
Sum of the clause penalty *values for clauses that become satisfied if this variable is flipped.*

property : penBreak **is** Real
Sum of the clause penalty *values for clauses that become unsatisfied if this variable is flipped.*

property : penScore **is** Real
*Net change in the penalized evaluation function if this variable is flipped (*penBreak - penMake*).*

Figure 2.12: The clause penalty **property and corresponding variable scoring properties.**

require us to explain *clause penalties*.

The clause penalties we describe in this dissertation are more often known as clause *weights* in the literature (including some of our own publications [61]). However, we use the term penalties to avoid any confusion with the clause weights in the weighted MAX-SAT problem, the weighted instances we study in Section 5.2 and the weighted algorithms we describe in Section 3.6 and Section 5.1.

In our parlance, each clause simply has a penalty property that is assigned a value that can change dynamically during the search. In many SLS algorithms for SAT without clause penalties, such as GSAT, the intrinsic evaluation function is used, and the variable score property corresponds to the change in this evaluation function that would occur if the variable were flipped. Typical DLS-CP algorithms use a *penalized* evaluation function, which is the sum of the clause penalty values over all unsatisfied clauses for a given variable assignment. Accordingly, a variable property named penScore measures the change in this penalized evaluation function, and is described in Figure 2.12. There are some subtle differences in the ways DLS-CP algorithms use their penalty properties, which we explore in more detail in Chapter 4.

As an example of how DLS-CP algorithms operate, we describe the BREAKOUT algorithm, one of the earliest and most influential DLS-CP algorithms [83]. The variable selection in BREAKOUT is very similar to the variable selection in GSAT with two notable differences. The first difference is that instead of selecting the variable with the best score property, it selects the variable with the best penScore property that optimizes the aforementioned penalized evaluation function. The second difference is that BREAKOUT only allows improving steps, and if no improving step is possible (*i.e.*, the algorithm is in a local minimum), the clause penalty values are *updated* until the algorithm is no longer in a local minimum. In other words, the algorithm can *break out* of a local minimum. As with most DLS-CP algorithms, the BREAKOUT `InitializeClausePenalties` procedure initializes all penalty values to one. As we will demonstrate in Chapter 4, often the largest difference between DLS-CP algorithms is how they dynamically update their clause penalty values. In the case of BREAKOUT, the `UpdateClausePenalties` procedure increments the clause penalty values of all *unsatisfied* clauses. The effect of this increase is that all the penMake values are increased, whereas the penBreak values remain the same, so eventually the penScore property (equivalent to penBreak minus penMake) will become negative for at least one variable, resulting in a possible improving step. We refer to this general approach, where clause penalties are updated at a local minimum, as a BREAKOUT strategy. We describe the BREAKOUT algorithm in Figure 2.13. We must note that the BREAKOUT algorithm is not considered a state-of-the-art DLS-CP algorithm, and we will discuss more prominent DLS-CP algorithms, including our SCALING AND PROBABILISTIC SMOOTHING (SAPS) algorithm in Chapter 4.

When we first introduced the term Dynamic Local Search (DLS) to the SAT community in 2002 with our SAPS algorithm, we were exclusively referring to DLS-CP algorithms, and the way they *dynamically* changed their evaluation function via clause penalties. As our work progressed, we embraced a more general definition of DLS that includes all SLS algorithms for SAT that incorporate history into the search. For example, GSAT and WALKSAT/SKC are non-dynamic or *stationary* algorithms, in that their behaviour at some search step j is only depends on the current variable assignment, and is completely independent of what occurred in steps $1\ldots(j-1)$. Conversely, the NOVELTY algorithm uses the age property, and therefore its behaviour depends on the actions taken in previous steps.

The concepts of history and memory are closely related. Fundamentally, memory is required to record history, so DLS algorithms require additional space to store the historical information. DLS algorithms do not typically record their entire search history, and instead capture only certain aspects of the search (*e.g.*, the age property). In the SLS literature, the *duration* or scope of historical information is also expressed in terms of memory, using memory in the more traditional use of the word [25, 41, 108]. For example, the age property is a *short-term memory* property because for each variable it only records information regarding the most recent flip. The flips property is a *long-term memory* property because it records the total number of times the variable has been flipped (*i.e.*, since the beginning of the search).

Throughout this dissertation we explore DLS in many different ways, and we return to this discussion in Section 8.2.

Procedure `InitializeClausePenalties`

```
foreach clause c in Clauses[ ] do
    c.penalty := 1
end
```

Procedure `PickVariableBreakout`

```
bestPenScore := minimum penScore from Variables[ ].penScore
if bestPenScore $< 0$ then
    flipVariable := select variable from Variables[ ] with minimum penScore (BTR)
else
    flipVariable := null
end
```

Procedure `UpdateClausePenaltiesBreakout`

```
foreach clause c in Clauses[ ] do
    if not c.satisfied then c.penalty := c.penalty + 1
end
```

Algorithm BREAKOUT

Input: formula
Output: solution

```
InitializeVariables
InitializeClausePenalties
while not formula.satisfied do
    PickVariableBreakout
    if flipVariable is null then
        UpdateClausePenaltiesBreakout
    else
        FlipVariable
    end
end
return solution (stored in Variables[ ].value)
```

Figure 2.13: The BREAKOUT algorithm.

Chapter 3

UBCSAT

Work it harder... make it better... do it faster... makes us stronger.
More than ever... hour after... our work is... never over.
— Daft Punk. "Harder, Better, Faster, Stronger"

In this chapter, we lay the cornerstone of our dissertation by introducing UBCSAT, a software framework for efficiently implementing and evaluating SLS algorithms for SAT. We started the UBCSAT project with the following six design principles and goals:

1. include highly efficient, conceptually simple and accurate implementations of a wide range of prominent SLS algorithms for SAT and MAX-SAT;

2. facilitate the development and integration of new algorithms (and algorithm variants);

3. provide support for advanced empirical analysis of the performance and behaviour of SLS algorithms without compromising implementation efficiency;

4. provide explicit support for algorithms designed to solve the weighted and unweighted MAX-SAT problems;

5. provide an open-source software package that is publicly available to the academic community; and

6. implement the project in a platform-independent way, avoiding non-standard programming language extensions.

The remainder of this chapter is structured as follows. First, in Section 3.1, we provide some time complexity analysis and design considerations for implementing SLS algorithms for SAT. Second, in Section 3.2, we give an overview of the UBCSAT architecture and illustrate the fundamental

Algorithm URWALK

```
Input: formula
Output: solution
InitializeVariables                                   $\Theta(|V|)$
while not formula.satisfied † do
    SelectRandomVariable                              $\Theta(1)$
    FlipVariable                                      $\Theta(1)$
end
return solution (stored in Variables[ ].value)
```

† formula.satisfied, when implemented as described in Figure 2.5, completes in $\Theta(|F|)$ time in the worst case.

Figure 3.1: The URWALK algorithm with time complexity. For this implementation of URWALK, the worst-case time to complete a search step is $\Theta(|F|)$, because of the time required to determine if formula.satisfied is true in the while statement above.

concept of *triggered procedures*, which lies at the core of UBCSAT's efficient, yet highly flexible, design and implementation. Next, in Section 3.3, we outline the collection of SLS algorithms for SAT that are currently implemented within UBCSAT and compare their performance against that of the respective native reference implementations. Then, in Section 3.4, we demonstrate how new algorithms are implemented within UBCSAT. Next, in Section 3.5, we discuss the importance of empirical analysis in SLS research, and how UBCSAT can help facilitate empirical analysis. In Section 3.6, we describe how UBCSAT supports SLS algorithms for weighted and unweighted MAX-SAT. In Section 3.7, we describe related work. Finally, in Section 3.8, we summarize our contributions.

3.1 Implementing SLS Algorithms Efficiently

Before we introduce UBCSAT, we explore some of the time complexity and practicality issues involved with implementing an SLS algorithm for SAT. In Figure 3.1, we reproduce the URWALK algorithm described in Figure 2.6 with additional time complexity notations added. The `InitializeVariables` procedure completes in $\Theta(|V|)$ time, but in general we are not concerned with initialization procedures and only focus on the procedures that occur in every search step. The other two procedures in URWALK (`SelectRandomVariable` and `FlipVariable`) are both straightforward and can be completed in $\Theta(1)$ time. In this example, the operation in each search step that requires the most amount of time is determining the satisfied property of the formula object, which, when implemented as described in Figure 2.5, completes in $\Theta(|F|)$ time in the worst case.

```
Object formula Property FalseClauses[ ] is list of clause objects        $\Theta(|F|)$
List of all clauses that are currently unsatisfied
-------------------------------------------------------------------------------------
FalseClauses[ ] := empty                                                  $\Theta(1)$
foreach clause c in Clauses[ ] do
    if not c.satisfied then                                               $\Theta(|c|)$
        add c to FalseClauses[ ]                                          $\Theta(1)$
    end
end
return FalseClauses[ ]
```

Figure 3.2: The FalseClauses[ ] property of the formula object.

To demonstrate how the implementation and data structures affect the run-time performance of an SLS algorithm, we improve the URWALK by adding a new FalseClauses[ ] property to the formula object. We describe the new property in Figure 3.2. We assume that the operations required to add to, remove from, determine the length of, and check membership in a list can all be completed in $\Theta(1)$ (constant) time. One advantage of adding the FalseClauses[ ] property to the formula object is that to determine if the formula is satisfied, we can simply determine if the FalseClauses[ ] list is empty. Unfortunately, this advantage is not realized if we consider that the worst-case time to determine the FalseClauses[ ] property, as implemented in Figure 3.2, is also $\Theta(|F|)$.

To improve upon the time complexity of URWALK, we use an approach we generally refer to as *bookkeeping* of *state information*, *a.k.a.* incremental updates or delta evaluations [55: p. 48], where we maintain in memory important information about the state of the algorithm. To use bookkeeping on the FalseClauses[ ] property, we require two procedures: one to *initialize* the list of false clauses, and one to *update* the list when necessary, which for this example is whenever a variable is flipped. We use procedures to implement bookkeeping in our syntax, and in Figure 3.3 we provide the two procedures `InitializeFalseClauses` and `UpdateFalseClauses` necessary for this example.

In Figure 3.4 we present the new URWALK algorithm variant with bookkeeping. The worst-case time to complete a search step (flipping variable v_i) has improved from $\Theta(|F|)$ (as implemented in Figure 3.1) to $\Theta(scl_i)$ (as implemented in Figure 3.4), where scl_i is the sum of the clause lengths for all clauses in which variable v_i appears. This change is an improvement since typically variables tend to occur in relatively small fractions of the clauses of a given formula, *i.e.*, $scl_i \ll |F|$. We provided this example to illustrate how bookkeeping can affect the run-time performance of an algorithm in practice, without changing the step behaviour of the original algorithm. This example is also interesting because there is nothing in the URWALK algorithm to suggest that a list of false clauses would be useful, unlike the CRWALK and WALKSAT algorithms where a list is required for the algorithm.

```
Procedure InitializeFalseClauses                                      $\Theta(|F|)$
To occur after the variables have been initialized
-----------------------------------------------------------------------------------
FalseClauses[ ] := empty                                              $\Theta(1)$
foreach clause c in Clauses[ ] do
    if not c.satisfied then                                           $\Theta(|c|)$
        add c to FalseClauses[ ]                                      $\Theta(1)$
    end
end
```

```
Procedure UpdateFalseClauses                                          $\Theta(scl_i)$
To occur after a variable $v_i$ has been flipped
-----------------------------------------------------------------------------------
foreach clause c in flipVariable.AppearClauses[ ] do
    if c.satisfied then                                               $\Theta(|c|)$
        if c in FalseClauses[ ] then remove c from FalseClauses[ ]    $\Theta(1)$
    else
        if c not in FalseClauses[ ] then add c to FalseClauses[ ]     $\Theta(1)$
    end
end
```

Figure 3.3: Bookkeeping for the FalseClauses[ ] property. To maintain the FalseClauses[ ] property of the formula object we require two additional procedures. The variable property AppearClauses[ ] is a list of clause objects that the variable appears in. For `UpdateFalseClauses`, we provide the worst-case time of $\Theta(scl_i)$ for a given flip variable v_i, where scl_i is the sum of the clause lengths for all clauses in which variable v_i appears. For a variable that appears in every clause, scl_i is equivalent to $|F|$.

We note that in practice, URWALK can be optimized even further with bookkeeping procedures that maintain a numFalseClauses property of the formula object, and a numTrueLiterals property for each clause. Such an implementation of URWALK completes a search step (flipping variable v_i) in $\Theta(ncv_i)$ time, where ncv_i is the number of clauses that variable v_i appears in.

The bookkeeping approach we have described in Figure 3.3 is a straightforward example and may suggest that bookkeeping can always improve performance, but special care must be taken when implementing algorithms to test the *empirical* results. To demonstrate this, we briefly consider the bookkeeping required to maintain the make and break properties in a typical implementation of an SLS algorithm. For example, after a variable is flipped, for each clause that has become unsatisfied as a result of the flip, the make property is incremented for each of the variables in the selected clause. There are numerous additional changes required, and the time to update the make

```
Algorithm URWALK
Input: formula
Output: solution
InitializeVariables                          $\Theta(|V|)$
InitializeFalseClauses                       $\Theta(|F|)$
while not FalseClauses[ ] is empty do
    SelectRandomVariable                     $\Theta(1)$
    FlipVariable                             $\Theta(1)$
    UpdateFalseClauses                       $\Theta(scl_i)$
end
return solution (stored in Variables[ ].value)
```

Figure 3.4: The URWALK algorithm variant with the FalseClauses[ ] property. For this implementation of URWALK, the worst-case time to complete a search step flipping a variable v_i is $\Theta(scl_i)$. In this example, we replaced the conditional in the while statement from (**not** satisfied) to (**not** FalseClauses[ ] **is empty**) to highlight the difference.

and break properties (after flipping variable v_i) is:

$$\Theta(scl_i) = \Theta\left(\sum_{c_j:v_i\in c_j} |c_j|\right). \tag{3.1}$$

As an alternative to maintaining the bookkeeping for the make and break properties, they can be calculated when needed. The time to calculate the make and break properties of a variable v_i in a standard SLS implementation (*i.e.*, the numTrueLiterals property is available) is $\Theta(ncv_i)$. In a WALKSAT algorithm that uses both make and break (*e.g.*, NOVELTY), the properties must be calculated for each of the variables that appear in the selected clause. Therefore, the time to complete a search step where the properties are calculated for each variable (v_i) in the selected clause (c_j) is:

$$\Theta\left(\sum_{v_i:v_i\in c_j} ncv_i\right). \tag{3.2}$$

For typical variables and an average clause length, we would expect Equation 3.1 to be the same as Equation 3.2, and for an instance with a fixed clause length and fixed distribution of variables they are identical. From this evidence it would appear that there is no difference between the bookkeeping variant of NOVELTY and the non-bookkeeping variant. However, in our experiments we found that NOVELTY was significantly faster on 3-SAT without bookkeeping (see also, [55: p. 273]). Fukunaga discovered the same result independently in his experimentation [33]. There are other examples where the best bookkeeping option to be used on an instance depends on the structure of the instance.

One of the goals of this section was to identify the importance of implementing bookkeeping strategies and the requirement to test them empirically. The UBCSAT framework we describe in the following section has been specifically designed to support multiple methods of bookkeeping for empirical testing.

3.2 The UBCSAT Architecture

One of the challenges of developing the UBCSAT project was to build a flexible, feature-rich environment without compromising algorithmic efficiency. To achieve our goals, UBCSAT has been designed according to what we have named a *triggered procedure architecture*. The main ideas underlying this architecture are closely related to certain concepts from object- and event-oriented programming.

The UBCSAT software is structured around a set of *event points* that occur throughout the execution of an SLS algorithm for SAT. For each event point, a list of procedures is maintained that are executed whenever the event point is reached; this list is called the *triggered procedure list* of the event point and its elements are called the *triggered procedures*. We present the straightforward eventPoint object in Figure 3.5. A *trigger* is simply a mapping of a software procedure to an event point. When a trigger is *activated*, its associated procedure is added to the triggered procedure list of the corresponding event point.

Initially, the triggered procedure lists for all of the event points are empty. It is only when triggers are activated that procedures become executed when an event point is reached. For example, you may have an elaborate procedure for displaying the real-time status of an algorithm as it searches. You can create a trigger that maps your procedure to update the display at an appropriate event point, perhaps at the end of each search step. Whenever you want to have the real-time status displayed you can activate your trigger, which will execute your procedure at the end of each search step. However, if you do not wish to have the status displayed then you do not have to do anything; your trigger will not be activated, no procedure will be added to a triggered procedure list and your algorithm will not be slowed down by your status display procedure.

In addition to its associated procedure, event point and activation status, a trigger can have a *dependency list* and a *deactivation list*, which are lists of other triggers that are activated or deactivated (respectively) when the trigger is activated. The dependency list is used, for example, to ensure that when the procedure of a trigger relies on the existence of some data structures, the triggers for the procedures that create and update those data structures are also activated. In addition, the dependency list ensures that any triggers that occur at the same event point are executed in the correct order. The deactivation list is intended for advanced UBCSAT users, and can be used to override default routines and avoid conflicts between incompatible routines. In practice, deactivation lists are used in UBCSAT to improve implementation efficiency by combining the

Object eventPoint
Execution Point in an SLS Algorithm

---

property: Procedures[ ] **is** list of `Procedures`
A list of procedures that are to be executed at the event point.

Object trigger
A mapping of a `procedure` *to an event point*

---

property: proc **is** `procedure`
Instructions for the trigger.

property: event **is** eventPoint object
The event point where the procedure is to be executed.

property: activated **is** Boolean
Activation status of the trigger.

property: DependencyTriggers[ ] **is** list of trigger objects
Additional triggers to be activated if this trigger is activated.

property: DeactivationTriggers[ ] **is** list of trigger objects
Triggers to be deactivated if this trigger is activated.

Object containerTrigger
A trigger *object that is a collection of other* trigger *objects , grouped together for convenience.*

---

property: SubTriggers[ ] **is** list of trigger objects
Collection of trigger objects to be activated if this trigger is activated.

Figure 3.5: The UBCSAT eventPoint **and** trigger **objects.**

functionality of multiple procedures. For example, consider triggers *a* and *b* that have procedures `UpdateA` and `UpdateB`, but when both triggers are activated it would be significantly more efficient if the functionality of procedures `UpdateA` and `UpdateB` were combined into one procedure. In this case, a new procedure `UpdateAB` could be created and assigned to a trigger *ab* which would include *a* and *b* in its deactivation list and be available to algorithms that require the functionality of both *a* and *b*. UBCSAT detects and produces a warning if deactivated triggers are somehow reactivated, which might indicate a flaw in the design of an SLS algorithm that is being developed within the UBCSAT framework.

There is also a special type of trigger called a *container trigger* that has no associated procedure. Instead, it has a list of secondary triggers that are activated whenever the container trigger is activated. Container triggers are used as convenient shortcuts for activating groups of triggers that are used simultaneously. Conceptually, container triggers are very similar to dependency lists; by activating one trigger several others are also activated. While dependency lists are an important part of ensuring the triggered procedure architecture works properly, container triggers simply provide shortcuts for added convenience. A description of the two types of trigger objects is provided in Figure 3.5.

In the previous section, we described an efficient bookkeeping mechanism for maintaining the FalseClauses[ ] property that required an `InitializeFalseClauses` and an `UpdateFalseClauses` procedure. We have not yet mentioned an additional procedure, `CreateFalseClauses`, required in practice to allocate the memory for the property. In UBCSAT, many properties require these three types of triggers to operate properly: one to create the data structure, one to initialize it and one to update it. A container trigger is typically created to activate all three of these triggers simultaneously.

UBCSAT has hundreds of triggers, most of which have associated procedures that fall into one of the following four categories: heuristic selection (*e.g.*, of variables), property state information bookkeeping, report and statistic data collection, and file I/O. Triggers are activated based on the SLS algorithm to be run, the reports or statistics requested and other command line parameters. In the UBCSAT implementation, the triggered procedure lists are simply arrays of function pointers. Therefore, when each event point is reached, it is very efficient to call its triggered procedures.

Figure 3.6 shows a pseudo-code representation of UBCSAT that indicates the event points in UBCSAT version 1.0. Returning to the example of the FalseClauses[ ] property, the three procedures (`CreateFalseClauses`, `InitializeFalseClauses`, `UpdateFalseClauses`) would each have a corresponding trigger that would map to the event points (*CreateStateInfo*, *InitStateInfo*, *UpdateStateInfo*), respectively. For convenience, all three triggers would be contained in a *FalseClauseList* container trigger. The following example further illustrates the use of event points and the concept of triggered procedures.

Let us consider WALKSAT/TABU, a well-known high-performance SLS algorithm for SAT that is based on the WALKSAT architecture [80]. As in most WALKSAT-based algorithms, WALKSAT/TABU starts each search step by uniformly selecting a clause from the set of currently unsatisfied clauses (FalseClauses[ ]). The variable in the selected clause with the best score property that is not tabu is selected as the flip variable (breaking ties randomly). A variable's tabu property is true if it has been flipped within the last *tabuTenure* search steps, where *tabuTenure* is a parameter of the WALKSAT/TABU algorithm. If all of the variables in the selected clause are tabu, then no flip occurs at that step.

In the UBCSAT implementation of WALKSAT/TABU, the main heuristic procedure is `PickWalksatTabu`, and a trigger of the same name exists which maps the procedure to the *ChooseCandidate* event point. Most algorithms in UBCSAT also activate the *DefaultProcedures* trigger, a container trigger that includes triggers for handling common tasks, such as keeping track of the current

UBCSAT

```
Setup UBCSAT
Parse Parameters
Activate Algorithm Triggers
Activate Report Triggers
run procedures PostParameters
run procedures ReadInInstance
run procedures CreateData
run procedures CreateStateInfo
run procedures PreStart
while run < numRuns do
    while (step < cutoff) and (not solutionFound) and (not terminateRun) do
        run procedures PreStep
        run procedures CheckRestart
        if (step = 1) or (restart) then
            run procedures InitData
            run procedures InitStateInfo
            run procedures PostInit
        else
            run procedures ChooseCandidate
            run procedures PreFlip
            run procedures FlipCandidate
            run procedures UpdateStateInfo
            run procedures PostFlip
        end
        run procedures PostStep
        run procedures StepCalculations
        run procedures CheckTerminate
    end
    run procedures RunCalculations
    run procedures PostRun
end
run procedures FinalCalculations
run procedures FinalReports
```

Figure 3.6: An Overview of UBCSAT.

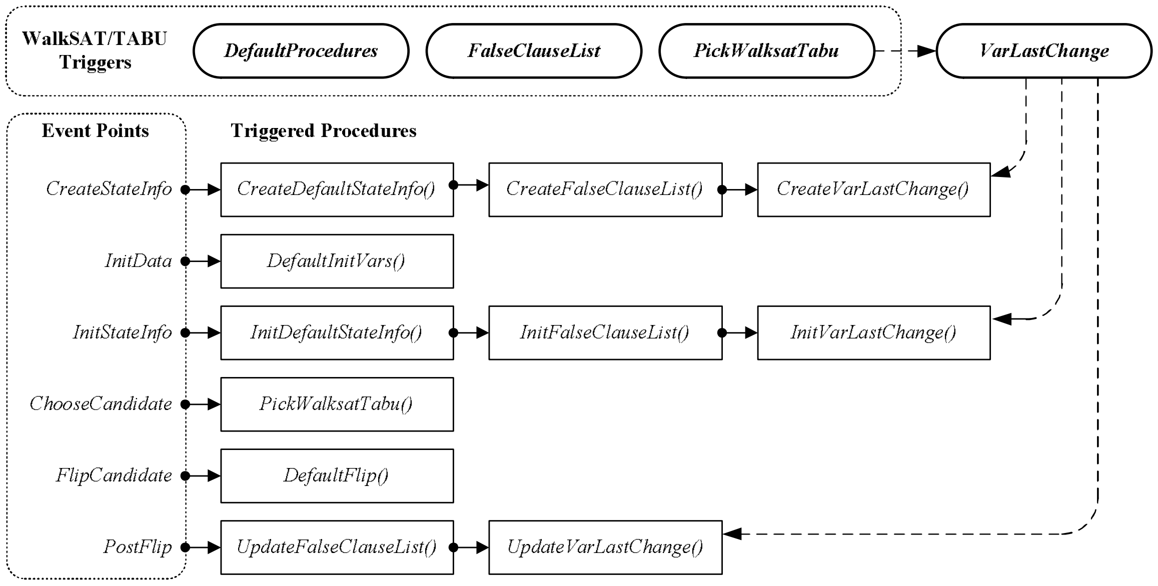


Figure 3.7: The WALKSAT/TABU algorithm in UBCSAT. The WALKSAT/TABU algorithm and the triggered procedures that appear in the event point triggered procedure lists. The dashed arrows illustrate how the *VarLastChange* procedures were added to the triggered procedure lists by the activation of the *PickWalksatTabu* trigger. Note that some procedures and event points are not listed, including a few additional procedures triggered by *DefaultProcedures*.

truth assignment and reading the formula into memory. Efficient implementations of WALKSAT-based algorithms require the FalseClauses[ ] property to be available and, as mentioned previously, a *FalseClauseList* container trigger will activate the three triggers required to create, initialize and update the data structure.

The WALKSAT/TABU algorithm needs to keep track of when each variable was last flipped to determine its tabu status. This requires adding a lastChange property to the variable object that is the search step of the last time the variable changed (see Figure 2.11). Like FalseClauses[ ], lastChange requires three procedures with three corresponding triggers which can be grouped into one container trigger we call *VarLastChange* (see Figure 3.7).

The primary advantage of the triggered procedure architecture lies in the fact that of the many procedures needed to realize the many SLS algorithms and report formats supported by UBCSAT, only those required in any given run are activated and used. The remaining inactive or non-triggered procedures do not affect UBCSAT's performance. A secondary advantage is that different algorithms and reports can share the same data structures and procedures, saving much programming effort. Potential drawbacks stem from the implementation overhead of registering all triggers, and in this framework, algorithms are typically split into many rather small procedures, which can lead to decreased performance compared to more monolithic implementations. However, we have found that these disadvantages are far outweighed by the advantages of UBCSAT's triggered procedure architecture. As we will demonstrate in the following section, the performance of UBCSAT is very competitive with native reference implementations of the respective SAT algorithms.

To achieve our goals of a platform-independent and highly efficient implementation, UBCSAT has been developed in strict ANSI C and tested on some of the most popular operating systems (Linux, Windows, SunOS, Mac OS X). We have incorporated the *Mersenne Twister* Pseudo-Random Number Generator (PRNG) [78] to provide a state-of-the-art and platform-independent source of pseudo-random numbers. UBCSAT is publicly available for academic (non-commercial) use without restriction to encourage free and open use throughout the SAT research community.

3.3 A Collection of Efficient Algorithm Implementations

One of the goals of UBCSAT is to be a large reference collection of SLS algorithms. By integrating algorithms into the UBCSAT framework, several advantages can be realized when compared to the reference native implementations. First, by using a single executable with a uniform interface, working with different algorithms becomes easier and more convenient. From an implementation point of view, different algorithms share common data structures and procedures, which reduces implementation effort and the likelihood of programming errors. Finally, from an empirical algorithmics point of view, comparing two algorithms is facilitated by the fact that UBCSAT allows fairer comparisons between algorithms that share components and use the same statistical calculations, input and output formats.

The UBCSAT software package version 1.0 implemented the following SLS algorithms for SAT:

- ADAPTIVE NOVELTY$^+$ [51]
- DERANDOMIZED SAPS (SAPS/NR) [114]
- GSAT WITH HISTORY (HSAT) [39]
- HSAT WITH RANDOM WALK (HWSAT) [40]
- ITERATED ROTS (IROTS) [105]
- GREEDY SEARCH FOR SAT (GSAT) [100]
- GSAT WITH RANDOM WALK (GWSAT) [98]
- GSAT WITH TABU (GSAT/TABU) [79]
- NOVELTY [80]
- NOVELTY$^+$ [49]
- R-NOVELTY [80]
- R-NOVELTY$^+$ [49]

- REACTIVE SAPS (RSAPS) [61]
- ROBUST TABU SEARCH (RoTS) [107]
- SCALING AND PROBABILISTIC SMOOTHING (SAPS) [61]
- STEEPEST ASCENT MILDEST DESCENT (SAMD) [44]
- WALKSAT/SKC [99]
- WALKSAT/TABU [80]

Version 1.1, released in 2008, additionally supports the following algorithms:

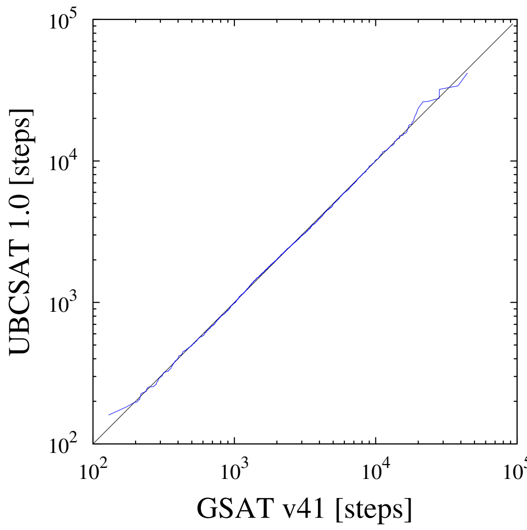
- ADAPTIVE G^{2}WSAT [77]
- ADAPTIVE G^{2}WSAT+P [77]
- CONFLICT-DIRECTED RANDOM WALK (CRWALK) [85]
- DETERMINISTIC ADAPTIVE NOVELTY$^+$ [114]
- DETERMINISTIC CRWALK (DCRWALK) [114]
- DIVIDE AND DISTRIBUTE FIXED WEIGHTS (DDFW) [63]
- GRADIENT-BASED GREEDY WALKSAT (G^{2}WSAT) [76]
- NOVELTY^{++} [76]
- NOVELTY$^+$P [77]
- PURE ADDITIVE WEIGHTING SCHEME (PAWS) [111]
- RESTARTING GSAT (RGSAT) [114]
- UNIFORM RANDOM WALK (URWALK)
- VARIABLE WEIGHTING SCHEME I (VW1) [88]
- VARIABLE WEIGHTING SCHEME II (VW2) [88]

Several of these algorithms have more than one variant (*e.g.*, a variant that uses an alternative book-keeping mechanism).

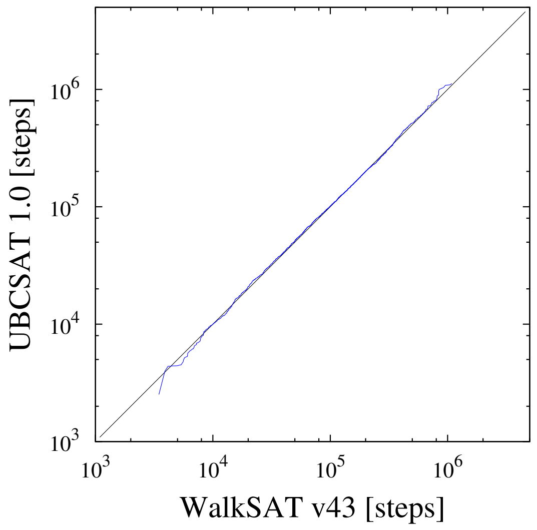
UBCSAT is designed to support weighted MAX-SAT variants (see also Section 3.6) as well as variants that may differ in their behaviour or implementation from the basic version of a given algorithm. Consequently, each algorithm within UBCSAT is identified as a triple (algorithm, variant, weighted) where the variant name can be empty and the weighted option is a Boolean option that has a default value of false. An algorithm is specified in UBCSAT on the command line as:

```
ubcsat -alg algorithm [-v variant] [-w].
```

For each of the previously listed algorithms, we ensured that the UBCSAT implementation behaves identically to the respective original reference implementation, taking into consideration the stochastic nature of the algorithms. This is illustrated in Figure 3.8, in which run-time distributions for the UBCSAT implementations of GWSAT and WALKSAT/SKC are compared to the original GSAT (version 41) and WALKSAT (version 43) implementations.



(a) GWSAT



(b) WALKSAT/SKC

Figure 3.8: Comparison of behaviour for original and UBCSAT implementations. Figures are quantile-quantile plots of the run-length distributions for 5000 runs. In (a), the algorithm is GWSAT and the instance is uf200-easy. In (b), the algorithm is WALKSAT/SKC and the instance is bw-large-a. All algorithms were executed with default parameters as described in Appendix A.

At the same time, the UBCSAT versions of all algorithms were optimized for efficiency, with the goal of matching or exceeding the run-time performance of the respective reference implementations. As we discussed in Section 3.1, we have implemented numerous bookkeeping mechanisms for a wide variety of properties. We strove to use data structures and incremental updating schemes that are efficient, yet reasonably straightforward to understand and implement. As we demonstrated with the two URWALK algorithm variants, the UBCSAT architecture supports functionally identical algorithm variants implemented using different bookkeeping methods, which makes it easy to implement new developments in this area.

The performance of the UBCSAT implementations of all supported algorithms in version 1.0 were tested against that of the respective reference implementations to ensure that the run-times of the former are at least as efficient as the latter. More importantly, for GSAT and WALKSAT algorithms, the UBCSAT implementations have been shown to be significantly faster (see Table 3.1 for representative results).

Algorithm	uuf100-01			uuf400-01		
	time [seconds]		*s.f.*	time [seconds]		*s.f.*
	Original	UBCSAT		Original	UBCSAT	
WALKSAT/SKC	144.7	97.7	**1.48**	150.3	98.5	**1.53**
NOVELTY	151.6	117.1	**1.29**	153.4	114.5	**1.34**
GSAT	305.0	106.7	**2.86**	316.5	114.1	**2.77**
GWSAT	590.1	172.1	**3.43**	768.2	266.8	**2.88**

Algorithm	jnh202			rg-200-2000-4-11		
	time [seconds]		*s.f.*	time [seconds]		*s.f.*
	Original	UBCSAT		Original	UBCSAT	
WALKSAT/SKC	217.2	134.0	**1.62**	310.7	142.1	**2.19**
NOVELTY	230.8	168.4	**1.37**	323.0	159.5	**2.02**
GSAT	1541.6	202.3	**7.62**	397.8	233.0	**1.71**
GWSAT	1894.7	254.3	**7.45**	1354.5	541.5	**2.50**

Table 3.1: Original implementations *vs* UBCSAT implementations. Total run-times are given in seconds for 100 000 000 search steps on *unsatisfiable* instances. Note by choosing unsatisfiable instances for this speed comparison we ensured that in all cases exactly the same number of search steps have been performed. The speedup factor (*s.f.*) shows the software speedups of the UBCSAT implementation over the original implementations (GSAT version 41 and WALKSAT version 43). See Appendix B for instance information. Execution environment: UBC BETA cluster (Section C.2). Unless otherwise noted, all algorithms executed with default parameters as described in Appendix A.

3.4 A Framework for Developing New Algorithms

As discussed in the previous section, the UBCSAT environment includes a wide variety of algorithms and data structures. To facilitate the development and integration of new SLS algorithms, UBCSAT has been designed so that new algorithms can easily re-use the existing procedures and data structures from other algorithms. The basis for this is provided by the triggered procedure architecture discussed in Section 3.2.

To illustrate how new algorithms are added to UBCSAT, we present the pseudo-code required to add a new WALKSAT/TABU algorithm variant to UBCSAT in Figure 3.9. We have named the new variant WALKSAT/TABU-NONULL, and it differs from the regular WALKSAT/TABU algorithm in only one detail: if all of the variables in the selected clause are tabu, then a variable will be selected from the clause at random and flipped. This variant is interesting from a practical point of view, since WALKSAT/TABU is one of the best-performing WALKSAT algorithms, but often suffers from search stagnation as a consequence of null-flips.

Within UBCSAT, the original WALKSAT/TABU algorithm is identified as the triple:

```
("walksat-tabu", "", false),
```

and our new algorithm variant will be identified as a triple:

```
("walksat-tabu", "nonull", false);
```

it differs from the already supported WALKSAT/TABU only in its variable selection procedure, which has a trigger we name *PickWalksatTabuNoNull*. An algorithm can explicitly specify the data structure procedures required, or it can *inherit* them from another algorithm. In this case, we simply inherit everything from regular WALKSAT/TABU. When an algorithm requires algorithm-specific command-line parameters (such as the *tabuTenure* parameter in WALKSAT/TABU) they must be defined or optionally inherited from an existing algorithm. In addition to creating and registering the new trigger in the system, its associated procedure (here also called `PickWalksatTabuNoNull`) has to be implemented, which in this example simply calls the regular WALKSAT/TABU variable selection procedure and then handles the special case when a null-flip occurs. While this example illustrates a particularly simple variant of an existing algorithm, the process of adding implementations of new SLS algorithms to UBCSAT is typically similarly straightforward.

3.5 An Empirical Analysis Tool

Empirical analysis plays an important role in the development and successful application of SAT algorithms. To characterize or measure the behaviour of an SLS algorithm, data typically needs to be collected from multiple independent runs of the algorithm. We note that each run corresponds to a complete execution of an SLS algorithm, as outlined in Figure 2.1, whereas the pseudo-code of Figure 3.6 performs multiple runs. As an example, consider the run-time data shown in Figure 3.8, which is based on 5 000 independent runs of each algorithm involved in the respective experiment. To facilitate the advanced empirical analysis of the SLS algorithms it implements, UBCSAT provides support for measuring and reporting basic descriptive statistics over multiple runs and strongly supports the analysis of Run-Time Distributions (RTDs) and Run-Length Distributions (RLDs) [55: p. 159]. In particular, UBCSAT can measure and report RTDs and RLDs in a format that can be easily plotted (see Figure 2.7) and be further analyzed with specialized statistical software.

Reports currently implemented in UBCSAT include the satisfying assignments found in each run, detailed information about the search state at each search step, flip statistics for individual variables and many others. In UBCSAT, statistics are special objects that are used to collect and summarize data for the default reports. Statistics can be shown for each individual run, or summarized

```
Procedure AddWalksatTabuNoNull
CreateAlgorithm("walksat-tabu"; "nonull", false                      /* new algorithm triple */
   "WALKSAT/TABU without null-flips",                                       /* description */
   "McAllester, Selman, Kautz [AAAI 97] (modified)",                            /* authors */
   PickWalksatTabuNoNull,                                             /* heuristic trigger(s) */
   ...)                                                              /* minor details omitted */
InheritDataTriggers("walksat-tabu", "", false)
InheritParameters("walksat-tabu", "", false)
CreateTrigger(PickWalksatTabuNoNull,                                         /* trigger name */
   ChooseCandidate,                                                            /* event point */
   PickWalksatTabuNoNull,                                                        /* procedure */
   ...)

Procedure PickWalksatTabuNoNull
PickWalksatTabu
if flipVariable is null then
 |  flipVariable := select random variable from selectedClause.Vars[ ]
end
```

Figure 3.9: The WALKSAT/TABU-NONULL algorithm in UBCSAT.

over all runs. Additional reports and statistics can easily be added to UBCSAT in a straightforward manner similar to the way in which new algorithms are added. Reports can be in any format and are implemented based on a list of triggered procedures that collect and output the required information.

We created a column object that will calculate the average of the age property for flipped variables during a run, as shown in Figure 3.10. We also added a formula property curVarAge that is determined by the current step, and the lastChange property of the variable that is to be flipped. The curVarAge property requires a procedure `UpdateCurVarAge` to update the property value, which requires a corresponding trigger *UpdateCurVarAge* to map the procedure to the event point *PreFlip*. The trigger *UpdateCurVarAge* depends on the trigger *VarLastChange* (see Figure 3.7), so if the algorithm already collects this data (*e.g.*, WALKSAT/TABU) then the statistic will simply share the existing variable property data. If the algorithm does not normally require this data, the trigger will ensure that the property is available. Because this column statistic has been identified as a *TypeMean* (average over all search steps of a run), an additional trigger is automatically activated to collect the statistical data at the end of each search step. Like many statistics added to UBCSAT, this age statistic is now available to *all* algorithms (that use a single-flip strategy). UBCSAT facilitates comparisons between algorithms on statistics such as these, which can help further our understanding of how SLS algorithms behave.

Object formula

property: curVarAge **is** Integer
Age of the most recently flipped variable.

Procedure `AddAgeStat`

```
AddColumn("agemean",                                  /* column name */
   "Mean Age of Variables when flipped",              /* description */
   curVarAge,                                         /* property to monitor */
   UpdateCurVarAge,                                   /* trigger to activate */
   TypeMean,                                          /* type of statistic to measure */
   ...)
CreateTrigger(UpdateCurVarAge,                        /* trigger name */
   PreFlip,                                           /* event point */
   UpdateCurVarAge,                                   /* procedure */
   VarLastChange,                                     /* trigger dependency */
   ...)
```

Procedure `UpdateCurVarAge`

```
curVarAge := step - flipVariable.lastChange
```

Figure 3.10: The agemean statistic in UBCSAT.

3.6 Weighted Algorithms for MAX-SAT

One area where SLS algorithms have been very successful and have defined the state-of-the-art for more than a decade, is in solving the MAX-SAT problem, and in particular, the weighted MAX-SAT problem [55: p. 315]. For this reason, supporting MAX-SAT was one of our goals in the UBCSAT project.

Although there are interesting differences between the state-of-the-art SLS algorithms for SAT and MAX-SAT, at the conceptual and implementation level, there are many similarities. Unweighted MAX-SAT can be seen as a special case of weighted MAX-SAT where all clauses have uniform weight properties. Therefore, in the following discussion, we focus on the weighted MAX-SAT problem. It should be noted that in terms of implementation, SLS algorithms for unweighted MAX-SAT are much more closely related to SLS algorithms for SAT. In UBCSAT, unweighted MAX-SAT algorithms are therefore typically equivalent to the corresponding SAT algorithm, while weighted MAX-SAT algorithms are implemented separately, facilitating conceptually simpler and highly efficient implementations for both cases.

Many existing SLS algorithms for SAT can be generalized easily and naturally to weighted algorithms by simply replacing their standard evaluation function with a weighted alternative, in a manner similar to penalized evaluation functions described in Section 2.6. For example, the WEIGHTED GSAT algorithm can simply add a new weightedScore variable property and select the variable with the smallest weightedScore. Often weighted algorithm variants are designed so that if one clause has twice the weight of another clause, then it should have the same relative effect on the algorithm as having an extra copy of the clause in the formula. For example, in WALKSAT algorithms, the random clause selection is no longer uniform, but instead the probability of selecting each unsatisfied clause is proportional to the clause weight value.

The main differences between SAT and MAX-SAT is that the optimal solution quality (*i.e.*, maximal total weight of satisfied clauses) for a given problem instance is often unknown. Hence, the best assignment encountered during the search, the so-called *incumbent assignment*, is memorized and returned at the end of the search. This memorization of the incumbent assignment is accomplished in UBCSAT via a report. Typically, SLS algorithms for MAX-SAT are not guaranteed to find *optimal solutions* (*i.e.*, maximal weight assignments), but many state-of-the-art SLS algorithms for MAX-SAT have the PAC property that if they search long enough, the probability of finding an optimal solution approaches one [49]. In many practical cases, assignments that are provably optimal or believed to be optimal can be found within reasonable run-times. UBCSAT supports termination criteria that end a run whenever a user-specified solution quality (*e.g.*, the known optimal solution quality for the given problem instance) is reached or exceeded. Alternatively, when dealing with instances whose optimal solution quality is unknown, UBCSAT can be configured with advanced criteria to determine when to terminate a run.

Currently, UBCSAT includes implementations of two dedicated algorithms for MAX-SAT, SAMD [44] and IROTS [105], and weighted MAX-SAT variants for many of the SLS algorithms listed in Section 3.3. The mechanism for implementing new MAX-SAT algorithms within UBCSAT is exactly the same as described for the case of SAT in Section 3.4. For unweighted MAX-SAT instances, the same DIMACS `(.cnf)` file format as for SAT is used. For weighted MAX-SAT instances, UBCSAT currently supports a straightforward extension of the this format known as the *weighted CNF file format* `(.wcnf)`. To support the empirical analysis of the behaviour and performance of SLS algorithms for MAX-SAT, in addition to the statistics and reports mentioned in Section 3.5, UBCSAT supports advanced analysis methods for stochastic optimization algorithms. In particular, the following types of empirical performance characteristics can be easily measured [55: p. 162]:

- Qualified Run-Time Distributions (QRTDs)– empirical probability distributions of the run-time required for reaching or exceeding a specific target solution quality measured over multiple runs of the algorithm;

- Solution Quality Distributions (SQDs)– empirical probability distributions of the best solution quality reached within a given amount of run-time, measured in terms of search steps or CPU time over multiple runs of the algorithm; and

- Solution Quality over Time (SQT) statistics – the development of descriptive statistics (such as quantiles) of the SQDs as run-time increases.

QRTDs, SQDs, and SQTs are determined from so-called *solution quality traces*, which contain information on every point in time the incumbent solution was updated during a given run of the algorithm. The solution quality traces are collected by UBCSAT with minimal overhead during the run of any MAX-SAT algorithm. While for the SAT problem there is no notion of solution quality, solution quality traces can also be useful to analyze the behaviour of SAT algorithms.

3.7 Related Work

In this section we discuss three projects similar to UBCSAT: COMET and OPENSAT preceded it, and SAT4J was developed afterward.

COMET [116] is an OOP language that supports a constraint-based architecture for local search. The COMET language is very sophisticated and can model SLS algorithms for solving complex constraint satisfaction problems, but it does not offer explicit support for SAT or MAX-SAT, nor provide tools for advanced empirical evaluation. While in principle both of these issues could be addressed by realizing the respective functionality within COMET, implementing UBCSAT in COMET seemed to pose the risk that in order to take full advantage of UBCSAT, users would have to understand both the idiosyncrasies of COMET as well as the architecture and interface of UBCSAT. We believed that as a consequence, UBCSAT would have been less accessible to its main target group, namely researchers interested in SAT and MAX-SAT. While there is evidence that COMET algorithm implementations are quite efficient, we do not have any insight as to how these would compare with the native reference implementations of the state-of-the-art SLS algorithms covered by UBCSAT.

The OPENSAT project [7] was developed as a Java-based open source project for DPLL-based solvers. A primary goal of OPENSAT was to make the advanced techniques and data structures used by state-of-the-art DPLL-based solvers openly available in order to accelerate the development of new SAT solvers. Generally, the architecture and implementation of DPLL-based solvers differs considerably from that of SLS-based SAT algorithms, and traditionally there has been very little overlap between the algorithmic and implementation details used in these two types of SAT solvers. Therefore, using OPENSAT as the basis for achieving the previously stated goals, while not completely infeasible, appeared to be problematic. Given the difficulty of supporting the development and implementation of SLS algorithms in a straightforward way, the current lack of support for MAX-SAT solvers, and the fact that OPENSAT does not provide dedicated support for the advanced empirical analysis of SAT algorithms, it is somewhat questionable whether its Java-based implementation makes it possible to achieve performance that is competitive with the native reference implementations of high-performance SLS algorithms such as WALKSAT/SKC or SAPS.

The SAT4J project [136] is similar to the OPENSAT project, but is targeted at end users who wish to solve SAT instances in a *black box* manner without concern for the implementation details. SAT4J implements the popular MINISAT software package [23]. SAT4J has been included in the eclipse open source IDE [125], giving SAT solving capabilities to millions of users. At one point there was an initiative at the SAT4J project to convert UBCSAT into Java and include it in the SAT4J distribution. Unfortunately that initiative has been abandoned.

3.8 Conclusions

We advanced the state-of-the-art for SLS algorithms for SAT (and MAX-SAT) by conceptualizing and developing the UBCSAT software package. UBCSAT meets all of the design goals we stated at the start of this chapter:

1. we included highly efficient, conceptually simple and accurate implementations of a wide range of prominent SLS algorithms for SAT and MAX-SAT;
2. we facilitated the development and integration of new algorithms (and algorithm variants);
3. we provided support for advanced empirical analysis of the performance and behaviour of SLS algorithms without compromising implementation efficiency;
4. we provided explicit support for algorithms designed to solve the weighted and unweighted MAX-SAT problems;
5. we provided an open-source software package that is publicly available to the academic community; and
6. we implemented the project in a platform-independent way, avoiding non-standard programming language extensions.

Overall, the UBCSAT project has been very successful. It has been well cited, has been used in numerous scientific experiments and has provided the framework for the development of new state-of-the-art SAT solvers. We will discuss its success and impact further in Chapter 8. UBCSAT provided the framework for most of the experiments in this dissertation. In the next chapter, we introduce our SCALING AND PROBABILISTIC SMOOTHING (SAPS) algorithm, one of the most successful algorithms included in UBCSAT.

Chapter 4

Scaling and Probabilistic Smoothing

So, Annie are you okay? Are you okay Annie?
You've been hit by... you've been struck by... a smooth criminal.
— Michael Jackson. "Smooth Criminal"

In this chapter, we advance the state-of-the-art for SLS algorithms for SAT by developing the SCALING AND PROBABILISTIC SMOOTHING (SAPS) algorithm. Our primary goal for this chapter was to develop a new algorithm that would reduce both the conceptual and computational complexity of the existing EXPONENTIATED SUBGRADIENT (ESG) algorithm, and exceed its performance on the same benchmark instances that established ESG as a state-of-the-art algorithm. Our secondary goal was to advance the understanding of the behaviour of ESG and our new algorithm. Chronologically, SAPS was the first body of work we completed in this dissertation, and as we will demonstrate, it was the genesis of our interest in Dynamic Local Search (DLS), which is the prevalent theme of our dissertation.

