Augmenting Local Search for Satisfiability

by

Finnegan D. J. Southey

A thesis
presented to the University of Waterloo
in the fulfilment of the
thesis requirement for the degree of
Doctor of Philosophy
in
Computer Science

Waterloo, Ontario, Canada, 2004

© Finnegan D. J. Southey 2004

AUTHOR'S DECLARATION FOR ELECTRONIC SUBMISSION OF A THESIS

I hereby declare that I am the sole author of this thesis. This is a true copy of the thesis, including any required final revisions, as accepted by my examiners.

I understand that my thesis may be made electronically available to the public.

Abstract

This dissertation explores approaches to the satisfiability problem, focusing on local search methods. The research endeavours to better understand how and why some local search methods are effective. At the root of this understanding are a set of metrics that characterize the behaviour of local search methods. Based on this understanding, two new local search methods are proposed and tested, the first, SDF, demonstrating the value of the insights drawn from the metrics, and the second, ESG, achieving state-of-the-art performance and generalizing the approach to arbitrary 0-1 integer linear programming problems. This generality is demonstrated by applying ESG to combinatorial auction winner determination. Further augmentations to local search are proposed and examined, exploring hybrids that incorporate aspects of backtrack search methods.

// Acknowledgements

I owe thanks to many people for the completion of this dissertation.

- To Jim Linders and Fakhri Karray for the opportunities they afforded me.
- To my committee members, Nancy Day and Peter van Beek, for their insights.
- To Bart Selman and John Thistle, for kindly agreeing to act as examiners.
- To Rob Holte and everyone at the University of Alberta.
- To my advisor, Dale Schuurmans, with whom it has been a pleasure and privilege to work (albeit often in the wee hours).
- To my schoolmates, colleagues, and friends for their comments, criticism, and fun: Relu, Ali, Dana, Fuchun, and Paul (Remember: Aim high... fall hard).
- To all the old gang, Brian, Min, James, Cynthia, Ian, Naomi, Anne, John Paul, Nick and Rebecca, most of whom swore to mooch off me if I ever made anything of myself – you picked the wrong horse, I'm an academic now.
- To my parents for, well, everything.
- To Stan, for even more.

Contents

1 Introduction **1**
- 1.1 Motivation . . . 1
- 1.2 Contributions . . . 2

2 Background **3**
- 2.1 Terminology . . . 3
  - 2.1.1 SAT and CNF . . . 3
  - 2.1.2 Assignments . . . 4
    - 2.1.2.1 Operations on Assignments . . . 4
    - 2.1.2.2 Comparing Two Assignments . . . 5
    - 2.1.2.3 Prefixes . . . 6
  - 2.1.3 Clause Statistics . . . 6
    - 2.1.3.1 Types of Clauses . . . 6
    - 2.1.3.2 Clause Statistics Under a Single Assignment . . . 7
    - 2.1.3.3 Clause Statistics Under a Change in Assignment . . . 8
  - 2.1.4 Resolution . . . 9
  - 2.1.5 Subsumption . . . 10
- 2.2 Instance Types . . . 10
  - 2.2.1 Instance Hardness . . . 10
  - 2.2.2 Instance Variety . . . 11
  - 2.2.3 Structured Instances . . . 12
    - 2.2.3.1 Natural Instances . . . 12
    - 2.2.3.2 Transformed Instances . . . 12
  - 2.2.4 Hard Random Instances . . . 13
  - 2.2.5 Sources of Instances . . . 14
- 2.3 Solution Approaches . . . 14
  - 2.3.1 Resolution Refutation . . . 14
  - 2.3.2 Model Finding . . . 14
  - 2.3.3 Systematic Search . . . 15
- 2.4 Backtracking Search Solvers . . . 15
  - 2.4.1 Backtracking Search . . . 16
    - 2.4.1.1 Naive Backtracking . . . 16

2.4.1.2 Conflict Pruning 16
2.4.2 Unit Propagation 19
2.4.2.1 Davis-Putnam-Logemann-Loveland 19
2.4.2.2 Choice Moves and Choice Levels 20
2.4.2.3 Open and Closed Prefixes 21
2.4.2.4 Notation for UP 22
2.4.3 Pure Literals 22
2.4.4 Variable Ordering and Value Ordering 24
2.4.4.1 Search Trees and Resolution Proofs 24
2.4.4.2 Value Ordering 26
2.4.4.3 MOM Heuristic 26
2.4.4.4 Balanced Heuristics 26
2.4.4.5 C-SAT 27
2.4.4.6 Failed Literals 27
2.4.4.7 UP Heuristics 28
2.4.5 Randomization 28
2.4.6 Data Structures 28
2.4.6.1 SATO 29
2.4.7 Backjumping 31
2.4.8 Clause Learning 32
2.4.8.1 relsat 32
2.4.8.2 GRASP 32
2.4.8.3 Chaff 35
2.4.9 Summary 35
2.5 Local Search Solvers 37
2.5.0.1 Local Minima vs. Plateaus 37
2.5.1 Random Restarts 38
2.5.1.1 Optimal Restarting 38
2.5.1.2 Effects of Restarts 40
2.5.2 Queens and Satisfiability 42
2.5.3 GSAT 42
2.5.4 HSAT 42
2.5.5 WalkSAT 44
2.5.6 Novelty and R-Novelty 48
2.5.7 Simulated Annealing 48
2.5.8 Weighted GSAT and Weighted WalkSAT 51
2.5.8.1 Weighted GSAT 51
2.5.8.2 Breakout 51
2.5.8.3 Cha and Iwama 53
2.5.8.4 Frank 54
2.5.9 DLM 55
2.5.9.1 Lagrangian Methods 55

2.5.9.2 Primal-Dual Methods 56
2.5.9.3 DLM and SAT 58
2.5.9.4 The Evolution of DLM 59
2.5.10 Guided Local Search 63
2.6 Survey Propagation 63