The remainder of this chapter is structured as follows. First, in Section 4.1, we provide background information on the research leading up to our SAPS algorithm, which we introduce in Section 4.2. Next, in Section 4.3, we present experimental results evaluating SAPS on benchmark instances and discuss the search behaviour of SAPS. Then, in Section 4.4, we review subsequent related work. Finally, in Section 4.5, we summarize our work.

4.1 Background and Previous Related Work

In Section 2.6, we introduced the class of algorithms that use Dynamic Local Search with Clause Penalties (DLS-CP), and described the penalty clause property (Figure 2.12). We explained how, in typical (non-DLS-CP) SLS algorithms for SAT, the evaluation function is the intrinsic evaluation function (*i.e.*, the number of unsatisfied clauses), whereas for typical DLS-CP algorithms the penalized evaluation function is the sum of the clause penalty values for all unsatisfied clauses. Thus, when all of the clause penalty values are identical and positive, the penalized evaluation function is equivalent to the intrinsic evaluation function. This is the initial behaviour for all of the DLS-CP algorithms we describe below, with all penalty values initialized to one, unless otherwise noted. One property of this penalized evaluation function we exploit in Section 4.2 is that multiplying all clause penalty values by a (positive) constant does not materially affect the evaluation function. Some algorithms use a slightly different notation where the weights are initialized to zero and the evaluation function includes the intrinsic evaluation function with an added term corresponding to the weights. For example, the GLSSAT and DLM algorithms described below were designed to have zero-based clause weights (λ) and our definition of a clause penalty value is equivalent to $(1+\lambda)$. To aid in the understanding of the algorithms, we describe all DLS-CP algorithms with our clause penalty value notation.

The general approach of all DLS-CP algorithms is the same. They use a penalized evaluation function, and throughout the search, the clause penalty values are *updated*. How and when those updates occur is the largest distinction between the DLS-CP algorithms and is our focus in this section. We also identify other interesting aspects of the DLS-CP algorithms and identify key contributions to the field.

The earliest known DLS-CP algorithm for SAT is the GSAT+CW algorithm by Selman and Kautz [98]. In GSAT+CW, the penalty values are only updated during a restart, where restarts occur after *maxTries* search steps (*maxTries* is a parameter of the algorithm). When the restart occurs, the variable assignment is re-initialized and the penalty values of all unsatisfied clauses are incremented. Selman and Kautz demonstrated that this method improved the performance dramatically over regular GSAT on instances with hidden asymmetries, such as gerrymandered graph colouring encodings, which we discuss in Section 5.2. They suggested that this method of DLS-CP was used to “fill in local minima” [98].

We described the BREAKOUT algorithm by Morris in Section 2.6 [83]. Whereas GSAT+CW updates clause penalties at restarts, BREAKOUT updates clause penalties whenever a local minimum is encountered. Morris described his method within the context of the search landscape topography, and suggested that BREAKOUT was raising the height of the current point in the search space. He acknowledged that changing the height at one point in the search space also affected the height at other points, and he discussed the theoretical FILL algorithm, where the height of only a single point in the search landscape would be changed. He argued that this FILL algorithm would be guaranteed to find a solution, but be infeasible in practice.

Frank developed two DLS-CP algorithm variants he named WGSAT and WGSAT WITH DECAY [29, 30], but we refer to them as GSAT+LR and GSAT+LR+D to avoid any confusion with the WEIGHTED GSAT algorithm. Unlike GSAT+CW or the BREAKOUT method, GSAT+LR updates penalty values after each search step, and if a restart occurs, all penalty values are reset to one.

The GSAT+LR clause update procedure, as with GSAT+CW and BREAKOUT, only applies to unsatisfied clauses, but instead of incrementing (*i.e.*, increasing by one) the penalty values it increases them by a *learning rate* parameter δ:

$$\mathsf{penalty} := \mathsf{penalty} + \delta. \tag{4.1}$$

The GSAT+LR+D algorithm adds a decay mechanism to GSAT+LR and was the first DLS-CP to introduce a reduction in clause penalty values. GSAT+LR+D adds a second stage to the penalty updates where a decay multiplier p $(0 < p \leq 1)$ is applied to all clauses:

$$\mathsf{penalty} := p \cdot \mathsf{penalty}. \tag{4.2}$$

Frank suggested that GSAT+LR could achieve *learning* as it searches, and that the GSAT+LR+D algorithm could also *"forget"* some of its learning to ensure that clause penalties have only a short term effect. This was motivated by the observation that for long search trajectories clause penalty values can become large, so incremental changes to clause penalties become increasingly insignificant. Frank also addressed the issue that GSAT+LR search steps are more complex than simpler algorithms (such as GSAT), and that it can be misleading to measure algorithm performance by search steps alone.

Mills and Tsang adapted the proven Guided Local Search (GLS) approach from other domains to SAT with the GLSSAT algorithm [81]. The variable selection in GLSSAT is unique, in that it selects the oldest variable that can achieve any improvement in the evaluation function, or if no improvement is possible, the oldest variable that is a sideways move. A clause penalty update occurs if a specified number of sequential sideways moves are taken or there is a strict local minimum (no sideways or improving moves possible). The clause penalty update mechanism is also unique, only incrementing the unsatisfied clauses with the minimal penalty value. An additional variant of GLSSAT with a decay mechanism was introduced (GLSSAT2) where after 200 penalty updates the clause penalty values of all clauses were updated as:

$$\mathsf{penalty} = \frac{4}{5} \cdot (\mathsf{penalty}) + \frac{1}{5}. \tag{4.3}$$

Several DISCRETE LAGRANGIAN METHOD (DLM) algorithms have been developed that were motivated by Lagrange multipliers for continuous and discrete domains [102, 118, 119]. The authors model clause weights as Lagrangian multipliers, and suggest the strong mathematical foundation behind Lagrangian methods are the reason DLS-CP algorithms are so effective. However, many of the desired theoretical properties of Lagrangian methods are not realized in DLM because of the numerous modifications to the underlying strategy that are necessary to make the algorithms efficient in practice [56: p. 152]. As we mentioned previously, in our parlance, the Lagrange multipliers (λ) are equivalent to the clause penalty values minus one. Shang and Wah introduced the first DLM algorithm, which is now referred to as DLM-98-BASIC-SAT [102]. The algorithm combines a clause penalty evaluation function with a simple tabu strategy, where a unique stagnation criteria is used and the clause penalty update procedure depends on algorithm parameters $(\theta_1, \theta_2, \delta_o, \delta_d)$. For every θ_1 steps in which no improvement has been made (*i.e.*, after θ_1 sideways moves), an adjustment step occurs where the unsatisfied clause penalty values are increased by δ_o. After θ_2 adjustment steps, the clause penalty values for all clauses are decreased by δ_d. The DLM-99-SAT algorithm by Wu and Wah adds an additional *special increase* mechanism that can increase the penalty value for a clause by an additional δ_s if the clause is frequently unsatisfied in local minima (see [118] for details). DLM-2000-SAT [119] uses the same clause penalty updates as DLM-98-BASIC-SAT, and introduced changes to the Lagrangian evaluation function to incorporate a history of recently visited search locations to avoid backtracking.

The SMOOTHED DESCENT AND FLOOD (SDF) algorithm by Schuurmans and Southey [95, 96] uses an advanced smoothing (flooding) mechanism, and it was the first DLS-CP algorithm to introduce multiplicative updates. The SDF algorithm incorporates a novel evaluation function that takes into account not just whether or not a clause is satisfied, but also how many variables satisfy each clause, thus reducing the number of ties and plateaus in the search space (*i.e.*, allowing for a smoother descent – see [95] for details). SDF uses the BREAKOUT strategy and has different update procedures for satisfied and unsatisfied clauses. We only provide a high-level description of the SDF algorithm here, which uses mechanisms to ensure that the sum of all clause penalty values remain constant (see [96] for details). For unsatisfied clauses, the penalty values are multiplied by a constant α_* that is calculated at each update to make the smallest penScore for all variables equal to $(-\delta)$, where δ is a parameter of SDF. For satisfied clauses, the penalty values are first updated by multiplying by a constant β_*:

$$\beta_* = (1 - \alpha_* \cdot \textstyle\sum_\bot) / \sum_\top, \tag{4.4}$$

where $\sum_\top$ and $\sum_\bot$ are the sum of the clause penalties for satisfied and unsatisfied clauses, respectively. The satisfied clauses penalty values are subsequently *smoothed* toward their mean value $(\frac{\sum_\top}{|C_\top|})$:

$$\mathsf{penalty} := \rho \cdot \mathsf{penalty} + (1-\rho) \cdot \frac{\sum_\top}{|C_\top|}. \tag{4.5}$$

An interesting observation made by Schuurmans and Southey is that SDF is almost completely deterministic, since ties are rare. We were fascinated by this observation, and explore this concept further in Chapter 6.

Schuurmans *et al.* followed up on SDF by introducing their EXPONENTIATED SUBGRADIENT (ESG) algorithm where the term *exponentiated* comes from the penalty value update mechanism (see below) and the term *subgradient* refers to a local search approach in Lagrangian parlance. Two stated objectives of Schuurmans *et al.* were to generalize SDF and develop a domain-independent Lagrangian approach for solving Binary Linear Programs (BLPs). This unfortunately resulted in (their words) some overbearing terminology. We refrain from introducing some of that terminology and present a very simplified version of what is described in the paper [97]. ESG uses a BREAKOUT strategy, enhanced with a random walk parameter η where when a local minimum is reached a random variable is flipped with probability η, otherwise, clause penalties are updated. In the ESG paper the greedy search steps are *primary steps* and the clause penalty updates are *dual steps*. This should not be confused with the following two stages of clause updates. In the first stage, all satisfied clause $\mathsf{penalty}$ values are updated as:

$$\mathsf{penalty} := \alpha^{-1/2} \cdot \mathsf{penalty}, \tag{4.6}$$

and all unsatisfied clause $\mathsf{penalty}$ values are updated as:

$$\mathsf{penalty} := \alpha^{+1/2} \cdot \mathsf{penalty}, \tag{4.7}$$

where α is an algorithm parameter. In the second stage, all clause penalties, as opposed to just satisfied clauses in SDF, are smoothed back to their mean value $\overline{\mathsf{penalty}}$ according to a smoothing parameter ρ:

$$\mathsf{penalty} := \rho \cdot \mathsf{penalty} + (1-\rho) \cdot \overline{\mathsf{penalty}}. \tag{4.8}$$

As we discuss in the following section, the authors' software implementation differs from the description in their paper.

When introduced, the performance of the ESG algorithm was impressive, with results showing ESG could dominate the performance of SDF and was competitive with state-of-the-art SLS algorithms for SAT. Encouraged by the performance of ESG, we developed our SAPS algorithm that led to much of the work in this chapter.

4.2 The Scaling and Probabilistic Smoothing Algorithm

Before we introduce the SAPS algorithm, we will briefly review some implementation details of the ESG algorithm. In the previous section we introduced the ESG algorithm and stated that the authors' software implementation (version 1.4) was different than the algorithm described in the paper [97]. Instead of multiplying the satisfied clause $\mathsf{penalty}$ values by $\alpha^{-1/2}$ and the unsatisfied clause $\mathsf{penalty}$ values by $\alpha^{1/2}$, only the unsatisfied values are multiplied by α_{raw}, where

$$\alpha_{\text{raw}} := 1 + \alpha \cdot \frac{|V|}{|C|}, \tag{4.9}$$

and $|V|$ is the number of variables in the formula and $|C|$ is the number of clauses (see Figure 2.8 for further symbol definitions). Setting aside the difference between α and α_{raw}, the change to modify only unsatisfied clause penalty values does not change the ESG algorithm, since multiplying all clauses by a positive constant (*i.e.*, $\alpha^{1/2}$) does not materially affect the evaluation function (although it does highlight the peculiarity of the original design). The smoothing in the ESG implementation was also changed by replacing the mean penalty value ($\overline{\mathsf{penalty}}$) with the constant value of one. This simplification altered the ESG algorithm, but was made to allow for a lazy update scheme to be feasible, where clause penalty values were only updated after one hundred updates (to avoid floating point numerical drift) or when necessary for calculations.

The lazy update scheme was implemented in ESG because in a straightforward implementation, the clause penalty update procedure is computationally expensive when compared to the search steps where no update is required. In a straightforward implementation, the time to complete an ESG search step, without a penalty update (flipping variable v_i) is:

$$\Theta(|V_\perp|) + \Theta(scl_i), \tag{4.10}$$

where the first term is the time to determine the variable to flip, and the second term is the time to maintain the bookkeeping for penScore (see Equation 3.1). The worst-case time to complete a search step with a penalty update is:

$$\Theta(|V_\perp|) + \Theta(|C|) + \Theta(|F|), \tag{4.11}$$

where the first term is the time of the (failed) variable selection, the second term is the time of the clause penalty update, and the third term is the time to maintain the bookkeeping for penScore. The difference in the time performance between the two types of search steps arises because for a typical variable v_i, $scl_i \ll |F|$.

In Table 4.1, we compare ESG and NOVELTY$^+$ on instances from the ESG paper. We selected NOVELTY$^+$ for this experiment because at the time it was a well-known state-of-the-art SLS algorithm for SAT [55: p. 278]. The run-length performance of ESG is rather impressive for a variety of problem instances; for some instances it outperforms NOVELTY$^+$ by more than an order of magnitude. Typically, ESG's run-time performance is somewhat better than that of NOVELTY$^+$, but even with the aforementioned software optimizations, ESG does not always outperform NOVELTY$^+$. Hence, it seems that the time of the penalty updates in the ESG implementation is limiting its performance.

The impact of the clause penalty update on the overall time complexity depends on the percentage of search steps that are update steps (*i.e.*, steps in local minima). In Table 4.1, we show the percent of steps in local minima range from around 7% (for flat instances) to more than 40% percent (for the ais instance). The results for logistics-c are especially illuminating, as the run-length

Problem Instance	NOVELTY$^+$ nov. noise	NOVELTY$^+$ steps $\times 10^3$	NOVELTY$^+$ time [sec]	ESG α_{raw}	ESG ρ	ESG η	ESG tot. steps $\times 10^3$	ESG % pen. updates	ESG time [sec]
bw-large-a	0.40	7.0	**0.014**	3.0	0.995	0.0015	2.7	10.3	0.016
bw-large-b	0.35	125.3	0.34	1.4	0.99	0.0005	31.6	14.6	**0.28**
bw-large-c	0.20	3 997.1	**16.00**	1.4	0.99	0.0005	1 625.7	11.9	38.10
logistics-c	0.40	101.7	**0.226**	2.2	0.99	0.0025	14.4	32.4	0.229
flat100-med	0.55	7.6	**0.008**	1.1	0.99	0.0015	7.5	15.5	0.013
flat100-hard	0.60	84.0	0.089	1.1	0.99	0.0015	22.9	12.2	**0.037**
flat200-med	0.60	198.4	**0.21**	1.01	0.99	0.0025	104.2	7.3	0.24
flat200-hard	0.60	18 147.7	18.86	1.01	0.99	0.0025	2 725.2	7.9	**5.89**
uf100-hard	0.55	30.0	0.046	1.15	0.99	0.001	2.9	22.3	**0.006**
uf250-med	0.55	9.9	**0.015**	1.15	0.99	0.003	8.4	16.4	0.020
uf250-hard	0.55	1 817.7	2.75	1.15	0.99	0.003	192.0	13.9	**0.46**
uf400-med	0.55	100.4	**0.16**	1.15	0.99	0.003	110.3	9.1	0.32
uf400-hard	0.55	14 419.9	22.30	1.15	0.99	0.003	3 297.8	8.6	**9.76**
ais10	0.40	1 332.2	4.22	1.9	0.999	0.0004	22.8	42.8	**0.14**

Table 4.1: NOVELTY$^+$ *vs* ESG performance. Bold text indicates the CPU time of the faster algorithm. For all runs of NOVELTY$^+$, $wp = 0.01$. ESG steps include clause penalty value updates (at local minima) where no flip occurs, with the percent of steps with updates shown. Parameter values for NOVELTY$^+$ were obtained by ad-hoc testing. ESG parameter values were obtained from the original ESG paper [97]. Software versions: ESG v1.4, NOVELTY$^+$ (WALKSAT) v36. Execution environment: UBC BETA cluster (Section C.2). Median run-lengths and run-times are reported, obtained from 100 runs. See Appendix B for instance information.

for ESG is approximately 14% that of NOVELTY$^+$, yet the run-time is approximately the same. In our empirical analysis of the ESG software, we found that despite the lazy update scheme, the time complexity of the clause penalty updates severely limited the performance of ESG, which has been a general criticism of DLS-CP algorithms in general [30].

Our goal for developing a new algorithm was to reduce both the conceptual and computational complexity of ESG while maintaining the excellent run-length performance. We decided to leave as much of the original ESG design (which itself has remnants of the SDF design) unchanged in order to highlight the impact of our changes. For example, the random walk mechanism in ESG is not very effective, but we chose to use the same mechanism to provide a more meaningful comparison. The only change we made to the ESG algorithm was the clause penalty update procedure. To help better illustrate the mechanisms involved in clause penalty value updates we divide the procedure into two separate stages: *scaling* and *smoothing*.

For the scaling stage, we avoided the Lagrangian notations introduced in the ESG paper and adopted the practice used in the ESG software of multiplying the $\mathsf{penalty}$ values of unsatisfied clauses by a simple parameter α_{raw}. However, we chose to use a simple constant value of α in lieu of the calculated α_* in SDF or the α_{raw} calculations introduced in the ESG software (Equation 4.9). To summarize, the scaling step from ESG

$$\mathsf{penalty} := \alpha^{\pm 1/2} \cdot \mathsf{penalty} \tag{4.12}$$

that applied to all clauses was simplified to

$$\mathsf{penalty} := \alpha \cdot \mathsf{penalty} \tag{4.13}$$

for only unsatisfied clauses. This straightforward change in the scaling procedure changes the worst-case time for a straightforward implementation of the scaling stage from

$$\Theta(|C|) + \Theta(|F|) \tag{4.14}$$

to

$$\Theta(|C_\perp|) + \Theta(|V_\perp|), \tag{4.15}$$

where $|C_\perp|$ is the number of false clauses, $|C_\perp| \ll |C|$, $|V_\perp| \ll |F|$, and as the search progresses, $|C_\perp|$ and $|V_\perp|$ both approach zero.

For the smoothing step, we made two changes. The first minor change is related to the observation we made previously that a uniform scaling of all clause penalties has no effect on the search algorithm, so we reduced the smoothing step from:

$$\mathsf{penalty} := \rho \cdot \mathsf{penalty} + (1-\rho) \cdot \overline{\mathsf{penalty}}, \tag{4.16}$$

to:

$$\mathsf{penalty} := \mathsf{penalty} + (1-\rho) \cdot \overline{\mathsf{penalty}}. \tag{4.17}$$

This change does not affect the time complexity significantly, but in practice the difference was measurable and it made the required bookkeeping more straightforward. However, we feel that this change makes a significant conceptual difference that highlights the simple yet fundamental difference between the two stages: scaling is simple multiplication and smoothing is simple addition[1]. We note that the term $(1-\rho) \cdot \overline{\mathsf{penalty}}$ could be replaced by a simple constant, but as mentioned previously we wanted to keep the general structure of ESG to highlight the most significant change made, namely *probabilistic smoothing*.

Due to our straightforward scaling procedure, the relative time complexity of the smoothing

[1]At one point in our development, we were considering alternate names for SAPS that reflected this multiplication and addition duality, but SAPS was determined to be more appealing and reflective of our contribution.

stage was now much more significant. As a result, we conducted experiments where we performed smoothing less frequently and with greater intensity and observed that it did not significantly affect the overall run-length performance of the algorithm. The procedure we eventually developed was called probabilistic smoothing, where for every scaling stage a smoothing stage was performed with probability *ps*. With probabilistic smoothing, the overall probabilistic worst-case time to perform a clause penalty update becomes:

$$\begin{array}{cc} \textit{with probability } (1-ps): & \Theta(|C_\perp|)+\Theta(|V_\perp|) \\ \textit{otherwise}: & \Theta(|C|)+\Theta(|F|). \end{array} \tag{4.18}$$

Obviously, there are other ways of achieving the same effect. For instance, similar to the mechanism found in DLM, smoothing could be performed deterministically after a fixed number of scaling stages. However, the probabilistic smoothing mechanism has the theoretical advantage of preventing the algorithm from getting trapped in cyclic behaviour (see also [51]). The other important distinction between our mechanism and those of other DLS-CP algorithms such as DLM is that our strategy is designed to reduce the time complexity of our approach, as opposed to a delayed decay mechanism. We present the core procedures of the SAPS algorithm in Figure 4.1.

Like the authors of ESG, our software implementation of the SAPS algorithm differs slightly from the published version. In the description in Figure 4.1, we describe the SAPS variant that corresponds to the UBCSAT software implementation of SAPS used in practice and note that the UBCSAT software also includes a variant that strictly implements SAPS as described in our paper [61]. There are three differences that exist between our software implementation and the version of SAPS we originally described. First, in the `PickVariableSAPS` procedure shown in Figure 4.1, a threshold parameter ε is introduced, where a change in the penScore is only considered an improving step if it exceeds this threshold. The second change was illustrated in the change from Equation 4.16 to Equation 4.17, where the penalty values are no longer multiplied by ρ. The removal of this multiplication caused the penalty values to increase over time, so to avoid numerical drift we introduced the third change, a normalization stage at the `UpdateClausePenaltiesSAPS` procedure as shown in Figure 4.1.

When we introduced SAPS, our co-author, Frank Hutter, developed the RSAPS algorithm. RSAPS adjusts the *ps* value during the search in a manner similar to the mechanism used by ADAPTIVE NOVELTY$^+$, which we discussed in Section 2.3. We do not discuss RSAPS further in this dissertation, and our original paper can be consulted for further details [61].

4.3 Experimental Results

To evaluate the performance of SAPS against ESG and NOVELTY$^+$, we conducted computational experiments on widely used benchmark instances for SAT. The results of this are presented in

Procedure `PickVariableSAPS`

```
bestPenScore := minimum penScore from Variables[ ].penScore
if bestPenScore <$\varepsilon$ then
    flipVariable := select variable from Variables[ ] with minimum penScore (BTR)
else
    with probability $wp$
        flipVariable := select random variable from Variables[ ]
    otherwise
        flipVariable := null
    end
end
```

Procedure `UpdateClausePenaltiesSAPS`

```
if flipVariable is null then
    foreach clause c in FalseClauses[ ] do
        c.penalty := c.penalty · $\alpha$
    end
    with probability $ps$
        avgPenalty := average of all Clauses[ ].penalty values
        smoothPenalty := (1 - $\rho$)· avgPenalty
        foreach clause c in Clauses[ ] do
            c.penalty := c.penalty + smoothPenalty
        end
    end
    maxPenalty := maximum penalty from FalseClauses[ ].penalty
    if maxPenalty > MAXPEN then
        foreach clause c in Clauses[ ] do
            c.penalty := c.penalty / MAXPEN
        end
    end
end
```

Figure 4.1: Variable selection and penalty updates in the SAPS algorithm. In the UBC-SAT implementation if SAPS, ε is a parameter defaulting to (-0.1), and MAXPEN $=$ 1 000.

Table 4.2. SAPS achieves superior performance over ESG and NOVELTY$^+$ with default values of α and *ps* with one exception. SAPS exhibited poor performance on the instance bw-large-c, despite some manual adjustment of the α parameter. We note that the aforementioned RSAPS algorithm was faster than both NOVELTY$^+$ and ESG on this instance. In Table 4.1, the instance logistics-c was identified as problematic for ESG, and we can see that the SAPS run-time outperforms ESG by a factor of six. For smaller instances, such as uf100-hard, the time is roughly the same; SAPS is never slower than ESG. Furthermore, for all problem instances considered in the ESG paper [97] where DLM outperformed ESG, SAPS (and RSAPS) outperform ESG by a greater margin. When comparing SAPS to NOVELTY$^+$, the performance differences are more apparent and the run-time performance of SAPS is often more than an order of magnitude superior. We note that since performing these experiments, we have collected evidence that there are numerous instances (*e.g.*, g125.17 and g125.18) where NOVELTY$^+$ performs substantially better than SAPS.

In the previous section, we described how the time complexity of a smoothing step is proportional to the problem size. This suggests that performance differences between ESG and SAPS should also increase with problem size. To avoid complications arising from different implementations, we use a variant of SAPS with $ps = 1$ to illustrate these differences. We refer to this variant as SAPS/ESG. The performance of this variant is very similar to ESG for small instances and is marginally better for larger instances. To demonstrate this proportional behaviour with respect to the problem size, we conducted experiments with SAPS/ESG, NOVELTY$^+$ and SAPS with $ps = 0.05$ on the instance sets uf100 and uf400. We present the results of these experiments in Figure 4.2 and Figure 4.3. We see impressive results for SAPS/ESG on uf100 (Figure 4.2 (a)), whereas its performance degrades for the instances in uf400 (Figure 4.3 (a)). The performance of SAPS is similar to SAPS/ESG on uf100 (Figure 4.2 (b)). However, for the larger instances in uf400 (Figure 4.2 (b)) it becomes obvious that the performance of SAPS can be far superior to that of SAPS/ESG.

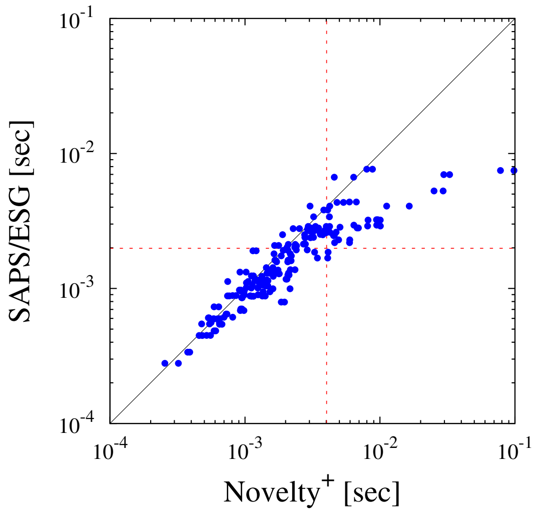
The experimental results in Table 4.2, Figure 4.2 and Figure 4.3 were originally presented with the introduction of SAPS in 2002 [61]. Subsequently, the SAPS algorithm has been a standard benchmark algorithm with numerous publications and algorithms measuring their performance against SAPS. More recent DLS-CP algorithms, such as PAWS (see Section 4.4), can outperform SAPS on a wide variety of instance domains. However, with the possible exception of SATENSTEIN [70], we are not aware of an SLS algorithm that has demonstrated the ability to dominate SAPS. The comparison to SATENSTEIN is unique, since SATENSTEIN subsumes SAPS and can itself be configured to behave as SAPS (see [70] and Section 7.5 for more details). A configuration of SATENSTEIN known as SATENSTEIN[FAC] is the best-known SLS algorithm for the fac benchmark set, where SAPS (with default settings) was identified as the strongest known competitor. In Figure 4.4, we demonstrate that an optimized configuration of SAPS can outperform the published settings of SATENSTEIN on the fac test set. This improvement is possible because the overhead of SAPS is marginally smaller than SATENSTEIN, and because SAPS has a much smaller

Problem Instance	NOVELTY$^+$ time [sec]	ESG time [sec]	SAPS α	SAPS ρ	SAPS tot. steps $\times 10^3$	SAPS % pen. updates	SAPS time [sec]	SAPS *s.f.*
bw-large-a	0.014	0.016	1.3	0.8	2.6	12.9	**0.009**	1.56
bw-large-b	0.34	0.28	1.3	0.8	32.7	9.8	**0.18**	1.56
bw-large-c	**16.00**	38.10	1.1	0.6	2 131.0	12.4	37.88	0.42
logistics-c	0.226	0.229	1.3	0.9	8.7	25.5	**0.037**	6.10
flat100-med	0.008	0.013	1.3	0.4	6.6	17.1	0.008	1.00
flat100-hard	0.089	0.037	1.3	0.8	25.6	13.7	**0.032**	1.16
flat200-med	0.21	0.24	1.3	0.4	59.9	7.8	**0.09**	2.39
flat200-hard	18.86	5.89	1.3	0.4	2 169.9	9.9	**3.05**	1.93
uf100-hard	0.046	0.006	1.3	0.8	3.8	21.5	0.006	1.00
uf250-med	0.015	0.020	1.3	0.4	6.6	17.5	**0.011**	1.36
uf250-hard	2.75	0.46	1.3	0.7	170.4	15.5	**0.29**	1.58
uf400-med	0.16	0.32	1.3	0.4	53.0	10.4	0.10	1.55
uf400-hard	22.30	9.76	1.3	0.2	1 033.8	12.9	**1.97**	4.95
ais10	4.22	0.14	1.3	0.9	19.9	32.4	**0.05**	2.73

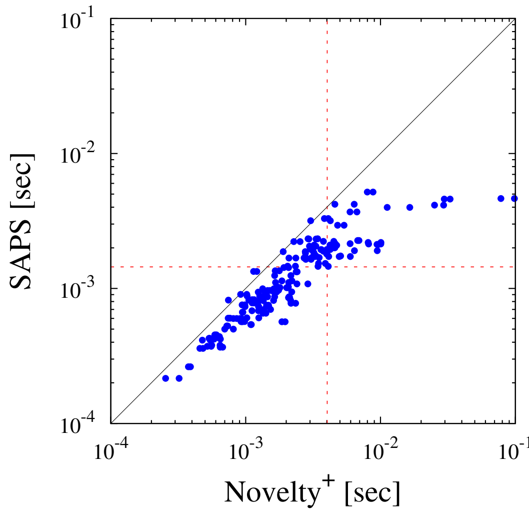
Table 4.2: SAPS *vs* NOVELTY$^+$ and ESG. The columns for NOVELTY$^+$ and ESG are repeated from Table 4.1. Bold text indicates the CPU time of the faster algorithm. For all runs of SAPS, $ps = 0.05$ and $wp = 0.01$. SAPS steps include clause penalty value updates (at local minima) where no flip occurs, with the percent of steps with updates shown. Parameter values for SAPS were obtained manually via ad-hoc testing. The speedup factor (*s.f.*) is the lesser of the NOVELTY$^+$ and ESG time divided by the SAPS time. Software versions: SAPS v1.0, Execution environment: UBC BETA cluster (Section C.2). Median run-lengths and run-times are reported, obtained from 100 runs.

number of parameters and is easier to optimize. However, our intent is not to claim that SAPS outperforms SATENSTEIN (a futile exercise because of the aforementioned subsumed behaviour), but rather demonstrate that despite the several years since SAPS was introduced, it is still relevant and able to outperform the current best-known state-of-the-art SLS algorithms for SAT on an interesting instance class from the literature.

We motivated our probabilistic smoothing approach with our observation that less frequent smoothing did not affect the run-length performance of the algorithm, and we provide evidence of this behaviour in Figure 4.5. In Figure 4.5, we illustrate the effect of varying the smoothing probability (ps) on the performance of SAPS, while simultaneously decreasing the smoothing parameter (ρ) to achieve an equivalent amount of smoothing overall. From Figure 4.5 (a), it is clear that smoothing less frequently and with greater intensity can achieve nearly identical run-length



(a) $ps = 1$ (ESG)



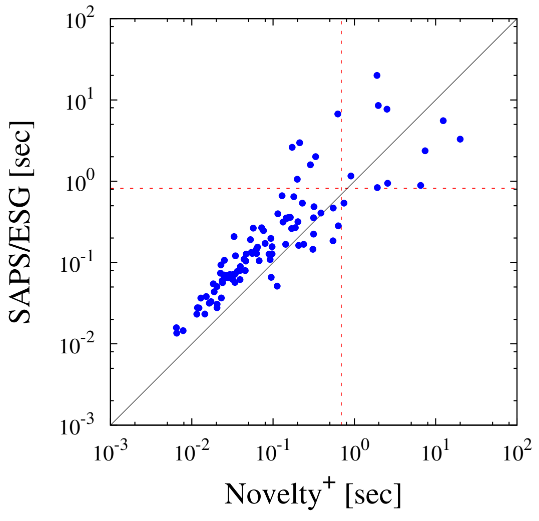
(b) $ps = 0.05$

Figure 4.2: SAPS on uf100: smoothing *vs* no smoothing. The speedup factor (*s.f.*) is (a) 2.03 and (b) 2.79. For NOVELTY$^+$, the *noveltyNoise* parameter is 0.55. For SAPS, the parameters in both plots are $(\alpha, \text{wp}) = (1.3, 0.01)$. For (a), $(ps, \rho) = (1.0, 0.99)$, for (b), $(ps, \rho) = (0.05, 0.8)$. (Each point corresponds to an instance in the set uf100 (100 instances). Median run-times obtained from 100 independent runs per instance. Execution environment: UBC BETA cluster (Section C.2). See Section C.5 for general correlation plot details.)

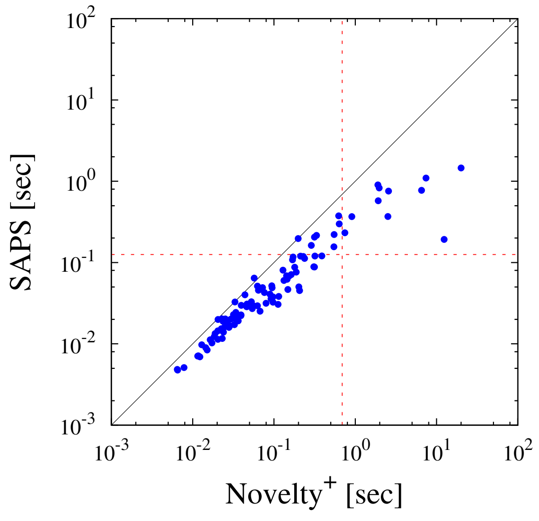
performance, while in Figure 4.5 (b) we can see the improvement in run-time performance achieved from smoothing less frequently.

When we introduced the SAPS algorithm, we were seeking to gain a deeper understanding of the role of the parameters α, ρ and *ps*, and we analyzed the evolution of clause penalty values over time. This analysis is the basis of the work we continue in Section 5.3. We provide an extended version of our original analysis in the following.

To illustrate the effect of the clause penalty values on the search evaluation function, we consider a very simple search sequence example that isolates three clauses (a, b, c) where two of the clauses are each unsatisfied for different search steps. Table 4.3 illustrates the changes in penalty values to these three hypothetical clauses. Because of the scaling and smoothing mechanism, we can see that although clause a and b were scaled the same number of times, the clause that was scaled more recently (b) has a larger penalty value. This example illustrates how scaling and smoothing can be seen as a mechanism for ranking the clauses by their history. Interestingly, these effects can be interpreted as that of a soft tabu mechanism on clauses, where recently unsatisfied clauses with higher weights are more tabu (*i.e.*, likely to stay satisfied longer).



(a) $ps = 1$ (ESG)



(b) $ps = 0.05$

Figure 4.3: SAPS on uf400: smoothing *vs* no smoothing. The speedup factor (*s.f.*) is (a) 0.84 and (b) 5.50. For both plots, NOVELTY$^+$ noise parameter is 0.55, SAPS parameters $(\alpha, \text{wp}) = (1.3, 0.01)$. For (a), $(ps, \rho) = (1.0, 0.97)$, for (b), $(ps, \rho) = (0.05, 0.4)$. (Each point corresponds to an instance in the set uf400 (100 instances). Median run-times obtained from 100 independent runs per instance. Execution environment: UBC BETA cluster (Section C.2). See Section C.5 for general correlation plot details.)

We believe that the observed multiplication and addition duality in scaling and smoothing and its historical effect on search behaviour illustrated by this example helps conceptualize one of the mechanisms that makes SAPS (and by extension ESG and SDF) effective SAT solvers. Ignoring issues of numerical precision and representation, these solvers have a very long-term history (*i.e.*, the clause penalty values reflect the entire history of the clause). By introducing probabilistic smoothing, we are essentially making the history more coarse, grouping all scaling activity that happens between two smoothing steps into the same period of search history. Scaling activity in the current period of history will have the most weight, then the previous period of history, and so on.

The parameters of SAPS (α, ρ, ps) clearly affect the clause penalty values dynamically during the search, and to illustrate their behaviour we examine Clause Penalty value Distributions (CPDs). To generate a CPD, we take a *snapshot* of the clause penalty values at a specific time during the search trajectory, normalize the values so that the minimum value is one, and finally sort the normalized values. Figure 4.6 shows typical CPDs for a given SAT instance after 400 scaling steps. In our experience, after a certain number of local minima, the CPD converges to a specific distribution that is determined by the problem instance and the parameter settings. We hypothesize that the shape of the CPD for a given problem instance determines the performance of SAPS. In Figure 4.6, we

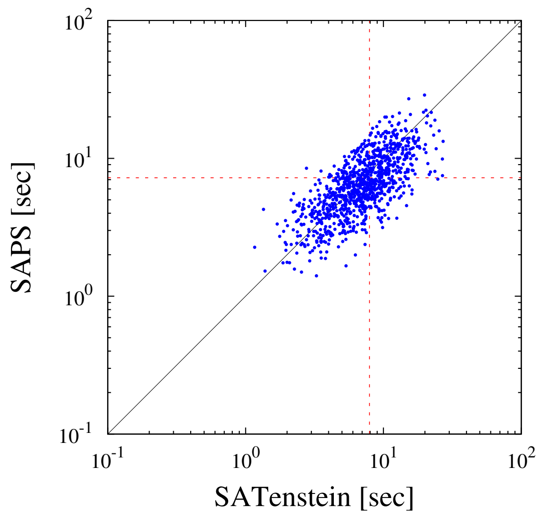
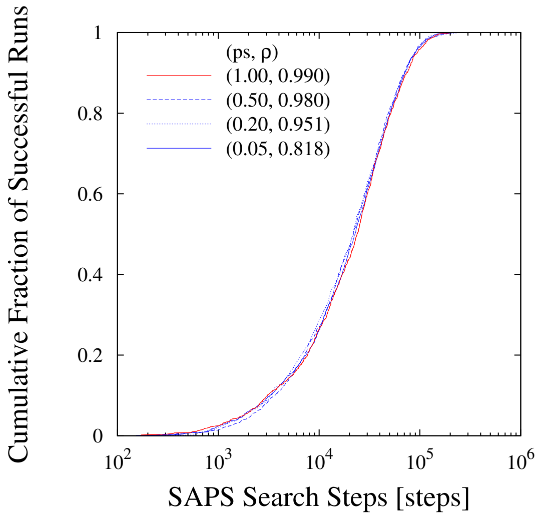


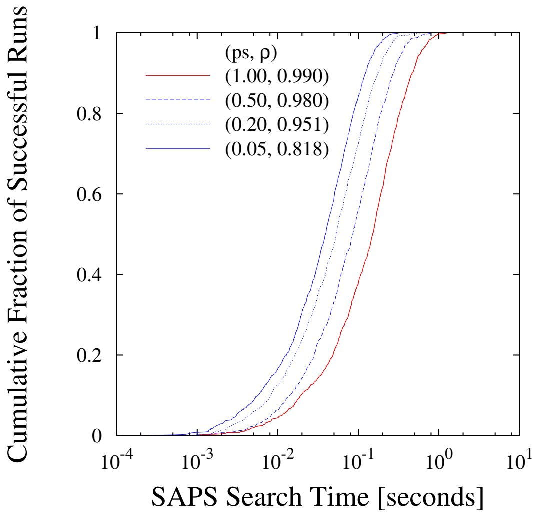
Figure 4.4: SAPS *vs* SATENSTEIN on fac. The speedup factor (*s.f.*) is 1.10. The SATENSTEIN parameters are the same as used in the SATENSTEIN paper [70]. The SAPS parameter settings of $(\alpha, ps, \rho, \mathrm{wp}) = (1.11, 0.025, 0.82, 0)$ were obtained manually. (Each point corresponds to an instance in the set fac (1000 instances). Median run-times obtained from 25 independent runs per instance. Execution environment: UBC arrow cluster (Section C.1). See Section C.5 for general correlation plot details.)

Description	a.penalty	b.penalty	c.penalty
Initialization	1	1	1
Scaling step affecting a	α	1	1
Scaling step affecting none	α	1	1
Smoothing step (see below)	$\alpha + k$	$1 + k$	$1 + k$
Scaling step affecting b	$\alpha + k$	$\alpha + k \cdot \alpha$	$1 + k$

Table 4.3: Scaling and smoothing example. The sequence of events in this example is as follows: Clause a is unsatisfied at a local minimum causing a scaling step, then another clause is unsatisfied at a local minimum (not shown here) causing a scaling step and a probabilistic smoothing step, and then finally clause b is unsatisfied at a local minimum causing another scaling step. The value of k is a positive constant not material to the discussion equal to $(1-\rho) \cdot \overline{\text{penalty}}$, where $\overline{\text{penalty}}$ is the mean of all penalty values after the second local minimum. The value of α is greater than one.



(a) run-length performance



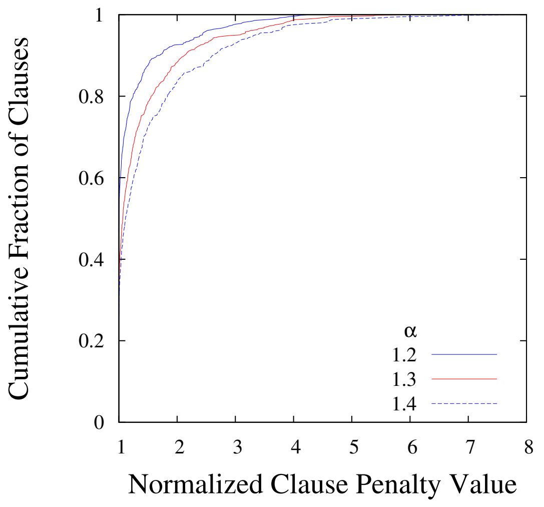
(b) run-time Performance

Figure 4.5: The effect of smoothing on SAPS performance. The smoothing parameter (ρ) and *ps* were adjusted to achieve an equivalent amount of smoothing overall. (Each configuration of SAPS was run on the instance ais10 1 000 times. Execution environment: UBC arrow cluster (Section C.1). Unless otherwise noted, all algorithms executed with default parameters as described in Appendix A.)

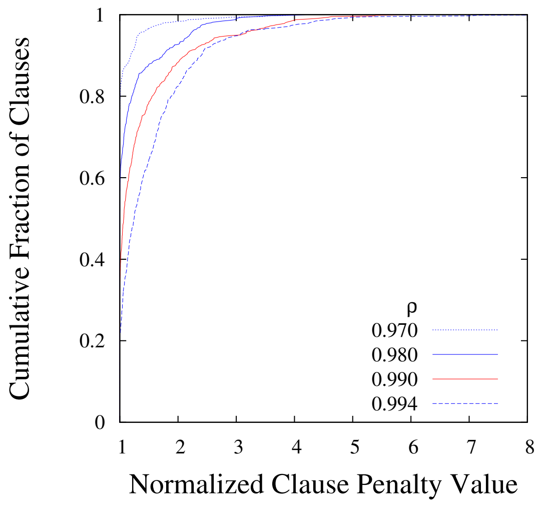
can see directly the effect of changing the parameters on the CPDs. For example, in Figure 4.6 (b) the smoothing parameter ρ has a significant impact on the shape of the CPD. Intuitively, the SAPS search procedure will place greater emphasis on satisfying and keeping satisfied the clauses with higher clause weights. For smaller values of ρ (*i.e.*, more smoothing), fewer clauses have high weights, leading to a greedier, more intensified search. Conversely, less smoothing leads to a more diversified search.

One of our more interesting observations was that if two different (α, ρ, ps) triplets result in nearly identical CPDs, they will also yield nearly identical performance results. Two such triplets are $(1.3, 0.99, 1.0)$ and $(1.3, 0.818, 0.05)$, where we demonstrated in Figure 4.5 they achieve similar run-length performance; as can be seen in Figure 4.6, the respective CPDs are very similar. Clearly, the parameters of SAPS are not independent, and studying CPDs could provide a way of modelling the dependency between the parameters to reduce the total number of parameters. Finally, this suggests that another DLS-CP algorithm, which produces CPDs similar in shape to the CPDs of SAPS and ESG, could achieve similar run-length performance with potentially less run-time complexity.

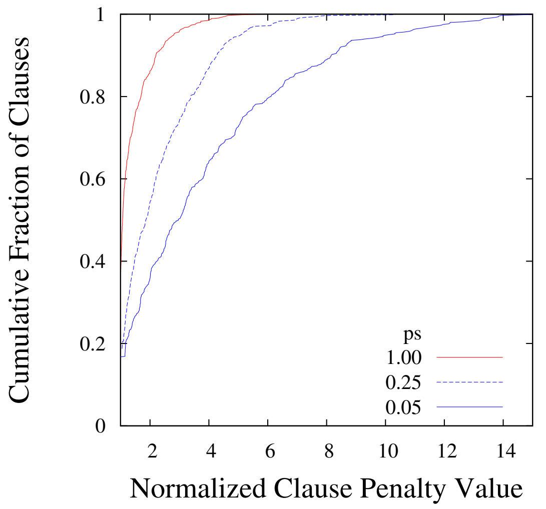
One final observation about the nature of the SAPS algorithm, originally observed by Schuurmans and Southey [95] on their SDF algorithm, is that as the search progresses, the variable



(a) Scaling (α)



(b) Smoothing (ρ)



(c) *ps*

Figure 4.6: SAPS clause penalty distributions. Each configuration of SAPS was terminated after 400 scaling operations on the instance uf150-hard. For each plot only one parameter was modified, with the (α, ρ, ps) parameters set to the values of $(1.3, 0.99, 1.0)$, and all other parameters set to defaults as described in Appendix A. The base configuration is identical for each plot and is highlighted. (Clause penalty values were normalized, *i.e.*, the penalty value of an always satisfied clause is one.)

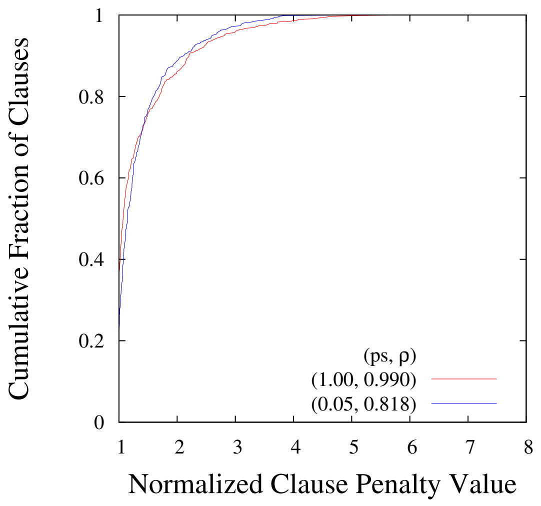


Figure 4.7: Approximately equivalent CPDs for two configurations of SAPS. (Each configuration of SAPS was terminated after 400 scaling operations on the instance uf150-hard. The values of (ρ, ps) are set as indicated in the plot, and all other parameters set to defaults as described in Appendix A. Clause penalty values were normalized, *i.e.*, the penalty value of an always satisfied clause is one.)

selection in SAPS essentially becomes deterministic. We will explore this further in Chapter 6, where we will introduce the deterministic SAPS/NR algorithm.

4.4 Subsequent Related Work

In this section, we briefly discuss the prominent DLS-CP algorithms that have been developed since SAPS was introduced in 2002. Since numerous experiments in the following chapter will be conducted with PAWS and DDFW, we will discuss those algorithms in more detail.

The Pure Additive Weighting Scheme (PAWS) algorithm by Thornton *et al.* [111] was developed as a response to SDF, ESG and SAPS and demonstrated that multiplicative clause penalty value updates are not always necessary and that a straightforward additive scheme can be quite effective. The PAWS clause penalty value update mechanism is to increment all unsatisfied clause penalty values, then (similarly to DLM) after every *maxInc* updates, all clause penalty values greater than one are decremented. Where PAWS differs from all previous DLS-CP algorithms is that it introduces a novel mechanism for taking random sideways steps. PAWS will always update the clause penalty values when a strict local minimum is encountered (*i.e.*, no sideways moves are possible), but when a sideways move is possible, PAWS will allow a sideways step with probability p_{flat}. The final interesting mechanism in PAWS is that for tie-breaking between variables (either for an improving or a sideways step) it uses a unique *Multiple Inclusion* roulette selection scheme whereby variables that appear in more than one false clause are proportionately more likely to be selected.

Thornton followed the original PAWS paper [111] with a publication studying the differences between PAWS and SAPS [109]. He conducted thorough empirical experiments to isolate the individual effects of the SAPS and PAWS algorithm mechanisms. He introduced five SAPS variants (SAPS+M, SAPS+R, SAPS+D, SAPS+A, SAPS+T) and four PAWS variants (PAWS+M, PAWS+R, PAWS+D, PAWS+A) that represent various hybrids or intermediate steps from SAPS to PAWS. His primary conclusion from his experiments was that while PAWS does not dominate SAPS, an additive strategy tends to perform better than a multiplicative one, particularly on the larger and more difficult problems. Thornton made additional interesting conclusions (*e.g.*, the negligible effect of multiple inclusion) and the full text can be consulted for further details [109].

Ferreira Jr. and Thornton developed a PAWS variant they named PAWS WITH USUAL SUSPECTS (PAWS+US) which allows for two different classes of clauses and penalizes them differently. PAWS+US will be discussed in Section 5.6.

The DIVIDE AND DISTRIBUTE FIXED WEIGHTS (DDFW) algorithm by Ishtaiwi *et al.* [63] takes a very unique approach to updating clause penalty values. To increase the penalty value of one clause, the penalty value of another clause has to be decreased by the same amount. As with the SDF algorithm, the sum of the clause penalties in DDFW remains constant. Aside from the clause penalty value updates, DDFW is identical to PAWS. For the DDFW algorithm, we set aside our convention that all clause penalty values are initialized to one, and initialize them to the value of the parameter W_{init}, where the default value is 8. In DDFW clause penalty value updates, each unsatisfied clause is increased and a satisfied source clause is selected to be decreased. If the source clause penalty value is greater than W_{init}, the change in penalty values is two, otherwise the change is one. To select the source clause, the satisfied neighbouring clause with the highest penalty value is selected. Two clauses are neighbours if they share a literal with the same variable and polarity. In the event that no such clause exists, a random clause with a penalty value greater than W_{init} is selected. Although not mentioned in the paper, the software version of DDFW provided by the authors also adds an additional noise parameter TL such that even if a satisfied same sign neighbouring clause is found, with probability $(1 - TL)$ a random clause with penalty value greater than W_{init} is selected instead. The UBCSAT implementation of DDFW that we use in our experiments includes this additional behaviour, which we found necessary to avoid search stagnation. Because of the need to search for the neighbouring clause with the highest penalty, the search steps in DDFW can take longer than search steps in SAPS or PAWS. In practice, the difference is often not significant, and DDFW can outperform those algorithms on numerous instances.

Ishtaiwi *et al.* followed up DDFW with DDFW$^+$ [62], which added an adaptive mechanism to DDFW to reduce the sensitivity of DDFW to the W_{init} parameter. The full text should be consulted for specific details, but essentially when stagnation is detected (more than $|V|$ steps without improvement) the algorithm alternates between a mechanism to increase the penalty values and a mechanism to reset the penalty values.

The RLS algorithm by Pullan and Zhao [91] migrated the technique of clause resolution from DPLL-based SAT algorithms to SLS methods successfully. RLS uses a DLS algorithm technique that we do not explore in this dissertation, namely the dynamic addition of clauses during search; however, RLS also uses clauses penalty values and a DLS-CP scoring function. Pullan and Zhao addressed not only the issue of assigning penalties to the clauses in the problem instance, but also assigning penalties to new learned clauses resulting from clause resolution.

Anbulagan *et al.* introduced the R+ family of algorithms [2] that included several R+ variants of DLS-CP algorithms (*e.g.*, R+SAPS, R+PAWS). The R+ variants do not actually change the underlying algorithm, but rather use a resolution-based pre-processor on the instance before attempting to solve the instance.

The VW1 and VW2 algorithms [88] by Prestwich use variable weight properties instead of clause penalty values and implement a smoothing mechanism for those variables. While VW1 and VW2 are not DLS-CP algorithms *per se*, they incorporated DLS-CP smoothing mechanisms. We will discuss VW2 in detail in Chapter 7.

Pham *et al.* introduced the gNovelty$^+$ algorithm [87] that has demonstrated excellent performance. In brief, gNovelty$^+$ can be seen as a hybrid of the G^{2}WSAT and Adaptive Novelty$^+$ algorithms with a DLS-CP scoring function that updates clause penalty values when there are no promising variables. The penalty value update mechanism is very straightforward, with a simple (additive) increment of unsatisfied clause penalty values and a SAPS-like probabilistic mechanism to decrement (smooth) all clause penalty values.

The most recent DLS-CP algorithm of note that we are aware of is the iPAWS algorithm by Thornton and Pham, which adds a sophisticated simulated annealing-based self-tuning mechanism to control a *maxThres* parameter, where clause penalty values are decremented when the total number of false clauses and penalized clauses each are greater than *maxThres* (see [110] for details).

4.5 Conclusions

We made two significant contributions toward advancing the state-of-the-art for SLS algorithms for SAT. First, we introduced SAPS, a new DLS-CP algorithm that dominates the performance of its predecessor, the ESG algorithm, and is still amongst the state-of-the-art SLS algorithms for SAT. Second, we provided a unique perspective of how SAPS and ESG behave, advancing our understanding of DLS-CP algorithms. As with the UBCSAT project we described in Chapter 3, SAPS has been very successful and has had a great impact on the academic community; it has been well cited, has been well studied, has led to several new advancements and has been used as a benchmark state-of-the-art algorithm in several publications, which we will discuss further in Chapter 8.

Chapter 5

Clause Penalties

What I've got's full stock of thoughts and dreams that scatter.
You pull them all together, and how, I can't explain.
— Hall & Oates. "You Make My Dreams"

In the previous chapter, we introduced the SCALING AND PROBABILISTIC SMOOTHING (SAPS) algorithm, a member of the class of algorithms known as Dynamic Local Search with Clause Penalties (DLS-CP). We made several interesting observations on the behaviour of SAPS, and in this chapter we extend this work by exploring the behaviour of SAPS and other prominent DLS-CP algorithms in more depth. Our goals for this chapter were to:

1. explore how clauses in an instance can be weighted to make the instance easier to solve;
2. follow up our study of clause penalties from Chapter 4 with a more comprehensive analysis of their behaviour;
3. investigate the dynamic landscapes generated by DLS-CP algorithms and the role they play in solving SAT instances; and
4. examine how clause penalty history affects the performance of DLS-CP algorithms.

This chapter is structured as follows. First, in Section 5.1, we motivate the algorithms and instances we use in our experiments throughout this chapter. Next, in Section 5.2, we introduce the concept of statically weighted instances and demonstrate the performance potential of SLS algorithms on those instances. Then, in Section 5.3, we examine the dynamic properties of the clause penalties during DLS-CP algorithm search trajectories. In Section 5.4, we combine the work from Section 5.2 and Section 5.3 to explore if DLS-CP algorithms weight their instances to render them easier to solve. Next, in Section 5.5, we examine how clause penalty history affects the performance

Feature	SAPS	PAWS	DDFW
Unsatisfied clause penalty update strategy	Multiplication $\times\alpha$	Addition $+1$	Addition $+1$ or $+2$
penalty Decay strategy	Probabilistic Smoothing $+(1-\rho)\cdot\overline{\text{penalty}}$	Periodic Subtraction -1	Distributive Subtraction -1 or -2
Clause penalty domain	Real	Integer	Integer
Initial clause penalty value	1	1	W_{init}
Allow sideways steps	No	with probability p_{flat}	with probability p_{flat}
Tie-breaking	Random	Multiple Inclusion	Multiple Inclusion
Random walks at local min	with probability wp	No	No

Table 5.1: Algorithmic comparison of SAPS, PAWS and DDFW.

of DLS-CP algorithms. Then, in Section 5.6, we discuss related work. Finally, in Section 5.7, we summarize our work.

5.1 Background

In this section we motivate our selection of algorithms and instances that we use throughout this chapter. To study the behaviour of DLS-CP algorithms, and the behaviour of their clause penalty values, we selected three prominent and representative algorithms: SAPS [61], the PURE ADDITIVE WEIGHTING SCHEME (PAWS) [111] and the DIVIDE AND DISTRIBUTE FIXED WEIGHTS (DDFW) algorithm [63]. We introduced SAPS in Section 4.2, and we described PAWS and DDFW in Section 4.4. We use these state-of-the-art DLS-CP algorithms because each has a different clause penalty update mechanism, and most prominent DLS-CP algorithms use mechanisms that are similar to one of these three algorithms. We summarize the differences between the three algorithms in Table 5.1.

In this chapter we conduct experiments that require a non-penalty-based weighted SLS algorithm. For those experiments we selected the prominent and state-of-the-art algorithms WEIGHTED ADAPTIVE NOVELTY$^+$ [51] and WEIGHTED G^{2}WSAT [76]. We described the original (unweighted) ADAPTIVE NOVELTY$^+$ and G^{2}WSAT algorithms in Section 2.5. We note that both algorithms use the score property to guide their search, which measures changes in the intrinsic evaluation function. We briefly described how weighted variants are developed in Section 3.6. The primary difference between the weighted and unweighted variants is that for weighted variants, the weightedScore is used (instead of score) to select variables and to determine if a variable is promising. In addition, instead of selecting unsatisfied clauses uniformly at random, the probability of selecting an unsatisfied clause is proportional to its clause weight value.

For experiments in Section 5.4, we created a new algorithm we call GSAT WITH NO WORSENING STEPS (GSAT/NW). GSAT/NW behaves the same as GSAT except that it does not perform any worsening steps and will terminate if it is in a strict local minimum. GSAT/NW is not very effective and is of interest solely because it behaves like the greedy search in BREAKOUT-based DLS-CP algorithms (*i.e.*, BREAKOUT without clause penalty updates). WEIGHTED GSAT/NW is a straightforward weighted implementation of GSAT/NW.

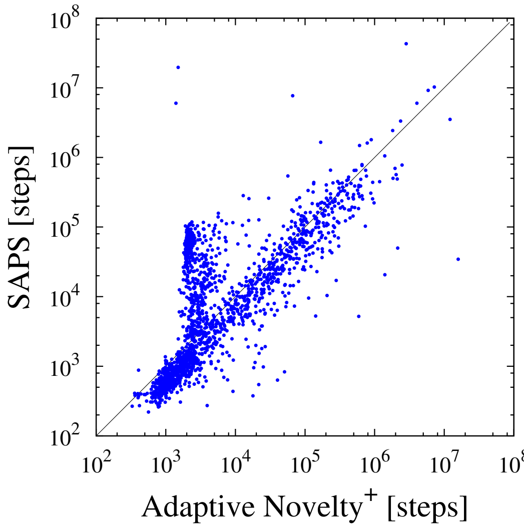
For the experiments in this chapter, we created a new instance set that we have named anp10m. This set contains 1931 instances and includes *all* structured (non-random) instances available on SATLIB where ADAPTIVE NOVELTY$^+$ has a median run-length between 10^3 and 10^7 steps. We selected these instances and this range to provide a reproducible and interesting set that includes a wide range of instance types and hardness without being prohibitively difficult to provide a thorough analysis. The majority of the instances are encodings of graph colouring problem instances, as they are the most abundantly available, and include instances from flat and swgcp. There are 19 additional types of instances in the set, including those from ais, bw-large, logistics, ii and parity.

We study structured instances in this chapter because, in our preliminary experiments, we observed some unusual and interesting behaviour for a small number of structured instances. The behaviour we observed for the majority of structured instances was consistent with our observations for random instances, so we believe our general observations and conclusions hold for both structured and random instances.

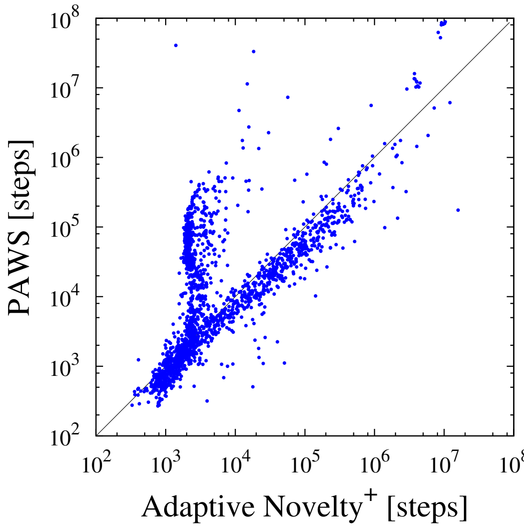
In our experiments we characterize behaviour over all instances in the set anp10m, but we also study individual instances. We selected the instance flat125-94 as a representative instance from the set anp10m. This instance was selected because it was the instance with median behaviour in Figure 5.14 (a). For the figures in Section 5.3 that present results for all of the instances in anp10m, we have identified where flat125-94 appears in the figure.

In this chapter, we perform experiments to better understand the general behaviour of DLS-CP algorithms by measuring changes in their run-length performance. We do not attempt to compare algorithms or claim superiority of one algorithm over another. We do not make any attempts to optimize the parameter settings of the algorithms, and use default settings for all algorithms as described in Appendix A. On the other hand, since we use the SAPS, PAWS, DDFW, ADAPTIVE NOVELTY$^+$ and G^{2}WSAT algorithms throughout this chapter, it is useful to observe how the performance of these algorithms compare on anp10m. We compared the run-length performance of ADAPTIVE NOVELTY$^+$ to the DLS-CP algorithms and present the results in Figure 5.1.

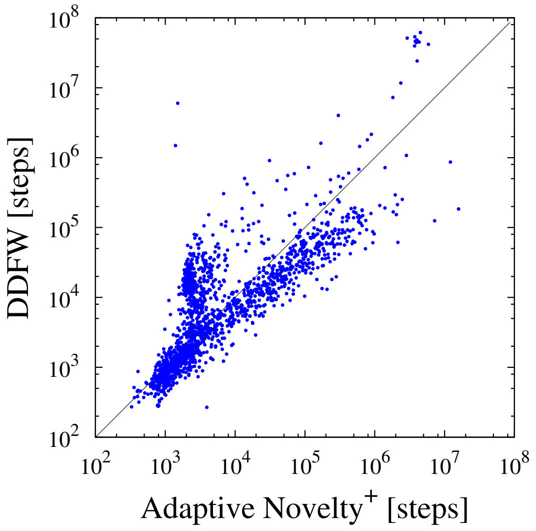
The most prominent feature of each of the plots is a cluster of instances for which ADAPTIVE NOVELTY$^+$ significantly outperforms the three DLS-CP algorithms. This cluster corresponds to the swgcp instances. Other significant outliers where ADAPTIVE NOVELTY$^+$ significantly outperforms all of the DLS-CP algorithms include all instances from the sets bitadd and the sgi sets, by factors



(a) SAPS

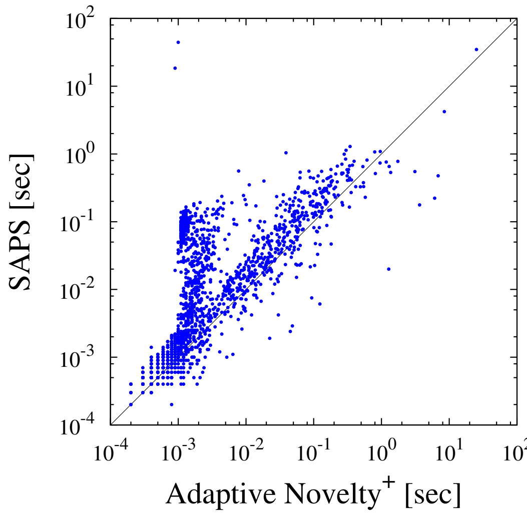


(b) PAWS

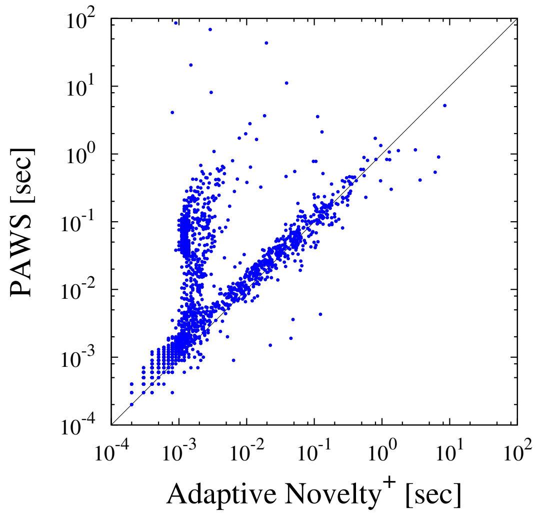


(c) DDFW

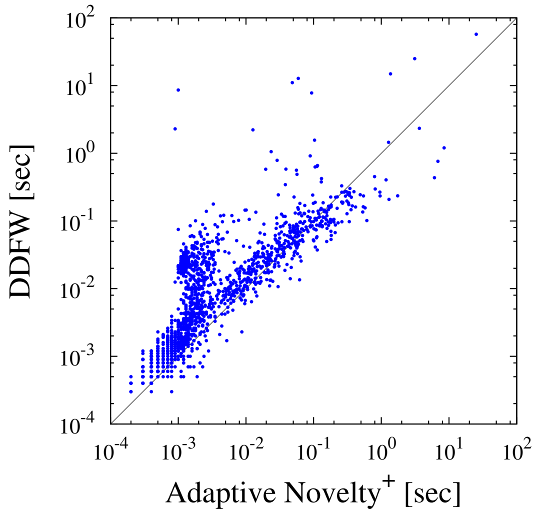
Figure 5.1: ADAPTIVE NOVELTY$^+$ *vs* DLS-CP algorithms on anp10m (run-length). (Each point corresponds to an instance in the set anp10m (1931 instances). Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)



(a) SAPS



(b) PAWS



(c) DDFW

Figure 5.2: ADAPTIVE NOVELTY$^+$ *vs* DLS-CP algorithms on anp10m (run-time). (Each point corresponds to an instance in the set anp10m (1931 instances). Median run-times obtained from 100 independent runs per instance. Execution environment: UBC arrow cluster (Section C.1). All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)

ranging from 2 to over 10 000. Conversely, the three DLS-CP algorithms all significantly outperform ADAPTIVE NOVELTY$^+$ on all the instances from the qg set by factors ranging from 5 to 400. SAPS and PAWS outperformed ADAPTIVE NOVELTY$^+$ on all instances from the ais set by factors ranging from 7 to 70, and in Section 5.2 we study this further. SAPS outperformed ADAPTIVE NOVELTY$^+$ on the ii instances, as did PAWS and DDFW with a few exceptions. SAPS performed poorly on some instances from the clus-1200 instances, whereas PAWS performed very well on these instances. DDFW performed poorly on instances from the bw-large and ferry sets.

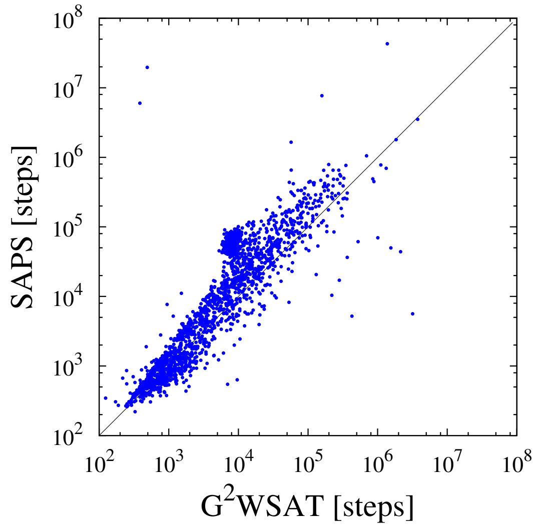
As we discussed in Chapter 2, comparing algorithms by run-length performance alone can be misleading. For additional perspective we also provide the results measured by run-time performance in Figure 5.2. From these figures we observe that for instances where the DLS-CP algorithms have better run-length performance than ADAPTIVE NOVELTY$^+$, the difference in run-time performance is less significant. This is because ADAPTIVE NOVELTY$^+$ does not perform penalty updates, and has less complex search steps. However, as we mentioned previously, in this chapter we are not comparing the run-time performance of different algorithms. In the remaining experiments in this chapter, we study the run-length performance of algorithms, and more importantly, the changes that occur in the run-length performance of an algorithm.

In Figure 5.3, we perform the same comparison as in Figure 5.1 with G^{2}WSAT (the other non-DLS-CP algorithm used in this chapter). In Figure 5.4, we compare the three DLS-CP algorithms to each other. The cluster of swgcp instances is still clearly visible in Figure 5.3 (b) when comparing G^{2}WSAT and PAWS, but is less pronounced for SAPS and DDFW. Otherwise, all of the outliers are the same as those previously observed. One important observation is that none of these algorithms completely dominates any of the other algorithms on anp10m.

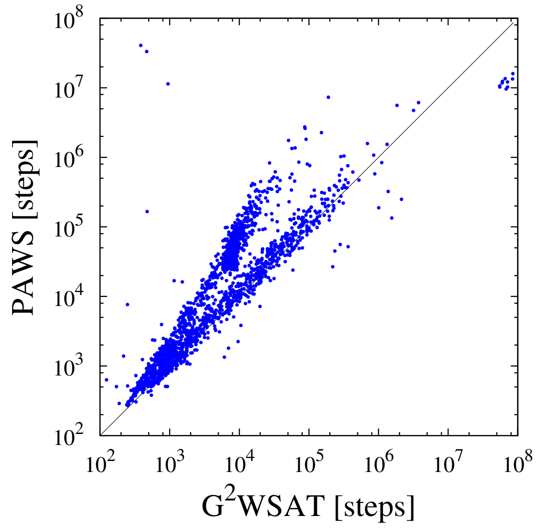
5.2 Weighted Instances

In Section 2.1, we briefly described the weighted MAX-SAT problem and in Section 3.6, we described how weighted MAX-SAT instances add a clause weight property. Just as DLS-CP algorithms typically use a dynamic penScore property, weighted MAX-SAT algorithms typically use a weightedScore property. The significant difference between the two approaches is that the clause penalty values are dynamic and change during the search, whereas the weight properties are static. The weightedScore property measures changes in the weighted evaluation function, which is the sum of the clause weight values for all unsatisfied clauses. Traditionally, in weighted MAX-SAT, the instances are unsatisfiable and the clause weight values are used to determine the optimal solution. In this section, we introduce *satisfiable* weighted instances, where the weight values do not determine the optimal solution, but instead help guide weighted algorithm variants to a solution.

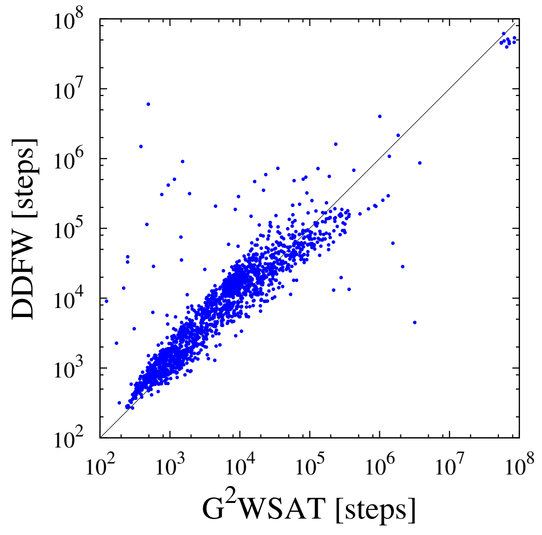
By changing the evaluation function, clause weights also change the search landscape (as described in Section 2.3). For a particular SAT instance, we refer to the search landscape defined by the unweighted instance as the *natural landscape* and the search landscape of a weighted instance



(a) SAPS

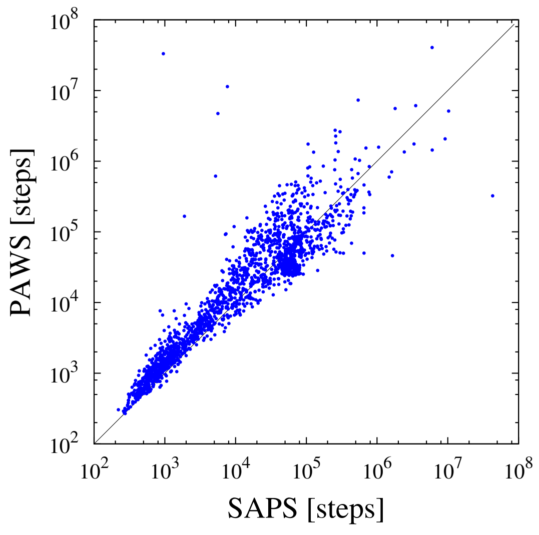


(b) PAWS

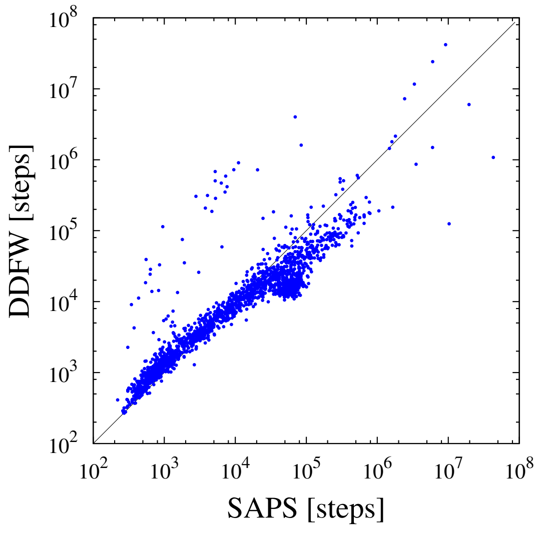


(c) DDFW

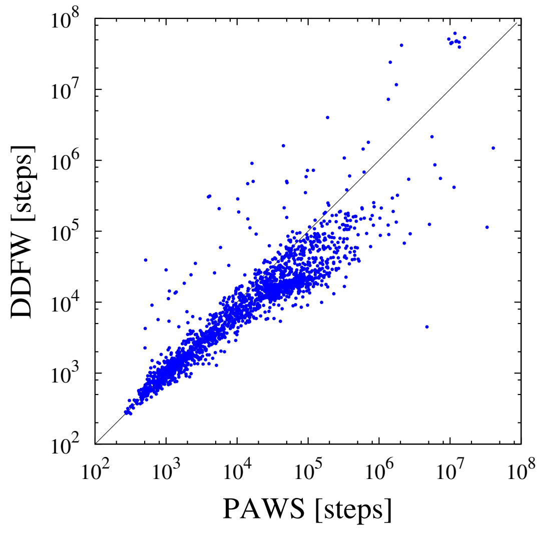
Figure 5.3: G^{2}WSAT *vs* DLS-CP algorithms on anp10m. (Each point corresponds to an instance in the set anp10m (1931 instances). Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)



(a) SAPS *vs* PAWS



(b) SAPS *vs* DDFW



(c) PAWS *vs* DDFW

Figure 5.4: SAPS *vs* PAWS *vs* DDFW on anp10m. (Each point corresponds to an instance in the set anp10m (1931 instances). Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)

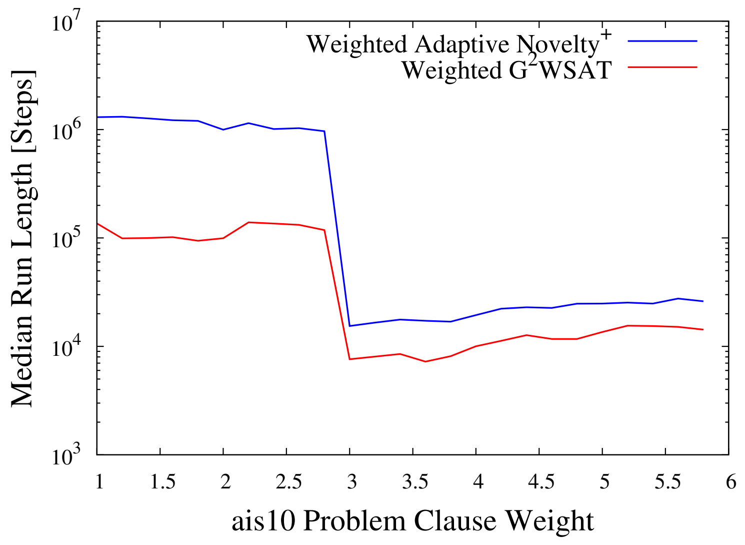


Figure 5.5: Weighted algorithm performance on modified ais10 instance. The ais10 instance was modified to weight the problem clauses (see text for details). (Median run-lengths were obtained from 1 000 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A.)

as the *warped landscape*. We propose that if (on average) a weighted algorithm variant can solve a weighted instance in fewer search steps than the corresponding unweighted instance, then the weighted instance is easier and the warped landscape is easier to search.

We will now provide an example of how a weighted satisfiable instance can be easier than the unweighted instance. We use the ais10 instance we described in Section 2.2. After studying the behaviour of DLS-CP algorithms on the ais instances, we found that of the six clause types, clauses that ensure each number occurs at least once were most frequently unsatisfied and, on average, had correspondingly larger clause penalties. We refer to these ten clauses of the ais10 instance as the *problem* clauses.

We changed the clause weights for the problem clauses of ais10 and measured the effect on WEIGHTED ADAPTIVE NOVELTY$^+$ and WEIGHTED G^{2}WSAT. The results are presented in Figure 5.5. We observed a dramatic improvement in the run-length performance of both algorithms when the clause weights were increased to a value of three or more. Since the critical weight value for the problem clauses for these algorithms is three and it is very straightforward to increase the weight of a clause by an integer factor by simply adding additional copies of the clause to the formula, we generated a new instance, ais10-problem3, with two duplicates of each of the ten problem clauses. We then compared the run-length performance of several algorithms from UBCSAT on the instances ais10 and ais10-problem3. The results are presented in Table 5.2. We observed that

Algorithm	ais10	ais10-problem3	*s.f.*
NOVELTY$^+$	1 213 750	14 094	86.1
ADAPTIVE NOVELTY$^+$	1 260 040	15 261	82.6
R-NOVELTY$^+$	1 233 370	15 493	79.6
G^{2}WSAT	144 195	6 621	21.8
NOVELTY^{++}	212 699	15 882	13.4
SAMD	632 624	50 930	12.4
RoTS	404 993	43 556	9.3
GWSAT	760 716	244 586	3.1
WALKSAT/SKC	271 210	171 700	1.6
HWSAT	866 298	561 448	1.5
PAWS	84 317	63 878	1.3
SAPS/NR	20 822	18 422	1.1
DDFW	708 348	639 538	1.1
SAPS	19 841	18 134	1.1
RSAPS	21 150	19 408	1.1

Table 5.2: SLS algorithm performance on ais10 *vs* ais10-problem3. Performance values correspond to the median run-length in search steps from 1 000 runs. The speedup factor (*s.f.*) is the ratio of the median run-lengths from the ais10-problem3 instance and the ais10 instance. Note that ais10-problem3 is same as the ais10 instance, with the exception that the 10 problem clauses have been duplicated twice each, for a total of 20 additional clauses (see text for description). All algorithm settings are their default values in UBCSAT 1.1.

in all cases the run-length performance of the algorithms improved on the modified instance.

After studying this ais example, we expected to observe similar behaviour for other instances. We hypothesized that all instances would have problem clauses that, when identified, could be used to re-weight the instance to make it easier to solve. We also hypothesized that those problem clauses could be found by examining the clause behaviour of DLS-CP algorithms, such as SAPS, on the instance and identifying clauses that were frequently unsatisfied. However, when we investigated further, we found that this phenomenon appears to be a rare. For most instances, identifying problem clauses and a weighting scheme that make the instance easier is very difficult. Any performance differences that occur on the weighted instance are often minor and not nearly as significant as these results would suggest. In Section 5.3, we investigate identifying problem clauses, and in Section 5.4 we examine how weighting schemes can affect instance hardness.

This behaviour is observed on the ais instances because they contain what Selman and Kautz describe as *asymmetries*, a characteristic they observed in crafted gerrymandered graph colouring instances [98]. The gerrymandered instances were so-named because specific clauses are frequently

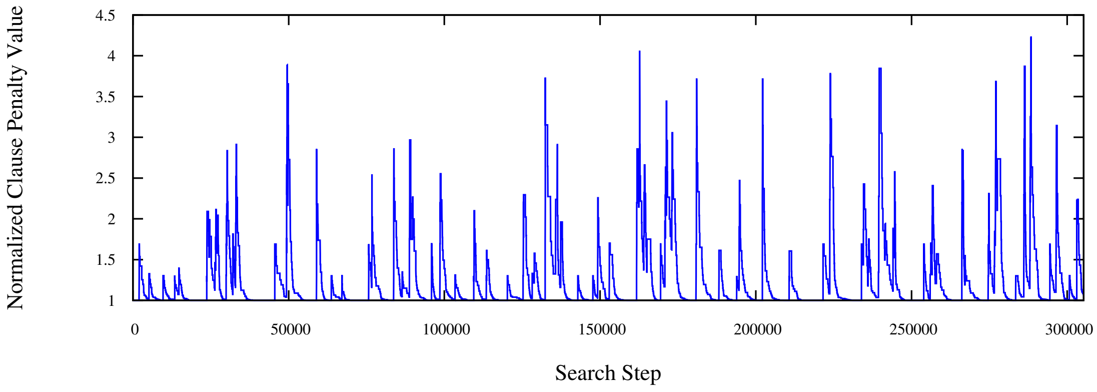


Figure 5.6: Typical clause penalty behaviour throughout a run of SAPS. A clause penalty value over time throughout a single run of SAPS on the instance flat125-94. (The clause selected had the median clause penalty variance for this run. The run selected was the median run from 25 independent runs. All algorithms were executed with default parameters as described in Appendix A.)

unsatisfied because they are often out-voted by other clauses. In the ais instances, the asymmetries occur because there are several clauses that can restrict a number from appearing in a position, whereas there is only one clause (*i.e.*, a problem clause) requiring that the number appear in at least one position. While the ais instances were not gerrymandered, the asymmetries resulting from the encoding has the same effect on the hardness of the instance. Our results suggest that encodings from other domains to SAT can avoid undesired asymmetries and become easier to solve by adding duplicate clauses or alternatively weighting clauses.

5.3 Clause Penalty Behaviour

In this section we examine the behaviour of a single clause penalty throughout a single search run, and expand our study until eventually we examine how all clause penalties behave across multiple independent runs on different instances and for multiple DLS-CP algorithms.

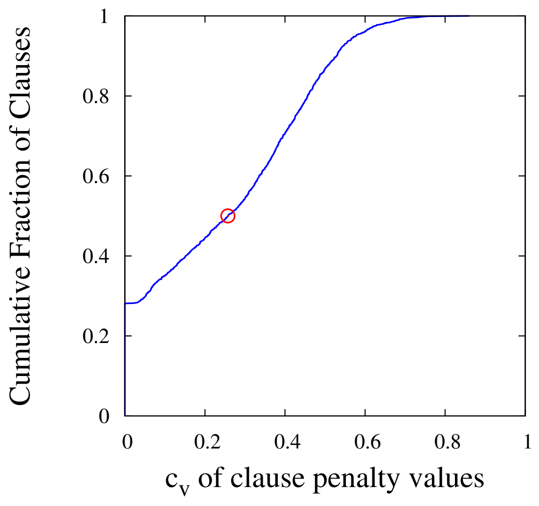
To properly compare the clause penalties from different algorithms, and between different runs of the same algorithm, we normalize our clause penalties. The clause penalties are normalized in such a way that a clause penalty that is not further penalized after initialization has a value of one. Because DDFW can decrease the penalty values of individual clauses, the normalized penalties can have values less than one.

Individual clause penalties can change frequently throughout the search trajectory of a DLS-CP algorithm. To demonstrate this behaviour we present, in Figure 5.6, a plot illustrating how a typical clause penalty changes in SAPS over time throughout a search trajectory. For this typical instance from our anp10m instance set, the individual clause selected had the median variance in penalty value in the run with the median run-length, suggesting this behaviour is representative. Clause penalties can rapidly increase and decrease numerous times throughout a search trajectory. We also observe that the clause penalty behaviour is typically stationary (*i.e.*, its general pattern of behaviour is no different near the end of the search than the middle). The duration for which the clause penalties have a large value before they are decreased (or smoothed) is specific to the algorithm parameters, but the fast decreases seen in the figure are very typical. The clause penalties in PAWS and DDFW are integers and exhibit more discrete changes in penalty value, but their general pattern of behaviour is very similar. We suggest that as the search progresses, individual clauses become relevant locally and then become less relevant in other areas of the search space. We explore this idea in more detail later in this chapter.

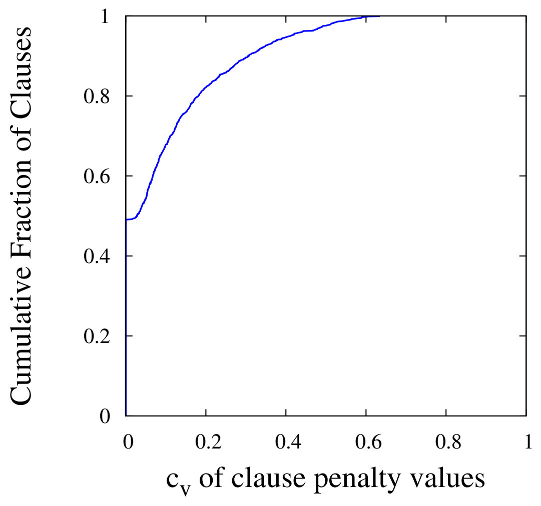
In Figure 5.6, we examined the behaviour of a single clause during a single run, where the clause selected had the median variance in a particular run. We would expect that not all clauses have the same amount of variation throughout a run. To study this effect quantitatively, we measured the variation of all the clause penalties over a run. In Figure 5.7, we show distributions of the coefficient of variation of individual clause penalties over the run. We observe from the figure that some clauses exhibit a large variation in their clause penalty value, whereas others show very little.

After examining the variance of clauses over a run on flat125-94, we now aggregate over anp10m. For each instance, we measured the coefficient of variation for each clause penalty over a run and determined the mean value of those variations. In Figure 5.8, we show the cumulative distribution of those means for all instances in the set anp10m. This measure is an indication of how much fluctuation there is in the clause penalties of an instance, and this figure shows the distribution of that fluctuation. In Figure 5.8, we observe that across anp10m there is a reasonably flat distribution of fluctuation, and each of the three algorithms have similar ranges of values. The amount of fluctuation is loosely correlated with instance hardness, where longer runs tend to have more variation. However, as per our previous observation on stationary clause penalty behaviour, it would seem that other instance characteristics have a larger effect than the length of the search.

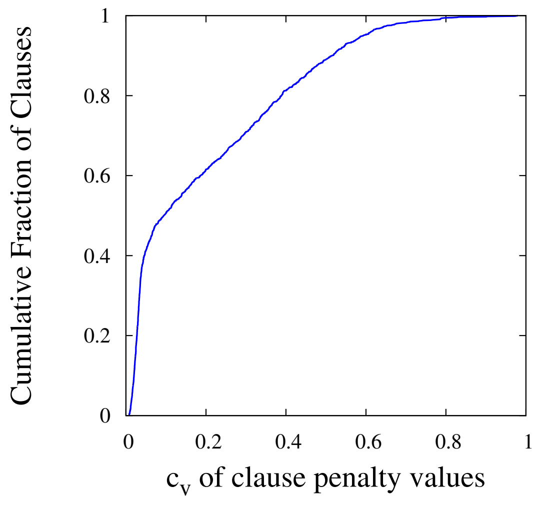
The results presented so far on clause penalty behaviour are not very surprising, but they help establish that individual clause penalties can vary considerably throughout search, with some clauses varying more than others, and some instances fluctuating more than others. We now turn to the actual clause penalty values, and examine *snapshots* of clause penalty values at a particular moment in time during a search trajectory. In Chapter 4, we observed that for a given instance the distribution of the clause penalties stabilized to a specific Clause Penalty value Distribution (CPD) (see Section 4.3). We hypothesized that the performance of an algorithm on a particular instance was determined largely by the CPD. When the SAPS scaling and smoothing parameters were changed, the change in the performance of the algorithm was greatest when there was a change in the CPD. Conversely, little difference was observed in algorithm performance when two different algorithm



(a) SAPS

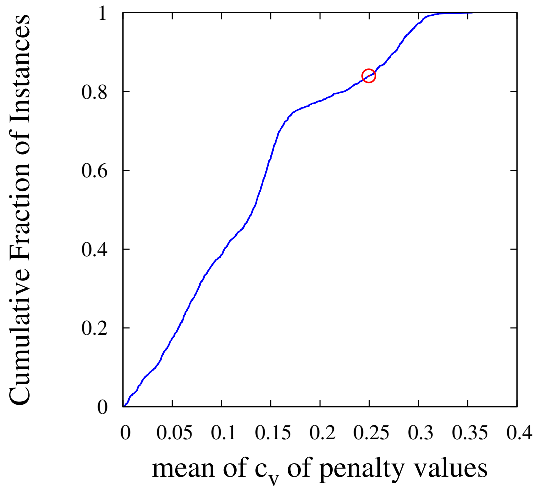


(b) PAWS

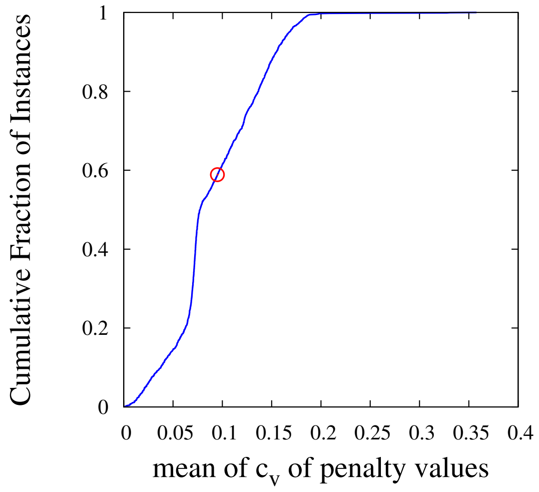


(c) DDFW

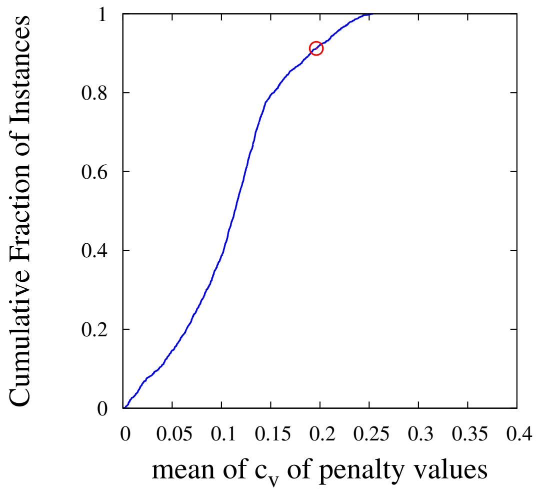
Figure 5.7: Clause penalty variation on flat125-94. For each clause in flat125-94 (1405 clauses), we measure the c_v (coefficient of variation) of the clause penalty value over all search steps in a run. (The run selected was the median run from independent 25 runs. In Figure 5.7 (a) the clause featured in Figure 5.6 is highlighted. All algorithms were executed with default parameters as described in Appendix A.)



(a) SAPS



(b) PAWS



(c) DDFW

Figure 5.8: Mean clause penalty variation per instance on anp10m. For each instance in anp10m (1931 instances), we measure the mean of the c_v of the clause penalty values as described in Figure 5.7. (The instance flat125-94 from Figure 5.7 is highlighted. All algorithms were executed with default parameters as described in Appendix A.)

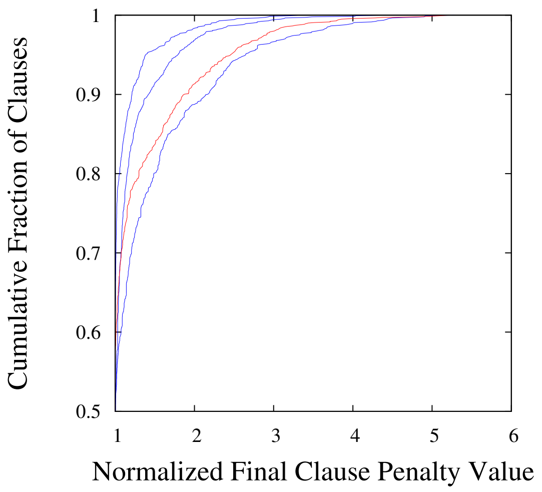
parameter settings produced a similar CPD (see Figure 4.7). The study of CPDs is of interest from an algorithm development perspective. If the CPD is the dominant factor in algorithm performance, then more efficient methods of achieving similar distributions can be explored. In this chapter, our focus is on what conclusions we can make from studying CPDs, instead of how the CPDs affect algorithm performance.

In Section 4.3, we compared the CPDs from different configurations of SAPS on the same instance, and we measured the clause penalty values after a fixed number of search steps for a meaningful comparison. In this section, we measure clause penalty values from different algorithms on multiple instances, so instead of measuring the penalty values after some arbitrary number of search steps, we measure them at the end of a search trajectory (*i.e.*, the final clause penalties). We measured the final clause penalties of SAPS, PAWS and DDFW from four different runs on the same instance, and present the results in Figure 5.9. From these figures we observe that the final CPDs do somewhat stabilize to a fixed distribution shape. We note that the vertical axis in Figure 5.9 and that over half of the clause penalties are either equal to one (unchanged) or very close to one. This observation is especially true for the PAWS algorithm, due to the aggressive decay mechanism in the default parameter settings.

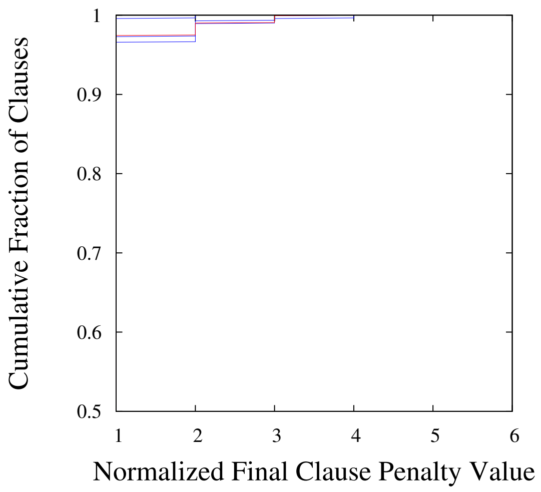
The clause penalties are heavily dependent on the instance, and in Figure 5.10 we explore the CPDs for SAPS on three different instances from anp10m (ais10, par8-1-c and qg1-07). The scales on the horizontal and vertical axes of these figures are different for the three instances, illustrating the difference in the shapes of the distributions. The distribution on instance par8-1-c shows that most of the clauses have been penalized and that the largest penalties approach values of 8, whereas the distribution on instance qg1-07 shows only a small fraction of the clauses have been penalized and the largest penalties approach values closer to 25. From these observations, and observations in our previous work, we suggest that the CPDs are very similar between runs, and are specific to the instance, the algorithm, and the algorithm settings.

In all instances we examined (not just those presented in Figure 5.9 and Figure 5.10) we observed that the CPDs seem to be qualitatively very similar between runs. To quantitatively summarize the distances between runs across anp10m, we present the data in Figure 5.11, where we measure the Kolmogorov-Smirnov distance[1] between the clause penalty distributions from the median and the longest runs. For PAWS, and to a lesser extent DDFW, we observe that most of the instances do tend to have final CPDs that are close, although there are some instances where the distance between the median and longest CPDs are significant. For SAPS, we observe that the distribution is flatter with more variation in the distances between CPDs. The distributions in Figure 5.11 suggest that it is difficult to quantitatively substantiate our qualitative observations for all of our algorithms and instances.

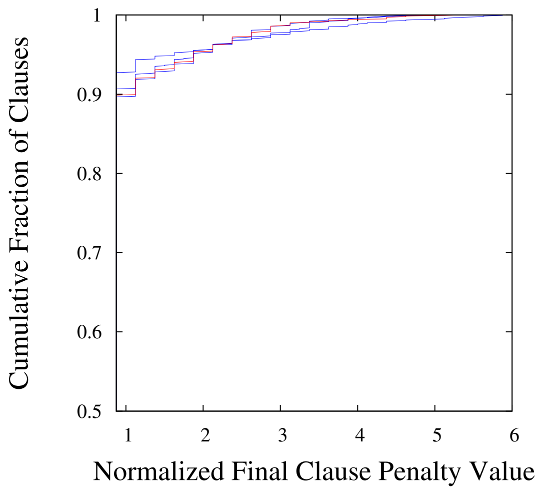
[1]We measure the distance between CPDs using the Kolmogorov-Smirnov distance function because it can be applied to cumulative distributions across different distribution sizes, as opposed to the Kullback Leibler divergence measure which cannot.