3 Metrics for Local Search 65
3.1 Depth 65
3.2 Primal Mobility 68
3.3 Primal Coverage 71
3.4 Necessary vs. Sufficient 75
3.5 Dual Space Metrics 76
3.5.1 Dual Mobility 76
3.5.2 Dual Coverage 77
3.5.3 Dual Metrics: Experimental Results 77
3.6 Other Metrics Research 81
3.7 Conclusion 82

4 Local Search: SDF 83
4.1 Smoothed Objective Function 84
4.2 Weighted Objective 86
4.3 Multiplicative Updates 87
4.4 Flooding 87
4.5 Weight Smoothing 88
4.6 SDF Performance 88
4.7 Conclusion 90

5 Local Search: ESG 99
5.1 Boolean Linear Programming 100
5.2 BLP Formulation of SAT 101
5.3 Subgradient Optimization 102
5.4 ESG: Three Enhancements for Subgradient Optimization 104
5.4.1 Multiplicative Weight Updates 104
5.4.2 Nonlinear Penalty 104
5.4.3 Smoothing 105
5.5 Primal Search 106
5.6 Related Work 106
5.6.1 DLM 106
5.6.2 SAPS 108
5.7 Implementation 108
5.7.1 ESGflt 108
5.7.2 ESGint 109
5.8 ESG Performance 110

5.8.1 Comparison with Standard ILP Methods 110
5.8.2 Comparison with Local Search SAT Methods 112
5.9 Conclusion 112

6 ESG for Combinatorial Auctions 135
6.1 Combinatorial Auctions 135
6.2 Other CA Solvers 136
6.3 CA Instances 136
6.4 Comparison of Subgradient Optimization Methods 137
6.5 Comparison of Available Methods 138
6.6 Related Work 140
6.6.1 WSAT(OIP) 140
6.6.2 Backtracking SAT-based ILP Methods 141
6.7 Conclusions 141

7 Extending Local Search Via Hybrids 143
7.1 Previous Work On Hybrids 143
7.2 Hybrid: DualRes 144
7.2.1 Resolution Strategy 144
7.2.2 DualRes Experiments 146
7.2.2.1 Local Search Comparison: ESG vs. DualRes 146
7.2.2.2 Backtrack Search Comparison: DualRes vs. Chaff 148
7.2.3 Comments on DualRes 155
7.3 Hybrid: RESG 155
7.4 Hybrid: ESGProbe 156
7.5 Experiments 156
7.5.1 Three Variations 156
7.5.2 Broader Comparisons 163
7.6 Conclusions 180

8 Conclusion 181

A Experimental Environments 183
A.1 Environment 1 183
A.2 Environment 2 183
A.3 Environment 3 183

B Software Suites 185
B.1 ESG-SAT 185
B.2 HybSAT 185

C Default Parameters 187

D Details in Implementing Unit Propagation 189

E Lots of Numbers **191**

Bibliography **215**

List of Figures

2.1 Naive Backtrack Search Tree 17
2.2 Conflict Pruning Backtrack Search Tree for Satisfiable Formula (2.20) . . . 19
2.3 An Example of Unit Propagation 20
2.4 UP Pruning Backtrack Search Tree 21
2.5 Conflict Pruning Backtrack Search Tree for Unsatisfiable Formula (2.21) . . 24
2.6 Resolution Tree for Unsatisfiable Formula (2.21). (Space constraints oblige us to use commas instead of $\vee$ to indicate disjunction in this diagram.) . . . 25
2.7 An example of backjumping. The right column shows the formula reduced by the current assignment at each step. The left column explains each step. Variable assignments are shown along with their reason. 33
2.8 Summary of Backtracking Solvers and their features 36
2.9 Original (1992) GSAT vs. DPLL results [106] 43
2.10 Current GSAT vs. DPLL results, log expected time for HR instances with varying # of variables 44

3.1 Average primal mobility for a range of window lengths, obtained by various search procedures, averaged across 100 repetitions of the uf100-0953 problem in SATLIB. 71
3.2 Same as 3.1, but results averaged over 100 runs on all 1000 uf100 problems from SATLIB. 72

4.1 Average number of unsatisfied clauses (depth in g_{unsat}) achieved before reaching a local optimum or plateau. Results are obtained by running initial descents in g_{unsat} and g_{smooth} on uf100-0953 until the first strict local minimum or plateau point is reached, and then reporting the average g_{unsat} value at a given time point if at least 95 of 100 runs have successfully descended that many steps. 85

5.1 (i) The Lagrangian, $L(\mathbf{x}, \lambda)$, plotted for three different values of $\mathbf{x}$. (ii) The resulting concave dual function, $D(\lambda)$, (min of the three lines). 103
5.2 (i) A differentiable function, $f()$, its tangent plane, and corresponding gradient $\mathbf{u}$ for a single point. (ii) Two possible tangent planes for a point on $D(\lambda)$ and one of the corresponding subgradients, $\mathbf{v}$. 103

5.3 (i) Linear and (ii) Hinge Penalty functions 105

List of Algorithms

1 Naive Backtracking Algorithm 17
2 Conflict Pruning Backtracking Algorithm 18
3 Davis-Putnam-Logemann-Loveland Algorithm 23
4 relsat Variable Selection Heuristic 34
5 Generic Local Search Algorithm 37
6 Restart Strategy 38
7 GSAT 43
8 WalkSAT 45
9 WalkSAT-B 46
10 WalkSAT-G 47
11 Novelty 49
12 R-Novelty 50
13 WGSAT 52
14 Primal-Dual Methods 57
15 SDF 89
16 ESG 107
17 SAT-CA Reduction: A transformation encoding a MAX-SAT problem as a single unit combinatorial auction. 137
18 DualRes Resolution Algorithm 147
19 ESGProbe 157

Chapter 1

Introduction

> *"How beautiful the world is, and how ugly labyrinths are," I said, relieved. "How beautiful the world would be if there were a procedure for moving through labyrinths," my master replied.*
> - **The Name of the Rose**, Umberto Eco

The *satisfiability* problem (SAT) is the prototypical NP-complete problem, first established as such by Cook in 1971 [19]. The problem arises originally from theorem proving in propositional logic and persists to this day as an area of both theoretical and practical interest. The problem is as follows:

> Given n boolean variables, $\mathbf{x} = x_1, .., x_n$, and a propositional logic formula, F, in *conjunctive normal form* (CNF), does there exist an assignment to the variables such that the formula is *satisfied* (true).