(a) SAPS

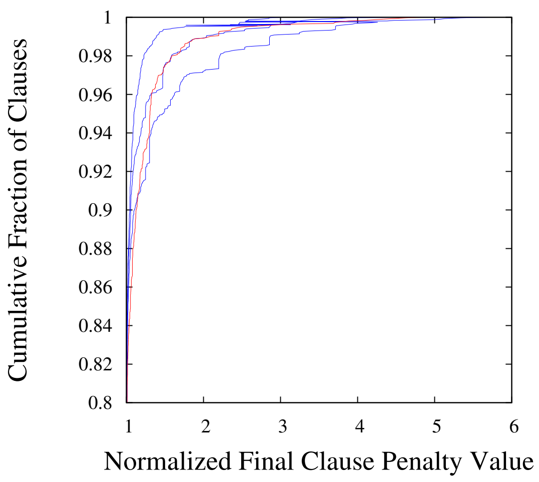


(b) PAWS

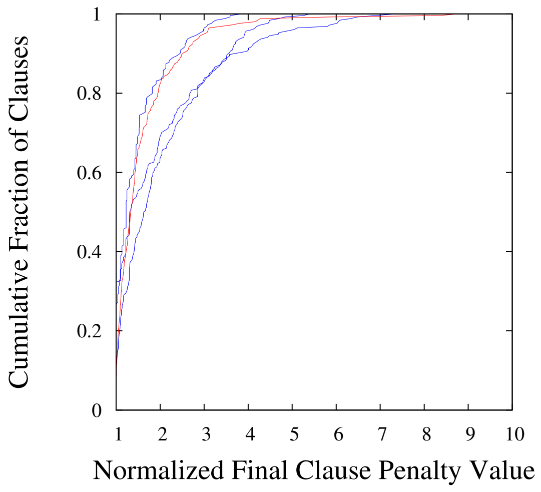


(c) DDFW

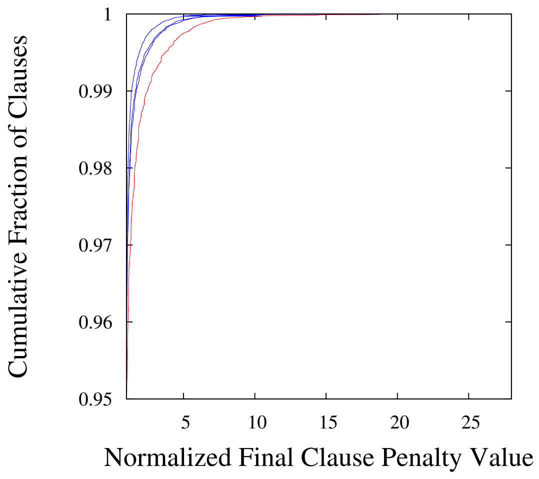
Figure 5.9: Final clause penalty distribution for flat125-94. For each clause in flat125-94 (1405 clauses), we measure the final clause penalty value. Clause penalty distributions from four runs are shown, corresponding to the $q_{0.24}$, median, $q_{0.76}$ and longest runs from an experiment with 25 runs. (The distribution from the longest run is highlighted. All algorithms were executed with default parameters as described in Appendix A. Clause penalty values were normalized, *i.e.*, the penalty value of an always satisfied clause is one.)



(a) ais10

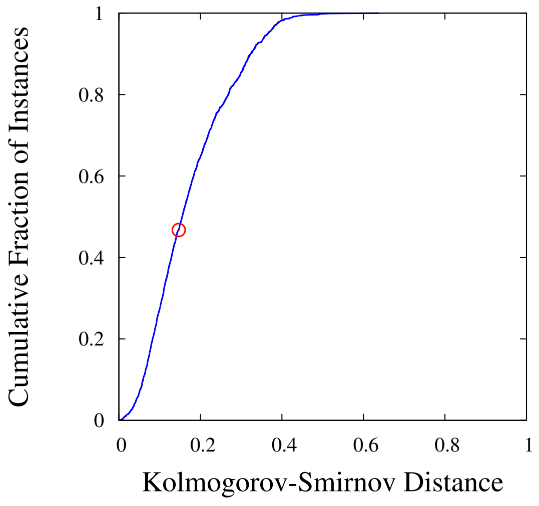


(b) par8-1-c

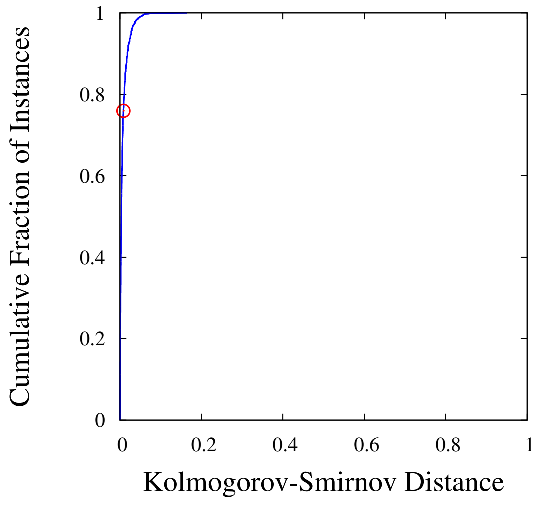


(c) qg1-07

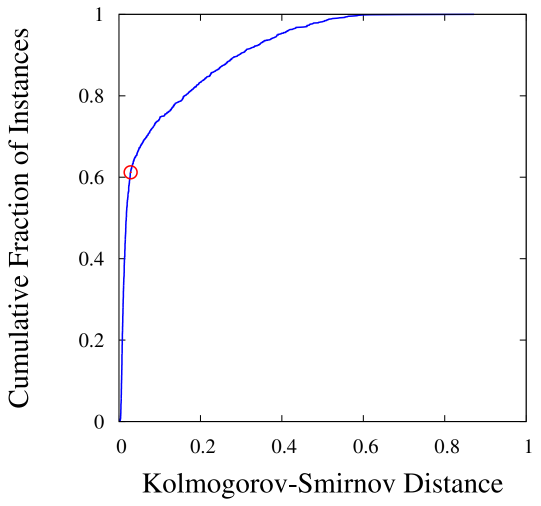
Figure 5.10: Additional SAPS final clause penalty distributions. Note that the axes ranges on each plot are different. Final clause penalties were collected from runs of SAPS on ais10 (3 151 clauses), par8-1-c (254 clauses) and qg1-07 (68 083 clauses). (see Figure 5.9 for details.)



(a) SAPS



(b) PAWS



(c) DDFW

Figure 5.11: The distance between final CPDs from two runs. For each instance in anp10m (1931 instances), we measure Kolmogorov-Smirnov distance between two CPDs (as described in Figure 5.9). (The runs selected were the median and the longest runs from 25 independent runs. The value corresponding to the instance flat125-94 is highlighted. All algorithms were executed with default parameters as described in Appendix A.)

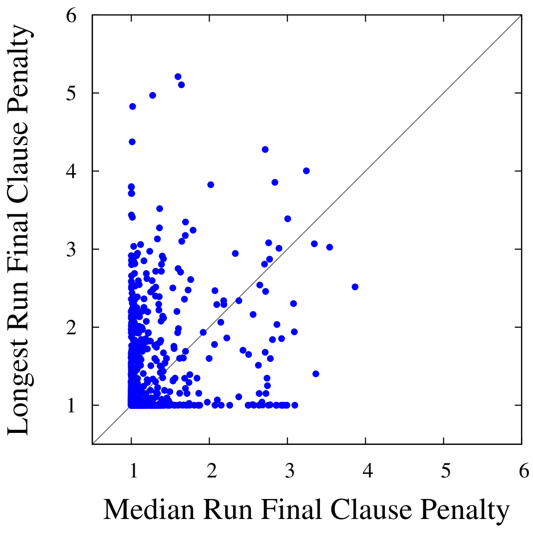
By observing these CPDs, we begin to see the larger picture of how clause penalties behave during search. At any given time during search there are a proportion of clauses that have been penalized more than the rest of the clauses. This could suggest that there are specific clauses that are problematic for instances, and that the number of them and the severity of how problematic they are is specific to the instance.

We have presented CPDs as cumulative distributions, abstracting away the values of individual clauses. We now turn our attention to the clause penalty values of individual clauses. To test for problem clauses, we first plot the final clause penalty values of one run against another run. In Figure 5.12, we observe that for most clauses there is a weak correlation between the final clause penalties of the two runs. The Pearson correlation coefficients between runs for SAPS, PAWS and DDFW are 0.53, 0.11 and 0.28. There are a few outlying clauses that are correlated, most of the final clause penalties show weak correlation. The lack of significant correlation shown in Figure 5.12 suggests that for these two particular runs on this particular instance there are no problem clauses, which does not support our hypothesis.

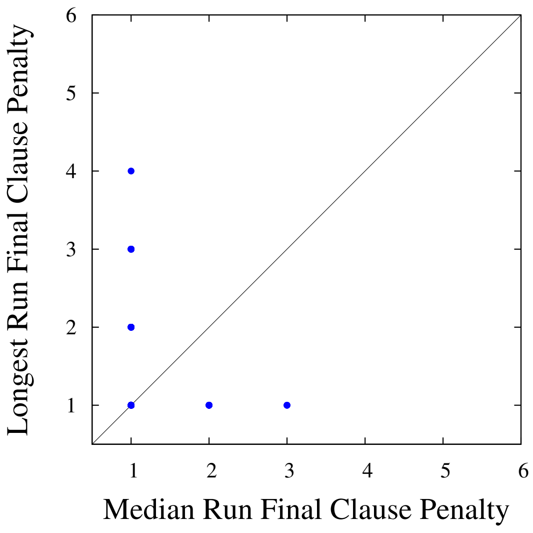
In Figure 5.6, we observed that throughout a search trajectory, clauses with high penalty values were quickly reduced. This suggests that sampling the clause penalties at any single point in time may not adequately capture problem clause behaviour. To better identify problem clauses, we measured the mean penalty value for each clause over a complete search trajectory. We reproduced our methodology used to generate the figures from Figure 5.12 using the mean clause penalty over a run instead of the final clause penalty value, and present the results in Figure 5.13. The Pearson correlation coefficients for SAPS, PAWS and DDFW are 0.94, 0.82 and 0.90, as shown in Figure 5.13. With the mean clause penalties we observe a much stronger correlation between the individual clause penalties, which suggests that DLS-CP algorithms can indeed identify problem clauses.

The results presented in Figure 5.12 indicate a low correlation in the final clause penalty values between two different runs, while Figure 5.13 suggests a very strong correlation between the mean clause penalty values from these same two runs. To measure this behaviour across anp10m, we measure the Pearson correlation coefficient[2] between the final clause penalty values from two runs (median and longest run) and the mean clause penalty values from those same two runs. In Figure 5.14, we observe that there is a much stronger correlation for the mean clause penalties than for the final clause penalties. The test instance (flat125-94) we have used throughout this section is the instance with median change in the correlation coefficient between the final and mean clause penalty values for SAPS. In other words, half of the instances in the set anp10m showed a stronger change in the correlation between the median and final clause penalties, whereas the other half did not.

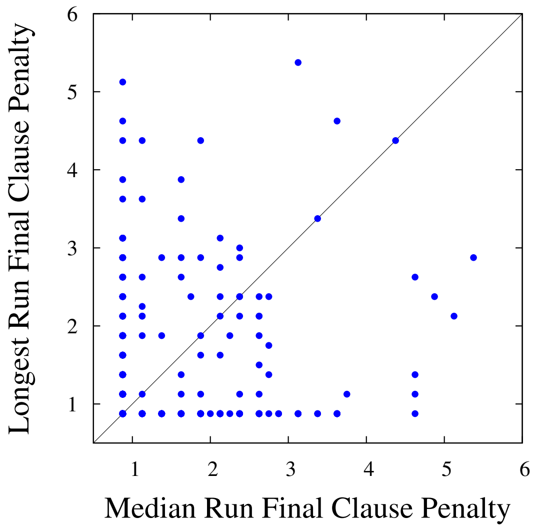
[2]For our analysis we generally prefer the Spearman rank correlation coefficient (ρ) over the Pearson correlation coefficient, but the ρ measure is problematic when there are multiple samples with equivalent values, such as in PAWS and DDFW final clause penalty values.



(a) SAPS
$r = 0.53$

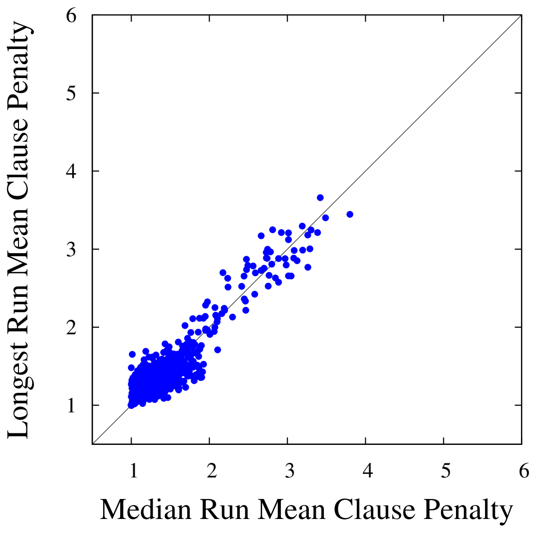


(b) PAWS
$r = 0.11$

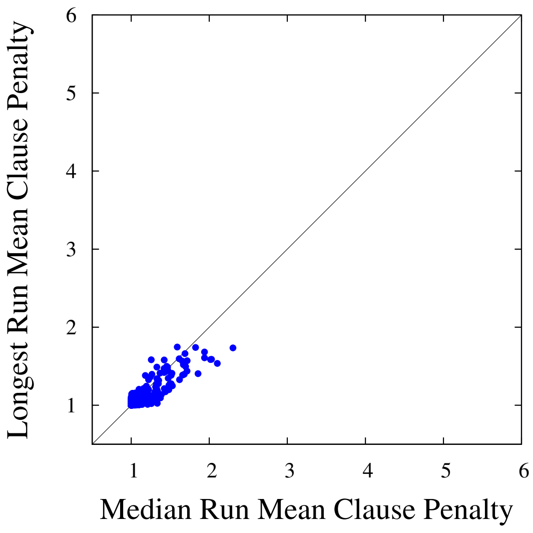


(c) DDFW
$r = 0.28$

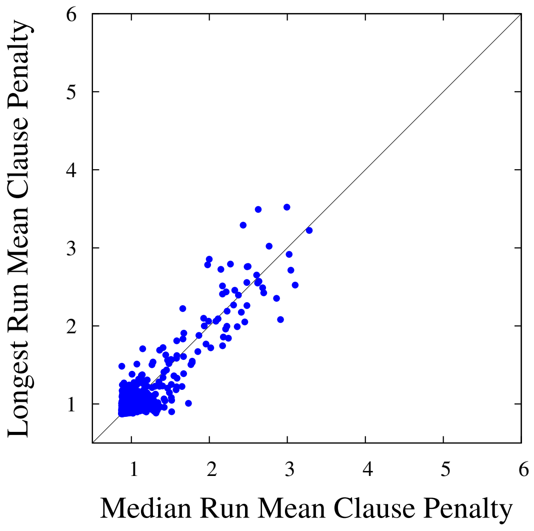
Figure 5.12: Final penalty values from two runs on flat125-94. For each clause in flat125-94 (1405 clauses), we measure the final clause penalty value for two independent runs. (Clauses with identical final penalty values for both runs are hidden (multiplicity). The runs selected were the median and the longest runs from 25 independent runs. The Pearson correlation coefficient (r) is shown for each data set above. All algorithms were executed with default parameters as described in Appendix A. Clause penalty values were normalized, *i.e.*, the penalty value of an always satisfied clause is one.)



(a) SAPS
$r = 0.94$

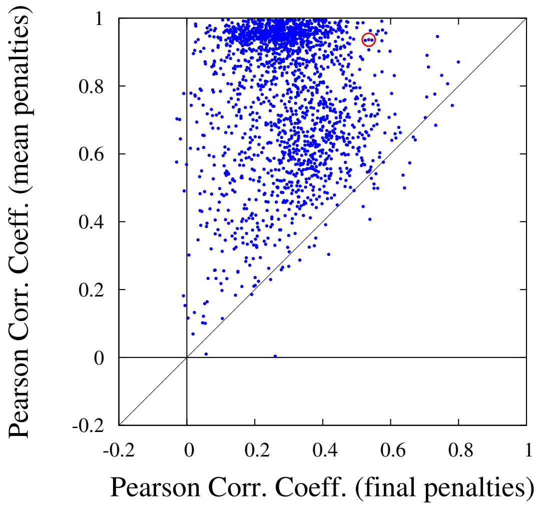


(b) PAWS
$r = 0.82$

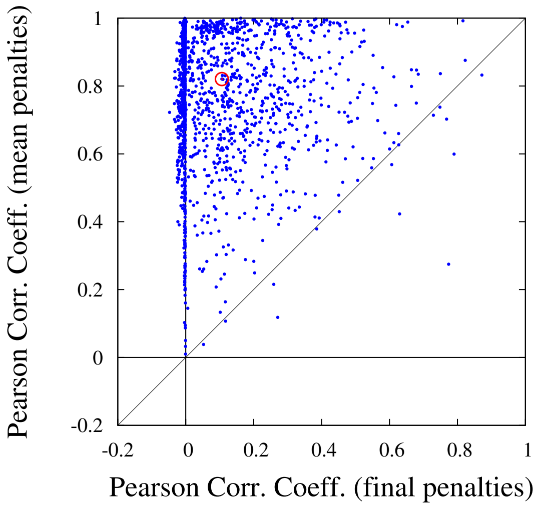


(c) DDFW
$r = 0.90$

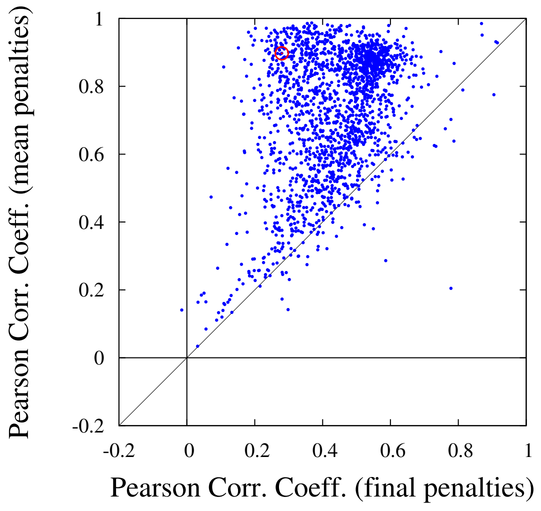
Figure 5.13: Mean penalty values from two runs on flat125-94. For each clause in flat125-94 (1405 clauses), we measure the mean clause penalty value over all search steps in two independent runs. (The runs selected were the median and the longest runs from 25 independent runs. The Pearson correlation coefficient (r) is shown for each data set above. All algorithms were executed with default parameters as described in Appendix A. Clause penalty values were normalized, *i.e.*, the penalty value of an always satisfied clause is one.)



(a) SAPS

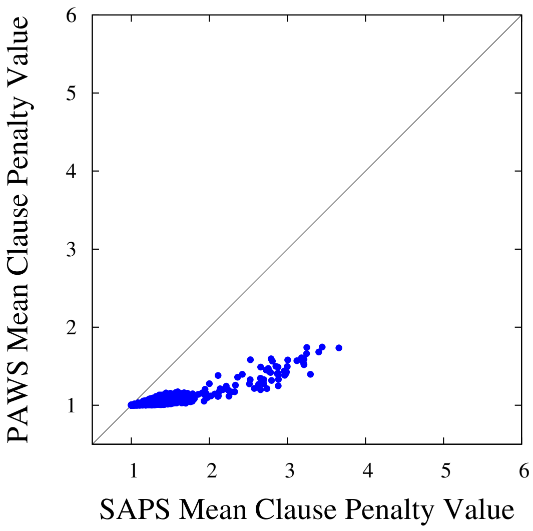


(b) PAWS

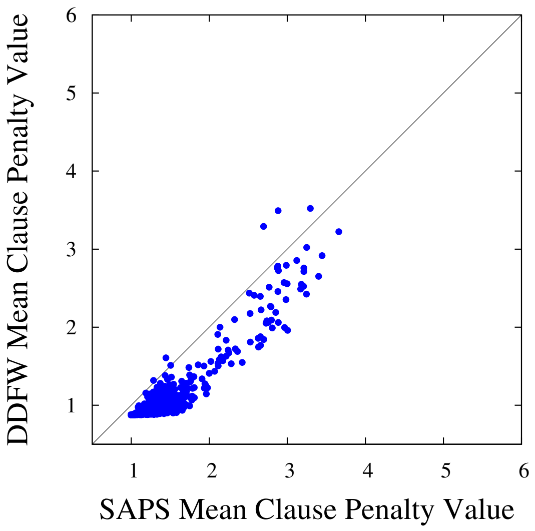


(c) DDFW

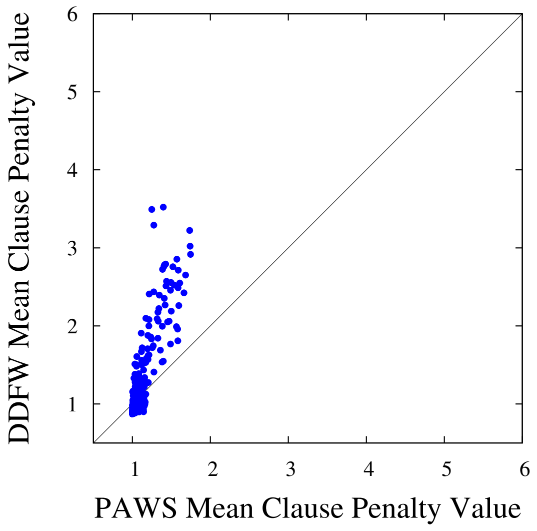
Figure 5.14: Correlation of penalty values from two runs: mean *vs* final. For each instance in anp10m (1931 instances), we measure the Pearson correlation coefficient of the final clause penalty values from two runs (as described in Figure 5.12) and the mean clause penalty values from two runs (as described in Figure 5.13). (The value corresponding to the instance flat125-94 is highlighted.)



(a) SAPS and PAWS
$r = 0.92$

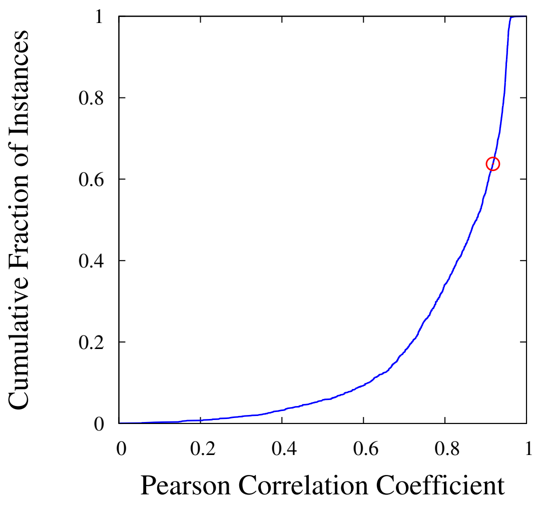


(b) SAPS and DDFW
$r = 0.91$

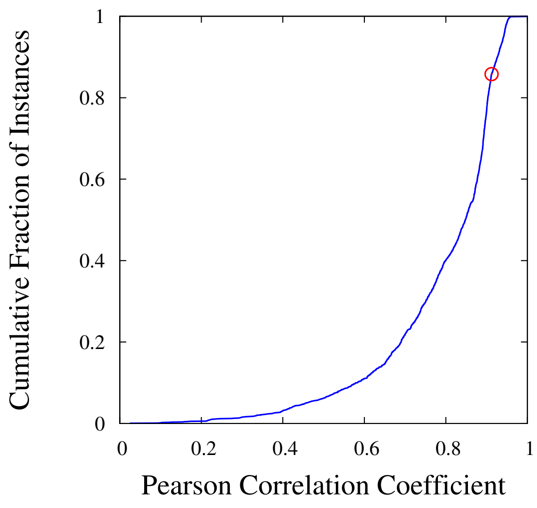


(c) PAWS and DDFW
$r = 0.89$

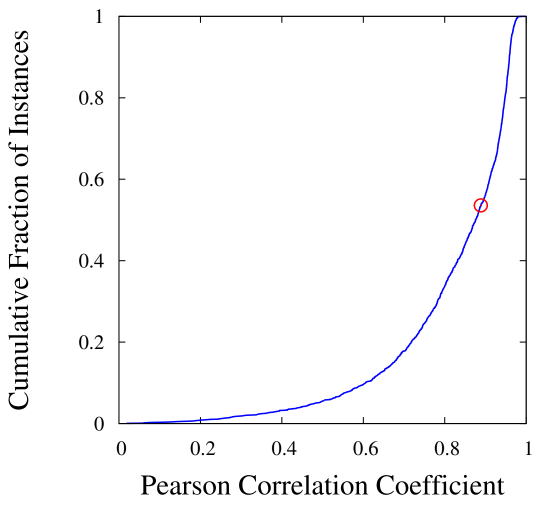
Figure 5.15: Mean penalty values from two different algorithms on flat125-94. For each clause in flat125-94 (1405 clauses), we measure the mean clause penalty value over all search steps in runs of two different algorithms. (The runs selected for each algorithm were longest run from 25 independent runs. The Pearson correlation coefficient (r) is shown for each data set above. All algorithms were executed with default parameters as described in Appendix A. Clause penalty values were normalized, *i.e.*, the penalty value of an always satisfied clause is one.)



(a) SAPS and PAWS



(b) SAPS and DDFW



(c) PAWS and DDFW

Figure 5.16: Correlation of mean penalty values from two algorithms on anp10m. For each instance in anp10m (1931 instances), we measure the Pearson correlation coefficient of the mean clause penalty values from runs of two different algorithms (as described in Figure 5.15). (The values from Figure 5.15 corresponding to the instance flat125-94 are highlighted.)

We have now seen evidence that for two different runs of the same algorithm there is a strong correlation between the mean clause penalty values. This suggests the presence of problem clauses, where the frequently penalized clauses are consistent from run to run. To explore whether or not two different algorithms contain the same problem clauses, we measured the mean clause penalties from a run (the longest run from 25 runs) on two different algorithms and measured the Pearson correlation coefficient between those penalties. We repeated this for all instances in anp10m and present the distribution of those correlation coefficients in Figure 5.16. In this figure, we observe that there are many instances with a very high correlation in the mean clause penalties between two different algorithms. This final piece of evidence strongly supports our hypothesis that instances have problem clauses and that DLS-CP algorithms are identifying those clauses by penalizing them more frequently.

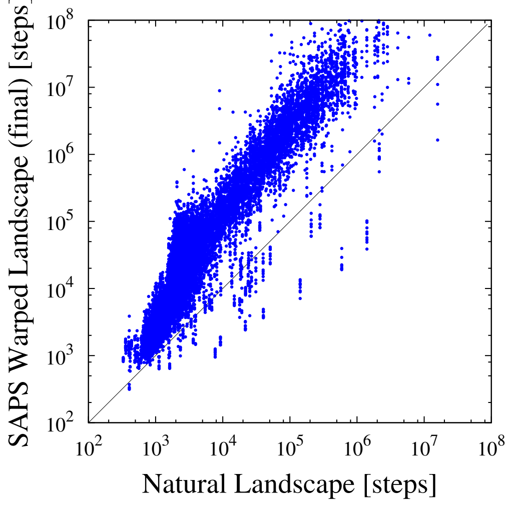
5.4 Warped Landscapes

With DLS-CP algorithms for SAT, the penalized search landscape of a SAT instance is continuously warped throughout the search trajectory. The landscape changes because each clause is assigned a dynamic penalty value, which affects the evaluation function, and thus changes the height of points in the search space. This process is typically repeated numerous times throughout the search, warping the landscape as it progresses.

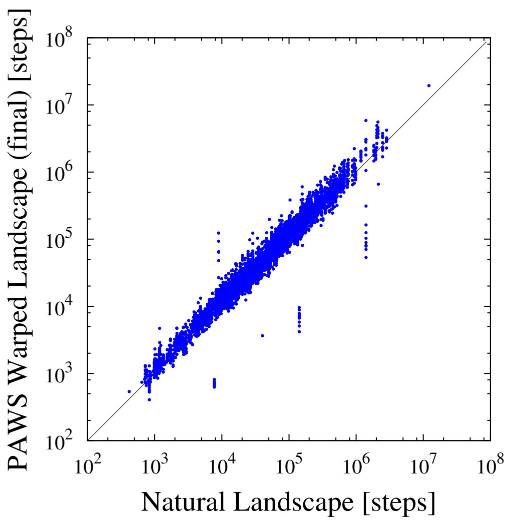
It has been suggested that the success of DLS-CP algorithms is a result of the fact that the clause penalties represent accumulated 'learned' knowledge about the search space [30]. In particular, the hypothesis that the resulting warped landscape will be easier to search than the original space appears to be widely accepted, even though there is little evidence to support it. Often, this reflects back to a popular analogy that a DLS-CP algorithm 'fills in the holes' (*i.e.*, the local minima), of a given search landscape. To investigate the validity of this proposed explanation, we look at the hardness of the landscapes generated by DLS-CP algorithms. We argue that if the warped landscapes generated by the DLS-CP algorithms are indeed easier to search, then the algorithms are warping the landscapes in an intelligent way. Moreover, this is a critical factor in explaining the efficiency of DLS-CP algorithms.

To study these warped landscapes, we generate (statically) weighted instances, where the clause weights are taken directly from the clause penalties of the DLS-CP algorithm. We then analyze these weighted instances to determine if the corresponding warped landscape is indeed easier to search.

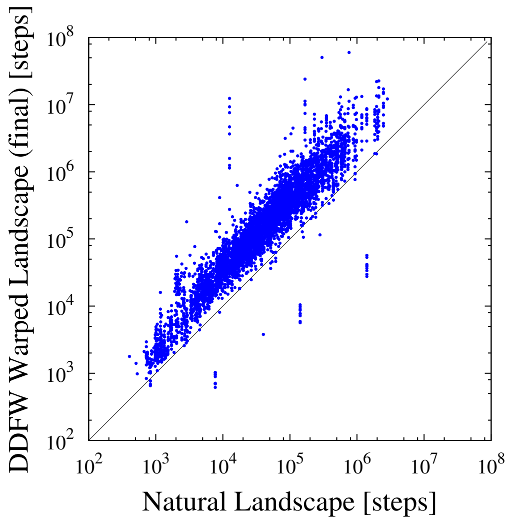
A DLS-CP algorithm starts with the natural landscape and ends with the warped landscape that was in place when the solution to the instance was found. Because this final landscape helped guide the algorithm to the final solution, we look at the hardness of these final landscapes. If the hypothesis that the algorithm is learning as it searches is true, then this final warped landscape will encapsulate all of the information the algorithm had learned when it found the solution.



(a) SAPS

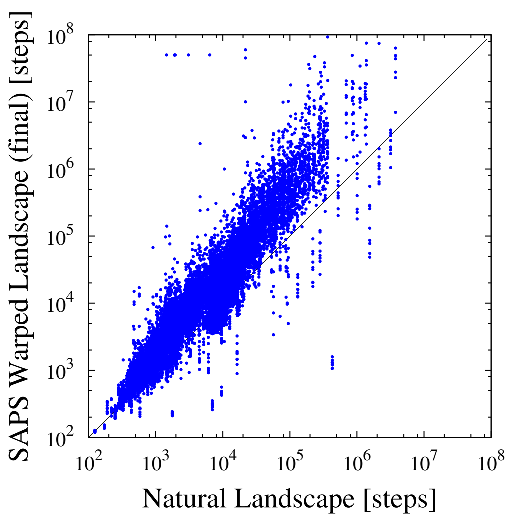


(b) PAWS

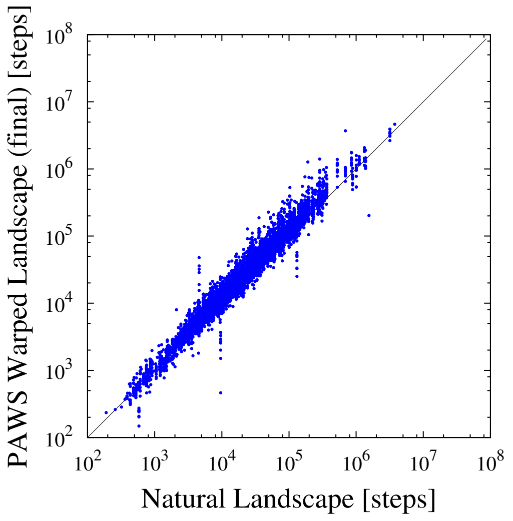


(c) DDFW

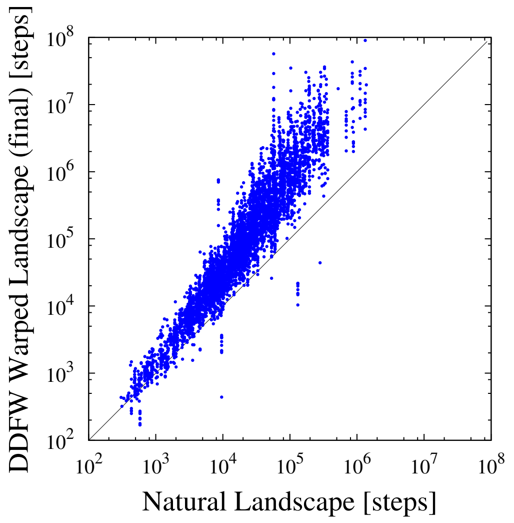
Figure 5.17: WEIGHTED ADAPTIVE NOVELTY$^+$: natural *vs* warped landscapes (final penalty values). For each instance in anp10m (1931 instances), we measure the run-length performance of WEIGHTED ADAPTIVE NOVELTY$^+$ on the original instance (natural landscape) and nine weighted instances (warped by DLS-CP algorithms). (The weights for the nine weighted instances were obtained from the final clause penalty values from nine runs. The nine runs correspond to the $(q_{0.10}, q_{0.20}, \ldots q_{0.90})$ runs from 100 independent runs. Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)



(a) SAPS



(b) PAWS



(c) DDFW

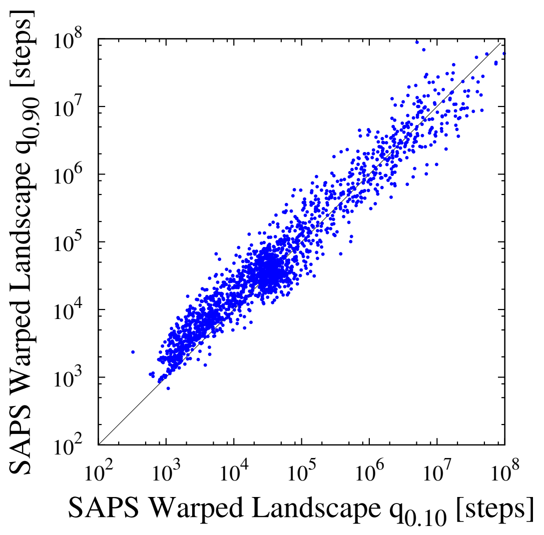
Figure 5.18: WEIGHTED G^2WSAT: natural *vs* warped landscapes (final penalty values). The run-length performance of WEIGHTED G^2WSAT is measured, otherwise see Figure 5.17.

In Figure 5.17 and Figure 5.18, we present the results from running WEIGHTED ADAPTIVE NOVELTY$^+$ and WEIGHTED G^{2}WSAT on the weighted instances generated from the final clause penalties from our three DLS-CP algorithms. As demonstrated in Section 5.3, the final clause penalties are often uncorrelated, so we repeated each experiment nine times, taking the final clause penalties from a different run each time. In Figure 5.17 and Figure 5.18 we observe that, in general, the final DLS-CP algorithm-warped landscapes are actually *harder* than the natural landscapes; however, there are exceptions. In Section 5.2, we demonstrated that the ais instances become significantly easier for WEIGHTED ADAPTIVE NOVELTY$^+$ when there are larger weights assigned to the identified problem clauses. While searching, the DLS-CP algorithms seem to quickly adopt high penalty values for the problem clauses, and as such, the resulting landscape becomes significantly easier. In addition to all of the ais instances, this phenomenon occurs for all of the qg instances and for three quarters of the ii instances. Together, those three instance types account for all of the statistically significant outliers seen by all three algorithms. The results of WEIGHTED G^{2}WSAT on PAWS- and DDFW-warped landscapes were very similar to those of WEIGHTED ADAPTIVE NOVELTY$^+$. However, WEIGHTED G^{2}WSAT on the SAPS-warped landscapes also produced statistically significant outliers on two of the bw-large instances, approximately half of the runs on parity instances, less 7% of the runs on the swgcp instances and less than 3% of the runs on the flat instances. These exceptions do not detract from our original observation that the warped landscapes are typically harder. From this evidence we conclude that making the landscapes easier cannot be an essential ingredient in the success of DLS-CP algorithms, since it only occurs under rare circumstances.

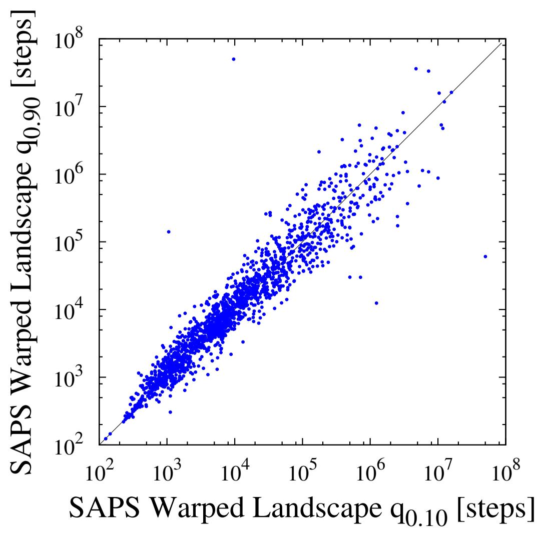
To further explore this idea of making the landscapes easier, we will speculate about the effect of run-length on the resulting warped landscape. If a DLS-CP algorithm is learning as it progresses, one could conclude that the longer the algorithm ran, the easier the landscape should become. Conversely, one could argue that shorter runs might have easier landscapes because those landscapes helped to find a solution faster. To test if there is any correlation between the length of the SAPS run and the hardness of its landscape, we compared the performance of WEIGHTED ADAPTIVE NOVELTY$^+$ and WEIGHTED G^{2}WSAT on SAPS-generated landscapes from short runs and long runs. We present the results in Figure 5.19, where we observe that there is no clear bias for shorter runs to be easier or harder than longer runs. Equivalent observations were also made for PAWS and DDFW.

We have presented evidence that, for most instances, the final SAPS landscapes do not make the instance easier than the natural landscape. While we have demonstrated that the landscapes were not becoming easier in a global sense, we have not considered the possibility that these landscapes were becoming easier in a local sense. To explore this concept we introduce the concept of a *solution basin*.

If a run of a local search algorithm is fortuitously initialized at an actual solution of the given



(a) WEIGHTED ADAPTIVE NOVELTY$^+$



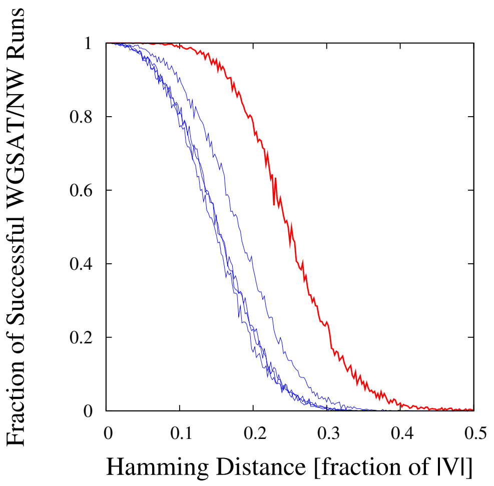
(b) WEIGHTED G^{2}WSAT

Figure 5.19: Instances weighted by penalties from short *vs* long runs. For each instance in anp10m (1931 instances), we measure the run-length performance of WEIGHTED ADAPTIVE NOVELTY$^+$ on two SAPS-weighted instances. (The weights for the weighted instances were obtained from the final clause penalty values from the $q_{0.10}$ and $q_{0.90}$ runs from 100 independent runs. Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)

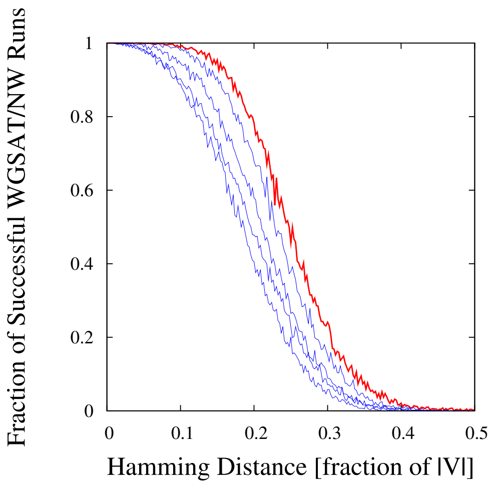
problem instance, then the algorithm will terminate immediately. When an algorithm is close to a solution, it can be drawn directly toward the solution with a very high probability. However, if the algorithm is initialized too far from the solution it will be pulled to numerous directions and there is a much lower probability that it will proceed directly to a solution. We refer to this phenomenon as the *probabilistic solution basin of attraction* for an instance or just solution basin for short.

We suggest that if one instance (or by extension, landscape) is easier than another, one reason could very well be that it has a larger (or wider) solution basin. Although many practical SAT instances have multiple solutions, in this section we only study instances that have a single solution, and therefore a single basin. This will make our analysis easier to interpret and avoid the confusion of competing solution basins. Our solution basins are not just instance specific, but are also algorithm specific and depend heavily on the greediness (or intensification) of the algorithm. For example, the URWALK algorithm has a much narrower basin for an instance than a greedier algorithm such as WALKSAT/SKC.

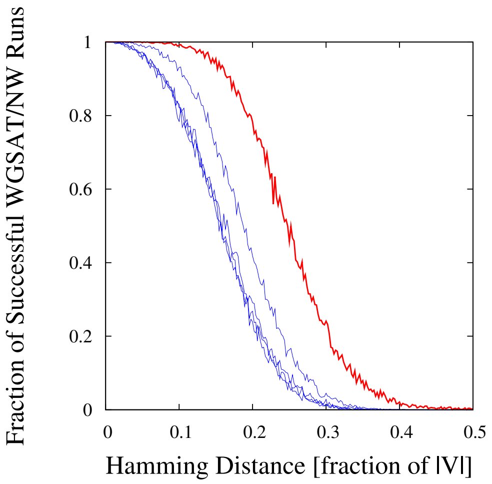
To measure the size and shape of a solution basin on a single solution instance, we use the



(a) SAPS

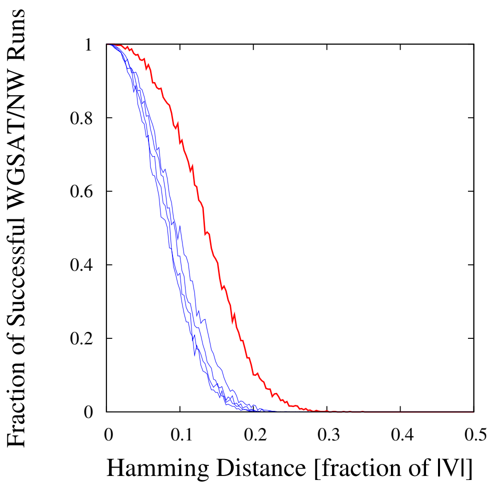


(b) PAWS

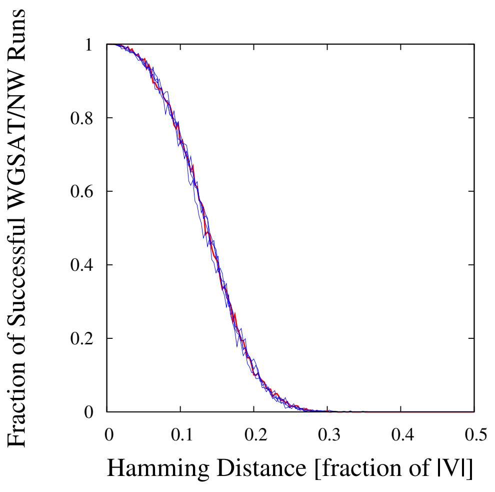


(c) DDFW

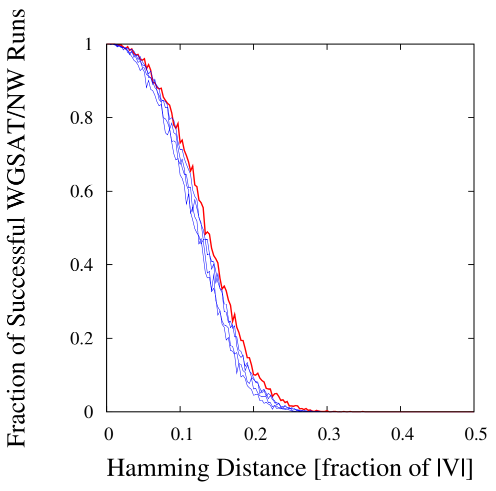
Figure 5.20: Probabilistic basins of attraction for bw-large-a. Each plot shows the probabilistic basin of attraction corresponding to the natural (not weighted) instance highlighted in addition to four probabilistic basins of attraction corresponding to weighted instances from the $q_{0.30}$, median, $q_{0.70}$ and $q_{0.90}$ runs from 100 runs of the DLS-CP algorithms, as described in Figure 5.17. The fraction of successful WEIGHTED GSAT/NW runs for a Hamming distance d is determined by measuring the number of successful runs from 1 000 runs of the WEIGHTED GSAT/NW algorithm when it is randomly initialized with a variable assignment that has d variables differing from the solution to the instance. The d variables are independently determined for each run, and the Hamming distance is normalized by the number of variables in the instance ($|V|$). All algorithms were executed with default parameters as described in Appendix A.



(a) SAPS



(b) PAWS



(c) DDFW

Figure 5.21: Probabilistic basins of attraction for par8-5. The instance is par8-5, otherwise see Figure 5.20.

GSAT/NW algorithm, where no worsening steps are allowed. The GSAT/NW algorithm is ideal for this analysis because it mimics the behaviour of our three DLS-CP algorithms on a fixed landscape. We empirically sample the probability that the GSAT/NW algorithm will reach the solution from a given initial Hamming distance from the solution. This is a probabilistic sample of the underlying basin of attraction for this specific algorithm. In Figure 5.20 and Figure 5.21, we present the GSAT/NW solution basins for three different algorithms on two different single solution instances from anp10m. From this figure we observe the general shape of the GSAT/NW solution basins is what we would expect: when very close to the solution, nearly all runs are successful, and the proportion of successful runs drops as the Hamming distance increases. The more interesting observation from Figure 5.20 is that the basins obtained from the warped landscapes narrow rather than widen. This provides conclusive evidence that the landscapes do not make the instance easier in a local sense.

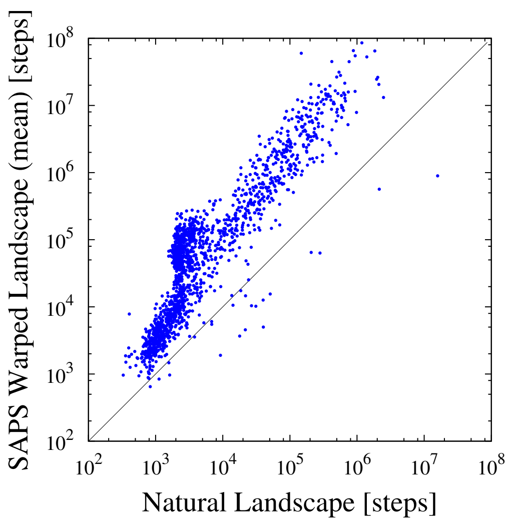
The work presented in the section so far has been based on the final clause penalties from search trajectories of DLS-CP algorithms. However, in Section 5.3, we saw that the mean clause penalties between runs were more strongly correlated than the final clause penalties, which we interpreted as evidence for the existence of problem clauses.

In Figure 5.22 and Figure 5.23 we present the results from running WEIGHTED ADAPTIVE NOVELTY$^+$ and WEIGHTED G^{2}WSAT on the weighted instances generated by taking the mean clause penalties from a run of our DLS-CP algorithms. As can be seen from the figure, the results are very similar to those in Figure 5.17 and Figure 5.18, where most of these new weighted instances are *harder* than the ones generated from final clause penalties. These weighted instances are artificial, in that the landscapes of these instances are essentially an aggregate of numerous individual landscapes. The individual DLS-CP algorithms never actually encounter these mean landscapes during their search, so they cannot exploit these landscape changes. However, this experiment demonstrates that, in practice, identifying the problem clauses observed by DLS-CP algorithms and weighting the instance does not make the instance easier.

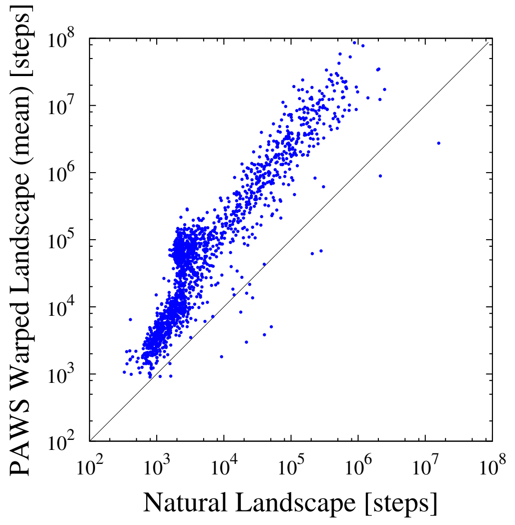
Overall, based on our empirical results, there is no evidence to support the hypothesis that the warped landscapes generated by DLS-CP algorithms are easier to search for any reasonably powerful SLS algorithm. Hence, the clause penalties determined over successful runs of a DLS-CP algorithm do not reflect any general knowledge on how to solve the given problem instance more efficiently.