1.1 Motivation

As the primary exemplar of the NP-complete complexity class, satisfiability is clearly worthy of study, if only to measure our progress in attacking hard problems of all kinds. Practical and theoretical progress on SAT teaches us how hard "hard" really is as we identify easier subclasses, algorithmic approaches offering acceptable average case performance, and specially hard instances.

Beyond this kind of exploration, the fact that other NP-complete problems can be translated into SAT means that effective SAT solvers represent potential approaches to these other problems, either by direct representation as SAT instances or by adapting SAT solution techniques for the new domain. A notable example of this kind of transfer is the use of SAT in planning problems [66]. The success of this approach spawned a wealth of planning research, and while subsequent research has moved away from SAT representations, the approach has left an indelible mark on the field of planning [114].

More directly, popular methods for formal verification of hardware and software involve determining the satisfiability of a CNF formula in order to detect errors in, or to prove

the correctness of, some system. This application is of substantial interest to hardware developers and receives attention from research groups attached to several manufacturers (e.g. Intel, AMD). Improvements to satisfiability testers translate into faster verification (see [110], a study of a wide range of satisfiability solvers applied to the verification of microprocessors).

There are a few distinct families of satisfiability solvers, each with its own strengths and weaknesses, and quite distinct in their approaches. In particular, backtracking search methods dating back to the 60's are widely researched and have seen substantial improvement in recent years. In the early 90's, a radically different family, the local search methods, arose and stimulated new interest in satisfiability. While this interest led to great activity on both sides, there has been surprisingly little exchange of ideas between these methodologies. Furthermore, successful local search is still only weakly understood, especially compared to backtrack search.

This thesis seeks to redress this failure, at least in part, by first exploring the somewhat poorly understood success of local search solvers and. secondly, by applying that knowledge to the design of a state-of-the-art local search method. Thirdly and finally, it explores augmentations of local search, hybridizing it with ideas from backtracking search in an attempt to augment its performance, and offering some insights into the interaction between these widely different methodologies.

1.2 Contributions

The key contributions of this research are:

- the definition and empirical evaluation of metrics for characterizing local search behaviour
- the development of SDF, a local search solver motivated by the intuitions corresponding to the metrics
- the development of ESG, an informed reconstruction of SDF with state-of-the-art performance and generalized to 0-1 integer linear programming problems
- the demonstration of ESG on combinatorial auctions
- the development of various hybrids between ESG and backtracking search solvers

We will start by establishing some basic terminology and proceed to a detailed background of both backtracking search and local search SAT solvers. From there, we explain each of our contributions in turn, providing empirical evidence at each step. We conclude with some short remarks on the value of the presented research.

Chapter 2

Background

And thus, by sleeping little and reading much, the moisture of his brain was exhausted to that degree that at last he lost the use of his reason.
- **Don Quixote**, Miguel de Cervantes

2.1 Terminology

Before considering approaches to the SAT problem it is important to establish some definitions. We start by restating the satisfiability problem as posed in the introduction and proceed to elaborate.

2.1.1 SAT and CNF

> Given n boolean variables, $\mathbf{x} = x_1, .., x_n$, and a propositional logic formula, F, in *conjunctive normal form* (CNF), does there exist an assignment to the variables such that the formula is *satisfied* (true).

A formula in CNF consists of a conjunction of m *clauses* (or *constraints*), $c_1, .., c_m$, where each clause consists of a disjunction over its *atoms*. Each atom is one of the *literals*, which are the set of variables and their negations (i.e. $x_1, .., x_n, \bar{x}_1, .., \bar{x}_n$). The unnegated literal of a variable x_i is called a *positive literal*. The negated literal (i.e. $\bar{x}_i$) is called a *negative literal*.

Thus, the general form for a CNF formula is:

$$F = (l_{11} \vee .. \vee l_{1k_1}) \wedge .. \wedge (l_{m1} \vee .. \vee l_{mk_m}) \tag{2.1}$$

where k_i is the *length* (number of atoms) of clause c_i, l_{ij} is the j-th atom in clause c_i, and $\{l_{i1}, .., l_{ik_i}\} \subset \{x_1, .., x_n, \bar{x}_1, .., \bar{x}_n\}$ such that $l_{ij} \neq \bar{l}_{ih}, 1 \leq j, h \leq k_i$ and $j \neq h$.[1]

[1] These conditions simply forbid duplicate literals in a clause and clauses containing a literal and its negation, which are trivially satisfied (*vacuous clauses*).

Such a formula is said to be an *instance* of the SAT problem.

If $k_i \le k, 1 \le i \le m$, then we say that an instance is in *k-CNF*. In particular, the literature frequently refers to 3-CNF instances. It is also fairly common to talk about *k-SAT instances*, which may generally be assumed to be in CNF.

The NP-complete SAT problem is a decision problem ("does a satisfying assignment exist"). However, practical solvers that prove satisfiability can answer the more general question ("return a satisfying assignment or report that no such assignment exists").

2.1.2 Assignments

Variables take a value of $\mathcal{T}$ (true) or $\mathcal{F}$ (false). Sometimes we will talk about variables being *unassigned* (or, identically, *undecided*). We will indicate this by assigning the value $\mathcal{U}$ (undecided) to the variable. Variables that have been assigned are called *assigned* (or, identically, *decided*).

An assignment of values to the variables will be indicated by a vector, $\mathbf{a} = a_1, .., a_n$, where a_i is the value currently assigned to variable x_i. In some cases we will describe one in a sequence of assignments to the variables by superscripting the assignment vector (e.g. $\mathbf{a}^{(t)} = a_1^{(t)}, .., a_n^{(t)}$). In cases where we wish to distinguish between only two or three assignments, we will use $\mathbf{a}$, $\mathbf{a}'$, and $\mathbf{a}''$.