5.5 Long-Term Memory

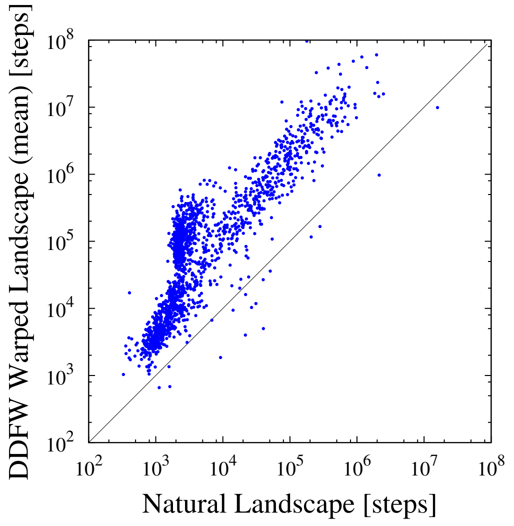
In Chapter 2, we explained how in the SLS literature, the *duration* or scope of historical information maintained by an SLS algorithm is expressed in terms of memory in the traditional use of the word (*e.g.*, short-term and long-term memory). In this section, we examine how long-term memory can affect the performance of DLS-CP algorithms. In Section 4.3, we established that the penalty values



(a) SAPS

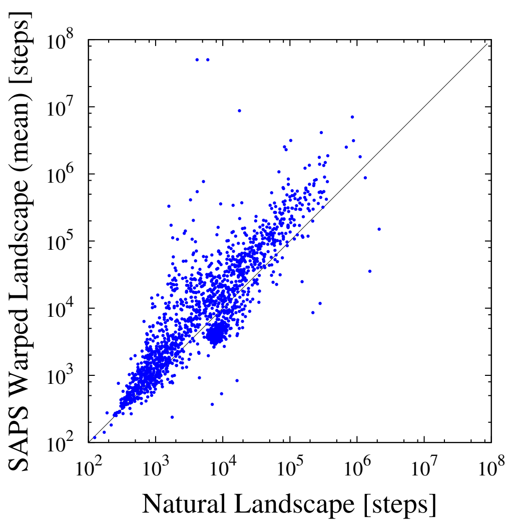


(b) PAWS

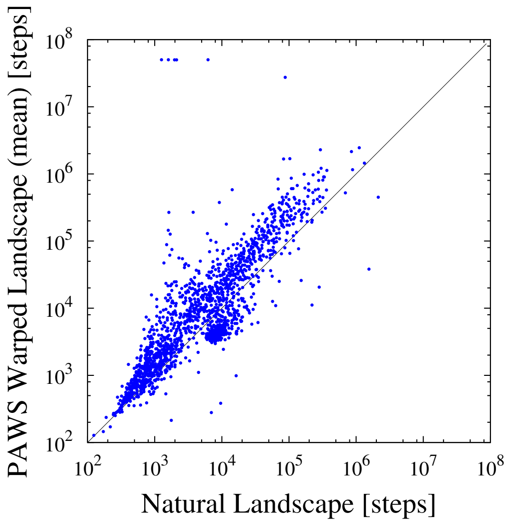


(c) DDFW

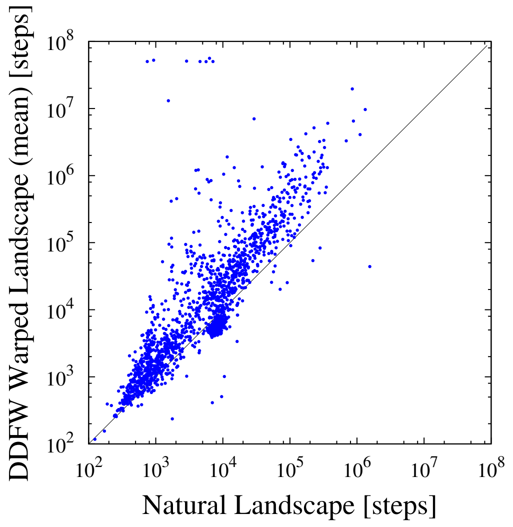
Figure 5.22: WEIGHTED ADAPTIVE NOVELTY$^+$: natural *vs* warped landscapes (mean penalty values). For each instance in anp10m (1931 instances), we measure the run-length performance of WEIGHTED ADAPTIVE NOVELTY$^+$ on the original instance (natural landscape) and a weighted instances (warped by a DLS-CP algorithm). (The weights for the weighted instances were obtained from the mean clause penalty values over the longest run from 25 independent runs. Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)



(a) SAPS



(b) PAWS



(c) DDFW

Figure 5.23: WEIGHTED G^{2}WSAT: natural *vs* warped landscapes (mean penalty values). The run-length performance of WEIGHTED G^{2}WSAT is measured, otherwise see Figure 5.22.

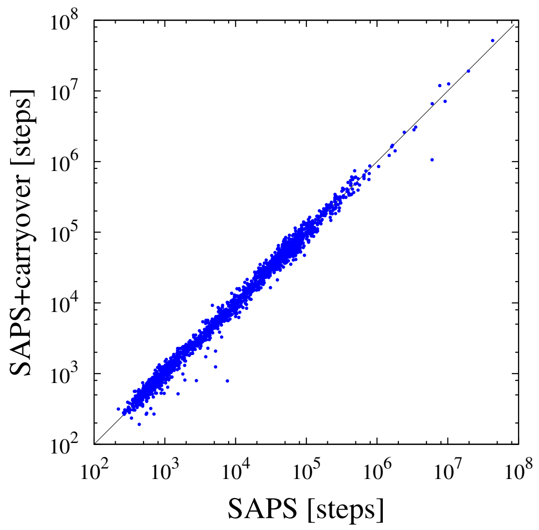
of SAPS can capture very long-term memory, since the smoothing never completely erases any of the past behaviour. Conversely, when PAWS decrements its penalty values, it is truly erasing past history. It is more difficult to characterize DDFW in such a manner, but we expect that it can retain information much longer than PAWS.

In Section 5.4, we examined how the non-DLS-CP algorithms, WEIGHTED G^2WSAT and WEIGHTED ADAPTIVE NOVELTY$^+$, performed on warped landscapes generated by DLS-CP algorithms. We observed that for most instances these landscapes were harder and no advantage could be achieved by searching the warped landscapes instead of the natural landscapes. While we have demonstrated that the non-DLS-CP algorithms do not typically benefit from this extra information, the question arises whether DLS-CP algorithms themselves could.

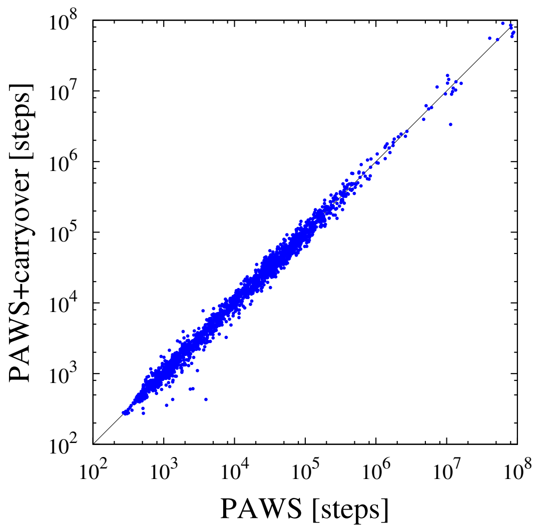
To explore this question, we developed +CARRYOVER variants. a +CARRYOVER variant of a DLS-CP algorithm retains all of the clause penalty values between successive runs and essentially extends the long-term memory of the algorithm. The runs of the +CARRYOVER variants are not independent, and in practice all that occurs between successive runs is that after a solution is found the variable assignment is re-initialized to a random assignment.

In Figure 5.24, we present the results of these new variants on anp10m. For most instances there is no statistically significant difference between the run-length performance of the +CARRYOVER variants when compared to the original algorithm. However, there are significant outliers for all three variants, and DDFW+CARRYOVER has the most. The run-length performance of all three +CARRYOVER variants was better on approximately half of the inductive inference ii instances, especially for DDFW+CARRYOVER. The performance improvement for both SAPS+CARRYOVER and DDFW+CARRYOVER on one of the bitadd instances was very significant. The performance of both SAPS+CARRYOVER and PAWS+CARRYOVER was better on a few of the flat instances. For DDFW+CARRYOVER the performance was worse on a few instances, the most significant being one of the gcp instances, all of the instances from the bw-large set, and one of the ii instances. Aside from these few exceptions, it would seem that extending the long-term memory is not helping for DLS-CP algorithms. These results are not overly surprising if we recall the behaviour of individual clause penalties demonstrated in Figure 5.6, where we saw that clause penalties are smoothed back down very quickly. This further suggests that clause penalties in DLS-CP algorithms are important locally, but not globally.

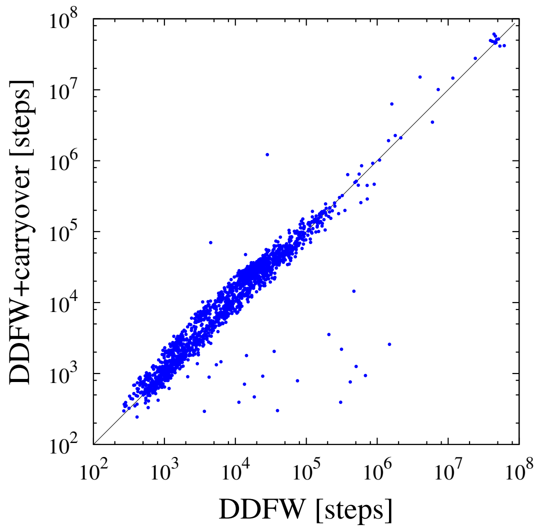
We have demonstrated that extending long-term memory does not improve the performance of a DLS-CP algorithm on most instances. We now explore what happens if the algorithm's long-term memory is reduced. To further study the effects of long-term memory on DLS-CP algorithms, we developed +AMNESIA variants of our DLS-CP algorithms. An +AMNESIA variant of a DLS-CP algorithm will periodically reset all of the clause penalties back to their initial values (*i.e.*, they will 'forget' all of the clause penalty information they have learned). Conceptually and in practice, an +AMNESIA variant simply performs periodic restarts, with the exception that the current variable



(a) SAPS



(b) PAWS



(c) DDFW

Figure 5.24: Original *vs* +CARRYOVER variants on anp10m. (Each point corresponds to an instance in the set anp10m (1931 instances). Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)

assignment is not changed when the restart occurs. +AMNESIA variants can also be thought of as algorithms with very aggressive (complete) periodic smoothing. We developed these variants to demonstrate that long-term memory does not play a critical role in the performance of current state-of-the-art DLS-CP algorithms. If long term memory were important to the success of the algorithm, then the performance of +AMNESIA variants should be substantially poorer than the original DLS-CP algorithm.

When implementing an +AMNESIA variant of a DLS-CP algorithm, we must establish the *duration* of the long-term memory (*i.e.*, the number of search steps that occur between periodic resets). A short duration may cripple the algorithm to the point that it cannot escape local minima. A long duration may not substantively affect the algorithm. We selected two durations of $3 \cdot |V|$ and $10 \cdot |V|$ search steps where $|V|$ is the number of variables in the instance. The choice of $3 \cdot |V|$ was chosen to be consistent with the restart mechanism in SCHÖNING'S ALGORITHM [94], and the $10 \cdot |V|$ parameter was arbitrarily selected for comparison.

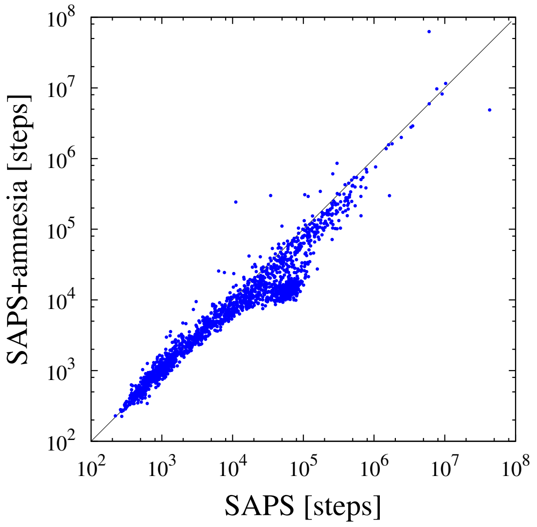
In Figure 5.25, we present the results of our experiment with a duration of $3 \cdot |V|$. For a large number of the instances in anp10m the amnesia had no effect or marginally improved the performance of the algorithm. Since the default parameter setting of the PAWS algorithm has aggressive smoothing, the amnesia did not affect it significantly. SAPS+AMNESIA performed poorly on a few statistically significant outliers, including all of the parity and qg instances, two of the bitadd instances, and a sixth of the ii instances. DDFW+AMNESIA was significantly worse on a quarter of the instances. This suggests that DDFW could be more reliant on long term memory, more sensitive to smoothing schedules than SAPS and PAWS, or that it requires more search steps to penalize the clauses necessary to escape from local minima. We increased the duration to $10 \cdot |V|$ and present the results in Figure 5.26, where the performance of the DDFW+AMNESIA significantly improved. However, there were still many significant outlier instances where DDFW+AMNESIA performed worse than regular DDFW, including most of the instances from the bitadd, ais, clus-1200, ii, logistics, parity and qg sets.

In this section we have demonstrated that while DLS-CP algorithms can have long-term memory and accumulate large amounts of information on the instance and the search landscape, typically this information is not helpful to the algorithm during the search, and is not the primary mechanism responsible for how DLS-CP algorithms find solutions effectively.

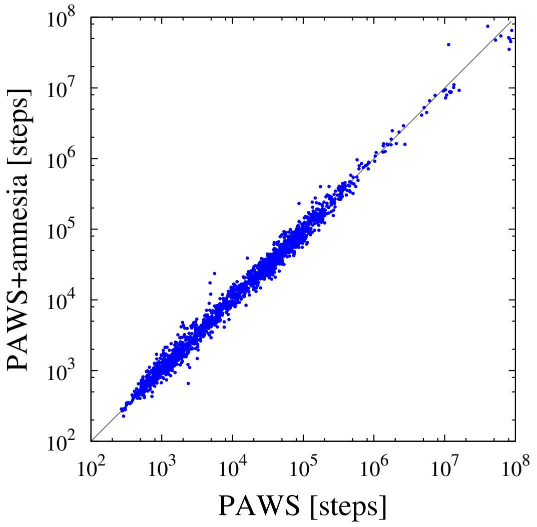
5.6 Related Work

The literature on search landscapes for combinatorial problems is quite extensive, with Stadler's work [106] often cited, and a thorough study available in the book by Hoos and Stützle [55: ch. 5].

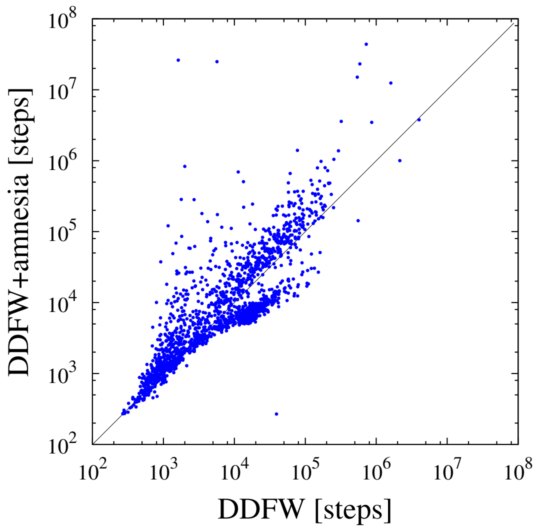
In Section 5.4, we introduced our notion of a probabilistic solution basin of attraction, which is closely related to the concept of a solution basin mentioned by Cheeseman *et al.* [15] and explored in depth by Flamm *et al.* [27]. Prestwich and Roli [90] independently developed a conceptual model



(a) SAPS

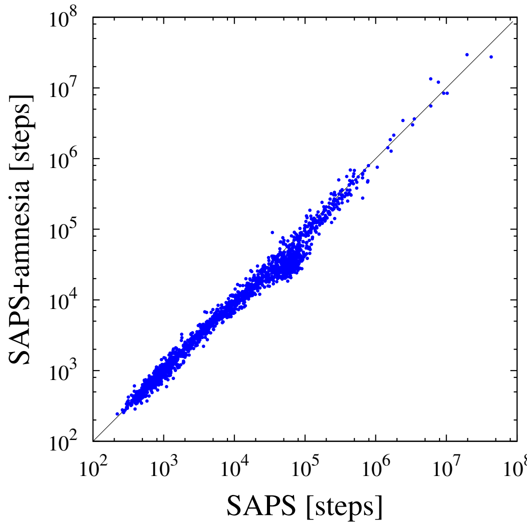


(b) PAWS

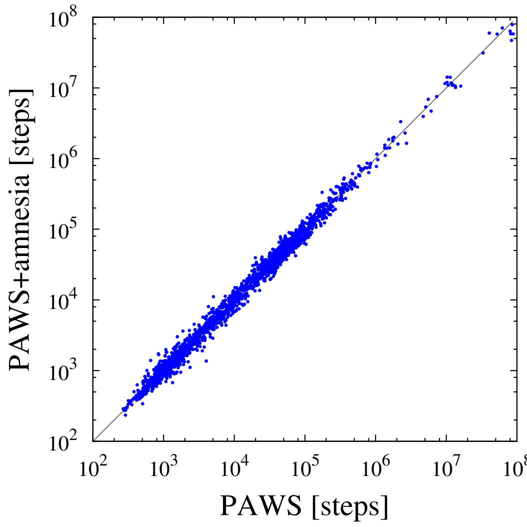


(c) DDFW

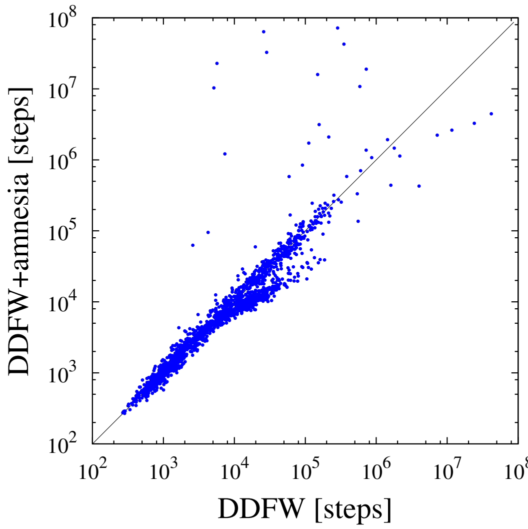
Figure 5.25: Original *vs* +AMNESIA variants $(3 \cdot |V|)$ **on anp10m.** (Each point corresponds to an instance in the set anp10m (1931 instances). Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)



(a) SAPS



(b) PAWS



(c) DDFW

Figure 5.26: Original *vs* +AMNESIA variants $(10 \cdot |V|)$ on anp10m. (Each point corresponds to an instance in the set anp10m (1931 instances). Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)

of solution basins very similar to our approach. They used solution basins to measure the change in hardness that occurs when instances are modified with symmetry breaking.

We became aware of Ferreira Jr. and Thornton's PAWS WITH USUAL SUSPECTS (PAWS+US) algorithm variant after we had conducted the experiments in this chapter. The PAWS+US algorithm is given additional information *a priori* about which clauses are so-called *usual suspects*, an analogous concept to the problem clauses we describe throughout this chapter. The PAWS+US algorithm treats the usual suspect clauses differently by increasing their penalty value at the initialization stage and increasing the clauses by a greater amount during the penalty update stage. Despite their methodology being different from our own, they observed similar behaviour, in that including extra clause information only significantly improved performance on a small number of instances. In their experiments, the bw-large instances were the most significant. Interestingly, they included the ais10 instance we studied in Section 5.2 in their experiments, but they did not observe any improvement in performance on those algorithms by identifying the usual suspect clauses.

5.7 Conclusions

The primary objective of this chapter was to explore how DLS-CP algorithms solve SAT instances and to answer the question of why they are effective. Based on our experimental results, we can rule out a number of plausible hypotheses and arrive at a somewhat surprising explanation.

In Section 5.4, we observed that for the vast majority of instances, the warped landscapes created by DLS-CP algorithms are no easier to search than the unweighted (natural) landscapes for the respective problem instances. Even if the DLS-CP algorithms we studied could search on aggregated landscapes reflecting the cumulative clause penalty information learned throughout the search, it appears that for most types of SAT instances, they would likely not benefit from this information. We have established that in practice, for most instances, a DLS-CP algorithm does not achieve success by manipulating its clause penalties in an intelligent manner. We observed that even when a DLS-CP algorithm was very close to a solution of a given problem instance, it was not the clause penalties themselves that led the algorithm to the solution. It is clear that while clause penalties can lead algorithms away from local minima, they do not lead algorithms toward solutions.

In Section 5.5, we demonstrated that long-term memory is not a vital ingredient to the success of a DLS-CP algorithm. Providing a DLS-CP algorithm *a priori* with the clause penalty information from an independent successful run typically does not help the algorithm to find a solution more effectively. This suggests that any information embedded in the long-term memory of a DLS-CP algorithm is not useful to the algorithm. To further explore this topic we created amnesic variants of the algorithms and demonstrated that removing the long-term memory typically does not affect performance. Clearly, the success of DLS-CP algorithms cannot be explained by the presence of long-term memory. If short-term memory were to be completely eliminated, most DLS-CP algorithms would be reduced to simple iterative improvement procedures and hence become highly susceptible to local minima. This suggests that short-term memory plays an important role in rendering DLS-CP effective.

Considering all the evidence presented in this chapter, it seems likely that the short-term memory represented by the clause penalties of DLS-CP algorithms is primarily useful for escaping minima in practice. In other words, DLS-CP algorithms are not successful because they utilize any guidance towards solutions, but because they are effective at escaping from non-solution areas of the search landscape.

Chapter 6

Random Decisions

It's action… reaction… random interaction.
So who's afraid of a little abstraction? Can't get no satisfaction.
— Rush. "Roll the Bones"

In Chapter 4, we observed that the variable selection mechanism in our SAPS algorithm was essentially deterministic for long search trajectories (an observation also made by Schuurmans and Southey on their SDF algorithm [95]). In SAPS, it would appear that the idiosyncrasies of the long search trajectory captured in the history (long-term memory) of the clause penalty values is an adequate substitute for the random decisions normally employed by SLS algorithms. For us, this raised many interesting questions regarding the role of random decisions in SLS algorithms: Why are most algorithms so heavily randomized? How important are those random decisions? How important is the quality of the underlying random numbers? How much randomness is necessary? Can randomness be eliminated altogether? In this chapter, our goal was to shed light on some of these questions.

The remainder of this chapter is structured as follows. First, in Section 6.1, we provide background information and related work. Second, in Section 6.2, we study the effect of the quality of the underlying random number sequence on the behaviour of some well-known SLS algorithms. Next, in Section 6.3, we investigate the amount of randomness required to achieve the typical behaviour of these algorithms using derandomization. Finally, in Section 6.4, we summarize our work.

6.1 Background and Related Work

In Figure 2.1, we illustrated that a typical SLS algorithm for SAT consists of an initialization phase, in which a truth value is assigned to each variable, and a search phase, during which the values of individual, heuristically selected variables are changed (flipped) in an attempt to reach a satisfying assignment. Stochastic (random) decisions are typically used in both phases, and in the following we describe the most common ways SLS algorithms for SAT make use of random decisions:

Variable initialization is heavily randomized in most SLS algorithms for SAT; typically, the initial variable assignment is obtained by assigning each variable a truth value chosen uniformly and independently at random. For example, the `InitializeVariables` procedure in URWALK as described in Figure 2.6.

Heuristic tie-breaking occurs when a choice needs to be made between several alternatives that are ranked identically by a given heuristic evaluation function; many SLS algorithms for SAT break these ties randomly. For example, the `PickVariableGSAT` procedure in GSAT [100] as described in Figure 2.9.

Variable selection often includes randomized choices. For example, the variable selection in NOVELTY [80] (see Figure 2.11) can make *noisy* decisions. Under certain circumstances, the second best variable is selected (a noisy decision) over the best variable (a greedy decision) with some probability *noveltyNoise*.

Neighbourhood selection occurs when an algorithm narrows the list of flip candidates to a subset of all the variables. For example, in the WALKSAT algorithms, at each step, an unsatisfied clause is selected uniformly at random, and then only variables occurring in this clause are considered as flip candidates. This is described in Figure 2.10.

Random walk steps involve flipping randomly selected variables; they can help to increase search diversification, to avoid stagnation and to render an algorithm Probabilistically Approximate Complete (PAC). In a uniform random walk all variables can be selected with uniform probability. In a conflict-directed random walk, only variables occurring in currently unsatisfied clauses can be selected, such as in CRWALK [85].

Random restarts cause an algorithm to randomly re-initialize all variables. Most SLS algorithms for SAT, including algorithms of purely theoretically interest, such as SCHÖNING'S ALGORITHM [94], perform periodic restarts instead of random restarts.

Search control mechanisms can also make use of randomized decisions; examples include the probabilistic smoothing mechanism in SAPS [61], described in Chapter 4, and the random selection of the *tabuTenure* parameter in ROBUST TABU SEARCH (ROTS) [107].

The prominent use of random decisions in many components of SLS algorithms suggests that it is an important area of study, and in the following we identify some related work.

Gent and Walsh investigated the role of random decisions in GSAT [39]. They found that random decisions were neither important in the initialization phase nor for tie-breaking, and that deterministic substitutions could be made in both cases. Much of their analysis revolved around the ability of the algorithm to diversify the search during the re-initialization that occurs during restarts. They did not study the impact of the quality of random decisions. It is not clear to what extent their observations apply to more powerful SLS algorithms for SAT that do not require restart mechanisms or to a broader range of SAT instances.

There has been a large body of work dedicated to the quest for increasingly higher quality random number generators. In the Monte Carlo simulation literature, there has been evidence that even 'good' random number generators can produce very undesirable results [11, 26]. In work related to our experiments, Ribeiro *et al.* surveyed RNGs to find a good candidate for randomized algorithms [92]. In our published work [112], we investigated the role of random decisions in SAPS, which we will develop further in Section 6.3.

In this chapter we perform experiments with three different DLS algorithms. The first algorithm we consider is CRWALK (*a.k.a.* Papadimitriou's algorithm) [85], as described in Section 2.5. We chose to include CRWALK in our study because it is a prominent, yet very simple, algorithm that is purely based on random decisions. Originally, we had decided to include SCHÖNING'S ALGORITHM in our study because of its provably excellent worst-case behaviour, but in preliminary experiments on a large set of instances from SATLIB we found no empirical evidence for any differences between its behaviour and that of CRWALK (which, given well-known empirical results that the WALKSAT algorithm does not benefit from restarts [55, 86] is not surprising). The results reported in the following sections clearly show CRWALK performs rather poorly when compared against high-performance SLS algorithms for SAT, because it lacks the heuristic guidance of an evaluation function.

The next algorithm we consider in this study, ADAPTIVE NOVELTY$^+$ [51] is described in Section 2.5 and uses a deterministic mechanism for adapting its *noveltyNoise* setting during the search and therefore requires no parameter tuning. ADAPTIVE NOVELTY$^+$ uses randomized neighbourhood selection, randomized heuristic variable selection, and conflict directed random walk steps (in addition to random initialization).

Finally, we used the SAPS algorithm [61] as described in detail in Chapter 4. It was our experience with SAPS that inspired much of the work in this chapter. We included it in this work because (as we will discuss in more detail later) in long search trajectories SAPS operates almost deterministically [112]. In all experiments reported in this study we used the default parameters for SAPS, as described in Appendix A.

Unless otherwise stated (as in Section 6.2) all experiments have been conducted using the default random number generator in UBCSAT, the Mersenne Twister (MT) [78].

6.2 The Quality of Random Decisions

Typically, SLS algorithms are designed to make perfectly random decisions without any concern as to how those decisions are made. When implementing SLS algorithms, all random decisions are realized using a Random Number Generator (RNG). In principle, a True Random Number Generator (TRNG), which obtains a sequence of random numbers from a truly random source could be used. Hardware implementations of TRNGs that obtain random data from physical phenomae, such as atmospheric noise or radioactive decay, are available and are popular in applications such as gambling [129] and cryptography [10]. However, most computer implementations use Pseudo-Random Number Generators (PRNGs) instead [71]. A PRNG is a finite state machine with memory, and performs deterministic mathematical operations on the state information to generate a sequence of numbers. Once a PRNG is initialized with a numerical seed, it will produce a series of numbers that may have the appearance of being random, but in fact can all be deterministically calculated from the seed. The quality of a PRNG is solely determined by the mathematical operations it performs. Ideally, sequences will be uniform and unbiased (*i.e.*, equal fractions of numbers from the sequence should fall into equal intervals), uncorrelated (*i.e.*, the numbers in the sequence should be statistically independent of one another) and have long periods (because the state information in a PRNG is finite, all PRNGs will eventually cycle, but the period between cycles should be very large) [55: p. 52].

Because of the importance of high quality random numbers in cryptography and other applications, tests have been developed that measure the quality of a sequence of random data. The American National Institute of Standards and Technology (NIST) has produced a document [93] with companion software [131] to test the quality of random data. The NIST software includes 16 groups of tests that cover a wide variety of statistical properties. Another popular software tool for quickly analyzing the quality of random numbers is known as simply ENT (short for entropy) and was developed by John Walker at Fourmilab [126].

There are numerous PRNGs available that use a wide variety of mathematical methods. We have selected a few characteristic PRNGs to test, in addition to data generated by a TRNG. The following are brief descriptions of the RNGs we used:

True Random Data This data was obtained online [135] and was generated by a hardware device measuring atmospheric noise.

C `random()` We chose the Linux gcc (C) `random()` function because it is the default PRNG for many programmers, and is also currently the default PRNG for the original WALKSAT software package by Kautz [99] when compiled under Linux. We used gcc v3.3.3 on SuSE Linux v9.1.

LCG The Linear Congruential Generator (LCG) we chose is based on the ANSI C specification:

Random Data	Bias	χ^2 Analysis	Monte Carlo π	NIST %
True random	0.5000290	235.9 (75%)	3.14094 (0.021%)	97.80
C `random()`	0.4999988	224.6 (90%)	3.14148 (0.004%)	99.50
LCG	0.5000000	0.0 (99.99%)	3.14123 (0.011%)	93.53
LFG	0.5000129	237.3 (75%)	3.14139 (0.007%)	96.69
MT	0.5000204	278.5 (25%)	3.14203 (0.014%)	98.37
Random: Skew 1.25:1	0.5554831	2165538.1 (0.01%)	2.76998 (11.829%)	16.39
Random: Cycled 16k	0.5000086	4327.3 (0.01%)	3.14631 (0.150%)	59.18

Table 6.1: Test of quality on RNG data. Tests were executed on 160MB of data. The Bias value is the average value of all bits (the ideal value is 0.5). The χ^2 analysis from ENT shows the distribution value and a percentage which indicates how frequently a TRNG would have a larger distribution value, where values $> 95\%$ or $< 5\%$ are highly suspect. The Monte Carlo π analysis from ENT gives an estimated value of π and the respective error. For the NIST tests, we report the overall percentage of the tests passed by the respective data, where each of the 16 groups of tests was weighted equally.

$I_{j+1} = (I_j \cdot 1103515245 + 12345)$ except that only one byte (bits 11-18) of random data was collected per iteration, a common practice to improve the quality of this particular PRNG.

LFG The Lagged Fibonacci Generator (LFG) we chose is from Knuth [71], and the source code is available from his website [130].

MT The Mersenne Twister (MT) we chose is the MT19937 algorithm [78], which has an astounding period of $(2^{19937} - 1)$. This is the default PRNG in the current release of the UBCSAT software package.

In Table 6.1, we examine the relative quality of some of these RNGs. There is little difference between the results for the PRNGs and the TRNG, with the exception of LCG, which is clearly the worst of the tested PRNGs. It is often the case that particular sequences of TRNGs fail more tests than particular sequences of PRNGs [93]. The bottom two rows of Table 6.1 will be discussed later.

We now investigate the extent to which implementations of existing SLS algorithms are affected by the quality of the source of randomness. For most random decisions made within an SLS algorithm, there are bad choices (that increase the length of the current run) and good choices. Correspondingly, there could be bad sources of biased randomness (that cause a given algorithm to make more bad choices) and good sources. Whether or not a source is good or bad would depend on the given SLS algorithm and instance. An extremely good source for a particular SLS algorithm and instance could cause the algorithm to make a series of good decisions, solving the instance in linear run-time. However, an extremely bad source could cause the same algorithm to search for an arbitrarily long time, and in the worst case make a solution unreachable. However, note that even when using a TRNG with extreme bias, as long as the probability of generating 0 or 1 at any position of the sequence is greater than zero, the PAC property of a given SLS algorithm would remain intact, since the required sequence of 'correct decisions' would still occur (albeit with much lower probability).

The effect of correlation in the random number sequence, as long as it does not involve deterministic dependencies, is very similar for analogous reasons. (Note that correlation, in this context, corresponds to bias for certain subsequences.)

Deterministic cycles in the random number sequence, on the other hand, can lead to a loss of the PAC property. In combination with the finite state information held by the algorithm (which, in addition to the search position, may include search control variables and properties, such as tabu status information and clause penalty values), cycles in the random number sequence could cause cycles in the search trajectory that do not include any solutions to the given problem instance. Note that all PRNGs are periodic; whether or not this leads to observable stagnation of a given SLS algorithm depends on the period of the PRNG as well as on the amount and nature of state information used by the SLS algorithm.

To empirically study the effect of poor quality RNGs on SLS algorithms, we generated some intentionally bad random number sequences by manipulating the data we had from the TRNG. First, we introduced a skew s in our data. We converted 32-bits of our random data to an unsigned integer value and then divided it by 2^{32} to obtain a fixed-point value in the range [0,1). We generated a one if the value was greater than $s/(s+1)$, and zero otherwise. Next, we generated cycled data where we simply truncated the random data at a fixed number of bytes and repeated the same sequence. We ran our new poor sequences through the same tests we performed on the PRNGs and, from Table 6.1, it is clear that our poor RNGs do not meet very high standards of quality. We then made the sequences progressively worse, so the data presented in Table 6.1 can be considered the best of the bad sequences we generated.

To examine the effects of different RNGs on our selected algorithms, we ran CRWALK, ADAPTIVE NOVELTY$^+$ and SAPS with the different sources of random data, and present the results in Table 6.2, Table 6.3 and Table 6.4 respectively. We provided the PRNG comparison for CRWALK, and we can see the algorithm was very robust to the selection of the PRNGs. Analogous observations were made for ADAPTIVE NOVELTY$^+$ and SAPS.

For the skewed data, the sequences had more ones, and we shall consider what effect it would have on the specific implementations of the algorithms. For CRWALK, the bias would be toward arbitrarily specific clauses and literals. For the ADAPTIVE NOVELTY$^+$ algorithm, the same bias would exist for clause selection, but more importantly the frequency of random walk steps and noisy variable selections would decrease. For SAPS, the only significant change is a decrease in the

	ii8c2		ssa7552-159		flat50-med		uf100-med		uf50-hard	
	steps	c_v	steps	c_v	steps	c_v	steps	c_v	steps	c_v
True random	300k	0.99	2.21M	0.97	631k	0.97	76.1M	0.94	372k	0.97
C `random()`	1.13	0.94	1.02	0.91	0.96	0.93	1.10	0.99	1.10	0.99
LCG	1.13	1.02	1.02	1.00	0.95	0.98	1.02	0.96	1.02	0.96
LFG	1.15	0.99	1.05	1.02	0.94	1.03	0.98	0.97	0.98	0.97
MT	0.97	0.95	0.99	0.98	0.90	0.93	0.93	0.96	0.93	0.96
Skew 1.25:1	0.48	0.97	3.39	1.08	0.93	0.97	0.97	1.03	0.97	1.03
Skew 1.5:1	0.29	0.92	**15.27**	0.97	0.85	1.04	1.10	0.96	1.10	0.96
Skew 2:1	0.13	0.94	$>$ **368**	0.97	0.93	1.03	1.03	0.99	1.03	0.99
Skew 4:1	**0.06**	1.00	$>$ **2k**	0.02	0.88	1.02	0.96	1.05	0.96	1.05
Cycled 16k	1.28	0.86	0.66	0.96	0.92	0.87	0.82	1.16	0.82	1.16
Cycled 4k	1.23	0.85	0.82	1.15	0.89	0.83	0.61	1.11	0.61	1.11
Cycled 1k	0.89	0.76	2.17	0.91	0.55	0.83	0.52	1.00	0.52	1.00
Cycled 512	0.68	1.22	∞	**0**	0.10	0.75	0.63	1.12	∞	**0**
Cycled 256	2.38	0.56	∞	**0**	0.41	0.70	0.41	0.69	∞	**0**

Table 6.2: The effect of RNG quality on CRWALK. For the true random data, the mean number of search steps (run-length) required to find a solution is given, while for all other sources the mean search steps is given as a fraction of the number required for the true random source. The c_v is calculated as the standard deviation divided by the mean. Note that $c_v = 1$ characterizes an exponential run-length distribution, which is typical for high-performance SLS algorithms for SAT. All experiments results are based on 500 runs with a maximum run-length of 2^{32} (4.3B) steps. For the cycled sequences, with a reported ∞ mean, we confirmed cyclic behaviour by examining the respective search trajectories. See Appendix B for instance information.

smoothing frequency. Not all of the changes were negative. In some cases, such as the CRWALK algorithm on the ii8c2 instance, the skew greatly improved the performance of the algorithm.

For the cycled data, we continued to shorten the length of the cycles and thereby increased the likelihood that the algorithms would cycle. In Table 6.2 and Table 6.3 we present results from situations where both the CRWALK and the ADAPTIVE NOVELTY$^+$ algorithm became stuck in endless loops. Note that although CRWALK and ADAPTIVE NOVELTY$^+$ are both PAC, our empirical results show that these algorithms are no longer complete when using cyclic random number sequences. The fact that all PRNGs eventually cycle implies that *no conventional implementation* of an SLS algorithm is truly PAC. (An implementation may be PAC for a given instance, but with a countably infinite number of SAT instances there is no hope of guaranteeing that an implementation will be PAC for any arbitrary instance.)

Given this conclusion, it might seem wise to implement algorithms with TRNGs. If efficient

Random Data	uf100-med		uf250-hard		bw-large-c		ferry9u	
	steps	c_v	steps	c_v	steps	c_v	steps	c_v
Random Source	998	0.63	3.00M	0.96	10.0M	0.99	880k	0.88
Skew 1.25:1	1.17	0.61	1.29	1.06	0.91	1.05	0.57	0.87
Skew 1.5:1	1.29	0.62	2.17	0.95	0.80	0.94	0.45	0.87
Skew 2:1	1.61	0.65	4.16	1.01	0.99	0.95	0.68	0.90
Skew 4:1	3.02	0.76	**96.31**	0.62	1.30	1.00	$>$ **3 122**	0.75
Cycled 16k	1.06	0.80	0.85	0.95	0.93	1.17	0.98	0.40
Cycled 4k	1.27	0.76	0.81	0.94	0.82	0.92	1.09	1.03
Cycled 1k	0.98	0.64	203.97	2.45	1.15	1.08	1.40	0.86
Cycled 512	1.26	0.50	∞	**0**	0.13	1.61	1.03	0.80
Cycled 256	0.33	0.79	∞	**0**	0.66	1.33	∞	**0**

Table 6.3: The effect of RNG quality on ADAPTIVE NOVELTY$^+$. See Table 6.2 for details.

Random Data	uf100-med		uf250-hard		bw-large-c		ferry9u	
	steps	c_v	steps	c_v	steps	c_v	steps	c_v
Random Source	1.06k	1.01	304k	1.07	14.6M	0.99	1.92M	1.01
Skew 1.25:1	1.31	0.97	1.33	1.01	0.54	1.04	0.39	0.97
Skew 1.5:1	1.53	1.13	1.79	1.02	0.35	1.02	0.32	0.98
Skew 2:1	1.89	1.16	**3.03**	1.08	0.34	0.97	0.26	0.97
Skew 4:1	**2.37**	1.09	**5.45**	1.04	**0.42**	1.02	**0.11**	0.90
Cycled 16k	1.10	0.99	0.99	1.00	0.95	0.97	0.78	0.90
Cycled 4k	0.91	0.91	1.12	0.95	0.87	0.90	0.41	1.03
Cycled 1k	0.60	0.88	0.73	0.97	1.26	1.37	1.59	0.86
Cycled 512	0.55	0.72	0.96	0.49	0.88	1.18	2.18	0.89
Cycled 256	1.39	0.89	1.44	0.83	1.26	0.99	0.39	1.23

Table 6.4: The effect of RNG quality on SAPS. See Table 6.2 for details. SAPS was executed with default parameters as described in Appendix A.

TRNGs were readily available it would be an ideal solution. However, TRNGs are far from efficient when compared to PRNGs. To add perspective to this discussion, we must consider how incredibly unlikely the aforementioned circumstances are with a good PRNG; for example, the Mersenne Twister PRNG has a period of $(2^{19937} - 1)$, which means that it will not cycle in practice. If cyclic behaviour is observed for an algorithm using a PRNG of this type, the behaviour is far more likely to be due to a design flaw, an implementation error, or simply because the algorithm is not PAC (even when using true random numbers).

When implementing an SLS algorithm and selecting a PRNG, there are several factors to be considered. To assess the quality of a given PRNG, one of the many available test suites can be used. However, any reasonable PRNG will be sufficiently unbiased and uncorrelated to render impacts on the performance of typical SLS algorithms very unlikely. To minimize the chance of encountering cycling behaviour of an SLS algorithm in practice, it is generally advisable to choose a PRNG with a sufficiently large period. We provided the Mersenne Twister as an example of a PRNG with an extremely large period, but note that much smaller periods appear to be sufficient in practice. Another potentially important factor is the efficiency of a PRNG. This is particularly relevant in the context of highly randomized SLS algorithms that make random decisions in every (or almost every) search step. Finally, especially in the context of scientific research, the use of platform-independent PRNGs makes it possible to reproduce unusual algorithm behaviour exactly across different hardware and operating systems. The previously mentioned Mersenne Twister has *all* of the qualities that are desirable for a PRNG and overall appears to be the best choice in the context of implementing SLS algorithms.

6.3 Quantity of Randomness

In the previous section, we examined how the quality of random numbers can affect SLS behaviour. In this section, we study the quantity of random decisions made by SLS algorithms, and consider how many random decisions are truly required. We first investigate random decisions in the SAPS algorithm. It has been observed that DLS-CP algorithms, such as SDF or SAPS, become essentially deterministic after an initial search phase [95]. Intuitively, the clause penalties become unique after numerous scaling and smoothing steps, so there is no need for heuristic tie-breaking. To further investigate the role of randomness in these algorithms, we have created and studied a mostly derandomized variant of SAPS known as SAPS/NR [114].

SAPS/NR does not perform any random walk steps at local minima, uses periodic smoothing after every ($\lfloor 1/ps \rfloor$) local minima, and breaks all ties by selecting the variable with the smallest index. At first glance, it may seem that SAPS/NR is completely deterministic, but we must emphasize that the initialization of SAPS/NR is identical to the initialization in SAPS, and consequently the initial starting position for each run of SAPS/NR is completely random. In Table 6.5 and Figure 6.1 we compare the performance differences between SAPS and SAPS/NR. The ferry9u

Instance	SAPS		SAPS/NR	
	Mean	c_v	Mean	c_v
uf100-med	1 075	0.95	1 041	1.01
uf250-hard	287 907	0.98	292 488	0.96
bw-large-c	13 413 962	0.98	14 510 361	1.05
ferry9u	1 883 606	1.03	3 179 808	1.06

Table 6.5: SAPS *vs* SAPS/NR. (Median run-lengths were obtained from 1 000 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A.)

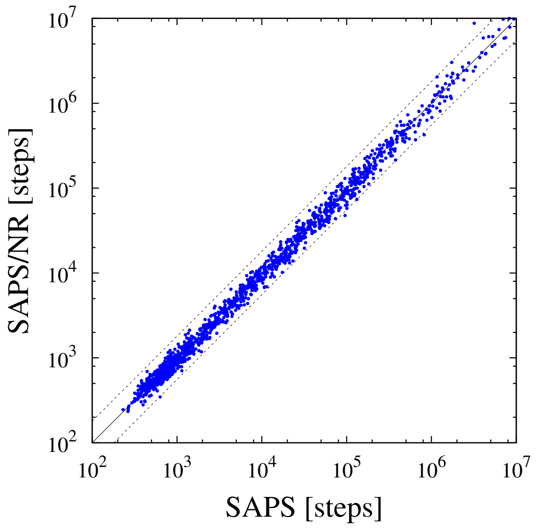


Figure 6.1: SAPS *vs* SAPS/NR on anp10m. Each point corresponds to an instance in the set anp10m (1931 instances). Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.

instance is one of the few cases in which we have found significant performance differences; in the overwhelming majority of cases, both algorithms show no significant difference in their performance.

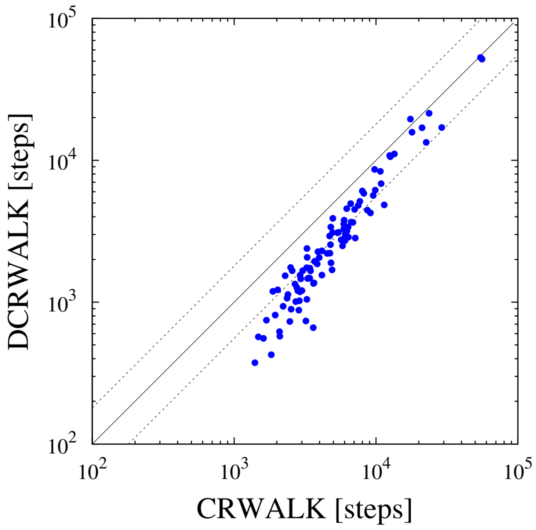
After restricting all of the random decisions to the initialization phase, we will next consider what happens when we remove the random decisions from the initialization phase as well. If we deterministically initialize the variable assignments, SAPS/NR will always take the same number of steps to solve an instance, reducing the variability in the run-time to zero. This can be seen in Figure 6.3 (a) as a vertical line. The deterministic initialization method we used was a simple greedy approach. For each variable, if the positive literal appears more frequently than the negative, the variable is assigned a value of true, otherwise false. When variables with an equal number of positive and negative literals are encountered, they are deterministically assigned a value of true or false, alternating between variables.

We next consider what happens if, between the initialization and the search phase, we select one variable uniformly at random and flip it. Remarkably, as seen in Figure 6.3 (a), the variability introduced by just that one random decision is close to the full variability seen by the regular, fully randomized version of SAPS. Because this instance has 250 variables, there are 250 discrete levels in the curve, corresponding to each of the 250 variables that could have been flipped. It is quite remarkable and rather counter-intuitive that flipping just one variable between the initialization and search phase could have such a dramatic effect on the run-time behaviour of the algorithm. We note that this phenomenon is very reminiscent of the extremely sensitive dependence on initial conditions found in chaotic dynamic systems [68: p. 56]

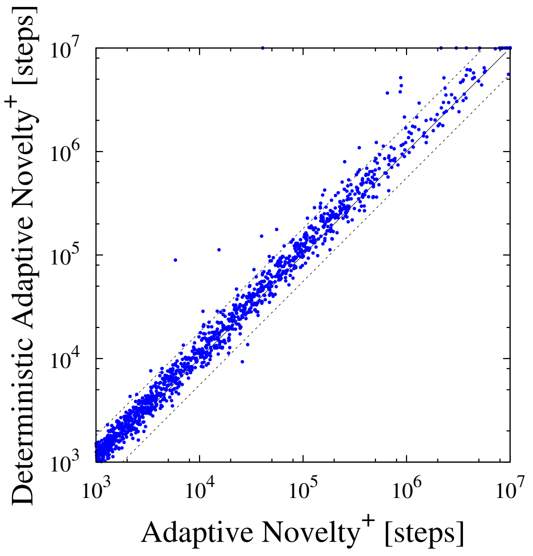
Next, we consider similar derandomizations for CRWALK and ADAPTIVE NOVELTY$^+$, which depend on random decisions to a much greater extent than SAPS. It should be noted that the derandomized versions of these algorithms described in the following were chosen for their simplicity rather than for their performance or their exceptionally strong correlation to the original algorithms. We did not invest time in tuning and engineering our algorithms with different derandomization strategies to meet higher quality standards. Our goal was to illustrate that our simple, straightforward approach works reasonably well for most instances.

Recall that CRWALK uses random decisions to select unsatisfied clauses and to decide which variable in a selected clause is to be flipped. To implement clause selection in DETERMINISTIC CRWALK (DCRWALK), we keep track of the number of times each clause has been selected (count) and the number of steps at which each clause has been unsatisfied (unsat), then we simply select the clause that has the smallest (count : unsat) ratio, breaking ties by selecting the clause with the smallest index. This method ensures that clauses are selected in a uniform, fair and deterministic manner. For literal selection, we simply keep a counter for each clause, selecting the first literal the first time the clause is selected, the second literal the second time, and so on, returning to the first literal when all have been exhausted. Thus, DCRWALK removes all of the randomness from the heuristic search phase, while still allowing for random decisions at the initialization phase. Note that our approach differs substantially from some of the published theoretical methods for derandomizing SCHÖNING'S ALGORITHM [19], which use Hamming balls to eliminate randomness from the initialization phase, departing from traditional SLS by using backtracking in the local search phase.