If no variable is undecided in an assignment, it is said to be a *full assignment*. A *partial assignment* may have zero or more undecided variables (typically at least one but it is notationally convenient to allow partial assignments to be full). A *strictly partial assignment* has at least one undecided variable.

We will refer to a literal (or atom) as being *satisfied*, by which we mean that there is an assignment to the corresponding variable that agrees with the literal (e.g. if the literal is $\bar{x}_i$ then it is satisfied if and only if the variable x_i is assigned the value $\mathcal{F}$). If a contrary assignment has been made, we will describe the literal as *unsatisfied*, and if no assignment has been made to the variable, we will call the literal *undecided*.

Similarly, we will refer to a clause as being *satisfied* when one or more of its literals are satisfied, *unsatisfied* when all of its literals are unsatisfied, and *undecided* if none of its literals are satisfied and at least one is undecided. Finally, a particular instance is *satisfied* if and only if all of its clauses are satisfied by some assignment to the variables, *satisfiable* if one or more such assignments exist, and *unsatisfiable* if no such assignment exists.

In all of the above, we also allow the obvious term *decided* (i.e. either satisfied or unsatisfied).

2.1.2.1 Operations on Assignments

We will now define some basic operations and notation for constructing and transforming assignments.

An assignment can be defined by an n-tuple of values:

$$\mathbf{a} \leftarrow (v_1, v_2, \ldots, v_{n-1}, v_n)$$

where $v_i \in \{\mathcal{F}, \mathcal{T}, \mathcal{U}\}$ and the i-th value in the tuple is assigned to x_i.

Since it is not usually convenient to specify every single value, we also allow a set of assignments to specific variables (all unspecified variables are assumed to be unassigned):

$$\mathbf{a} \leftarrow \{x_i = v_i, x_j = v_j, \ldots, x_k = v_k\}$$

where $v_i \in \{\mathcal{F}, \mathcal{T}\}$.

For example, $\mathbf{a} \leftarrow \{\}$, specifies an empty assignment with all variables undecided.

Setting one particular variable in an assignment looks like this:

$$a_i \leftarrow v_i$$

where $v_i \in \{\mathcal{F}, \mathcal{T}, \mathcal{U}\}$.

More commonly, we will describe transformations of existing assignments that form new assignments, keeping the two distinct. We indicate the construction of a new assignment, $\mathbf{a}'$, by the change of a single value in the original assignment, $\mathbf{a}$, using:

$$\mathbf{a}' \leftarrow \mathbf{a}[x_i \leftarrow v_i]$$

where $v_i \in \{\mathcal{F}, \mathcal{T}, \mathcal{U}\}$.

One of most common transformations is assigning a single, previously unassigned variable, for which we give special syntax, distinguishing it with a plus sign:

$$\mathbf{a}' \leftarrow \mathbf{a} + [x_i \leftarrow v_i]$$

It is useful to indicate the *negation* of a currently assigned value thus:

$$neg(a_i)$$

where $a_i \in \{\mathcal{F}, \mathcal{T}\}$, $neg(\mathcal{T}) = \mathcal{F}$, and $neg(\mathcal{F}) = \mathcal{T}$.

In the interest of conciseness, wherever we have used a value and variable, we may substitute the corresponding literal. Examples are $\mathbf{a} \leftarrow \{x_1, \bar{x}_3\}$ (instead of $\mathbf{a}' \leftarrow \{x_1 = \mathcal{T}, x_3 = \mathcal{F}\}$) and $\mathbf{a}' \leftarrow \mathbf{a} + [\bar{x}_3]$ (instead of $\mathbf{a}' \leftarrow \mathbf{a} + [x_3 \leftarrow \mathcal{F}]$). The expression $\mathbf{a}' \leftarrow \mathbf{a}[neg(a_2)]$ 'flips' the value of variable x_2 in the new assignment.

2.1.2.2 Comparing Two Assignments

We now define some useful functions that provide comparisons and differences between two assignments.

$$\mathbf{ASSIGNED}(\mathbf{a}', \mathbf{a}'') = \{i \in [1, n] \mid a_i' = \mathcal{U} \wedge a_i'' \neq \mathcal{U}\} \tag{2.2}$$

$$\mathbf{UNASSIGNED}(\mathbf{a}', \mathbf{a}'') = \{i \in [1, n] \mid a_i' \neq \mathcal{U} \wedge a_i'' = \mathcal{U}\} \tag{2.3}$$

$\mathbf{ASSIGNED}()$ and $\mathbf{UNASSIGNED}()$ are the sets of variable unassigned/assigned in $\mathbf{a}'$and assigned/unassigned in $\mathbf{a}''$.

Note that $\mathbf{UNASSIGNED}(\mathbf{a}', \mathbf{a}'') = \mathbf{ASSIGNED}(\mathbf{a}'', \mathbf{a}')$.

$$\mathbf{FLIPPED}(\mathbf{a}', \mathbf{a}'') = \{i \in [1, n] \mid a'_i \neq \mathcal{U} \wedge a''_i \neq \mathcal{U} \wedge a'_i = neg(a''_i)\} \tag{2.4}$$

$\mathbf{FLIPPED}()$ is the set of variables flipped to turn one assignment into another. We will frequently assume without notice that the size of the flip set is one, especially when examining local search algorithms.

We may also wish to compare assignments in terms of sets of assigned variables. For example, $\mathbf{a}' \subseteq \mathbf{a}''$ iff $\mathbf{ASSIGNED}(\mathbf{a}'', \mathbf{a}') = \emptyset$ and $\mathbf{UNASSIGNED}(\mathbf{a}', \mathbf{a}'') = \emptyset$. We define proper subsets, supersets, intersections, and unions along similar lines. All such set operations and relations on assignments should be interpreted as the equivalent of those operations/relations on the sets of assigned variables in each assignment.

2.1.2.3 Prefixes

In some cases we wish to regard a set of assignments to variables as having occurred in some particular *order*. The order is defined over all assigned variables and the unassigned variables follow in no particular order. We will call such ordered assignments *prefixes* and may distinguish them by using the variable ρ instead of $\mathbf{a}$. The meaning of the various operations on assignments defined previously all apply in the same way to prefixes with only a few exceptions.

In particular, subscripting of ρ has a different meaning. Whereas a_i means the value assigned to x_i, ρ_i refers to the value of the i-th variable assigned. For example, if $\rho \leftarrow \{x_1, \bar{x}_3, x_4, x_7\}$ then $\rho_2 = \bar{x}_3$ and $\rho_3 = x_4$. Furthermore, we consider the assignment of a previously unassigned variable (e.g. $\rho' \leftarrow \rho + [x_i \leftarrow v_i]$) to be an assignment ordered immediately after all currently assigned variables.

We also define the following function only on prefixes:

$$AL(\rho,i) = \begin{cases} None & \text{if } x_i = \mathcal{U} \\ j & \text{if } \rho_j = a_i \end{cases} \tag{2.5}$$

The function $AL()$ computes the position of variable x_i within ρ if it is assigned, or $None$ if it is unassigned. This is the *assignment level* of the variable within the prefix.

2.1.3 Clause Statistics

2.1.3.1 Types of Clauses

A clause containing a single literal is called a *unary clause* or *unit clause*. A clause with two literals is a *binary clause*. A clause with three literals is a *ternary clause*.

A clause with no literals is called an *empty clause* and is designated by $\emptyset$. An empty clause is always unsatisfied (false). Therefore, any formula containing an empty clause is unsatisfiable.

2.1.3.2 Clause Statistics Under a Single Assignment

We will often need to consider the effects of assignments as regards which clauses are satisfied or unsatisfied and decided or undecided. We will often talk about the state of a particular clause or set of clauses *under* some assignment, $\mathbf{a}$.

Let $consistent(\mathbf{a})$ be a predicate that is true if no clauses are unsatisfied under assignment $\mathbf{a}$, and false otherwise.

Let $satisfying(\mathbf{a})$ be a predicate that is true if all clauses are satisfied under assignment $\mathbf{a}$, and false otherwise.

Let $\mathbf{SAT}(\mathbf{a})$ be the set of clauses that are satisfied under assignment $\mathbf{a}$, and $sat(\mathbf{a}) = |\mathbf{SAT}(\mathbf{a})|$.

Let $\mathbf{UNSAT}(\mathbf{a})$ be the set of clauses that are unsatisfied under assignment $\mathbf{a}$, and $unsat(\mathbf{a}) = |\mathbf{UNSAT}(\mathbf{a})|$.

Let $\mathbf{DEC}(\mathbf{a})$ be the set of clauses that are decided under assignment $\mathbf{a}$, and $dec(\mathbf{a}) = |\mathbf{DEC}(\mathbf{a})|$.

Let $\mathbf{UNDEC}(\mathbf{a})$ be the set of clauses that are undecided under assignment $\mathbf{a}$, and $undec(\mathbf{a}) = |\mathbf{UNDEC}(\mathbf{a})|$.

Let $\mathbf{SATLIT}(c, \mathbf{a})$ be the set of satisfied literals in clause c under assignment $\mathbf{a}$, and $satlit(c, \mathbf{a}) = |\mathbf{SATLIT}(c, \mathbf{a})|$.

Let $\mathbf{UNSATLIT}(c, \mathbf{a})$ be the set of unsatisfied literals in clause c under assignment $\mathbf{a}$, and $unsatlit(c, \mathbf{a}) = |\mathbf{UNSATLIT}(c, \mathbf{a})|$.

We define $\mathbf{DECLIT}(c, \mathbf{a})$ and $\mathbf{UNDECLIT}(c, \mathbf{a})$ similarly.

Let $\mathbf{POS}(x, \mathbf{a})$ be the set of clauses under assignment $\mathbf{a}$ that are undecided and contain the undecided, positive literal of x, and $pos(x, \mathbf{a}) = |\mathbf{POS}(x, \mathbf{a})|$.

Let $\mathbf{NEG}(x, \mathbf{a})$ be the set of clauses under assignment $\mathbf{a}$ that are undecided and contain the undecided, positive literal of x, and $neg(x, \mathbf{a}) = |\mathbf{NEG}(x, \mathbf{a})|$.

Let $\mathbf{BINPOS}(x, \mathbf{a})$ be the set of clauses under assignment $\mathbf{a}$ that are undecided, have exactly two undecided literals,[2] and contain the undecided, positive literal of x, and $binpos(x, \mathbf{a}) = |\mathbf{BINPOS}(x, \mathbf{a})|$.

Let $\mathbf{BINNEG}(x, \mathbf{a})$ be the set of clauses under assignment $\mathbf{a}$ that are undecided, have exactly two undecided literals, and contain the undecided, negative literal of x, and $binneg(x, \mathbf{a}) = |\mathbf{BINNEG}(x, \mathbf{a})|$.

We may also view the transformation of an individual clause by a given partial assignment as a clause itself. The effect of an assignment will be either to unsatisfy the clause, satisfy it, or *reduce* it into a shorter clause consisting only of its undecided literals. We will refer to this transformation as the *reduction* or a *reduced clause*.

Let $c^{\mathbf{a}}$ be the reduction of a clause, c, of length k, under some assignment, $\mathbf{a}$.

$$c^{\mathbf{a}} = \begin{cases} \mathcal{T} & \text{if } satlit(c, \mathbf{a}) \geq 1 \\ \emptyset & \text{if } unsatlit(c, \mathbf{a}) = k \\ \mathbf{UNDECLIT}(c, \mathbf{a}) & otherwise \end{cases}$$

[2]These are *binary clauses* under a given assignment, $\mathbf{a}$. Binary clauses are important to many SAT heuristics.