To derandomize the ADAPTIVE NOVELTY$^+$ algorithm, we need to replace three types of random decisions: clause selection, random walk steps and noisy variable selection. For clause selection, we maintain a list of the currently false clauses and simply step through that list, selecting the clause in the list that is the current search step number modulo the size of the list. Instead of random walk



(a) CRWALK on flat30

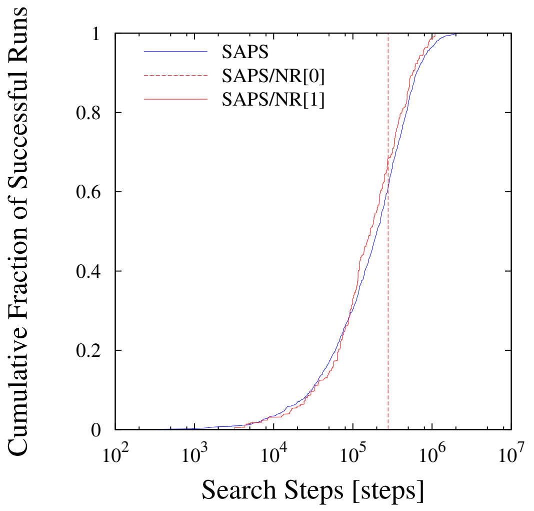


(b) ADAPTIVE NOVELTY$^+$ on anp10m

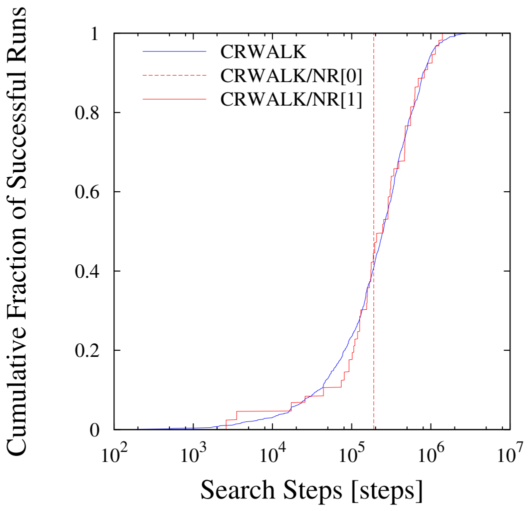
Figure 6.2: Original *vs* deterministic implementations. ((a) Each point corresponds to an instance in the set flat30 (100 instances). (b) Each point corresponds to an instance in the set anp10m (1931 instances). Median run-lengths were obtained from 100 independent runs per instance. All algorithms were executed with default parameters as described in Appendix A. See Section C.5 for general correlation plot details.)

steps, every $(\lfloor 1/wp \rfloor)$ steps a variable is selected to be flipped using the same variable selection scheme used by DCRWALK. For the noisy variable selection, we use two integer variables n and d. If the ratio $(\frac{n}{d})$ is less than the current noise setting *noveltyNoise* a noisy decision is made and n is incremented. Conversely, if $(\frac{n}{d})$ is greater than *noveltyNoise* the greedy decision is made and d is incremented. Whenever the adaptive mechanism modifies the noise parameter *noveltyNoise*, the values of n and d are reinitialized to $\lfloor 256 \cdot noveltyNoise \rfloor$ and $(256 - n)$, respectively.

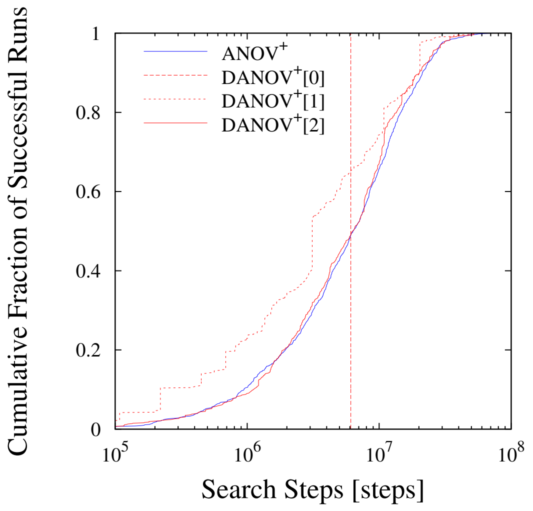
In Figure 6.2, we compare the performance of DCRWALK and DETERMINISTIC ADAPTIVE NOVELTY$^+$ with their fully randomized versions. In general, we do not see the same tight correlation observed for SAPS/NR. However, for the most part our derandomized algorithms show very similar behaviour. Our DCRWALK algorithms seems to outperform CRWALK for the vast majority of instances in this set, possibly because the clause selection scheme is fair and unbiased. Gent and Walsh observed similarly improved behaviour for a fair deterministic version of GSAT [39]. Our DETERMINISTIC ADAPTIVE NOVELTY$^+$ algorithm suffers from slightly worse performance on average, and there are significant outliers that indicate some inherent problems with our derandomization approach on some specific instances, but for most instances the performance of DETERMINISTIC ADAPTIVE NOVELTY$^+$ resembles that of ADAPTIVE NOVELTY$^+$.



(a) SAPS on uf250-hard



(b) CRWALK on uf50-hard



(c) ADAPTIVE NOVELTY$^+$ on bw-large-c

Figure 6.3: The variability of deterministic algorithms with few random decisions. A run-length distribution comparison of SAPS, CRWALK and ADAPTIVE NOVELTY$^+$ and their deterministic variants (with [N] total random decisions per run) based on 1 000 runs. The deterministic algorithms were initialized according to a deterministic method (see text for details), and then [N] variables were selected at random to be flipped after initialization. In other words, the vertical bar, [0], reflects when *all* random decisions have been replaced, while the [1] curve shows the behaviour when one single random decision is made in each run. (Instances are (a) uf250-hard, (b) uf50-hard and (c) anp10m (see Appendix B). Each run-length distribution is for 1 000 runs. All algorithms were executed with default parameters as described in Appendix A.)

In Figure 6.3 (b) and Figure 6.3 (c), we see evidence that the same 'chaotic' behaviour observed for SAPS/NR is also present for DCRWALK and DETERMINISTIC ADAPTIVE NOVELTY$^+$. Using the same deterministic initialization as in SAPS/NR, we obtain the same behaviour. With just one simple random decision in DCRWALK and two in DETERMINISTIC ADAPTIVE NOVELTY$^+$, the full variability found in the run-time distributions of the original, heavily randomized versions of these algorithms is achieved. What makes this observation remarkable is not so much that in principle, the amount of random decisions can be drastically reduced without any substantial effect on the behaviour of the algorithm (after all, any implementation of an SLS algorithm using a PRNG is fully deterministic), but rather that it can be done using very simple derandomization schemes.

6.4 Conclusions

Most SLS algorithms heavily use various types of random decisions. We argued that, from a theoretical point of view, their performance can be expected to be severely compromised by poor-quality random number sequences. Nevertheless, our empirical results indicate that in practice, the behaviour of these algorithms is remarkably robust with respect to the quality of the RNG used to implement these random decisions. As a consequence, there is no reason to consider the use of true random number generators (which have the disadvantage of typically being rather slow), or to worry about minor differences in the quality of readily available PRNGs, especially if their period is high. Because of its extremely high period, efficiency and platform-independent availability, we recommend using the Mersenne Twister PRNG for the implementation of SLS algorithms.

We demonstrated that three prominent SLS algorithms for SAT (SAPS, ADAPTIVE NOVELTY$^+$, CRWALK) can be almost completely derandomized using very simple mechanisms to replace the random decisions without significantly changing their behaviour. In particular, versions of these algorithms that only use a single random decision during initialization basically exhibit the full variability in the run-time required to solve a given SAT instance as the original, fully randomized algorithms. Eliminating this last random decision leads to completely deterministic algorithms which, on average, may often perform similarly well as their fully randomized versions. At the same time, these deterministic algorithms can no longer benefit from easy and efficient parallelization by means of performing multiple independent tries in parallel [53]. Additionally, at least for the deterministic version of ADAPTIVE NOVELTY$^+$ we observed substantially degraded performance on a very small number of instances. Therefore, we see no practical advantages in using completely or partially derandomized SLS algorithms.

Overall, our results are fully consistent with the widely held view that the role of random decisions in SLS algorithms is primarily to provide search diversification. Therefore, neither the quality of the RNG nor the quantity of random decisions used by an SLS algorithm is of crucial importance to its behaviour.

Chapter 7

Variable Expressions

Call on me...
...I'm the same boy I used to be.
— Eric Prydz. "Call On Me"
(Original lyrics by Steve Winwood)

In this chapter, we present our most recent and significant work, which stands as the capstone of our dissertation. Our work in this chapter was motivated by two significant trends we had observed within our research group and elsewhere. The first trend we observed was that DPLL-based algorithms were continuing to outperform SLS algorithms on application instances such as software verification encodings [137]. As a result of this observation, our primary goal was to improve the state-of-the-art of SLS algorithms for SAT on encodings of software verification benchmark instances. In our pursuit of this goal, we developed a new conceptual model for representing SLS algorithms for SAT, based on our notion of Variable Expressions (VEs). The second significant trend we observed was the success of automated algorithm configuration tools, in particular PARAMILS [59]; we believe that this trend is heralding a new era of automated algorithm design. As a result of this observation, we were motivated to revisit our UBCSAT architecture, and we created the DESIGN ARCHITECTURE FOR VARIABLE EXPRESSIONS (DAVE), a highly flexible SLS algorithm design architecture designed to leverage recent tools such as PARAMILS to automate many of the tedious aspects of algorithm design. By following our new algorithm design approach, we were able to achieve significant improvements over previous state-of-the-art SLS algorithms for SAT on encodings of software verification benchmark instances, thus accomplishing our goal.

This chapter is structured as follows. First, in Section 7.1, we provide some background and describe some of our experimental methodologies. Then, in Section 7.2, we introduce VEs, and demonstrate their potential. Next, in Section 7.3, we present our general conceptual model and briefly discuss its implementation (DAVE). Then, in Section 7.4, we introduce a new, highly parametric algorithm named VE-SAMPLER to demonstrate how DAVE facilitates the automated design of SLS algorithms. In Section 7.5, we discuss related work from the literature. Finally, in Section 7.6, we summarize our contributions.

7.1 Background

In this chapter, we propose a new conceptual model for specifying SLS algorithms for SAT. We introduce the concept of Variable Expressions (VEs) to generalize scoring functions; while VEs are ultimately used for variable selection, they can transcend the traditional notion of score. VEs are mathematical expressions that compute numerical values from one or more properties of a variable in combination with constants, operators and functions. We described variable properties in Section 2.4, including GSAT's score property [100] and the age property used by NOVELTY and WALKSAT/TABU [80]. A VE can be a simple property (*e.g.*, $\langle \mathsf{age} \rangle$) or any mathematical expression with one or more properties, such as $\langle \mathsf{score} + 3 \cdot \log(\mathsf{age}) \rangle$. Most existing SLS algorithms for SAT select variables based on scoring functions that correspond to a single, rather simplistic VE. In Section 7.2, we present evidence that potentially complex VEs can be very effective.

We also introduce the concept of Variable-Selection Mechanisms (VSMs), which evaluate VEs to determine the variable to flip during a search step. For example, the GSAT algorithm (see Figure 2.9) uses a very simple min VSM to select the variable with the minimum $\langle \mathsf{score} \rangle$ (breaking ties uniformly at random).

Our model was developed to provide a clean conceptual separation between the VEs and the VSM of an algorithm. A more complicated example of a VSM, that demonstrates the potential of this conceptual separation, is the NOVELTY algorithm (see Figure 2.11). In our model, the VSM of NOVELTY uses three VEs (e_1, e_2, e_3). The first VE is the primary scoring function, which in the NOVELTY algorithm is ($e_1 = \langle \mathsf{score} \rangle$). The second VE ($e_2 = \langle -\mathsf{age} \rangle$) is used for tie-breaking (we negated the age property so that it is consistent with score, *i.e.*, a minimal value is preferred). The third VE ($e_3 = \langle \mathsf{age} \rangle$) is coincidentally similar to e_2, and is used to identify when a randomized noisy decision is made. The generalized NOVELTY VSM is as follows: the variable with the minimal e_1 (breaking ties by e_2) is selected, unless the variable has the minimal e_3, in which case with probability *noveltyNoise* the variable with the second minimal e_1 (breaking ties by e_2) is selected. In our model, the traditional NOVELTY algorithm can be implemented by providing the VSM the VEs ($\langle \mathsf{score} \rangle, \langle -\mathsf{age} \rangle, \langle \mathsf{age} \rangle$), but now that we have separated the VEs from the VSM, we can use the VSM of NOVELTY with *any* three VEs, such as ($\langle \mathsf{break} \rangle, \langle \mathsf{flips} \rangle, \langle \mathsf{age}/\mathsf{flips} \rangle$). We will explore our model in more detail in Section 7.3

In this chapter, we mostly focused on the cbmc software verification instance sets. This set is interesting to us primarily because it has some of the structural properties of larger and more complicated software verification problems, which are still somewhat intractable for SLS solvers.

For example, well-known state-of-the-art SLS solvers from the 2009 SAT Competition [137], such as ADAPTIVE G^{2}WSAT [77] and GNOVELTY$^+$ [87], require over an hour to solve the hardest cbmc instance, whereas many of the DPLL-based solvers from the competition, such as PICOSAT [12], can solve the hardest cbmc instance in less than one second. At the same time, a significant number of the instances can be solved by SLS algorithms within a low enough time to allow for extensive experiments. In Section 7.4, we also provide, for the first time, experimental data for SLS algorithms on the software verification benchmark set swv generated by the CALYSTO static checker [9] and used as a benchmark for DPLL-based solvers by Hutter *et al.* [58].

In the experiments presented throughout this chapter, we used the PARAMILS automated algorithm configurator by Hutter *et al.* [60]. PARAMILS is an SLS method that searches a parameter configuration space. The three primary inputs to PARAMILS are an algorithm, a set of instances and a configuration file that specifies the parameters of the algorithm and the possible values for each parameter. The primary output of PARAMILS is a configuration of the algorithm that has been optimized for the given instance set. We used PARAMILS to optimize the parameter settings of various SLS-based SAT algorithms for the aforementioned sets. To ensure that our results generalize to instances other than those used during the optimization process, we randomly split each set into two halves, a *training* set and a *test* set. In Section C.6 we provide more detail on how we split the training and test sets. The optimized configuration is found by running PARAMILS on only the training set, and only the test set is used to report our results.

7.2 Advanced Variable Expressions

In Section 2.5, we described several SLS-based algorithms known from the literature and identified the various variable properties they use. Before we start introducing new ideas, we provide a brief review of how variable properties are used by existing algorithms, in the new light of VEs.

Perhaps the most popular VE currently used by SLS algorithms is $\langle \text{score} \rangle$, which is equivalent to the VE $\langle \text{break} - \text{make} \rangle$ where the properties make and break measure the number of clauses that would become satisfied and unsatisfied, respectively, if the variable were to be flipped. The WALKSAT/SKC algorithm was the first algorithm to use the even simpler VE $\langle \text{break} \rangle$ for scoring variables and also introduced a Boolean freebie property that is true if, and only if, break equals zero. Algorithms with dynamic clause penalties, such as SAPS, use a penalized property penScore. The G^{2}WSAT algorithm uses a Boolean promising property that indicates a negative score property value, but only under certain circumstances (see Section 2.5 and [76] for details).

Another variable property that is prominently used in existing SLS algorithms for SAT is age. The age property is defined as the number of search steps that have occurred since the given variable was last flipped. The age property is closely related to the flips property (*a.k.a.* flipCount) used by the HSAT algorithm [39] as a tie-breaking mechanism. The flips property measures how many times a variable has been flipped. An interesting and effective combination of the freebie, break, age and flips properties is used in the VW2 algorithm [88].

7.2.1 Deconstructing VW2

In many ways, Prestwich's VW2 algorithm [88] provided the starting point for our work on VEs, and we describe VW2 in the following.[1] Each variable is assigned a *weight*, which we call the vw property, initialized to zero. At each search step a WALKSAT strategy is used, where the flip candidates are those variables that appear in a randomly selected unsatisfied clause. If there are any candidates with a true freebie value, one of those is selected (breaking ties uniformly at random); otherwise, with probability wp, a candidate is selected uniformly at random, and in the remaining cases (*i.e.*, with probability $(1-wp)$), the candidate is selected with the smallest value of the VE:

$$\mathsf{break} + c \cdot (\mathsf{vw} - \overline{\mathsf{vw}}) \tag{7.1}$$

(breaking ties uniformly at random), where the constant c is a parameter and $\overline{\mathsf{vw}}$ denotes the average of the vw property across all variables. When a variable is flipped, its vw property is updated according to:

$$\mathsf{vw} := (1-s) \cdot (\mathsf{vw}+1) + s \cdot \mathrm{step}\ , \tag{7.2}$$

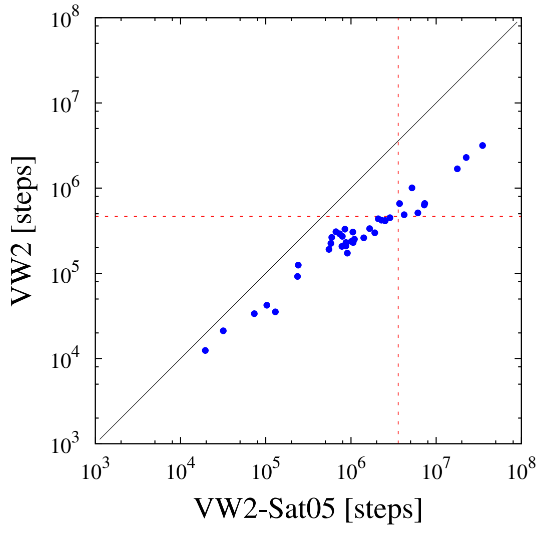
where s is another constant parameter, and step is the current search step iteration value.

A variant of VW2 that we call VW2-SAT05 received the bronze medal in the satisfiable random category of the 2005 SAT competition [73]. This variant eliminates the three VW2 parameters (s, c, wp) by setting wp to zero and introducing a randomized mechanism to change the behaviour of c and s during the search; it has been included recently in the SATENSTEIN [70] and HYBRID [117] algorithms. In our experiments, we found that the original VW2 procedure with parameter settings optimized for a given set of benchmark instances will often outperform VW2-SAT05. In particular, we observed this performance difference on the cbmc software verification instances, as illustrated in Figure 7.1 (a). In Figure 7.1 (b), we present evidence showing that VW2 can outperform SATENSTEIN, the previously best-known SLS algorithm for SAT for cbmc. This excellent performance of VW2 motivated us to study it in more depth.

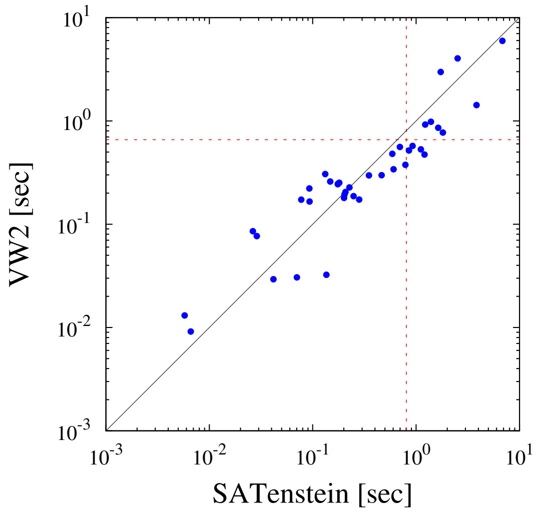
Upon closer examination of the VW2 VE shown in Equation 7.1, we noticed that the $\overline{\mathsf{vw}}$ term can be removed without changing the behaviour of VW2, since this term is constant over all variables and therefore does not affect the variable selection. In the vw property update procedure, the s parameter is a *smoothing* parameter. if s is set to one, the VE becomes equivalent to $\langle \mathsf{break} - c \cdot \mathsf{age} \rangle$. If s is set to zero, as in the optimal setting for cbmc, the VE becomes equivalent to $\langle \mathsf{break} + c \cdot \mathsf{flips} \rangle$.

For very small values of c, it may appear as though the vw property acts as a tie-breaking mechanism, and Prestwich observed that when s is zero, VW2 behaves like HSAT [39]. While it may be easy to dismiss the mechanics of VW2 as a simple tie-breaking scheme, this simplification

[1] For consistency with our dissertation, we chose to use our notations instead of Prestwich's when describing VW2.



(a) VW2 *vs* VW2-SAT05



(b) VW2 *vs* SATENSTEIN

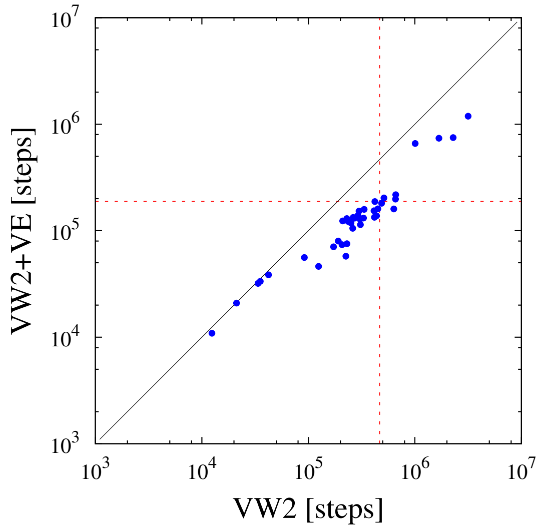
Figure 7.1: VW2 *vs* VW2-SAT05 and SATENSTEIN on cbmc. The speedup factor (*s.f.*) is (a) 7.66 and (b) 1.22. (The parameters for VW2 are $(s, c, wp) = (0, 0.01, 0.2)$, found by PARAMILS (see Section C.7.1). The configuration of SATENSTEIN is from the author [69]. Each point corresponds to an instance in the set cbmc (test) (39 instances). Median run-lengths and run-times are reported, obtained from 25 runs. Execution environment: UBC arrow cluster (Section C.1). See Section C.5 for general correlation plot details.)

does not seem justified when considering the parameter settings obtained for VW2 and the length of typical runs required for solving cbmc instances. In our analysis of VW2 on the hardest cbmc instance, we observed that for over half of the search steps the break and flips properties were interacting in a complex way, and VW2 was making trade-offs between satisfying additional clauses (intensification) and changing the values of rarely flipped variables (diversification).

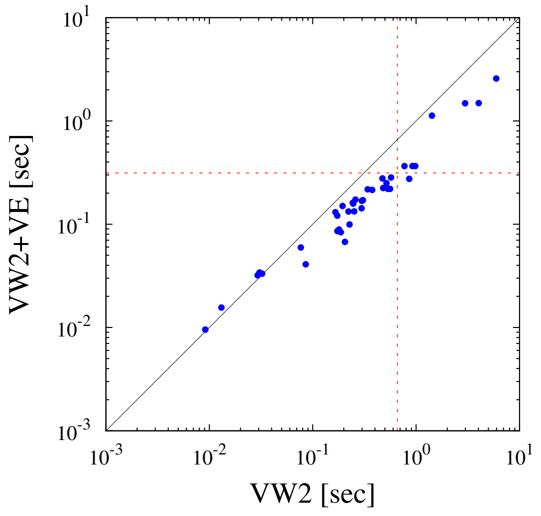
7.2.2 VW2+VE: Modifying the VE in VW2

Considering this type of complementarity in the role of the break and flips properties and the strong performance of VW2, it seemed promising to explore different ways of constructing a VE based on those two properties. Because the difference in scale between the two properties becomes increasingly larger as the search progresses, we decided to *normalize* the values of these properties to the interval $[0, 1]$. We achieved this using the formula $\frac{\mathsf{p}}{\max(\mathsf{p})}$, where $\max(\mathsf{p})$ refers to the maximum value of the property p for all flip candidates, which for VW2 would be those variables in the currently selected clause.

In addition to normalizing the property values, we also allowed for *non-linear* interaction be-



(a) run-length



(b) run-time

Figure 7.2: VW2+VE *vs* VW2 on cbmc. The speedup factor (*s.f.*) is (a) 2.47 and (b) 2.10. (The parameters for VW2 are $(s,c,wp) = (0,0.01,0.2)$, found by PARAMILS (see Section C.7.1). The configuration of VW2+VE is $(c,a,wp) = (0.95,8,0.05)$, found by PARAMILS (see Section C.7.2). Each point corresponds to an instance in the set cbmc (test) (39 instances). Median run-lengths and run-times are reported, obtained from 25 runs. Execution environment: UBC arrow cluster (Section C.1). See Section C.5 for general correlation plot details.)

tween the two properties. Our motivation was that the *relative* difference in magnitude between two different property values could have an important impact on the behaviour of the algorithm. Since the values have already been normalized, we used a simple polynomial transformation on the normalized values of the flips property, to obtain the generalized VE:

$$\frac{\mathsf{break}}{\max(\mathsf{break})} + c \cdot \left(\frac{\mathsf{flips}}{\max(\mathsf{flips})}\right)^{a}. \tag{7.3}$$

We used this to replace the scoring function of VW2. We refer to the resulting variant of VW2, in which we also disabled smoothing, as VW2+VE. Automated configuration of this algorithm for our cbmc training set using PARAMILS yielded the parameter configuration $(c,a,wp) = (0.95,8,0.05)$ (see Section C.7.2).

As can be seen from Figure 7.2 (a), the use of this generalized VE leads to improved performance in terms of local search steps required for solving the cbmc instances (as always, we show results for the test set, which is disjoint from the training set used for parameter optimization). However, the VE is more complex, and evaluating it requires an additional initial iteration to determine the maximum values. This leads to a less pronounced improvement in terms of run-time performance, which is illustrated in Figure 7.2 (b). Still, VW2+VE performs better than VW2 on the cbmc benchmark which, based on our earlier findings, makes it the best SLS-based SAT algorithm for that benchmark currently available.

7.2.3 Normalization in VEs

In VW2+VE, we normalized the break and flips properties so they would fall within the interval $[0,1]$. We now generalize this further, using from here on the notation $\|x\|$ in VEs to indicate that the value x has been normalized using one of several different methods. The method used in VW2+VE,

$$\|x\| = \frac{x}{\max(x)}, \tag{7.4}$$

preserves ratios between the values being normalized. Alternatively, a flat normalization:

$$\|x\| = \frac{x - \min(x)}{\max(x) - \min(x)} \tag{7.5}$$

forces the maximum and minimum to be one and zero, respectively, and a summation normalization:

$$\|x\| = \frac{x}{\text{sum}(x)} \tag{7.6}$$

forces the sum of the values to be one. Of course, numerous other normalizations are possible, including non-linear normalizations and normalizations more suitable for both positive and negative values.

In the literature, some scoring functions are designed to select variables with the *minimum* value (such as VW2's), whereas others select the variable with the *maximum* value. Both cases are common, and choosing which one should be used is usually obvious from the context; however, this may not always be the case as we consider more complicated VEs. To address this issue, we first note that the question of favouring minimum or maximum values already arises for variable properties. For example, a small value of flips is considered favourable, while the opposite is true for age. To facilitate the construction of more complex VEs, we require that all properties be transformed to favour maximum values. To this end, we revise our notation for normalization so that $\|\mathsf{p}\|$ indicates that p has been normalized and transformed (if necessary). A simple transformation and normalization would be $(1 - \|\mathsf{p}\|)$, and we found that $\|\max(\mathsf{p}) + \min(\mathsf{p}) - \mathsf{p}\|$ worked quite effectively in practice.

When normalizing the make and break properties, we observed that the number of clauses in which the variable appears can also be used for normalization. We introduce the variable properties relMake and relBreak to correspond to the *relative* number (fraction) of clauses that become satisfied or unsatisfied, respectively, as a result of flipping a given variable. For example, if the positive literal x occurs in numPosOcc clauses and the negative literal $\neg x$ occurs in numNegOcc clauses, then the value of relMake is equivalent to $\langle \mathsf{make}/\mathsf{numPosOcc} \rangle$ when x is false and $\langle \mathsf{make}/\mathsf{numNegOcc} \rangle$ when x is true. For randomly generated instances with uniform structure, normalizing the score in this manner would have no material effect. For structured formulae, such as the cbmc instances, there is often large variability in the number of clauses each variable appears in, and consequently, this normalization can make a substantial difference. Ansótegui *et al.* explored the *scale-free* structure of industrial instances and the impact of this structure on DPLL-based solvers [3], and we believe that there is potential for SLS algorithms to exploit this structure as well.

Another observation we made is that existing algorithms combine make and break symmetrically, but there may be an advantage to constructing VEs in which they are weighted differently. We therefore consider the generalized VE $\langle c_1 \cdot \mathsf{break} - c_2 \cdot \mathsf{make} \rangle$, which uses simple scaling to weight the two variable properties differently. We note that WALKSAT/SKC can be seen as using a special case of this VE where $c_2 = 0$. While it is possible that in many cases choosing $c_1 = c_2$ may lead to the best performance, there is no reason to assume this would always be the case.

Finally, we observed that the summation normalization (Equation 7.6) behaved rather differently than the one we used in VW2+VE (Equation 7.4), even though at first glance it would appear that they should only differ by a constant factor. However, that constant factor is the *clause length*, which is constant for any particular search step, but can differ between search steps. In other words, we discovered that normalization of the clause length can be beneficial, and we believe that such normalizations merit further study.

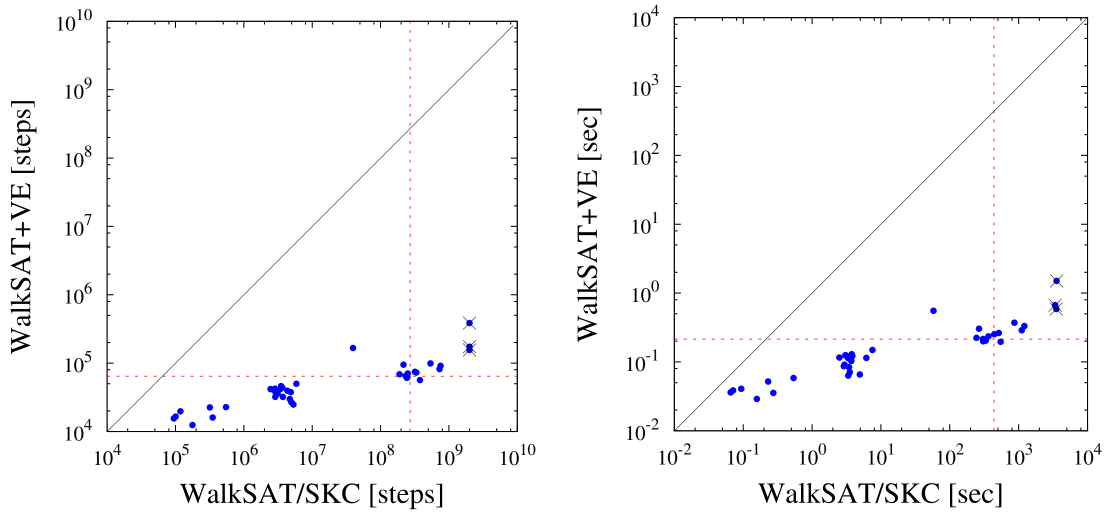
7.2.4 WALKSAT+VE: Modifying the VE in WALKSAT

To investigate the potential latent in the generalizations introduced up to this point, we constructed a new SLS algorithm we call WALKSAT+VE. This algorithm is obtained from the original WALKSAT/SKC algorithm by replacing the VE $\langle \mathsf{break} \rangle$ with the following VE that makes use of scaling, normalizations and non-linear transformations:

$$c_1 \cdot \|\mathsf{make}\|^{a_1} + c_2 \cdot \|\mathsf{relMake}\|^{a_2} + c_3 \cdot \|\mathsf{break}\|^{a_3} + c_4 \cdot \|\mathsf{relBreak}\|^{a_4} \ . \tag{7.7}$$

Whereas VW2+VE benefited from the flips property providing diversification, this VE uses only greedy components (make and break) and a standard random walk mechanism. To test the effectiveness of our new algorithm, we ran PARAMILS to optimize the values of the constants and the normalization parameters (hidden in the $\|\mathsf{p}\|$ notation) on the cbmc training set (see Section C.7.3).

The performance of the configuration thus obtained on the cbmc test set is illustrated in Figure 7.3 and Figure 7.4. WALKSAT+VE solves this benchmark set more than three orders of magnitude faster than WALKSAT/SKC, and outperforms the previously best known SLS algorithm for



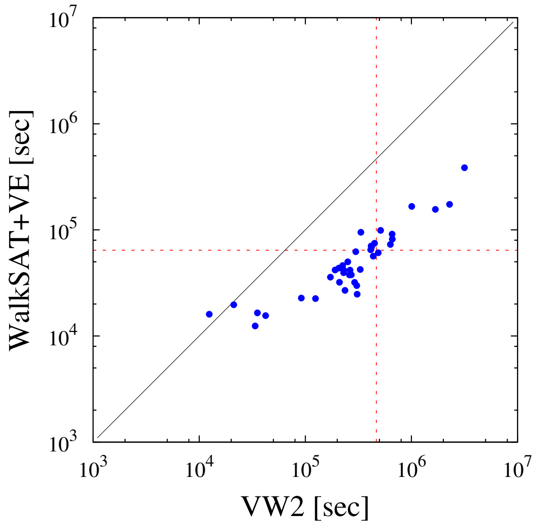
(a) run-length **(b)** run-time

Figure 7.3: WALKSAT+VE *vs* WALKSAT/SKC on cbmc. Unsuccessful runs of WALKSAT/SKC that were terminated after two billion search steps are indicated with an 'x'. The speedup factor (*s.f.*) is (a) $> 4\,194$ and (b) $> 2\,043$. (The random walk parameter of WALKSAT/SKC is $(wp) = (0.15)$, and was found manually. The configuration of WALKSAT+VE was found by PARAMILS and is described in Section C.7.3. Each point corresponds to an instance in the set cbmc (test) (39 instances). Median run-lengths and run-times are reported, obtained from 25 runs. Execution environment: UBC arrow cluster (Section C.1). See Section C.5 for general correlation plot details.)

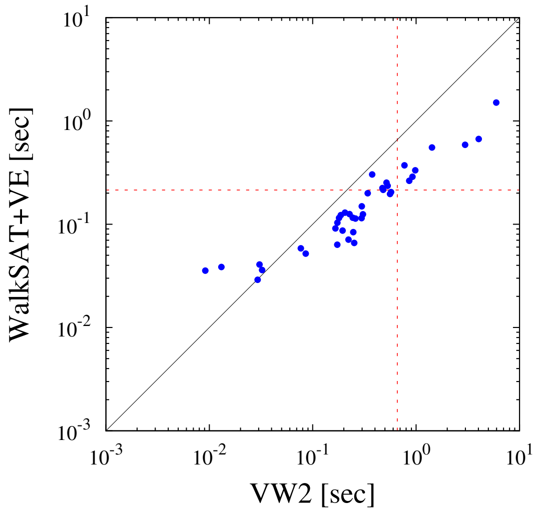
this benchmark (VW2). These results are especially impressive when examining run-length performance, but because of the complexity involved with this advanced VE, the results of the run-time performance is somewhat less impressive, but still significant. We were genuinely surprised that with this relatively modest modification to the venerable, but rather dated WALKSAT/SKC algorithm, we were able to outperform all known SLS algorithms. This experiment clearly demonstrates the potential of complex VEs as a basis for the development of new, high-performance SLS algorithms.

7.3 Modeling and Designing SLS Algorithms with VEs

Now that we have motivated our interest in VEs, we present our VE-based model. Our model, illustrated in Figure 7.5, includes an *algorithm controller* and three core stages: a variable filter stage, a VE evaluation stage and a variable selection stage. There is a final stage that simply flips the selected variable and updates the state information resulting from the flip (*e.g.*, property values) and any algorithm state information (such as the noise value in algorithms with adaptive noise). We



(a) run-length



(b) run-time

Figure 7.4: WALKSAT+VE *vs* VW2 on cbmc. The speedup factor (*s.f.*) is (a) 7.25 and (b) 3.07. (The parameters for VW2 are $(s, c, wp) = (0, 0.01, 0.2)$, found by PARAMILS (see Section C.7.1). The configuration of WALKSAT+VE was found by PARAMILS and is described in Section C.7.3. Each point corresponds to an instance in the set cbmc (test) (39 instances). Median run-lengths and run-times are reported, obtained from 25 runs. Execution environment: UBC arrow cluster (Section C.1). See Section C.5 for general correlation plot details.)

first describe the three core stages and then describe the algorithm controller.

The **Variable Filter Stage** outputs a list of variables that are candidates to be flipped in this search step. For example, the clause-based filter used in WALKSAT/SKC [99] and VW2 [88] selects an unsatisfied clause uniformly at random, and then only the variables that appear in that clause are flip candidates. Other examples include the GSAT algorithm [100] which considers all variables, the SAPS algorithm which includes all variables that appear in unsatisfied clauses and the G^2WSAT algorithm [76] which includes a filter that only allows variables with a true promising property.

The **VE Evaluation Stage** is very straightforward. The input is the list of n flip candidates from the filter stage and k VEs from the controller. The output is an array of $n \cdot k$ values where each of the VEs are evaluated for each candidate.

The **Variable Selection Stage** makes the final decision as to which of the candidates will be flipped, based on the array of values from the VE evaluation stage. For simplicity, we assume that a single candidate is selected and flipped in each step, but in practice, the VSM could select zero or many candidates. For most existing SLS algorithms, the variable selection mechanism (VSM)

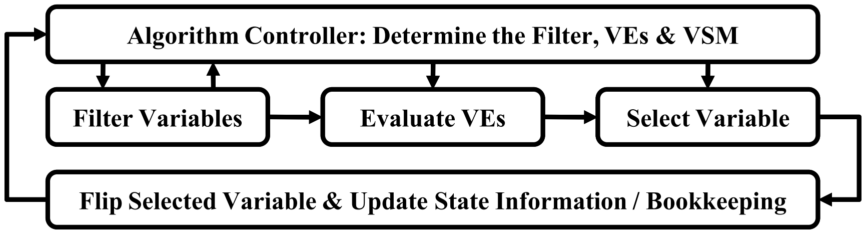

Algorithm Controller: Determine the Filter, VEs & VSM
Filter Variables
Evaluate VEs
Select Variable
Flip Selected Variable & Update State Information / Bookkeeping

Figure 7.5: Our conceptual SLS algorithm model.

is a simple max (or min) operation, where the candidate with the maximum value of the first VE is selected; additional VEs can be used for tie-breaking, and any remaining ties will be broken randomly. The NOVELTY algorithm we described earlier is an example of an algorithm with a VSM that incorporates multiple VEs.

The **Algorithm Controller** controls behaviour at each step by determining the components of each of the three stages: the filter, the set of VEs and the VSM. The controller may use the same components for every step, make independent random decisions for each step or it may use a more sophisticated decision mechanism. The GSAT algorithm represented in our model uses a simple controller, where the components are the same at every step: no filter (consider all variables), use a simple VE of ⟨score⟩ and a min VSM. The GWSAT algorithm adds a random walk to GSAT, and is represented in our model by a randomized controller that, with some probability, selects an alternate filter (only variables that appear in unsatisfied clauses) and a VSM that selects candidates randomly. In Figure 7.5, we indicate control flow from the filter back to the controller to allow for controllers that may wish to re-filter the variables or defer the determination of the VEs or VSM until after the filter results are known. For example, as a form of clause normalization (see Section 7.2.3), a controller could use a random-clause-based filter and choose VEs based on the length of the selected clause.

In our model, complex controllers can be constructed that do not directly decide the components for the three stages, but instead utilize a number of sub-controllers. Since each sub-controller can correspond to a unique algorithm (or the same algorithm with different parameter settings), this allows the construction of *hybrid* algorithms. A hybrid algorithm can switch between different algorithms randomly, periodically, when some criteria is met (*e.g.*, search stagnation is detected) or according to some other customized mechanism. G^{2}WSAT is one such hybrid algorithm, where if any variables have a true promising property, a GSAT-based step occurs, otherwise, a WALKSAT-based step occurs [76].

Now that we have presented our highly flexible model, we briefly outline our Design Architecture for Variable Expressions (DAVE), based on our versatile UBCSAT architecture [113]. One of the design goals of DAVE was to reduce (and potentially eliminate) the programming component of algorithm design by allowing the entire algorithm behaviour to be specified at run-time. The user can specify the algorithm controller (and sub-controllers), the filter(s), the VE(s), the VSM(s) and any additional UBCSAT triggers to activate. The only programming required is to introduce *new* variable properties, controllers, filters or VSMs. Because the *configuration space* of DAVE is actually an *algorithm specification space*, when we use DAVE in combination with an automated configurator, we can find optimized algorithm specifications automatically. To further facilitate the use of a configurator, DAVE supports a sophisticated macro-based syntax that allows controllers, filters, VEs, and VSMs to be highly parameterized.

In DAVE, most variable properties depend on the current value of the variable assignment. We use the notation p' to correspond to the property value for the negation of a given variable. For example, the flips property in DAVE is actually half of the total flip count $(\mathsf{flips}+\mathsf{flips}')$. Similarly, age' ignores the most recent flip and measures the number of search steps that have occurred since the flip prior to the most recent flip.

The only other implementation detail of DAVE that we address here, as it is specifically relevant to the presentation and understanding of the performance results we report later, is the interpreted nature of the algorithms specified in DAVE. Since DAVE receives the algorithm specification and VEs at run-time, the code is not natively compiled. Instead, each operation is individually interpreted and executed. This means that an algorithm in DAVE will not achieve the same run-time performance as the equivalent algorithm in compiled source code, which is why we encourage measuring DAVE algorithms by run-length performance where there is no such penalty. In preliminary experiments, we have seen algorithms in DAVE run 1.5-3 times slower than their native implementations, where the speed of DAVE is often more a function of the number of operators used in the VE, as opposed to the true complexity of the algorithm. This is one reason why we present DAVE as a design architecture that facilitates the exploration of new algorithmic ideas. It is our intent that new and robust algorithms developed in DAVE will subsequently be incorporated directly in UBCSAT as stand-alone optimized algorithms. We are currently in the preliminary stages of developing a software tool that can automatically generate fast, native source code to implement an algorithm specified in DAVE.

7.4 VE-SAMPLER: Exploring New SLS Methods using DAVE

In this section we introduce a new algorithm framework we call VE-SAMPLER. VE-SAMPLER uses a randomized controller that selects between six sub-controllers, where each sub-controller is selected with a probability proportional to a configurable weight. Each of the six sub-controllers uses a simple max VSM, and has a configurable clause-based filter, where the unsatisfied clause selected is either random, the clause unsatisfied the longest, or the clause most frequently unsatisfied. The VE of the first sub-controller is $\langle\mathsf{freebie}\rangle$, similar to the random walk in WALKSAT/SKC [99], where no random walk occurs if a freebie exists. The max VSM will select all freebie candidates, or all candidates if no freebies exist, and then break ties randomly.

The VEs for the other five sub-controllers are all of the form:

$$\|\mathsf{p1}\|^{a_1} + \mathrm{clw}(s,m,l) \cdot \|\mathsf{p2}\|^{a_2} \ , \tag{7.8}$$

where p1 and p2 are configurable, and correspond to variable properties (or a ratio of properties) selected from lists we describe below. The $\mathrm{clw}()$ function represents a simple mechanism we created to address clause normalization (briefly discussed in Section 7.2.3) in a practical, yet interesting way. The three configurable parameters of $\mathrm{clw}(s,m,l)$ correspond to scaling coefficients that depend on whether the clause length is small (< 3), medium ($= 3$), or large (> 3) (*i.e.*, if the clause length is two then $\mathrm{clw}(s,m,l) = s$).

The normalization and non-linear transformation used in Equation 7.8 is similar to VW2+VE and WALKSAT+VE. We chose to use only two properties to avoid the poor run-time performance we observed with four properties in WALKSAT+VE. However, we believe that our approach of using multiple VEs via a controller can provide a similar level of algorithm robustness without significantly degrading per-step time complexity.

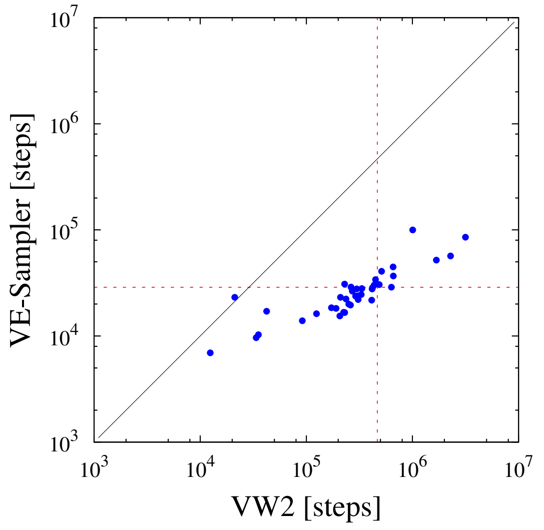
Of the five sub-controllers, one was configured to have only *greedy* properties similar to WALKSAT+VE, while the remaining four were configured to have one greedy property (p1) and one *diversification* property (p2) similar to VW2+VE. The five greedy properties available were score, make, relMake, break and relBreak.

We wanted diversification properties that were independent of the greedy variable properties and required little or no computational overhead to maintain. For VE-SAMPLER, we created the following new properties: filtCount is incremented every search step where the variable (with its current value) has appeared in the list of flip candidates, relFiltCount is similar, but increases by $1/\mathrm{clauselen}$, and goodFlips and badFlips are incremented every time the variable (with its current value) is flipped and the number of satisfied clauses goes up or down, respectively. In total, there were thirteen diversification properties (or ratios of properties) available in VE-SAMPLER:

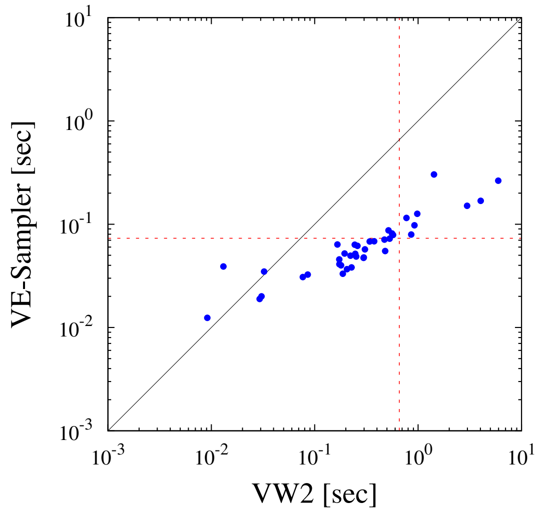
flips,	age/flips,	relFiltCount,	goodFlips/flips,
age,	age′/age,	relFiltCount/flips,	goodFlips/goodFlips′,
age′,	filtCount,	relFiltCount/relFiltCount′,	goodFlips/badFlips

and rand, which draws a number uniformly at random from the interval $[0,1]$. While some of these properties are based on prior evidence and intuition, others are simply interesting ideas that we thought might be effective.

Our goal with VE-SAMPLER was to make very few decisions at design time and to configure the resulting, highly parameterized algorithm automatically for optimized performance [52]. In total, VE-SAMPLER has over 10^{50} possible configurations, which, to the best of our knowledge, is the largest design space searched using PARAMILS so far. The experiments for the cbmc and swv



(a) run-length



(b) run-time

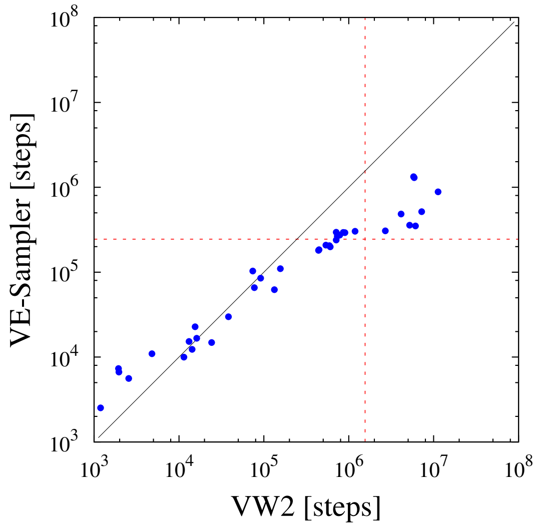
Figure 7.6: VE-SAMPLER *vs* VW2 on cbmc. The speedup factor (*s.f.*) is (a) 16.2 and (b) 9.0. (The parameters for VW2 are $(s,c,wp) = (0, 0.01, 0.2)$, found by PARAMILS (see Section C.7.1). The configuration of VE-SAMPLER was found by PARAMILS and is described in Section C.7.4. Each point corresponds to an instance in the set cbmc (test) (39 instances). Median run-lengths and run-times are reported, obtained from 25 runs. Execution environment: UBC arrow cluster (Section C.1). See Section C.5 for general correlation plot details.)

test sets were conducted independently, and PARAMILS found a different configuration for each set, as described in Section C.7. We present the results of our PARAMILS-configured VE-SAMPLER in Figure 7.6, Figure 7.7 and Table 7.1. We compared VE-SAMPLER against the SLS-based solvers VW2 and SATENSTEIN (see Section 7.5), both also configured with PARAMILS (see Section C.7). The results we present were obtained using a compiled version of VE-SAMPLER, where the original version, implemented in DAVE, was approximately 1.5 times slower.