Note that in discussing the length of a clause, c, we may refer to its length as specified in the instance (i.e. k) or, when discussing partial assignments, we may refer to the length of the reduced clause which we will designate by $k^{\mathbf{a}}$. In the latter case, we are generally only interested in undecided clauses, particularly when the reduced clause is a unit or binary clause, but for completeness we will define the following:

$$k^{\mathbf{a}} = \begin{cases} \text{undefined} & \text{if } satlit(c, \mathbf{a}) \geq 1 \\ 0 & \text{if } unsatlit(c, \mathbf{a}) = k \\ undeclit(c, \mathbf{a}) & otherwise \end{cases}$$

When discussing partial assignments, we will frequently make references to reduced clauses and their lengths without being explicit about the reduction. This should generally be understandable from the context of the discussion.

2.1.3.3 Clause Statistics Under a Change in Assignment

We also need to consider the effects of changing from one assignment to another. We will consider several functions of two assignments, $\mathbf{a}'$ and $\mathbf{a}''$. In particular, we want to describe the set of clauses that are unsatisfied under assignment $\mathbf{a}'$ and satisfied under assignment $\mathbf{a}''$ (*made clauses*) and the set of clauses that are satisfied under assignment $\mathbf{a}'$ and unsatisfied under assignment $\mathbf{a}''$ (*broken clauses*). We define these more formally below, along with several other similar functions.[3]

$$\mathbf{MAKE}(\mathbf{a}', \mathbf{a}'') = \mathbf{UNSAT}(\mathbf{a}') \cap \mathbf{SAT}(\mathbf{a}'') \tag{2.6}$$

$$make(\mathbf{a}', \mathbf{a}'') = \left|\mathbf{MAKE}(\mathbf{a}', \mathbf{a}'')\right| \tag{2.7}$$

$$\mathbf{BREAK}(\mathbf{a}', \mathbf{a}'') = \mathbf{SAT}(\mathbf{a}') \cap \mathbf{UNSAT}(\mathbf{a}'') \tag{2.8}$$

$$break(\mathbf{a}', \mathbf{a}'') = \left|\mathbf{BREAK}(\mathbf{a}', \mathbf{a}'')\right| \tag{2.9}$$

$$loss(\mathbf{a}', \mathbf{a}'') = unsat(\mathbf{a}'') - unsat(\mathbf{a}') = break(\mathbf{a}', \mathbf{a}'') - make(\mathbf{a}', \mathbf{a}'') \tag{2.10}$$

$$gain(\mathbf{a}', \mathbf{a}'') = -loss(\mathbf{a}', \mathbf{a}'') \tag{2.11}$$

$$\mathbf{STAYSAT}(\mathbf{a}', \mathbf{a}'') = \mathbf{SAT}(\mathbf{a}') \cap \mathbf{SAT}(\mathbf{a}'') \tag{2.12}$$

[3] The terms *gain* and *loss* are based on terminology found in [41], which uses the term *net gain*. It also uses the term *positive gain* for $make()$ and *negative gain* for $break()$.

$$staysat(\mathbf{a}', \mathbf{a}'') = \left|\mathbf{STAYSAT}(\mathbf{a}', \mathbf{a}'')\right| \tag{2.13}$$

$$\mathbf{STAYUNSAT}(\mathbf{a}', \mathbf{a}'') = \mathbf{UNSAT}(\mathbf{a}') \cap \mathbf{UNSAT}(\mathbf{a}'') \tag{2.14}$$

$$stayunsat(\mathbf{a}', \mathbf{a}'') = \left|\mathbf{STAYUNSAT}(\mathbf{a}', \mathbf{a}'')\right| \tag{2.15}$$

$$\mathbf{DECIDE}(\mathbf{a}', \mathbf{a}'') = \mathbf{UNDEC}(\mathbf{a}') \cap \mathbf{DEC}(\mathbf{a}'') \tag{2.16}$$

$$decide(\mathbf{a}', \mathbf{a}'') = \left|\mathbf{DECIDE}(\mathbf{a}', \mathbf{a}'')\right| \tag{2.17}$$

$$\mathbf{UNDECIDE}(\mathbf{a}', \mathbf{a}'') = \mathbf{DEC}(\mathbf{a}') \cap \mathbf{UNDEC}(\mathbf{a}'') \tag{2.18}$$

$$undecide(\mathbf{a}', \mathbf{a}'') = \left|\mathbf{UNDECIDE}(\mathbf{a}', \mathbf{a}'')\right| \tag{2.19}$$

2.1.4 Resolution

Proofs in propositional logic are frequently based on the operation called *resolution* ([26, 92], also see [94], pages 166-174, for a concise account of propositional resolution). Two clauses may be *resolved* if each has one of an opposing pair of literals (e.g. A and $\bar{A}$). The variable A cannot be assigned to satisfy both clauses, so an additional constraint is placed on the other variables in the two clauses. This constraint consists of a disjunction of all literals in the original two clauses except the two opposing literals. The result is called a *resolvent*.

The following example demonstrates the resolution operation, indicated by the horizontal line separating the original clauses (or *resolvands*) and the resolvent.

$$\begin{array}{c} A \vee B \\ \bar{A} \vee C \\ \hline B \vee C \end{array}$$

Clearly, either B or C must be true in order to satisfy the original two clauses and so that is the new constraint. We will occasionally use a phrase like "resolve on A" to indicate the opposing literals used in resolution.

There are a few interesting special cases of resolution. In the following case, the resolvent is strictly shorter than either of the resolvands. We call this a *contracting resolution*.

$$\begin{array}{c} A \vee B \vee C \\ \bar{A} \vee B \vee C \\ \hline B \vee C \end{array}$$

In the next case, resolution produces an empty clause (designated by $\emptyset$).

$$\frac{\begin{array}{c}A\\ \bar{A}\end{array}}{\emptyset}$$

Obviously, the two resolvands cannot both be satisfied under a single assignment. This means that any formula with those two clauses is unsatisfiable. So if resolution can be applied to a formula F to produce the empty clause, then F is unsatisfiable. This situation $(A \wedge \bar{A})$ is called a *contradiction* and the use of resolution to prove a contradiction is called *resolution refutation.* Note that several resolution steps may be required to arrive at the empty clause.