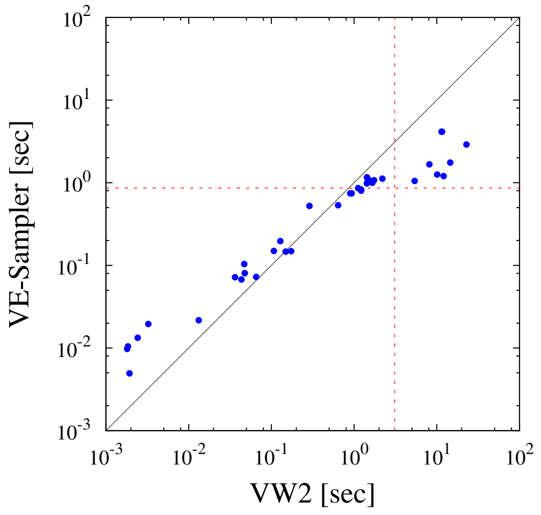
VE-SAMPLER performs substantially better than VW2 and SATENSTEIN on our cbmc test set, especially in terms of search steps. On the much more challenging real-world software verification instances from the swv set, VE-SAMPLER also performs significantly better than VW2 and SATENSTEIN. We note that none of the SLS algorithms we are aware of can solve more than about half of the complete set of swv instances within our 600 second cutoff, but VE-SAMPLER does solve this half of the instances more efficiently than any other SLS algorithm. While the results in Table 7.1 are impressive and represent the current state-of-the-art in SLS-based SAT solvers on these types of instances, the DPLL-based solver PICOSAT [12] is twice as fast as VE-SAMPLER on cbmc, seven times as fast on swv (partial) and can solve any instance from the full swv set in

Algorithm	cbmc			swv (partial)				swv (full)	
	Steps	Time		Steps	Time		%	PAR	%
	$\times 10^3$	sec.	*s.f.*	$\times 10^3$	sec.	*s.f.*	Compl.		Compl.
VW2-SAT05	3 577	6.22	0.11	10 089	19.20	0.16	100	3 008	50.1
VW2	467	0.66	*ref.*	1 555	3.10	*ref.*	100	3 042	49.3
SATENSTEIN	228	0.80	0.82	1 465	12.50	0.25	100	3 040	49.5
VE-SAMPLER	**29**	**0.07**	**9.00**	**245**	**0.90**	**3.61**	100	**2 664**	**50.7**

Table 7.1: VE-SAMPLER *vs* VW2 and SATENSTEIN on cbmc and swv. Values shown are the means of the median run-length and run-time from (a) 25 runs on instances from the cbmc test set and (b) 10 runs on instances from swv. The *s.f.* is measured against the run-time of VW2. All algorithms completed 100% of the cbmc instances. The PAR (Penalized Average Run-time) is the average from all runs on all instances, where incomplete runs after 600 seconds are penalized by a factor of 10 (6 000 seconds) (see [70] for details). All algorithms (except the parameterless VW2-SAT05) were optimized by PARAMILS (see Section C.7).



(a) run-length



(b) run-time

Figure 7.7: VE-SAMPLER *vs* VW2 on swv (partial). The speedup factor (*s.f.*) is (a) 6.36 and (b) 3.61. (The parameters for VW2 are $(s, c, wp) = (0, 0.1, 0.05)$, found by PARAMILS (see Section C.7.1). The configuration of VE-SAMPLER was found by PARAMILS and is described in Section C.7.4. Each point corresponds to an instance in the set swv (partial) (test) (39 instances). Median run-lengths and run-times are reported, obtained from 10 runs. Execution environment: UBC arrow cluster (Section C.1). See Section C.5 for general correlation plot details.)

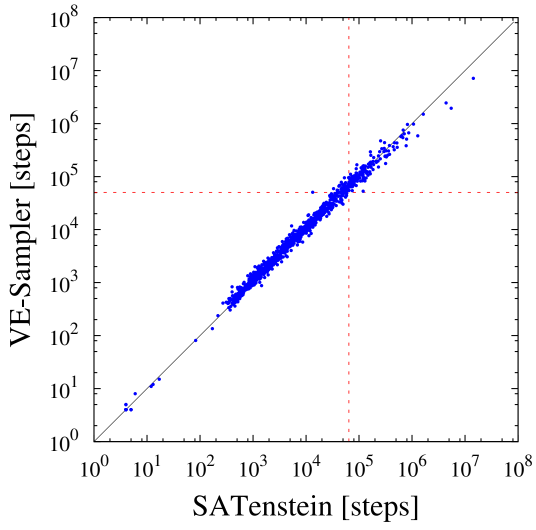
just a few CPU seconds. Thus, while we have considerably reduced the performance gap between SLS-based and DPLL-based SAT solvers on these software verification instances, there is still much room for improvement.

When studying the VE-SAMPLER configurations found by PARAMILS, we observed that configurations with similarly good performance often had substantially different configurations. This might suggest that VE-SAMPLER is somewhat robust with respect to its configuration, and that PARAMILS was far from finding the true optimal configuration of VE-SAMPLER (with over 10^{50} possible configurations, this is not surprising). We also observed configurations where two sub-controllers would be configured to use the same variable properties, but to be rather different otherwise. This was the case in the configurations featured in the results above, where the final cbmc configuration heavily weighted two sub-controllers with the properties relMake and age′, and the final swv configuration heavily weighted two sub-controllers with the properties break and flips (see Section C.7.4). We believe this suggests that a hybrid algorithm including multiple configurations of the same underlying mechanism can achieve very robust performance.

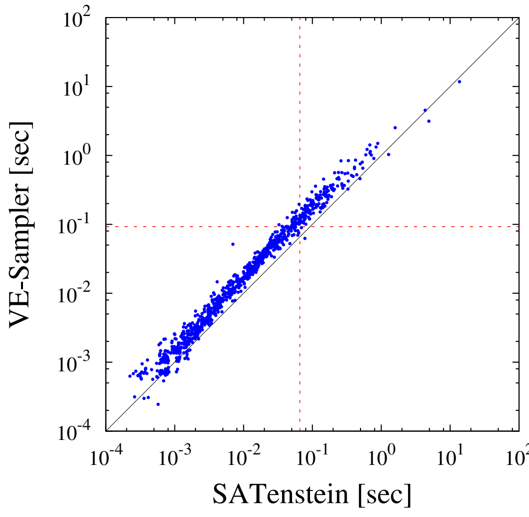
After experimenting with VE-SAMPLER on the software verification instance sets cbmc and swv, we were curious to see how VE-SAMPLER would perform on instance sets where SLS algorithms are currently the best known approach. We selected the crafted set qcp and the random set r3sat used by KhudaBukhsh *et al.* in their work on SATENSTEIN [70], which they showed to be the best performing SAT solver on these sets. As with our previous experiments, the sets were split into a training set and a test set, and for qcp and r3sat we used the same subsets as the SATENSTEIN author. We used PARAMILS to configure VE-SAMPLER on each of the training sets and present the results of our experiments on the test sets in Figure 7.8, Figure 7.9 and Table 7.2. The results we present were obtained using a compiled version of VE-SAMPLER, where the original version, implemented in DAVE, was approximately 1.5 times slower.

On the qcp test set, VE-SAMPLER outperforms both SATENSTEIN and VW2 in run-lenth performance (which is the criterion we optimized), but is slower with respect to run-time performance. On the r3sat test set, VE-SAMPLER outperforms only VW2 in run-length performance, and is almost twice as slow as both SATENSTEIN and VW2 with respect to run-time performance. However, based on the results presented by KhudaBukhsh *et al.* [70], VE-SAMPLER significantly outperforms all eleven of the state-of-the-art challenger algorithms on qcp and six of the challengers on r3sat.

We designed VE-SAMPLER to demonstrate the power of VEs, and our results on the software verification instances are especially impressive because the other components of the algorithm design (*e.g.*, algorithm controllers, filters, variable selection mechanisms) are very straightforward. However, while the results for VE-SAMPLER on the the qcp and r3sat domains are still very competitive, to advance the state-of-the-art in these domains it is not sufficient to focus solely on VEs, and additional components of our algorithm model also need to be explored.



(a) run-length

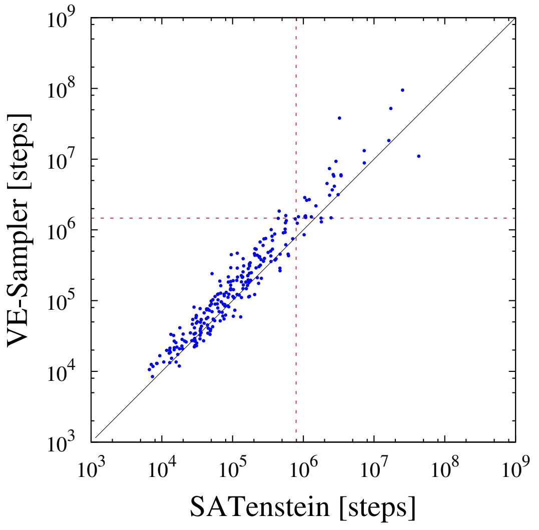


(b) run-time

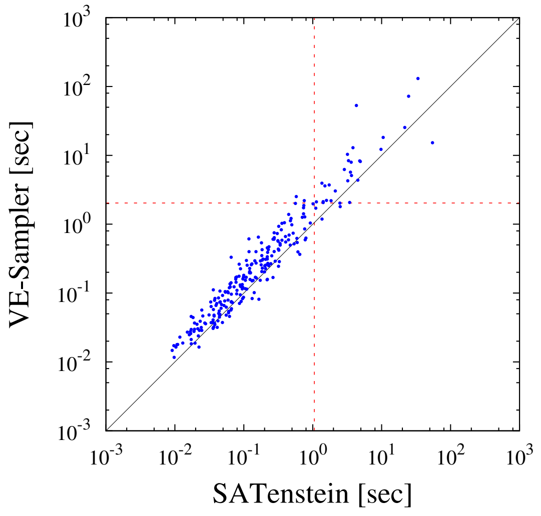
Figure 7.8: VE-SAMPLER *vs* SATENSTEIN on qcp. The speedup factor (*s.f.*) is (a) 1.28 and (b) 0.71. (The configuration of SATENSTEIN is from the author [69]. The configuration of VE-SAMPLER was found by PARAMILS and is described in Section C.7.4. Each point corresponds to an instance in the set qcp (test) (1000 instances). Median run-lengths and run-times are reported, obtained from 25 runs. Execution environment: UBC arrow cluster (Section C.1). See Section C.5 for general correlation plot details.)

Algorithm	qcp			r3sat		
	Steps	Time		Steps	Time	
	$\times 10^3$	sec.	*s.f.*	$\times 10^3$	sec.	*s.f.*
VW2-SAT05	$>$ 21 048	$>$ 23	$<$ 0.01	4 341	2.82	0.38
VW2	95.4	0.074	0.89	1 801	**1.04**	1.02
SATENSTEIN	64.3	**0.066**	*ref.*	**790**	1.06	*ref.*
VE-SAMPLER	**50.3**	0.092	0.71	1 466	2.03	0.52

Table 7.2: VE-SAMPLER *vs* VW2 and SATENSTEIN on qcp and r3sat. Values shown are the means of the median run-length and run-time from 25 runs on instances from the qcp and r3sat test sets. The *s.f.* is measured against the run-time of SATENSTEIN. All algorithms completed 100% of the runs except for VW2-SAT05 on qcp, where 188 of 1000 instances had a median beyond the cutoff of 10^8 steps. All algorithms (except the parameterless VW2-SAT05) were optimized by PARAMILS (see Section C.7).



(a) run-length



(b) run-time

Figure 7.9: VE-SAMPLER *vs* SATENSTEIN on r3sat. The speedup factor (*s.f.*) is (a) 0.54 and (b) 0.52. (The configuration of SATENSTEIN is from the author [69]. The configuration of VE-SAMPLER was found by PARAMILS and is described in Section C.7.4. Each point corresponds to an instance in the set r3sat (test) (250 instances). Median run-lengths and run-times are reported, obtained from 25 runs. Execution environment: UBC arrow cluster (Section C.1). See Section C.5 for general correlation plot details.)

7.5 Related Work

The manner in which SLS algorithm hybrids can be implemented in DAVE can be seen as a generalization of the HYBRID algorithm by Wei *et al.* [117]. HYBRID implements a clever heuristic to select between the algorithms VW2-SAT05 and ADAPTIVE G^2WSAT at each search step. Their heuristic corresponds to a specific algorithm controller in our model, and once implemented in DAVE, it becomes a universal controller that can be used to select between *any* two algorithms. Furthermore, the selection of the algorithms to be hybridized can be achieved by using an automated configurator.

DAVE is conceptually related to the SATENSTEIN solver by KhudaBukhsh *et al.* [70], which also extends UBCSAT, albeit in a different direction. SATENSTEIN incorporates proven components from over two dozen existing SLS algorithms, including GNOVELTY$^+$ [87], ADAPTIVE G^2WSAT+P [77], SAPS [61] and PAWS [111] and can be configured to instantiate any of those algorithms, as well as many complex hybrids. SATENSTEIN is very efficient when properly configured and is the best known SLS algorithm on several benchmark sets [70]. Whereas the SATENSTEIN authors liken their generated algorithms to Frankenstein's monster, stitched together from existing algorithm parts, we believe that our model is more akin to a mad scientist experimenting with algorithmic DNA. The significant difference is that SATENSTEIN has a bounded configuration space, whereas DAVE is a design environment that supports arbitrarily complex algorithms in a potentially unbounded space.

In that latter respect, DAVE is similar in nature to the COMPOSITE HEURISTIC LEARNING ALGORITHM FOR SAT SEARCH (CLASS) by Fukunaga [35]. CLASS is a genetic programming system that constructs new variable selection heuristics. Our work with VEs is somewhat orthogonal to the research direction underlying CLASS; our goal has been to decouple the scoring functions (VEs) from the VSMs and focus on the VEs, whereas in CLASS they are tightly coupled. There is potential for combining the strategies of DAVE and CLASS, and we are considering incorporating a CLASS-like syntax for VSMs into a future version of DAVE. Conversely, CLASS could be extended by incorporating our concept of VEs.

7.6 Conclusions

In this work, we proposed a new conceptual model for SLS algorithms based on variable expressions (VEs), and we demonstrated that algorithms with complex VEs can be very effective in practice. We created a new software framework for designing new SLS algorithms and algorithm hybrids in our model, and we demonstrated that by combining our software with an automated algorithm configuration tool, it was rather easy to construct a new algorithm that is nine times faster than the existing state-of-the-art SLS-based SAT solvers on a set of software verification instances known from the literature. In Section 8.3, we discuss the future potential of this work and identify ways we believe it can be extended.

Chapter 8

Conclusions

. . . with satisfaction when we're done. . .
— Deee-Lite. "Groove Is in the Heart"

This final chapter is structured as follows. First, in Section 8.1, we summarize the primary contributions we have made in this dissertation. Next, in Section 8.2, we reflect on the journey we have taken in this dissertation, and how the theme of Dynamic Local Search (DLS) has been prevalent throughout. Finally, in Section 8.3, we identify ways in which our work can be extended.

8.1 Contributions

As we stated in Chapter 1, our primary goal in this dissertation was to advance the state-of-the-art for SLS-based SAT solving. We accomplished this goal explicitly by developing new SLS algorithms that outperform the current state-of-the-art SLS-based SAT solvers on interesting benchmark problems, and implicitly by advancing the understanding of current SAT solvers and introducing development tools for the next generation of SAT solvers. More specifically, our contributions are as follows:

1. We developed UBCSAT, a framework for efficiently implementing and empirically evaluating SLS algorithms (Chapter 3).
2. We created the SCALING AND PROBABILISTIC SMOOTHING (SAPS) algorithm, and demonstrated that SAPS dominates the performance of its predecessor, the EXPONENTIATED SUBGRADIENT (ESG) algorithm [97], and is amongst the state-of-the-art SLS algorithms for SAT for some benchmark instances (*e.g.*, the fac instances) (Chapter 4).
3. We provided an in-depth study of DLS-CP algorithms, advancing our understanding of their behaviour. We discovered that there are interesting examples of instances where DLS-CP

algorithms can identify problem clauses that can be weighted to make solving the instance easier, but we demonstrated that this behaviour is rare and not how DLS-CP algorithms solve most instances in practice. We concluded that typically only the very short-term memory of DLS-CP algorithms is useful, and primarily for escaping local minima (Chapter 5).

4. We studied the role of random decisions in SLS algorithms, and performed an empirical analysis on both the quality and quantity of random decisions. We concluded that SLS algorithms are very robust with respect to the quality of their randomness source, and that widely available Pseudo-Random Number Generators (PRNGs) are of sufficient quality for implementing SLS algorithms. We presented evidence that even highly randomized SLS algorithms can be derandomized in a straightforward manner without significantly changing their behaviour. We observed an interesting phenomenon where, by making only one or two changes in their initial variable assignment, derandomized algorithms can exhibit the same full variability in run-time observed for randomized algorithms. (Chapter 6).

5. We introduced a new conceptual model for representing and designing new SLS algorithms with Variable Expressions (VEs). We developed the DESIGN ARCHITECTURE FOR VARIABLE EXPRESSIONS (DAVE), an extension of UBCSAT that implements our model. DAVE was designed to leverage the use of recent automated algorithm configuration tools for the automated development of new algorithms. We demonstrated that by following our new algorithm design approach, we achieved significant improvements over previous state-of-the-art SLS-based SAT solvers on software verification benchmark instances (Chapter 7).

Almost all of the experiments in our work were conducted from within the UBCSAT framework, which is why in Chapter 1 we referred to it as the cornerstone of our dissertation. The task of collecting the clause penalty data from multiple DLS-CP algorithms (Chapter 5) would have been very onerous without UBCSAT. The use of weighted algorithm variants and our ability to create new variants, such as the +AMNESIA algorithms in Chapter 5 and the derandomized algorithms in Chapter 6, was also facilitated by UBCSAT. In addition, we were able to easily extend UBCSAT to add support for providing arbitrary sources of random data. Most importantly, we were able to develop DAVE by extending the UBCSAT framework.

Beyond this dissertation, UBCSAT has been a significant contribution to the research community. It is difficult to obtain exact usage statistics for UBCSAT, but based on the Google analytics tool [124], in the eight month period from September 1, 2009 through April 30, 2010 the UBCSAT website [141] was visited by over 1 000 visitors (over 750 unique visitors), from over 60 different countries. Since launched in 2004, UBCSAT has been downloaded hundreds of times. Google Scholar [139] lists over 50 publications that cite UBCSAT [113]. We find this evidence very encouraging, and when combined with the positive anecdotal feedback we have received from UBCSAT users, it reinforces our belief that UBCSAT has been a worthwhile and meaningful endeavour.

One of the most recent uses of UBCSAT is the SATENSTEIN solver [70], which was able to extend UBCSAT to create over 10^{11} unique algorithm hybrids from the algorithms that exist in UBCSAT. Another interesting use of UBCSAT is its inclusion in SATZILLA [122], the overall winner of the 2009 SAT Competition [137]. SATZILLA is a portfolio solver that selects a SAT solver to solve an instance based on features of the instance. It uses the robust statistical reporting abilities of UBCSAT to collect features, including several measures of the behaviour of the SAPS algorithm during a brief search through the instance. Interestingly, the behaviour of SAPS reported by UBCSAT is useful even when selecting from amongst DPLL-based algorithms to solve an instance [84].

This use of the SAPS algorithm in SATZILLA also speaks to the success of SAPS. Google Scholar lists over 100 publications that cite SAPS [61], and it has been used as a state-of-the-art benchmark algorithm in numerous publications. SAPS was one of the algorithms included in the aforementioned SATENSTEIN solver, where many components of the SAPS algorithm were combined with components from other algorithms to make interesting hybrids. A noteworthy and frequent use of SAPS has been in the study of automated algorithm configuration tools. The developers of both PARAMILS [59] and the GENDER-BASED GENETIC ALGORITHM (GGA) [4] have used SAPS in their experiments as an algorithm with a small, but interesting parameter configuration space.

In Chapter 5, we provided a comprehensive analysis of how DLS-CP algorithms behave in practice. We believe that our empirical study and methodology was consistent with the principals prominently advocated by Hooker [46, 47], and that insights such as these will ultimately make it possible to design better algorithms. The experimental approach we adopted to study random decisions in Chapter 6 was motivated similarly, and was recognized as a significant contribution when its publication [114] won the best paper award at the 2006 Canadian Conference on Artificial Intelligence.

Our work in Chapter 7 was only recently published, so it is too early to assess its reception. Ultimately, we believe that our DAVE project will have the most significant impact on the development of new state-of-the-art SLS algorithms for SAT. In Section 8.3, we discuss the future potential of this work and identify ways we believe it can be extended.

8.2 Overview: Dynamic Local Search for SAT

As we foreshadowed in Chapter 1, Dynamic Local Search (DLS) has been the prevalent theme throughout this dissertation. We have defined a DLS algorithm as an algorithm that incorporates its search history to dynamically adjust its search behaviour. Throughout this dissertation, we have described numerous variable and clause properties that algorithms use in their search. In this context, a useful alternate definition of a DLS algorithm is an algorithm that uses dynamic properties. Non-dynamic properties are either static properties, such as the length of a clause, or scoring properties that reflect the current variable assignment, such as the score variable property and the satisfied clause property. Dynamic properties incorporate elements of the search history, such as the age and flips variable properties and the penalty clause property. Our work with UBCSAT aided our study and development of DLS algorithms by creating a framework that facilitates the creation of dynamic properties and the ability to collect statistical information on their behaviour.

At the beginning of our journey, we were fascinated by how SAPS incorporated history into its search, not directly as with the age variable property, but indirectly with the use of the penalty clause property. This fascination led us down a path that began with SAPS in Chapter 4 and eventually led to the rigorous experimental analysis of DLS-CP algorithms we conducted in Chapter 5. Along this journey we made some interesting observations and arrived at some important conclusions. In particular, we concluded that for some instances we can study the dynamic clause penalties generated by a DLS-CP algorithm such as SAPS to identify problem clauses that be used to make solving the instance easier. (see Chapter 5).

We continued along this path of discovery by studying the role of random decisions in SLS algorithms. In Chapter 6, we were able to replace the random decisions in SAPS with dynamic properties. From this we concluded that a duality may exist between random decisions and the dynamic search history (*i.e.*, they both give rise to the same behaviour). Surprisingly, we were able to demonstrate this duality on algorithms that heavily rely on random decisions. We were able to successfully replace the random decisions in CRWALK and ADAPTIVE NOVELTY$^+$ with dynamic properties.

When we started our work in Chapter 7, our interest in dynamic clause properties such as penalty had waned, and we saw more promising directions for further exploration. We decided to revisit DLS algorithms from a much broader perspective, and we started to develop a new conceptual model for SLS algorithms. We soon realized that even the most straightforward of dynamic properties, such as flips, could be used more effectively by SLS algorithms, and we demonstrated this in Chapter 7. By shifting the focus from Variable-Selection Mechanisms (VSMs) to Variable Expressions (VEs), we also shifted the focus to dynamic properties. We believe there is great potential for new DLS algorithms that can use new and existing dynamic clause and variable properties in novel and interesting ways. Ultimately, we are looking forward to continuing this journey of exploration and to discovering what lies ahead.

8.3 Future Work

In the previous two sections we summarized our work and our contributions. In this section, we identify several exciting directions in which our research can be extended.

We believe that much of our work can be extended beyond SAT to other problem domains. Our UBCSAT strategy, to design a framework for efficiently implementing and empirically analyzing algorithms, can be extended to nearly any domain. We believe our method of implementing triggered procedures will provide insight to other developers struggling with the practical implementation issues of such frameworks. Algorithm designers from other domains will be able to emulate our approach with DAVE, by developing flexible design architectures that can benefit from the availability of automated configuration tools. For domains where SLS approaches are prominent and successful, we believe that many of the lessons we learned from our work in SLS for SAT will also apply. For example, we believe that our conclusions on the role of random decisions in SLS algorithms for SAT are applicable to all SLS algorithms. The BREAKOUT approach is popular in other domains (*e.g.*, the Constraint Satisfaction Problem (CSP)), and our work on analyzing and understanding the behaviour of DLS-CP algorithms for SAT can provide new insight into the behaviour and efficacy of similar algorithms in other domains. In addition, we believe that our work with variable expressions, where we combine properties together in new and interesting ways, will be especially applicable and effective in other domains.

The UBCSAT project is an ongoing effort, and we are continuously expanding and enhancing the software. We anticipate that whenever new, effective and interesting SLS algorithms are developed we will want to incorporate them into UBCSAT, and we often receive such requests from the community. We also receive requests to enhance UBCSAT with new features. The two most popular requests are to extend UBCSAT to a true 64-bit architecture, allowing for search trajectories and data collection that extend beyond the 32-bit barrier, and to improve the library interface of UBCSAT so it can be more easily incorporated into larger projects. Both of these requested features are forthcoming.

Following our success with DAVE, we believe that automated algorithm configuration tools (such as PARAMILS) can be used to not only configure a single algorithm, but also to select from amongst a collection of configurable algorithms. While existing algorithm configuration tools are already sufficient, we propose that future tools may be more effective if they are designed specifically as portfolio-based (multiple algorithm) configuration tools. In future releases of UBCSAT we plan to include a PARAMILS configuration (specification) file that will allow PARAMILS to determine the best configuration of UBCSAT (*i.e.*, the best algorithm and the best configuration of that algorithm) for a given instance (or instance set). We believe that this combination of UBCSAT and an automated algorithm configuration tool can be used to automatically establish a large online repository (*e.g.*, on SATLIB), where for any given instance or set of instances the best-known configuration of UBCSAT would be widely available. This information would help new algorithm developers measure their performance against the existing state-of-the-art algorithms. The combination of UBCSAT and PARAMILS can also be used in the automatic construction of portfolio-based solvers, a new and exciting approach recently demonstrated by Xu *et al.* with their HYDRA procedure [120].

As we look to the future of SAT solving, there are two features missing from UBCSAT that we believe could help facilitate the next generation of SLS solvers: support for dynamic instances and distributed message passing. Algorithms with dynamic instances could change the instance during the search by adding and removing clauses and variables; DPLL-based solvers often use this strategy where new clauses are learned during the search. While this is not a new idea for SLS algorithms [14, 45, 91], we believe there is greater potential for SLS algorithms to employ similar strategies. With distributed message passing, UBCSAT could communicate with other applications (including other instantiations of UBCSAT), to exchange information and coordinate search effort. With dynamic message passing, new algorithms could be developed to exploit the parallel architectures that are becoming increasingly abundant in hardware. As an example of the potential of message passing, Kroc *et al.* extended UBCSAT to communicate with a DPLL-based solver to coordinate information to solve certain types of MAX-SAT instances instances very effectively [72].

We believe that in the light of our experience with SAPS and the knowledge we have gained in more recent work, there is the potential for developing new, powerful DLS-CP algorithms. We have seen that the random walk mechanism in SAPS is ineffective, and we believe that adding a more explicit parameterized diversification mechanism would be more useful. We have encountered instances where multiple scaling steps are necessary to escape a local minima and we would like to avoid such repetition by determining the minimum scaling factor α required to escape minima in a manner similar to SMOOTHED DESCENT AND FLOOD (SDF). As we have seen from our study in Chapter 5, there are instances where problem clauses can be identified and exploited, but for the majority of instances this is not the case, and dynamic penalties are simply a diversification mechanism. We believe there is potential to develop two flavours of DLS-CP algorithms: one specifically designed to detect and exploit problem clauses, and another primarily focused on using DLS-CP as a diversification strategy. We believe that experimenting with normalizations, non-symmetric and non-linear interactions in the penalized scoring functions can produce strong results, similar to those we observed in Chapter 7. Finally, we believe that within the DAVE architecture, we can combine penalized properties with other dynamic properties (such as flips and age) to develop interesting new DLS-CP algorithms.

With respect to our work on randomization in Chapter 6, we believe the biggest potential for improving state-of-the-art SAT solvers is in the further study of the circumstances when derandomized algorithms perform better than or worse than the corresponding randomized algorithm. For those instance and problem domains where derandomized algorithms perform better, perhaps new algorithms could be developed that are more fair and balanced, and would use random decisions more sparingly. Conversely, for instances on which derandomized algorithms are found to perform poorly (*i.e.*, ferry9u for SAPS/NR), it would be interesting to further explore the reasons underlying the loss of performance, and to investigate in which mechanism of the algorithm derandomization is causing the problem. This information could be useful to help identify how to use random decisions more effectively.

The area of our dissertation where we see the biggest potential for further improvement in the state-of-the-art for SAT solving is our work in Chapter 7. Apart from the previously mentioned work on CLASS-based Variable-Selection Mechanisms (VSMs) (Section 7.5) and the automated generation of source code from DAVE configurations (Section 7.3), we see several other promising directions for future work. We expect that there are more variable properties that can be effectively incorporated into VEs, as well as more sophisticated ways of combining variable properties beyond the simple scaling and non-linear transformations we presented in this work. We especially believe that there are more effective ways to handle clause normalization. Now that we have conceptually separated the components of algorithm controllers, filters, VEs and VSMs, we believe that algorithm designers will be able to focus on those individual components. With the ability to quickly and automatically test ideas in DAVE, we anticipate rapid development in each of these components of algorithm design. Overall, we believe that the utilization of rich and flexible design environments, such as DAVE, in combination with powerful automated configuration tools will make it possible to achieve further, substantial progress in the state-of-the-art in SLS-based SAT solving.

Bibliography

[1] T. Alsinet, R. Béjar, A. Cabiscol, C. Fernàndez, and F. Manyà. Minimal and redundant SAT encodings for the all-interval-series problem. In *Proceedings of the Fifth Catalonian Conference on Artificial Intelligence (CCIA-02)*, volume 2504 of *Lecture Notes in Computer Science*, pages 139–144, 2002.
Referenced in text: page(s) 6

[2] Anbulagan, Duc Nghia Pham, John Slaney, and Abdul Sattar. Old resolution meets modern SLS. In *Proceedings of the Twentieth National Conference on Artificial Intelligence (AAAI-05)*, pages 354–359, 2005.
Referenced in text: page(s) 62

[3] Carlos Ansótegui, María Luisa Bonet, and Jordi Levy. On the structure of industrial SAT instances. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP-09)*, volume 5732 of *Lecture Notes in Computer Science*, pages 127–141, 2009.
Referenced in text: page(s) 125

[4] Carlos Ansótegui, Meinolf Sellmann, and Kevin Tierney. A gender-based genetic algorithm for the automatic configuration of algorithms. In *Proceedings of the Fifteenth International Conference on Principles and Practice of Constraint Programming (CP-09)*, volume 5732 of *Lecture Notes in Computer Science*, pages 142–157, 2009.
Referenced in text: page(s) 139, 161

[5] Călin Anton and Lane Olson. Generating satisfiable SAT instances using random subgraph isomorphism. In *Proceedings of the Twenty-Second Conference of the Canadian Society for Computational Studies of Intelligence (AI-06)*, volume 5549 of *Lecture Notes in Computer Science*, pages 16–26, 2009.
Referenced in text: page(s) 175

[6] Bengt Aspvall, Michael F. Plass, and Robert Endre Tarjan. A linear-time algorithm for testing the truth of certain quantified Boolean formulas. *Information Processing Letters*, 8:121–123, 1979.
Referenced in text: page(s) 5

[7] Gilles Audemard, Daniel Le Berre, Olivier Roussel, Inês Lynce, and João Marques-silva. OpenSAT: an open source SAT software project. In *Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*, pages 502–509, 2003.
Referenced in text: page(s) 41, 165

[8] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In *Proceedings of the Twenty-First International Joint Conference on Artificial Intelligence (IJCAI-09)*, pages 399–404, 2009.
Referenced in text: page(s) 8, 161

[9] Domagoj Babić and Alan J. Hu. Calysto: Scalable and precise extended static checking. In *Proceedings of the Thirtieth International Conference on Software Engineering (ICSE-08)*, pages 211–220, 2008.
Referenced in text: page(s) 120, 158, 176

[10] Vittorio Bagini and Marco Bucci. A design of reliable true random number generator for cryptographic applications. In *Proceedings of the First International Workshop on Cryptographic Hardware and Embedded Systems (CHES-99)*, volume 1717 of *Lecture Notes in Computer Science*, pages 204–218, 1999.
Referenced in text: page(s) 107

[11] Heiko Bauke and Stephan Mertens. Pseudo random coins show more heads than tails. *Journal of Statistical Physics*, 114:1149–1169, 2004.
Referenced in text: page(s) 106

[12] Armin Biere. PicoSAT essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:75–97, 2008.
Referenced in text: page(s) 120, 131, 165

[13] Armin Biere, Marijn J. H. Heule, Hans van Maaren, and Toby Walsh, editors. *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.
Referenced in text: page(s) 1, 8, 153

[14] Byungki Cha and Kazuo Iwama. Adding new clauses for faster local search. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 332–337, 1996.
Referenced in text: page(s) 142

[15] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the really hard problems are. In *Proceedings of the Twelfth International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 331–337, 1991.
Referenced in text: page(s) 6, 99

[16] Vašek Chvátal and Endre Szemerédi. Many hard examples for resolution. *Journal of the ACM*, 35(4):759–768, 1988.
Referenced in text: page(s) 176, 177

[17] Edmund Clarke, Daniel Kroening, and Flavio Lerda. A tool for checking ANSI-C programs. In *Proceedings of the Tenth International Conference on Tools and Algorithms for Construction and Analysis of Systems (TACAS-04)*, volume 2988 of *Lecture Notes in Computer Science*, pages 168–176, 2004.
Referenced in text: page(s) 172

[18] Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Third IEEE Symposium on the Foundations of Computer Science*, pages 151–158, 1971.
Referenced in text: page(s) 1, 5

[19] Evgeny Dantsin, Andreas Goerdt, Edward A. Hirsch, and Uwe Schöning. Deterministic algorithms for *k*-SAT based on covering codes and local search. In *Proceedings of the Twenty-Seventh International Colloquium on Automata, Languages and Programming*, pages 236–247, 2000.
Referenced in text: page(s) 114

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

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

[22] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 61–75, 2005.
Referenced in text: page(s) 167

[23] Niklas Eén, Alan Mishchenko, and Niklas Sörensson. An extensible SAT-solver. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 502–518, 2003.
Referenced in text: page(s) 42, 164

[24] Hai Fang and Wheeler Ruml. Complete local search for propositional satisfiability. In *Proceedings of the Ninteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 161–166, 2004.
Referenced in text: page(s) 10

[25] Valnir Ferreira Jr. and John Thornton. Longer-term memory in clause weighting local search for SAT. In *Proceedings of the Seventeenth Australian Joint Conference on Artificial Intelligence (AI-04)*, volume 3339 of *Lecture Notes in Computer Science*, pages 730–741, 2004.
Referenced in text: page(s) 21, 165

[26] Alan. M. Ferrenberg, D. P. Landau, and Y. Joanna Wong. Monte Carlo simulations: Hidden errors from "good" random number generators. *Physical Review Letters*, 69(23):3382–3384, 1992.
Referenced in text: page(s) 106

[27] Christoph Flamm, Ivo L. Hofacker, Peter F. Stadler, and Michael T. Wolfinger. Barrier trees of degenerate landscapes. *Z. Phys. Chem.*, 216:155–173, 2002.
Referenced in text: page(s) 99

[28] John Franco and Marvin Paull. Probabilistic analysis of the Davis Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5(1):77–87, 1983.
Referenced in text: page(s) 174

[29] Jeremy Frank. Weighting for Godot: Learning heuristics for GSAT. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 338–343, 1996.
Referenced in text: page(s) 44, 162

[30] Jeremy Frank. Learning short-term clause weights for GSAT. In *Proceedings of the Fifteenth International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 384–389, 1997.
Referenced in text: page(s) 44, 49, 87, 162

[31] Eugene C. Freuder, Rina Dechter, Matthew L. Ginsberg, Bart Selman, and Edward Tsang. Systematic versus stochastic constraint satisfaction. In *Proceedings of the Fourteenth International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 2027–2032, 1995.
Referenced in text: page(s) 11

[32] Alex Fukunaga. Automated discovery of composite SAT variable-selection heuristics. In *Proceedings of the Eighteenth National Conference in Artificial Intelligence (AAAI-02)*, pages 641–648, 2002.
Referenced in text: page(s) 159

[33] Alex Fukunaga. Efficient implementations of SAT local search (extended abstract). In *Proceedings of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, pages 287–292, 2004.
Referenced in text: page(s) 27

[34] Alex Fukunaga. Evolving local search heuristics for SAT using genetic programming. In *Proceedings of the 2004 Genetic and Evolutionary Computation Conference (GECCO-2004)*, volume 3103 of *Lecture Notes in Computer Science*, pages 483–494, 2004.

Referenced in text: page(s) 159

[35] Alex S. Fukunaga. Automated discovery of local search heuristics for satisfiability testing. *Evolutionary Computation*, 16(1):31–61, 2008.
Referenced in text: page(s) 136, 159

[36] Michael R. Garey and David S. Johnson. *Computers and Intractability: A Guide to the Theory of NP-Completeness*. Books in the Mathematical Sciences. W. H. Freeman and Company, 1979.
Referenced in text: page(s) 5

[37] Ian P. Gent, Holger H. Hoos, Patrick Prosser, and Toby Walsh. Morphing: Combining structure and randomness. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 654–660, 1999.
Referenced in text: page(s) 176

[38] Ian P. Gent, Iain McDonald, and Barbara M. Smith. Conditional symmetry in the all-interval series problem. In *Proceedings of the Third International Workshop on Symmetry in Constraint Satisfaction Problems (SYMCON-03)*, pages 55–65, 2003.
Referenced in text: page(s) 6

[39] Ian P. Gent and Toby 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.
Referenced in text: page(s) 19, 33, 105, 115, 120, 121, 163

[40] Ian P. Gent and Toby Walsh. Unsatisfied variables in local search. In *Hybrid Problems, Hybrid Solutions*, pages 73–85, 1995.
Referenced in text: page(s) 19, 33, 163

[41] Fred Glover. Future paths for integer programming and links to artificial intelligence. *Computers & Operations Research*, 13(5):533–549, 1986.
Referenced in text: page(s) 19, 21

[42] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, 1997.
Referenced in text: page(s) 19

[43] Carla P. Gomes and Bart Selman. Problem structure in the presence of perturbations. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 221–226, 1997.
Referenced in text: page(s) 174

[44] P. Hansen and B. Jaumard. Algorithms for the maximum satisfiability problem. *Computing*, 44:279–303, 1990.
Referenced in text: page(s) 34, 40, 166

[45] Edward A. Hirsch and Arist Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. *Annals of Mathematics and Artificial Intelligence*, 43(1):91–111, 2005.
Referenced in text: page(s) 142

[46] J. N. Hooker. Needed: An empirical science of algorithms. *Operations Research*, 42(2):201–212, 1994.
Referenced in text: page(s) 139

[47] J. N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1:33–42, 1995.
Referenced in text: page(s) 139

[48] Holger H. Hoos. *Stochastic Local Search - Methods, Models, Applications*. PhD thesis, Darmstadt University of Technology, 1998.
Referenced in text: page(s) 171

[49] Holger H. Hoos. On the run-time behaviour of stochastic local search algorithms for SAT. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 661–666, 1999.
Referenced in text: page(s) 16, 17, 33, 40, 164, 166

[50] Holger H. Hoos. SAT-encodings, search space structure, and local search performance. In *Proceedings of the Sixteenth International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 296–302, 1999.
Referenced in text: page(s) 6

[51] Holger H. Hoos. An adaptive noise mechanism for WalkSAT. In *Proceedings of the Eighteenth National Conference in Artificial Intelligence (AAAI-02)*, pages 655–660, 2002.
Referenced in text: page(s) 19, 33, 51, 64, 106, 158, 170

[52] Holger H. Hoos. Computer-aided design of high-performance algorithms. Technical Report TR-2008-16, University of British Columbia, 2008.
Referenced in text: page(s) 130

[53] Holger H. Hoos and Thomas Stützle. Evaluating Las Vegas algorithms – pitfalls and remedies. In *Proceedings of the Fourteenth Annual Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 238–245, 1998.
Referenced in text: page(s) 117

[54] Holger H. Hoos and Thomas Stützle. SATLIB: An online resource for research on SAT. In *SAT2000: Highlights of Satisfiability Research in the year 2000*, volume 63 of *Frontiers in Artificial Intelligence and Applications*, pages 283–292, 2000.
Referenced in text: page(s) 6, 175, 176, 177

[55] Holger H. Hoos and Thomas Stützle. *Stochastic Local Search: Foundations and Applications*. Morgan Kaufmann, 2005.
Referenced in text: page(s) 2, 8, 11, 13, 25, 27, 37, 39, 40, 48, 99, 106, 107, 179

[56] Holger H. Hoos and Edward Tsang. Local search methods. In Francesca Rossi, Peter van Beek, and Toby Walsh, editors, *Handbook of Constraint Programming*, Foundations of Artificial Intelligence, chapter 5, pages 135–168. Elsevier, 2006.
Referenced in text: page(s) 45

[57] Frank Hutter. *Automated Configuration of Algorithms for Solving Hard Computational Problems*. PhD thesis, University of British Columbia, 2009.
Referenced in text: page(s) 165

[58] Frank Hutter, Domagoj Babić, Holger H. Hoos, and Alan J. Hu. Boosting verification by automatic tuning of decision procedures. In *Proceedings of the Seventh International Conference on Formal Methods in Computer-Aided Design (FMCAD-07)*, pages 27–34, 2007.
Referenced in text: page(s) 120, 176

[59] Frank Hutter, Youssef Hamadi, Holger H. Hoos, and Kevin Leyton-Brown. Performance prediction and automated tuning of randomized and parametric algorithms. In *Proceedings*

of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP-06), volume 4204 of *Lecture Notes in Computer Science*, pages 229–243, 2006.
Referenced in text: page(s) xxiv, 118, 139, 165

[60] Frank Hutter, Holger H. Hoos, Kevin Leyton-Brown, and Thomas Stützle. ParamILS: An automatic algorithm configuration framework. *Journal of Artificial Intelligence Research*, 36:267–306, 2009.
Referenced in text: page(s) 120, 165

[61] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the Eighth International Conference on Principles and Practice of Constraint Programming (CP-02)*, volume 2470 of *Lecture Notes in Computer Science*, pages 233–248, 2002.
Referenced in text: page(s) xix, xxiv, 20, 34, 51, 53, 64, 105, 106, 135, 139, 166, 167

[62] Abdelraouf Ishtaiwi, John Thornton, Anbulagan, Abdul Sattar, and Duc Nghia Pham. Adaptive clause weight redistribution. In *Proceedings of the Twelfth International Conference on Principles and Practice of Constraint Programming (CP-06)*, volume 4204 of *Lecture Notes in Computer Science*, pages 229–243, 2006.
Referenced in text: page(s) 61, 160

[63] Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, and Duc Nghia Pham. Neighbourhood clause weight redistribution in local search for SAT. In *Proceedings of the Eleventh International Conference on Principles and Practice of Constraint Programming (CP-05)*, volume 3709 of *Lecture Notes in Computer Science*, pages 772–776, 2005.
Referenced in text: page(s) 34, 61, 64, 160

[64] Kazuo Iwama and Suguru Tamaki. Improved upper bounds for 3-SAT. In *Proceedings of the Fifteenth Annual ACM-SIAM Symposium on Discrete Algorithms (SODA-04)*, pages 328–328, 2004.
Referenced in text: page(s) 17

[65] David S. Johnson, Cecilia R. Aragon, Lyle A. McGeoch, and Catherine Schevon. Optimization by simulated annealing: An experimental evaluation; part II, graph coloring and number partitioning. *Operations Research*, 37(6):865–892, 1989.
Referenced in text: page(s) 173

[66] Anil P. Kamath, Narendra K. Karmarkar, K.G. Ramakrishnan, and Mauricio G.C. Resende. A continuous approach to inductive inference. *Mathematical Programming*, 67:215–238, 1992.
Referenced in text: page(s) 172, 173

[67] Henry Kautz and Bart 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.
Referenced in text: page(s) 172, 174

[68] Stephen H. Kellert. *In the Wake of Chaos : Unpredictable Order in Dynamical Systems*. University of Chicago Press, 1993.
Referenced in text: page(s) 114

[69] Ashiqur R. KhudaBukhsh. SATenstein: Automatically building local search SAT solvers from components. Master's thesis, University of British Columbia, 2009.
Referenced in text: page(s) 122, 134, 135, 167

[70] Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin 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.
Referenced in text: page(s) 8, 53, 57, 121, 132, 133, 135, 139, 167, 172, 173, 174, 175

[71] Donald E. Knuth. *The Art of Computer Programming, Volume 2 / Seminumerical Algorithms*. Addison-Wesley, 1969.
Referenced in text: page(s) 107, 108

[72] Lukas Kroc, Ashish Sabharwal, Carla P. Gomes, and Bart 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.
Referenced in text: page(s) 142

[73] Oliver Kullmann. The SAT 2005 solver competition on random instances. *Journal on Satisfiability, Boolean Modeling and Computation*, 2:61–102, 2006.
Referenced in text: page(s) 121

[74] Tracy Larrabee. Test pattern generation using boolean satisfiability. *IEEE Transactions on Computer-Aided Design*, 11(1):6–22, 1992.
Referenced in text: page(s) 176

[75] Daniel Le Berre and Laurent Simon. The essentials of the SAT 2003 competition. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*, volume 2919 of *Lecture Notes in Computer Science*, pages 452–467, 2003.
Referenced in text: page(s) 173

[76] Chu Min Li and Wen Qi Huang. Diversification and determinism in local search for satisfiability. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 158–172, 2005.
Referenced in text: page(s) 19, 34, 64, 120, 127, 128, 161, 164, 170

[77] Chu Min Li, Wanxia Wei, and Harry Zhang. Combining adaptive noise and look-ahead in local search for SAT. In *Proceedings of the Tenth International Conference on Theory and Applications of Satisfiability Testing (SAT-07)*, volume 4501 of *Lecture Notes in Computer Science*, pages 121–133, 2007.
Referenced in text: page(s) 34, 120, 135, 158, 164

[78] Makoto Matsumoto and Takuji Nishimura. Mersenne twister: a 623-dimensionally equidistributed uniform pseudo-random number generator. *ACM Transactions on Modeling and Computer Simulation*, 8(1):3–30, 1998.
Referenced in text: page(s) 33, 106, 108

[79] Bertrand Mazure, Lakhdar Saïs, and Éric Grégoire. Tabu search for SAT. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 281–285, 1997.
Referenced in text: page(s) 33, 162

[80] David McAllester, Bart Selman, and Henry Kautz. Evidence for invariants in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, pages 321–326, 1997.
Referenced in text: page(s) 17, 19, 30, 33, 34, 105, 119, 164, 166, 169

[81] Patrick Mills and Edward Tsang. Guided local search applied to the satisfiability (SAT) problem. In *Proceedings of the Fifteenth National Conference of the Australian Society for Operations Research (ASOR-99)*, pages 872–883, 1999.
Referenced in text: page(s) 45, 161

[82] David Mitchell, Bart Selman, and Hector Levesque. Hard and easy distributions of SAT problems. In *Proceeding of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.
Referenced in text: page(s) 6

[83] Paul Morris. The breakout method for escaping from local minima. In *Proceedings of the Eleventh National Conference in Artificial Intelligence (AAAI-93)*, pages 40–45, 1993.
Referenced in text: page(s) 20, 44, 158

[84] Eugene Nudelman, Kevin Leyton-Brown, Holger H. Hoos, Alex Devkar1, and Yoav Shoham1. Understanding random SAT: Beyond the clauses-to-variables ratio. In *Proceedings of the Tenth International Conference on Principles and Practice of Constraint Programming (CP-04)*, volume 3258 of *Lecture Notes in Computer Science*, pages 438–452, 2004.
Referenced in text: page(s) 139, 167

[85] Christos H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS-91)*, pages 163–169, 1991.
Referenced in text: page(s) 17, 34, 105, 106, 159

[86] Andrew J. Parkes and Joachim P. Walser. Tuning local search for satisfiability testing. In *Proceedings of the Thirteenth National Conference on Artificial Intelligence (AAAI-96)*, pages 356–362, 1996.
Referenced in text: page(s) 106

[87] Duc Nghia Pham, John Thornton, Charles Gretton, and Abdul Sattar. Advances in local search for satisfiability. In *Proceedings of the Twentieth Australian Joint Conference on*

Artificial Intelligence (AI-07), volume 4830 of *Lecture Notes in Computer Science*, pages 213–222, 2007.
Referenced in text: page(s) 62, 120, 135, 161

[88] Steven Prestwich. Random walk with continuously smoothed variable weights. In *Proceedings of the Eighth International Conference on Theory and Applications of Satisfiability Testing (SAT-05)*, volume 3569 of *Lecture Notes in Computer Science*, pages 203–215, 2005.
Referenced in text: page(s) 34, 62, 121, 127, 168, 169

[89] Steven Prestwich. CNF encodings. In Biere et al. [13], chapter 2, pages 75–97.
Referenced in text: page(s) 5, 6