In the following case, resolving on either A or B will produce a *tautology* (or *vacuous clause*), a clause that is true under any assignment and is therefore useless. Useful resolution can only be performed on clauses with only one pair of opposing literals.

$$\frac{\begin{array}{c}A \vee \bar{B}\\ \bar{A} \vee B\end{array}}{B \vee \bar{B}}$$

2.1.5 Subsumption

Formulas can contain redundant clauses whose omission does not change the set of satisfying assignments. One case of this is when one clause *subsumes* some other clause ([94], page 286). Given two clauses, c_i and c_j, $i \neq j$, c_i is said to subsume c_j iff $c_i^{\mathbf{a}} = \mathcal{T} \implies c_j^{\mathbf{a}} = \mathcal{T}, \forall \mathbf{a}$. For example, $c_1 = (x_1, x_3)$ subsumes $c_2 = (x_1, \bar{x_2}, x_3)$. In such a case, we can simply delete the subsumed clause from the formula, potentially saving work when solving the instance. Clauses may be redundant even if not subsumed by any other clause, but in general there is no simple way to detect this (e.g. a subsuming clause may be discoverable only by a long chain of resolutions).

2.2 Instance Types

There are many sources for instances of the SAT problem and some understanding of the variation in the resulting instances is important to understanding the relative strengths and weaknesses of the various solution methods.

2.2.1 Instance Hardness

The *difficulty* or *hardness* of an instance is frequently referred to in the literature without any formal definition. In such cases, it should be treated as a relative term used to compare instances. Typically, a *hard* instance is understood to be one that is computationally expensive for all known solvers, compared with other instances of similar size.[4] Even so, a

[4]The "size" of an instance has no widely accepted definition either. One might use the number of variables, the number of clauses, the number of literals, or some combination of these to characterize a formula's size.

hard instance may be easier (require less computation) for one solver than for another. The distinction has chiefly arisen from research on randomly generated instances, where some generators were found to create mostly easy instances, while other generators can reliably produce more difficult instances (see Section 2.2.4 for references and further discussion).

More formal definitions of hardness are typically related to algorithms for solving the problem. We will discuss various approaches in Section 2.3, but will briefly state some natural hardness measures related to common approaches here.

For unsatisfiable instances, the size of a minimal resolution proof is a natural measure or the smallest search tree traversed by a backtracking search with pruning (these two are essentially equivalent). For satisfiable instances, it is more difficult to quantify since there is no guaranteed lower bound (e.g. a random assignment to the variables could find a solution right away). One possibility is to consider the number of solutions for the instance. Another possibility is to consider how many assignments are considered by a solver during its search. In many cases, only probabilistic measures are appropriate (e.g. expectations).

More strict notions of hardness may be obtainable for particular classes of instances (e.g. measures from statistical physics [81] applied to so-called *hard, random instances* – see Section2.2.4). In the end, most empirical SAT research compares the runtimes of various solvers to judge the hardness of instances and the term "hardness" itself is used quite loosely.

2.2.2 Instance Variety

When studying SAT solvers, experimentation reveals two important facts, which we simply assert here but which the wealth of SAT research and our own results amply confirm:

1. SAT instances are not all equally difficult
2. no single solver offers the best performance on every instance

The first fact is true even of instances whose gross characteristics are similar (e.g. same number of variables and/or clauses). The construction of an instance can be critical in determining its difficulty.

The second fact reflects the variety of approaches used to attack SAT instances. Different approaches involve different heuristics and, by chance or design, emphasize different aspects of the solution process. There are dramatic differences amongst the best solvers currently available and progress is made on several fronts rather than by incremental improvements to a single, basic approach.

This makes it important to identify the types of instances we are dealing with in experimentation and ensure that we do not evaluate any solver on too narrow a range of instances.

Because the number of variables gives a natural (albeit trivial) upper bound on the search (i.e. $O(2^n)$), it is the most commonly used size comparison. However, studies of instance hardness must select their framework for comparison and must be judged within that framework as there is no universally accepted criterion for comparison.

We present a crude categorization of instance types for use in our experiments and discussion.

2.2.3 Structured Instances

2.2.3.1 Natural Instances

There are certain applications whose problems are most naturally posed as the satisfiability of some propositional formulae. Where this is the case, instances can be obtained from research on the application or from "real-world" work. These instances are particularly important since there is usually a practical value in solving them. We call these *natural instances*.

One source of natural instances has considerable interest to both research and industrial communities. *Formal verification* attempts to prove the presence or absence of some particular properties in a computer hardware or software system. These problems are frequently posed in terms of propositional logic. Verification, especially of hardware systems, is of great interest to industries such as microprocessor manufacturers. These instances can be extremely large and their study has significantly pushed the envelope of SAT solvers in recent years [110]. While we will not pursue the subject of formal verification specifically, we will demonstrate results on instances of this kind. For a concise tutorial on the use of SAT in formal verification, see [11].

2.2.3.2 Transformed Instances

Since all NP-complete problems can be transformed into one another, there is a wealth of SAT instances to be obtained by transforming other NP-complete problem instances. Given a *source problem*, there may be existing instances from earlier research or from a particular application. In particular, we are interested in well-studied instances of known difficulty, instances that have been analyzed and characterized by experts in that domain. However, one may often randomly generate instances of the source problem and transform those into SAT.

We group these transformed instances under "structured instances" even when the source instances are randomly generated because even random instances typically have some noticeable structural commonalities when transformed to CNF and tend to behave similarly. In other words, they are structured from the SAT point of view, even if they are not from the point of view of the source problem.

Transformed problems vary widely in difficulty and it is tricky to judge their value. Converting all NP-complete problems to SAT in order to solve them is an attractively simple idea but, in practice, the SAT formulation may be much harder to solve than the original [8]. These instances are of interest chiefly for what we can learn about the solvers we attack them with.