[90] Steven Prestwich and Andrea Roli. Symmetry breaking and local search spaces. In *Proceedings of the Second International Conference on Integration of Artificial Intelligence (AI) and Operations Research (OR) techniques in Constraint Programming (CPAIOR-05)*, volume 3524 of *Lecture Notes in Computer Science*, pages 273–287, 2005.
Referenced in text: page(s) 99

[91] Wayne Pullan and Liang Zhao. Resolvent clause weighting local search. In *Proceedings of the Seventeenth Conference of the Canadian Society for Computational Studies of Intelligence (AI-2004)*, volume 3060 of *Lecture Notes in Artificial Intelligence*, pages 233–247, 2004.
Referenced in text: page(s) 62, 142, 166

[92] Celso C. Ribeiro, Reinaldo C. Souza, and Carlos Eduardo C. Vieira. A comparative computational study of random number generators. *Pacific Journal of Optimization*, 1(3):565–578, 2005.
Referenced in text: page(s) 106

[93] Andrew Rukhin, Juan Soto, James Nechvatal, Miles Smid, Elaine Barker, Stefan Leigh, Mark Levenson, Mark Vangel, David Banks, Alan Heckert, James Dray, and San Vo. A statistical test suite for random and pseudorandom number generators for cryptographic applications. Technical Report 800-22, National Institute of Standards and Technology, 2000.
Referenced in text: page(s) 107, 108

[94] Uwe Schöning. A probabilistic algorithm for *k*-SAT and constraint satisfaction problems. In *Proceedings of the Fourtieth Annual Symposium on Foundations of Computer Science (FOCS-99)*, page 410, 1999.
Referenced in text: page(s) 17, 99, 105, 168

[95] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. In *Proceedings of the Seventeenth National Conference in Artificial Intelligence (AAAI-00)*, pages 297–302, 2000.
Referenced in text: page(s) 46, 58, 104, 112, 168

[96] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132(2):121–150, 2001.
Referenced in text: page(s) 46

[97] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic Boolean programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, pages 334–341, 2001.
Referenced in text: page(s) 2, 47, 49, 53, 137, 161

[98] Bart Selman and Henry Kautz. Domain-independant extensions to GSAT : Solving large structured variables. In *Proceedings of the Thirteenth International Joint Conference on Artificial Intelligence (IJCAI-93)*, pages 290–295, 1993.
Referenced in text: page(s) 16, 33, 44, 72, 162, 163

[99] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (AAAI-94)*, pages 337–343, 1994.
Referenced in text: page(s) 17, 34, 107, 127, 129, 169

[100] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465, 1992.
Referenced in text: page(s) 16, 33, 105, 119, 127, 162

[101] Bart Selman, David Mitchell, and Hector Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81:17–29, 1996.
Referenced in text: page(s) 174

[102] Yi Shang and Benjamin W. Wah. A discrete Lagrangian-based global-search method for solving satisfiability problems. *Journal of Global Optimization*, 12(1):61–100, 1998.
Referenced in text: page(s) 45, 46, 160

[103] Andrew Slater. Modelling more realistic SAT problems. In *Proceedings of the Fifteenth Australian Joint Conference on Artificial Intelligence (AI-02)*, volume 2557 of *Lecture Notes in Artificial Intelligence*, pages 591–602, 2002.
Referenced in text: page(s) 172

[104] Peter J. Slater and William Yslas Velez. Permutations of the positive integers with restrictions on the sequence of differences, ii. *Pacific Journal of Mathematics*, 82(2):527–531, 1979.
Referenced in text: page(s) 6

[105] Kevin Smyth, Holger H. Hoos, and Thomas Stützle. Iterated robust tabu search for MAX-SAT. In *Proceedings of the Sixteenth Conference of the Canadian Society for Computational Studies of Intelligence (AI-03)*, volume 2671 of *Lecture Notes in Artificial Intelligence*, pages 129–144, 2003.
Referenced in text: page(s) 33, 40, 164

[106] Peter F. Stadler. Towards a theory of landscapes. In *Complex Systems and Binary Networks*, volume 461, pages 77–163, 1995.
Referenced in text: page(s) 99

[107] Éric D. Taillard. Robust taboo search for the quadratic assignment problem. *Parallel Computing*, 17(4-5):443–455, 1991.
Referenced in text: page(s) 34, 105, 166

[108] Éric D. Taillard, Luca M. Gambardella, Michel Gendreau, and Jean-Yves Potvin. Adaptive memory programming: A unified view of metaheuristics. *European Journal of Operational Research*, 135(1):1–16, 2001.
Referenced in text: page(s) 21

[109] John Thornton. Clause weighting local search for SAT. *Journal of Automated Reasoning*, 35(1-3):97–142, 2005.
Referenced in text: page(s) 61, 165

[110] John Thornton and Duc Nghia Pham. Using cost distributions to guide weight decay in local search for SAT. In *Proceedings of the Ninteenth National Conference on Artificial Intelligence (AAAI-04)*, pages 191–196, 2004.
Referenced in text: page(s) 62, 163

[111] John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira Jr. Additive versus multiplicative clause weighting for SAT. In *Proceedings of the Tenth Pacific Rim International Conference on Artificial Intelligence (PRICAI-08)*, pages 405–416, 2008.
Referenced in text: page(s) 34, 60, 61, 64, 135, 165

[112] Dave A. D. Tompkins and Holger H. Hoos. Warped landscapes and random acts of SAT solving. In *Proceedings of the Eighth International Symposium on Artificial Intelligence and Mathematics (SAIM-04)*, 2004.
Referenced in text: page(s) xix, xx, 106

[113] Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In *Revised Selected Papers of the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT-04)*, volume 3542 of *Lecture Notes in Computer Science*, pages 306–320, 2005.
Referenced in text: page(s) xx, 128, 138, 168

[114] Dave A. D. Tompkins and Holger H. Hoos. On the quality and quantity of random decisions in stochastic local search for SAT. In *Proceedings of the Nineteenth Conference of the Canadian Society for Computational Studies of Intelligence (AI-06)*, volume 4013 of *Lecture Notes in Artificial Intelligence*, pages 146–158, 2006.
Referenced in text: page(s) xx, 33, 34, 112, 139, 159, 160, 165, 167, 171

[115] Dave A. D. Tompkins and Holger H. Hoos. Dynamic scoring functions with variable expressions: New SLS methods for solving SAT. In *Proceedings of the Thirteenth*

International Conference on Theory and Applications of Satisfiability Testing (SAT-10), volume 6175 of *Lecture Notes in Computer Science*, pages 278–292, 2010.
Referenced in text: page(s) xx, 159, 168, 169, 170

[116] Pascal Van Hentenryck and Laurent Michel. Control abstractions for local search. In *Proceedings of the Ninth International Conference on Principles and Practice of Constraint Programming (CP-03)*, volume 2833 of *Lecture Notes in Computer Science*, pages 65–80, 2003.
Referenced in text: page(s) 41, 159

[117] Wanxia Wei, Chu Min Li, and Harry Zhang. A switching criterion for intensification and diversification in local search for SAT. *Journal on Satisfiability, Boolean Modeling and Computation*, 4:219–237, 2008.
Referenced in text: page(s) 121, 135, 163

[118] Zhe Wu and Benjamin W. Wah. Trap escaping strategies in discrete Lagrangian methods for solving hard satisfiability and maximum satisfiability problems. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, pages 673–678, 1999.
Referenced in text: page(s) 45, 46, 160

[119] Zhe Wu and Benjamin W. Wah. An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In *Proceedings of the Seventeenth National Conference in Artificial Intelligence (AAAI-00)*, pages 310–315, 2000.
Referenced in text: page(s) 45, 46, 161

[120] Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. Hydra: Automatically configuring algorithms for portfolio-based selection. In *Proceedings of the Twenty-Fourth National Conference on Artificial Intelligence (AAAI-10)*, pages 210–216, 2010.
Referenced in text: page(s) 141, 163

[121] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla-07: The design and analysis of an algorithm portfolio for SAT. In *Proceedings of the Thirteenth International Conference on Principles and Practice of Constraint Programming (CP-07)*, volume 4741 of *Lecture Notes in Computer Science*, pages 712–727, 2007.
Referenced in text: page(s) 167

[122] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. SATzilla: Portfolio-based algorithm selection for SAT. *Journal of Artificial Intelligence Research*, 32:565–606, 2008.
Referenced in text: page(s) 139, 167

[123] Hantao Zhang and Mark Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1-2):277–296, 2000.
Referenced in text: page(s) 175

Online Bibliography

[124] http://www.google.com/analytics.

[125] http://www.eclipse.org.

[126] http://www.fourmilab.ch/random.

[127] http://www.is.titech.ac.jp/~watanabe/gensat/a2/index.html.

[128] http://webdocs.cs.ualberta.ca/~joe/Coloring/Generators/flat.html.

[129] http://www.first.fraunhofer.de/owx_download/keno-engl.pdf.

[130] http://www-cs-faculty.stanford.edu/~knuth/programs/rng.c.

[131] http://csrc.nist.gov/rng.

[132] ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/crawford/README.

[133] http://fmv.jku.at/precosat/.

[134] http://www.satcompetition.org/2003/TOOLBOX/.

[135] http://www.random.org.

[136] http://www.sat4j.org.

[137] http://www.satcompetition.org.

[138] http://www.satlib.org.

[139] http://scholar.google.com.

[140] ftp://dimacs.rutgers.edu/pub/challenge/satisfiability/contributed/UCSC/instances/cnf-ssa/README.

[141] http://www.satlib.org/ubcsat.

Appendix A

Algorithm Index

A.1 ADAPTIVE G^2WSAT

Original citation: [77]

Referenced in text: page(s) 34, 120, 135

A.2 ADAPTIVE G^2WSAT+P

Original citation: [77]

Referenced in text: page(s) 34, 135

A.3 ADAPTIVE NOVELTY$^+$

ADAPTIVE NOVELTY$^+$ is described in Section 2.5

Original citation: [51]

Default parameter(s): $(wp) = (0.01)$

Referenced in text: page(s) 19, 33, 51, 62, 64, 65, 68, 72, 106, 109, 110, 114–117, 140, 171

A.4 BREAKOUT

BREAKOUT is described in Section 2.6

Original citation: [83]

Referenced in text: page(s) 20, 21, 44–47, 65, 141

A.5 CALYSTO

Original citation: [9]

Referenced in text: page(s) 120, 176

A.6 CLASS

CLASS is described in Section 7.5

Original citation: [35]

Additional citation(s): [32, 34]

Alternate name(s): Composite heuristic Learning Algorithm for SAT Search

Referenced in text: page(s) 136, 143

A.7 COMET

COMET is described in Section 3.7

Original citation: [116]

Referenced in text: page(s) 41

A.8 CRWALK: Conflict-Directed Random Walk

CRWALK is described in Section 2.5

Original citation: [85]

Alternate name(s): Papadimitriou's Algorithm

Referenced in text: page(s) 17, 19, 25, 34, 105, 106, 109, 110, 114–117, 140

A.9 DAVE: Design Architecture for Variable Expressions

DAVE is described in Section 7.3

Original citation: [115]

Referenced in text: page(s) 2, 3, 118, 119, 128, 129, 131, 133, 135, 136, 138, 139, 141–143, 179, 180

A.10 DCRWALK: Deterministic CRWALK

DCRWALK is described in Section 6.3

Original citation: [114]

Referenced in text: page(s) 34, 114, 115, 117

A.11 DDFW: Divide and Distribute Fixed Weights

DDFW is described in Section 4.4

Original citation: [63]

Default parameter(s): $(W_{init}, p_{flat}, TL) = (8, 0.15, 0.99)$

Referenced in text: page(s) 34, 60, 61, 64–70, 72–78, 80–86, 88–90, 92, 93, 95–101

A.12 DDFW$^+$: Adaptive DDFW

DDFW$^+$ is described in Section 4.4

Original citation: [62]

Referenced in text: page(s) 61

A.13 Deterministic Adaptive Novelty$^+$

Deterministic Adaptive Novelty$^+$ is described in Section 6.3

Original citation: [114]

Default parameter(s): $(wp) = (0.01)$

Referenced in text: page(s) 34, 115, 117

A.14 DLM: Discrete Lagrangian Method

DLM is described in Section 4.1

Original citation: [102]

Referenced in text: page(s) 44–46, 51, 53, 60

A.15 DLM-98-BASIC-SAT

DLM-98-BASIC-SAT is described in Section 4.1

Original citation: [102]

Alternate name(s): DLM $\mathcal{A}_3$

Referenced in text: page(s) 46

A.16 DLM-99-SAT

DLM-99-SAT is described in Section 4.1

Original citation: [118]

Referenced in text: page(s) 46

A.17 DLM-2000-SAT

DLM-2000-SAT is described in Section 4.1

Original citation: [119]

Referenced in text: page(s) 46

A.18 ESG: Exponentiated Subgradient

ESG is described in Section 4.1

Original citation: [97]

Referenced in text: page(s) 2, 43, 46–51, 53–56, 58, 60, 62, 137

A.19 G^{2}WSAT: Gradient-based Greedy WalkSAT

G^{2}WSAT is described in Section 2.5

Original citation: [76]

Default parameter(s): $(noveltyNoise, dp) = (0.5, 0.05)$

Referenced in text: page(s) 19, 34, 62, 64, 65, 68, 72, 120, 127, 128

A.20 GGA: Gender-based Genetic Algorithm

Original citation: [4]

Referenced in text: page(s) 139

A.21 GLSSAT: Guided Local Search for SAT

GLSSAT is described in Section 4.1

Original citation: [81]

Referenced in text: page(s) 44, 45

A.22 Glucose

Original citation: [8]

Referenced in text: page(s) 8

A.23 gNovelty$^+$

gNovelty$^+$ is described in Section 4.4

Original citation: [87]

Referenced in text: page(s) 62, 120, 135

A.24 GSAT: Greedy Search for SAT

GSAT is described in Section 2.5

Original citation: [100]

Referenced in text: page(s) 16, 17, 19–21, 33–36, 40, 44, 45, 65, 105, 115, 119, 127, 128

A.25 GSAT+CW: GSAT with Clause Weights

GSAT+CW is described in Section 4.1

Original citation: [98]

Alternate name(s): GSAT Strategy I: Clause Weights, WEIGHT

Referenced in text: page(s) 44, 45

A.26 GSAT+LR: GSAT with Learning Rates

GSAT+LR is described in Section 4.1

Original citation: [30]

Additional citation(s): [29]

Alternate name(s): WGSAT

Referenced in text: page(s) 44, 45

A.27 GSAT+LR+D: GSAT+LR with Decay

GSAT+LR+D is described in Section 4.1

Original citation: [30]

Alternate name(s): WGSAT with decay

Referenced in text: page(s) 44, 45

A.28 GSAT/NW: GSAT with No Worsening Steps

GSAT/NW is described in Section 5.1

Referenced in text: page(s) 65, 94

A.29 GSAT/Tabu: GSAT with TABU

Original citation: [79]

Default parameter(s): (*tabuTenure*) = (10)

Referenced in text: page(s) 33

A.30 GWSAT: GSAT WITH RANDOM WALK

GWSAT is described in Section 2.5

Original citation: [98]

Alternate name(s): GSAT Strategy III: Random Walk, GRSAT

Default parameter(s): $(wp) = (0.5)$

Referenced in text: page(s) 16, 17, 33–36, 72, 128

A.31 HSAT: GSAT WITH HISTORY

HSAT is described in Section 2.5

Original citation: [39]

Referenced in text: page(s) 19, 33, 120, 121

A.32 HWSAT: HSAT WITH RANDOM WALK

HWSAT is described in Section 2.5

Original citation: [40]

Alternate name(s): HRSAT

Default parameter(s): $(wp) = (0.1)$

Referenced in text: page(s) 19, 33, 72

A.33 HYBRID

HYBRID is described in Section 7.5

Original citation: [117]

Referenced in text: page(s) 121, 135

A.34 HYDRA

Original citation: [120]

Referenced in text: page(s) 141

A.35 IPAWS: SELF-TUNING PAWS

IPAWS is described in Section 4.4

Original citation: [110]

Referenced in text: page(s) 62

A.36 IRoTS: Iterated RoTS

Original citation: [105]

Default parameter(s): (*lTabu*,*tabuInterval*,*eSteps*,*pSteps*,*pTabu*,*pNoise*) = $(0.1 \cdot |V| + 4, 25, |V|^2/4, 0.9 \cdot |V|, pSteps/2, 0.1)$

Referenced in text: page(s) 33, 40

A.37 MiniSAT

Original citation: [23]

Referenced in text: page(s) 42

A.38 Novelty

Novelty is described in Section 2.5

Original citation: [80]

Default parameter(s): (*noveltyNoise*) = (0.5)

Referenced in text: page(s) 17, 19, 21, 27, 33, 36, 105, 119, 128

A.39 Novelty$^+$

Novelty$^+$ is described in Section 2.5

Original citation: [49]

Default parameter(s): (*noveltyNoise*,*wp*) = $(0.5, 0.01)$

Referenced in text: page(s) 17, 19, 33, 48, 49, 51, 53–56, 72

A.40 Novelty^{++}

Original citation: [76]

Default parameter(s): (*noveltyNoise*,*dp*) = $(0.5, 0.05)$

Referenced in text: page(s) 19, 34, 72

A.41 Novelty$^+$p

Original citation: [77]

Default parameter(s): (*noveltyNoise*,*wp*) = $(0.5, 0.01)$

Referenced in text: page(s) 34

A.42 OPENSAT

OPENSAT is described in Section 3.7

Original citation: [7]

Referenced in text: page(s) 41, 42

A.43 PARAMILS

PARAMILS is described in Section 7.1

Original citation: [59]

Additional citation(s): [57, 60]

Referenced in text: page(s) xxi, xxiv, 118, 120, 122, 123, 125–127, 130–135, 139, 141, 179–183

A.44 PAWS: PURE ADDITIVE WEIGHTING SCHEME

PAWS is described in Section 4.4

Original citation: [111]

Additional citation(s): [109]

Default parameter(s): $(maxInc, p_{flat}) = (10, 0.15)$

Referenced in text: page(s) 34, 53, 60, 61, 64–70, 72, 74–78, 80–86, 88–90, 92, 93, 95–101, 135

A.45 PAWS+US: PAWS WITH USUAL SUSPECTS

PAWS+US is described in Section 5.6

Original citation: [25]

Referenced in text: page(s) 61, 102

A.46 PICOSAT

Original citation: [12]

Referenced in text: page(s) 120, 131

A.47 PRECOSAT

Original Citation: [133]

Referenced in text: page(s) 8

A.48 RGSAT: RESTARTING GSAT

Original citation: [114]

Referenced in text: page(s) 34

A.49 RLS: Resolvent clause weighting Local Search

RLS is described in Section 4.4

Original citation: [91]

Referenced in text: page(s) 62

A.50 R-Novelty

Original citation: [80]

Default parameter(s): (*noveltyNoise*) = (0.5)

Referenced in text: page(s) 33

A.51 R-Novelty$^+$

Original citation: [49]

Default parameter(s): (*noveltyNoise*,*wp*) = $(0.5, 0.01)$

Referenced in text: page(s) 33, 72

A.52 RoTS: Robust TABU Search

Original citation: [107]

Alternate name(s): Robust Taboo

Default parameter(s): (tabu,tabuInterval) = $(10, 25)$

Referenced in text: page(s) 34, 72, 105

A.53 RSAPS: Reactive SAPS

RSAPS is described in Section 4.2

Original citation: [61]

Default parameter(s): (α,ρ,*ps*,*wp*,*sapsThresh*) = $(1.3, 0.8, 0.05, 0.01, -0.1)$

Referenced in text: page(s) xxiv, 34, 51, 53, 72

A.54 SAMD: Steepest Ascent Mildest Descent

Original citation: [44]

Default parameter(s): (tabuTenure) = (10)

Referenced in text: page(s) 34, 40, 72

A.55 SAPS: Scaling and Probabilistic Smoothing

SAPS is described in Section 4.2

Original citation: [61]

Default parameter(s): $(\alpha,\rho,ps,wp,sapsThresh) = (1.3, 0.8, 0.05, 0.01, -0.1)$

Referenced in text: page(s) ii, xxi, xxiv, 2, 3, 21, 34, 42, 43, 47, 50–70, 72–86, 88–93, 95–101, 104–106, 109, 111–114, 116, 117, 120, 127, 135, 137, 139, 140, 142

A.56 SAPS/NR: Derandomized SAPS

SAPS/NR is described in Section 6.3

Original citation: [114]

Default parameter(s): $(\alpha,\rho,ps,wp,sapsThresh) = (1.3, 0.8, 0.05, 0.01, -0.1)$

Referenced in text: page(s) 33, 60, 72, 112, 113, 115, 117, 142

A.57 SAT4J

SAT4J is described in Section 3.7

Referenced in text: page(s) 41, 42

A.58 SatELite

Original citation: [22]

Referenced in text: page(s) 172

A.59 SATenstein

SATenstein is described in Section 7.5

Original citation: [70]

Alternate name(s): SATenstein-LS

Additional citation(s): [69]

Referenced in text: page(s) 53, 54, 57, 121, 122, 131–136, 139, 180, 183

A.60 SATzilla

Original citation: [122]

Additional citation(s): [84, 121]

Referenced in text: page(s) 139

A.61 Schöning's Algorithm

Schöning's Algorithm is described in Section 2.5

Original citation: [94]

Referenced in text: page(s) 17, 99, 105, 106, 114

A.62 SDF: Smoothed Descent and Flood

SDF is described in Section 4.1

Original citation: [95]

Referenced in text: page(s) 46, 47, 49, 50, 56, 58, 60, 61, 104, 112, 142

A.63 UBCSAT

UBCSAT is described in Section 3.2

Original citation: [113]

Referenced in text: page(s) ii, xxi, 2, 3, 23, 24, 28–30, 32–42, 51, 52, 61, 62, 71, 72, 106, 108, 118, 128, 129, 135, 137–142

A.64 URWALK: Uniform Random Walk

URWALK is described in Section 2.4

Referenced in text: page(s) 12–16, 24–27, 34, 35, 91, 105

A.65 VE-Sampler

VE-Sampler is described in Section 7.4

Original citation: [115]

Referenced in text: page(s) 119, 129–135, 180–183

A.66 VW1: Variable Weighting Scheme I

Original citation: [88]

Default parameter(s): $(wp) = (0.5)$

Referenced in text: page(s) 34, 62

A.67 VW2: Variable Weighting Scheme II

VW2 is described in Section 7.2.1

Original citation: [88]

Default parameter(s): (c,s,wp) = $(0.01, 0.01, 0.5)$

Referenced in text: page(s) 34, 62, 121–124, 126, 127, 131–134, 180

A.68 VW2-Sat05: VW2 - 2005 Competition Variant

VW2-Sat05 is described in Section 7.2.1

Original citation: [88]

Referenced in text: page(s) 121, 122, 132, 134, 135

A.69 VW2+VE

VW2+VE is described in Section 7.2.2

Original citation: [115]

Referenced in text: page(s) 123–125, 130, 180, 181

A.70 WalkSAT: (Family of Algorithms)

WalkSAT is described in Section 2.5

Original citation: [99]

Referenced in text: page(s) 17, 19, 25, 27, 30, 32, 34–36, 40, 49, 105–107, 121, 128

A.71 WalkSAT/SKC

WalkSAT/SKC is described in Section 2.5

Original citation: [99]

Alternate name(s): WSAT, WalkSAT

Default parameter(s): (wp) = (0.5)

Referenced in text: page(s) 17, 21, 34–36, 42, 72, 91, 120, 125–127, 129

A.72 WalkSAT/Tabu

WalkSAT/Tabu is described in Section 3.2

Original citation: [80]

Alternate name(s): TABU, WTABU, Taboo

Default parameter(s): ($tabuTenure$) = (10)

Referenced in text: page(s) 19, 30, 32, 34, 36–38, 119

A.73 WALKSAT+VE

WALKSAT+VE is described in Section 7.2.4

Original citation: [115]

Referenced in text: page(s) 125–127, 130, 181

A.74 WEIGHTED ADAPTIVE NOVELTY$^+$

WEIGHTED ADAPTIVE NOVELTY$^+$ is described in Section 5.1

Original citation: [51]

Default parameter(s): $(wp) = (0.01)$

Referenced in text: page(s) 64, 71, 88, 90, 91, 94, 95, 97

A.75 WEIGHTED G^{2}WSAT

WEIGHTED G^{2}WSAT is described in Section 5.1

Original citation: [76]

Default parameter(s): $(noveltyNoise, dp) = (0.5, 0.05)$

Referenced in text: page(s) 64, 71, 89–91, 94, 96, 97

A.76 WEIGHTED GSAT/NW

WEIGHTED GSAT/NW is described in Section 5.1

Referenced in text: page(s) 65, 92

Appendix B

Instance Set Index

We follow the common practice in the literature of identifying instances by their abbreviated file name, often in the form prefix-suffix where the prefix is the instance set and the suffix is an identifier.

B.1 anp10m: SATLIB: Adaptive Novelty$^+$ 1k-10M

Description: This is a large set we created with 1931 structured instances from SATLIB where ADAPTIVE NOVELTY$^+$ has a median run-length between 1 000 and 10 000 000 steps. The anp10m set is described in Section 5.1.
Original citation: [114]
Set categories: Structured, Crafted
Referenced in text: page(s) 65–67, 69, 70, 74, 76, 80, 81, 84, 86–88, 91, 95, 97, 98, 100, 101, 113, 115, 116

B.2 ais: All Interval Series

Description: Encoding of a numerical sequence problem inspired by music theory. ais-N instances are sequences of length N. The ais instances are described in Section 2.2.
Original citation: [48]
Set categories: Structured, Crafted
Referenced in text: page(s) 6, 7, 48, 49, 54, 58, 65, 68, 71–73, 77, 79, 90, 99, 102

B.3 bw-large: Blocks World

Description: Encoding of a temporal planning problem to stack blocks. The letter (*e.g.*, bw-large-a) indicates the size of the instance.
Original citation: [67]
Set categories: Structured, Application
Referenced in text: page(s) 35, 49, 53, 54, 65, 68, 90, 97, 102, 116

B.4 bitadd: Bit Adders

Description: Encoding of VLSI Boolean circuit synthesis problem.
Original citation: [66]
Source: 1996 International Competition on SAT Testing in Beijing
Set categories: Structured, Application
Referenced in text: page(s) 65, 97, 99

B.5 cbmc: 'C' Bounded Model Checking

Description: Encoding of a software verification problem with bounded model checking. The instances are 'C' code of a binary search algorithm with different array sizes and loop-unwinding values.
Original citation: [17]
Pre-processing: SATELITE tool with full processing settings (+ve)
Set categories: Structured, Application
Additional citation(s): [70]
Referenced in text: page(s) 7, 119–127, 130–133, 180–182

B.6 clus: Clustered 3-SAT

Description: Randomly generated structured 3-SAT instances that exhibit real-world clustering behaviour.
Original citation: [103]
Source: 2003 SAT Competition
Set categories: Structured, Crafted
Referenced in text: page(s) 68, 99

B.7 fac: Factorial

Description: Encoding of a factorization problem.
Original Citation: [127]
Additional citation(s): [70]
Set categories: Structured, Crafted
Referenced in text: page(s) 53, 57, 137

B.8 ferry: Ferry Trafficking

Description: Encoding of a temporal planning problem to ferry cars from a source to a destination, similar to the bw-large instances.
Original citation: [75]
Set categories: Structured, Application
Referenced in text: page(s) 68, 111–113, 142

B.9 flat: Flat Graph Colouring

Description: Encoding of a randomly generated flat graph 3-colouring problem. flatN instances have N vertices. The hardest, median and easiest instances from a set are referred to as flatN-hard, flatN-med and flatN-easy.
Original Citation: [128]
Set categories: Structured, Crafted
Referenced in text: page(s) 7, 48, 49, 54, 65, 73–76, 78, 80–86, 90, 97, 110, 115

B.10 gcp: Graph Colouring Problem

Description: Encoding of a randomly generated graph 3-colouring problem. The instance named gN.C is a graph with N nodes where nodes are connected with a fixed probability p, and there are C colours.
Original citation: [65]
Set categories: Structured, Crafted
Referenced in text: page(s) 53, 97

B.11 ii: Inductive Inference

Description: Encoding of a Boolean Function Synthesis Problem.
Original citation: [66]
Set categories: Structured, Crafted
Referenced in text: page(s) 65, 68, 90, 97, 99, 110

B.12 jnh: Random P-SAT

Description: Random generated constant-density model instances, known as Random P-SAT. In Random P-SAT, clauses are generated by including a variable in a clause with some probability p and then randomly negated, with empty and unit clauses removed.
Original citation: [28]
Additional citation(s): [101]
Set categories: Random
Referenced in text: page(s) 36

B.13 logistics: Logistics Planning

Description: Encoding of a temporal planning problem to move packages between different locations in different cities.
Original citation: [67]
Set categories: Structured, Application
Referenced in text: page(s) 48, 49, 53, 54, 65, 99

B.14 parity: Parity Function

Description: Encoding of a parity function learning problem.
Original Citation: [132]
Set categories: Structured, Crafted
Referenced in text: page(s) 65, 77, 79, 90, 93, 99

B.15 qcp: Quasi-Group Completion

Description: Encoding of a quasi-group (Latin square) completion problem, similar to qg, generated at the phase transition [43]
Original citation: [43]
Additional citation(s): [70]
Set categories: Structured, Crafted
Referenced in text: page(s) 133, 134, 180, 182, 183

B.16 qg: Quasi-Group

Description: Encoding of a quasi-group (Latin square) problem. The instance named qgT-S is of type T and size S.
Original citation: [123]
Set categories: Structured, Crafted
Referenced in text: page(s) 7, 68, 77, 79, 90, 99

B.17 r3sat: Random 3-SAT

Description: Random 3-SAT instances, similar to the uf instances, generated by the 2002 SAT Competition generator.
Original Citation: [134]
Additional citation(s): [70]
Set categories: Random
Referenced in text: page(s) 133–135, 180, 183

B.18 rg: Random Graph

Description: Unsatisfiable Random Graph.
Set categories: Random, Unsatisfiable
Referenced in text: page(s) 36

B.19 SATLIB

Description: SATLIB is an online collection of benchmark problems [138].
Original citation: [54]
Referenced in text: page(s) 6, 65, 106, 141

B.20 sgi: Subgraph Isomorphism

Description: Encoding of the subgraph isomorphism problem to determine if a graph is isomorphic to a subgraph of another graph.
Original citation: [5]
Set categories: Structured, Crafted
Referenced in text: page(s) 65

B.21 ssa: Single-Stuck-At fault circuit analysis

Description: Encoding of a single-stuck-at fault circuit analysis problem.
Original citation: [74]
Additional citation(s): [140]
Set categories: Structured, Application
Referenced in text: page(s) 110

B.22 swv: Software Verification

Description: Encoding of a software verification problem generated by the CALYSTO static checker.
Original citation: [9]
Additional citation(s): [58]
Set categories: Structured, Application
Referenced in text: page(s) 7, 120, 130–133, 180, 182, 183

B.23 swgcp: Morphed Graph Colouring Problems

Description: Encoding of a graph colouring problem where the graph is generated by morphing randomly generated graph and a structured ring lattice graph.
Original citation: [37]
Set categories: Structured, Crafted
Referenced in text: page(s) 65, 68, 90

B.24 uf: Unforced Uniform Random 3-SAT

Description: Random instances with 3 unique variables per clause, each selected at random and negated at random, with no forced solution. ufN instances have *N* variables and a number of clauses corresponding to the phase-transition. The hardest, median and easiest instances from a set are referred to as ufN-hard, ufN-med and ufN-easy. The uf instances are described in Section 2.2.
Original citation: [16]
Additional citation(s): [54]
Set categories: Random
Referenced in text: page(s) 6, 35, 49, 53–56, 59, 60, 110, 111, 113, 116

B.25 uuf: Unsatisfied Unforced Uniform Random 3-SAT

Description: These are the same as uf instances, except that they are unsatisfiable.

Original citation: [16]

Additional citation(s): [54]

Set categories: Random, Unsatisfiable

Referenced in text: page(s) 36

Appendix C

Experimental Details

C.1 UBC arrow cluster

The UBC arrow cluster at UBC is composed of 55 dual 3.2GHz Intel Xeon PCs with 2GB RAM, 2MB cache, running SuSE Linux.
Referenced in text: page(s) xxi, 57, 58, 67, 122, 123, 126, 127, 131, 132, 134, 135, 180

C.2 UBC BETA cluster

The *original* UBC BETA cluster at UBC is composed of Pentium III (Coppermine) PCs with 256KB cache and 1GB RAM, running either Red Hat Linux or SuSE Linux.
Referenced in text: page(s) 36, 49, 54–56

C.3 WestGrid glacier cluster

The WestGrid glacier cluster is composed of 840 computational nodes, each with two 3.06 GHz Intel Xeon 32-bit processors with at least 2GB of RAM, running Red Hat Linux.

C.4 WestGrid orcinus cluster

The WestGrid orcinus cluster is composed of 12 chassis, each containing 16 blades with two compute servers on each blade and each server has two 3.0 GHz Intel Xeon E5450 quad-core processors, with each server sharing 16 GB of RAM, running Red Hat Enterprise Linux Server.

C.5 Correlation plots

Throughout this dissertation, we provided correlation plots to compare the performance of two different algorithms on an instance set. We also use the same methodology to compare two variants of the same algorithm or the same algorithm on two different weightings of the same instance set. Each point in the plot corresponds to a single instance, and the location of the point is the resulting median run-length from multiple runs of the two algorithms on the instance. In some cases, we indicate the mean value of all of the medians with dashed lines, and provide the speedup factor (*s.f.*) which is the ratio of the two means. The line of equivalent performance is also shown, and the relative difference between the performance of the algorithms is represented by the perpendicular distance from this line. In some cases, we also provide significance bands showing the distance from the line of equivalent performance that is considered statistically significant. To determine the location of these significance bands we use the Mann-Whitney U-test [55: p. 179] for significance level 0.01 and a power of 0.99. In Chapter 5, we do not include instances in the plots where one or both of the algorithms did not have at least half of the runs be successful at the cutoff value (10^8, or the largest axis value) and instead identify the instances as significant outliers in the text.

C.6 Training sets

For our experiments in Chapter 7, we split the instance sets into halves: a *test* set, and a *training* set. To split the instances we used a stratification strategy to ensure that the test and training set were of approximately equal hardness. As an approximate measure of hardness, we measured the file size of the instance. We first sorted the instances by their file size, and then segmented the instances into pairs of consecutive instances. For each consecutive pair, we randomly placed one instance in the test set, and the other in the training set. Only the instances in the training set were used to determine good parameter settings, and only instances in the test set were used to report experimental results.

C.7 PARAMILS experimental information

All PARAMILS experiments were conducted with the UNIX binary of PARAMILS version 2.3.2. We used the default *FocusedILS* configuration of PARAMILS with settings of (deterministic, overall-obj) = (0, mean10). Because PARAMILS can be very sensitive to the ordering of the instance list, we performed several runs of PARAMILS, each with a randomized instance list, and selected the configuration with the best performance on the training set. To measure this performance we ran each configuration five times on each instance in the training set and measured the median run-length from those five runs, and then measured the mean of those medians. The current PARAMILS software implementation only supports adaptive capping of algorithm runs after a given run-time, not after a given run-length. Since we were interested in optimizing our algorithms in DAVE for run-length performance, but still wanted to take advantage of PARAMILS's excellent adaptive capping feature, we reported the run-length information to PARAMILS as run-time information. Whether or not PARAMILS uses the solution quality to compare unsuccessful runs changed between different PARAMILS versions due to a bug we identified, so to err on the side of caution we included it in our run-time as follows. For a run with a run-length of rl with a solution quality of u unsatisfied clauses, we modified the run-length to be $rl + \frac{u}{10000}$. Instead of simply reporting the run-length of DAVE as a run-time, we decided to add an additional transformation to provide PARAMILS with a run-time in the same order of magnitude it typically encounters, to avoid introducing any unintended numerical precision errors. We divided the run-length by 10^6, so that one million search steps in DAVE corresponded to one second in PARAMILS. We had to use a *wrapper script* around DAVE to convert cutoff times received from PARAMILS back to search steps (*i.e.*, multiply by 10^6). For SATENSTEIN and VW2 experiments, the instance cutoff time was 60 seconds, and for DAVE the cutoff was 10 seconds, which is the equivalent of 10^7 search steps. Because PARAMILS measures its total execution time by relying on the reported run-time data, the manner in which we were reporting DAVE performance run-length data to PARAMILS as run-time data was problematic. As a result, we specified a very large amount of cutoff time to PARAMILS and controlled the total amount of CPU time used by PARAMILS through our computation environment.

In our experiments with PARAMILS and VE-SAMPLER, we encountered some difficulties that we believe were caused by the very large parameter space of VE-SAMPLER. Due to Ruby's overhead in some data structures, we observed that the binary Ruby implementation of PARAMILS could consume a large amount of RAM (over 1 GB), and as a consequence would be automatically terminated in our computation environment. In addition, we observed that some of the PARAMILS runs would stagnate, not improving over the initial (default) configuration despite parallel runs achieving great improvement. As a result of these two observations, we used an *iterative* strategy, where instead of executing PARAMILS for some amount of time t, we executed k iterations of PARAMILS, each with time t/k. For each iteration, we would use the best configuration from the previous iteration as the default configuration (see Section C.7.4).

C.7.1 VW2

The possible configurations for each parameter of VW2 were:

```
s {1, 0.33, 0.1, 0.033, 0.01, ... 0.000033, 0.00001, 0}
c {1, 0.33, 0.1, 0.033, 0.01, ... 0.00000033, 0.0000001, 0}
wp {0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5}
```

We ran PARAMILS 10 times on the UBC arrow cluster for 24 hours. The best VW2 configuration (s,c,wp) found by PARAMILS for cbmc is $(0,0.01,0.2)$, for swv is $(0,0.1,0.05)$, for r3sat is $(0.33,0.0001,0.4)$, and qcp is $(1,0.0000001,0.1)$.

C.7.2 VW2+VE

The possible configurations for each parameter of VW2+VE were:

```
w {0, 1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 20, 25, 30, 35,
   40, 45, 50, 60, 70, 80, 90, 100, 110, 120, 130, 140, 150, 175, 200}
c {0, 0.05, 0.1, 0.15, 0.2, 0.25, 0.3, 0.35, 0.4, 0.45, 0.5, 0.55, 0.6,
   0.65, 0.7, 0.75, 0.8, 0.85, 0.9, 0.95, 1.0, 1.05, 1.1, 1.15, 1.2,
   1.25, 1.3, 1.35, 1.4, 1.45, 1.5, 1.6, 1.7, 1.8, 1.9, 2.0, 2.25, 2.5,
   2.75, 3, 3.5, 4, 4.5, 5, 5.5, 6, 6.5, 7, 8, 9, 10, 15, 20, 25, 30,
   35, 40}
a {0.125, 0.25, 0.5, 1, 2, 4, 8}
```

The value of wp is calculated from w as $\frac{1}{1+w}$. We ran PARAMILS 20 times on the WestGrid glacier cluster for 24 hours. The best VW2+VE configuration found by PARAMILS for cbmc is $(c, a, wp) = (0.95, 8, 0.05)$.

C.7.3 WALKSAT+VE

For each term in WALKSAT+VE, the possible configurations for the parameters c, a and w (wp) were the same as for VW2+VE. For the normalizations, the possible configurations for make and relMake were:

$$\|\mathsf{p}\|_{\text{flat}} = \frac{\mathsf{p} - \min(\mathsf{p})}{\max(\mathsf{p}) - \min(\mathsf{p})} \tag{C.1}$$

$$\|\mathsf{p}\|_{\max} = \frac{\mathsf{p}}{\max(\mathsf{p})} \tag{C.2}$$

$$\|\mathsf{p}\|_{\text{sum}} = \frac{\mathsf{p}}{\text{sum}(\mathsf{p})} \tag{C.3}$$

and the possible configurations for break and relBreak included all of the above normalizations in the form of $(1 - \|\mathsf{p}\|)$, as well as:

$$\|x\|_{\text{-max}} = \frac{\max(x) + \min(x) - x}{\max(x)} \tag{C.4}$$

We ran PARAMILS 20 times on the WestGrid glacier cluster for 96 hours. The best WALKSAT+VE configuration found by PARAMILS for cbmc is $(wp) = (0.5)$ and the scoring function is:

$$\begin{gathered} 1.05 \cdot (\|\mathsf{make}\|_{\text{flat}})^8 + 1.35 \cdot (\|\mathsf{relMake}\|_{\text{flat}})^4 \\ +8 \cdot (1 - \|\mathsf{break}\|_{\max})^{1/2} + 2.25 \cdot (1 - \|\mathsf{relBreak}\|_{\max})^4 \end{gathered} \; . \tag{C.5}$$

C.7.4 VE-SAMPLER

For VE-SAMPLER, the possible configurations for the weight of each sub-controller (w), the exponents in the VEs (a) and the co-efficients of the clw function (c) were:

```
w {0, 1, 1.5, 2, 2.5, 3, 4, 5, 7.5, 10, 12.5, 15, 20,
   25, 30, 40, 50, 75, 100}
```

```
a {0.0625, 0.125, 0.25, 0.5, 1, 2, 4, 8, 16}
c {0, 0.1, 0.25, 0.5, 0.667, 0.8, 0.9, 0.95, 1, 1.05,
   1.1, 1.25, 1.5, 1.75, 2, 2.5, 3, 3.5, 4, 5, 7.5, 10}
```

The possible configurations for the properties (or ratio of properties) used in the VEs are listed in Section 7.4. For each property, we allowed only two possible normalizations, depending on whether or not the property is a maximal or minimal property (see Section 7.2.3). The possible normalizations were $\|\mathsf{p}\|_{\text{flat}}$, $\|\mathsf{p}\|_{\text{max}}$, $(1-\|\mathsf{p}\|_{\text{flat}})$ or $\|\mathsf{p}\|_{\text{-max}}$ (see Section C.7.3 above).

For cbmc we used the WestGrid glacier cluster, and ran PARAMILS for 4 iterations, where for each iteration we ran PARAMILS 40 times for 24 hours. The best VE-SAMPLER configuration found by PARAMILS on cbmc is:

$$\begin{aligned}
w_1 &= 3 & e_1 &= \mathsf{freebie}\\
w_2 &= 30 & e_2 &= (\|\mathsf{break}\|_{\text{-max}})^{1/4} + \mathrm{clw}(0,0,2)\cdot(1-\|\mathsf{relBreak}\|_{\text{flat}})^{16}\\
w_3 &= 50 & e_3 &= (\|\mathsf{relMake}\|_{\text{max}}) + \mathrm{clw}(3.5,0.9,5)\cdot(\|\mathsf{age}'\|_{\text{max}})^{8}\\
w_4 &= 3 & e_4 &= (\|\mathsf{make}\|_{\text{flat}}) + \mathrm{clw}(0.25,3.5,1.05)\cdot(\|\mathsf{flips}\|_{\text{-max}})^{1/8}\\
w_5 &= 30 & e_5 &= (\|\mathsf{relMake}\|_{\text{flat}})^{2} + \mathrm{clw}(0.25,0.25,3)\cdot(\|\mathsf{age}'\|_{\text{flat}})\\
w_6 &= 1 & e_6 &= (\|\mathsf{make}\|_{\text{max}})^{8} + \mathrm{clw}(0.95,3.5,0.5)\cdot(\|\mathsf{age}\|_{\text{max}})
\end{aligned} \tag{C.6}$$

For swv we used the WestGrid orcinus cluster, and ran PARAMILS for 4 iterations, where for each iteration we ran PARAMILS 40 times for 24 hours. The best VE-SAMPLER configuration found by PARAMILS on swv (partial) is:

$$\begin{aligned}
w_1 &= 3 & e_1 &= \mathsf{freebie}\\
w_2 &= 15 & e_2 &= (1-\|\mathsf{break}\|_{\text{flat}})^{1/16} + \mathrm{clw}(0,0.25,1)\cdot(\|\mathsf{make}\|_{\text{max}})^{1/2}\\
w_3 &= 50 & e_3 &= (\|\mathsf{break}\|_{\text{-max}})^{16} + \mathrm{clw}(0.1,5,0.25)\cdot(\|\mathsf{flips}\|_{\text{-max}})^{1/2}\\
w_4 &= 50 & e_4 &= (\|\mathsf{break}\|_{\text{-max}})^{1/16} + \mathrm{clw}(0.1,3.5,1.75)\cdot(\|\mathsf{flips}\|_{\text{-max}})^{1/16}\\
w_5 &= 3 & e_5 &= (\|\mathsf{relBreak}\|_{\text{-max}}) + \mathrm{clw}(1,5,7.5)\cdot(1-\|\mathsf{flips}\|_{\text{flat}})^{1/16}\\
w_6 &= 5 & e_6 &= (\|\mathsf{make}\|_{\text{flat}})^{1/2} + \mathrm{clw}(2.5,0.1,1.05)\cdot(\|\mathsf{flips}\|_{\text{-max}})
\end{aligned} \tag{C.7}$$

For qcp we used the WestGrid glacier cluster, and ran PARAMILS for 4 iterations, where for each iteration we ran PARAMILS 20 times for 24 hours. The best VE-SAMPLER configuration found by PARAMILS on qcp is:

$$\begin{aligned}
w_1 &= 3 & e_1 &= \mathsf{freebie} \\
w_2 &= 3 & e_2 &= (1-\|\mathsf{break}\|_{\text{flat}})^2 + \text{clw}(1.5,1.5,3.5)\cdot(1-\|\mathsf{break}\|_{\text{flat}})^{1/8} \\
w_3 &= 12.5 & e_3 &= (\|\mathsf{make}\|_{\text{flat}})^4 + \text{clw}(1.5,1.25,0)\cdot(1-\|\mathsf{flips}\|_{\text{flat}})^4 \\
w_4 &= 12.5 & e_4 &= (\|\mathsf{relMake}\|_{\text{flat}})^{1/16} + \text{clw}(1.25,0.95,1)\cdot(\|\mathsf{age}'\|_{\text{flat}})^{1/4} \\
w_5 &= 100 & e_5 &= (\|\mathsf{break}\|_{\text{-max}})^{16} + \text{clw}(0.9,0.9,0.667)\cdot(\|\mathsf{age}\|_{\text{max}})^{1/2} \\
w_6 &= 1.5 & e_6 &= (\|\mathsf{relMake}\|_{\text{flat}})^4 + \text{clw}(2.0,0.8,0.667)\cdot\left(\left\|\frac{\mathsf{age}'}{\mathsf{age}}\right\|_{\text{flat}}\right)^8
\end{aligned} \tag{C.8}$$

For r3sat we used the WestGrid glacier cluster, and ran PARAMILS for 4 iterations, where for each iteration we ran PARAMILS 20 times for 24 hours. The best VE-SAMPLER configuration found by PARAMILS on r3sat is:

$$\begin{aligned}
w_1 &= 3 & e_1 &= \mathsf{freebie} \\
w_2 &= 1 & e_2 &= (\|\mathsf{relMake}\|_{\text{flat}})^{1/2} + 1.05\cdot(1-\|\mathsf{relBreak}\|_{\text{flat}})^{16} \\
w_3 &= 50 & e_3 &= (\|\mathsf{break}\|_{\text{-max}}) + 1.1\cdot\left(\left\|\frac{\mathsf{age}}{\mathsf{flips}}\right\|_{\text{max}}\right)^{1/4} \\
w_4 &= 2.5 & e_4 &= (1-\|\mathsf{score}\|_{\text{flat}})^8 + 2\cdot(\|\mathsf{filtCount}\|_{\text{max}})^2 \\
w_5 &= 2.5 & e_5 &= (\|\mathsf{make}\|_{\text{flat}})^8 \\
w_6 &= 75 & e_6 &= (\|\mathsf{break}\|_{\text{-max}})^{16} + 1.1\cdot(\|\mathsf{age}'\|_{\text{max}})^{1/8}
\end{aligned} \tag{C.9}$$

C.7.5 SATENSTEIN

For SATENSTEIN, we used two PARAMILS configuration files provided by the SATENSTEIN authors. We ran PARAMILS 40 times (20 times for each configuration file) on the WestGrid orcinus cluster for 96 hours. The configuration of SATENSTEIN found by PARAMILS on swv (partial) is:

```
-adaptive 0 -adaptivenoisescheme 1 -adaptiveprom 0
-adaptpromwalkprob 0 -adaptwalkprob 0 -alpha 1.066 -c 0.00001
-clausepen 1 -decreasingvariable 3 -dp 0.05 -heuristic 2
-maxinc 20 -novnoise 0.5 -performalternatenovelty 1
-performrandomwalk 1 -pflat 0.05 -phi 5 -promdp 0.05
-promisinglist 0 -promnovnoise 0.5 -promphi 5 -promtheta 6
-promwp 0.01 -ps 0 -randomwalk 4 -rdp 0.05 -rfp 0.15 -rho 0.8
-rwp 0.1 -rwpwalk 0.05 -s 0.001 -sapsthresh -0.1
-scoringmeasure 3 -selectclause 1 -singleclause 0
-smoothingscheme 1 -tabu 5 -tabusearch 0 -theta 6
-tiebreaking 2 -updateschemepromlist 3 -varinfalse 1 -wp 0.05
-wpwalk 0.7
```