2.2.4 Hard Random Instances

In 1992, work on GSAT (see Section 2.5.3) was coupled with an attempt to find a reliable generator of hard instances [83]. The distributions of instances created by some random generators have led to deceptively favourable results in the past, as Franco and Paull pointed out in 1983 [33]. Existing random instance generators tended to generate trivial instances. It was discovered that the difficulty of random instances generated via the following algorithm depends on the *clause-variable ratio*: $r = \frac{m}{n}$. The generator for 3-CNF is simple:

1. Select a ratio, r, and a number of variables, n.
2. Generate $m = nr$ clauses in the following fashion:
    (a) randomly select three variables, without replacement, to form the clause
    (b) for each variable in the clause, negate it with probability 0.5

For low values of r ($\lesssim 4.2$ for 3-CNF), instances tend to be underconstrained, satisfiable and easy to decide. For large values of r ($\gtrsim 4.26$ for 3-CNF), instances tend to be overconstrained and are easily proved unsatisfiable. A region in the middle, commonly referred to as the *phase transition* is where the difficult instances lie (see Cheeseman et al. for other examples of phase transitions [17]). At a particularly interesting point, instances are produced in roughly equal numbers of satisfiable and unsatisfiable. These instances are, on average, harder than at any other ratio (i.e. every known solver typically takes longer to solve such instances). For 3-CNF, this point has been experimentally determined to be around $r = 4.26$ [20].

The discovery of a source of *hard, random instances* (*HR instances*) was both beneficial and, in a sense, diverting. While it led to more stringent testing of existing algorithms, it has also guided research on solvers to produce several that perform well on this artificial class but comparatively poorly on practical, real-world instances. The field has fragmented along lines devoted to different subclasses of the problem. Whether this is, in fact, a necessity rather than an accident is a key question, since there is still no solver that dominates across the board. The study of HR instances is nonetheless important. Some researchers (e.g. [22]) suggest that such instances represent the "hard" essence of a structured instances after all structure has been exploited, although this has not been clearly demonstrated. Along similar lines, Williams et al. have identified *backdoors,* small subsets of the variables that, if searched, suffice to solve the instance [115].

There have been various theoretical attempts to explain the phase transition effect. Cook and Mitchell have written a concise survey in this area that also discusses several solvers [20]. Also see [22, 38] for extensive empirical results. Most recently, research on spin-glasses from statistical physics has provided new insight into the phase transition via the *survey propagation* algorithm, placing tighter bounds on the region and producing a solver capable of solving satisfiable HR instances of much greater size than seemed feasible before (e.g. tens or even hundreds of thousands of variables) [81]. Some groups have also worked

on other methods for generating hard problems (e.g. [7]) or instances whose properties can be systematically controlled (e.g. the quasigroup or Latin squares problem [47]).

2.2.5 Sources of Instances

A wide variety of SAT instances are available from various researchers. Two popular collections are the DIMACS collection, used for the second DIMACS challenge [1], and SATLIB [60]. These two are the main sources of instances used in this thesis although a few instances were generated by us or obtained from a variety of sources. The selection includes many hard, random instances, transformed instances, and natural instances from planning and formal methods. Instances are described in the captions of experimental result tables throughout this document. It is worth noting that a wealth of newer instances are now available as the result of the recent SAT competitions although we have not explored these much as of yet.

2.3 Solution Approaches

There are a few general ways to approach solving a SAT instance, and the choice depends in part on whether the instance is satisfiable or not. In most cases, we do not know this (it is, after all, the answer), but we may believe one answer to be more likely than the other. In some cases, we may be fairly confident that the instance is satisfiable but, in addition to the satisfiable decision, we would like a satisfying assignment.

Some methods are *complete*, which means that they will correctly determine an instance to be satisfiable or unsatisfiable. Other methods are *incomplete*, and can only prove an instance satisfiable. Clearly, if we suspect an instance may be unsatisfiable and require proof that it is, an incomplete solver is not appropriate. Contrariwise, if we seek a satisfying assignment an incomplete solver may find a solution faster than a complete solver.

After identifying some broad classes of approach, we will turn to considering specific solvers and their relations to these classes (starting with Section 2.4).

2.3.1 Resolution Refutation

For unsatisfiable instances one may apply *resolution refutation*. The operation of *resolution* is repeatedly applied to derive an empty clause, which proves the instance is unsatisfiable. Proving satisfiability, in the most simplistic fashion, is done by performing all possible resolutions without discovering the empty clause. The method is therefore, complete.

2.3.2 Model Finding

For satisfiable instances, it is sufficient to find a single assignment to the variables that satisfies the instance. Such an assignment is called a *model*, and any approach that searches for a satisfying assignment is a *model finder*. Naturally, there are no models for an unsatisfiable instance, so many (but not all) model finders are incomplete methods. The chief differences

among model finders lie in the way they explore the space of assignments, and this has a fundamental effect on their completeness.

2.3.3 Systematic Search

Enumerating all possible assignments is a complete method because, if a model exists, it will be found during the search, and if no model exists, then the search will terminate when all assignments have been examined. This is an example of a model finder that is complete. Of course, with 2^n possible assignments, a full exhaustive search is infeasible for all but trivial instances. However, the search space can frequently be pruned so that large sets of assignments can be discounted without examining them individually. Such searches are still complete because we only prune assignments that we are certain do not contain a model.

We call such methods *systematic searchers* or *systematic model finders* because they explore the entire space of assignments and can determine when all assignments have been considered. This is usually done by considering partial assignments rather than full assignments.

2.4 Backtracking Search Solvers

The best known and currently most successful family of complete SAT solvers is rooted in the Davis-Putnam solver (DP) [26]. This work was an attempt to automatically refute quantified predicate logic statements by generating a sequence of propositional instantiations of the original formula. If any subset of these instantiations can be proved unsatisfiable then the original predicate logic formula is false. The DP approach generates a propositional formula in CNF and applies resolution refutation to eliminate variables from the formula one by one, together with some important optimizations.

DP is a resolution refutation solver, constructing a full resolution proof explicitly in memory using its *elimination rule* (resolving away variables one by one). Because it constructs an explicit resolution refutation, the elimination rule can result in exponential space requirements while solving. This space issue was a critical problem in the 1960's and rendered the approach infeasible for all but the smallest instances. The Davis-Putnam-Logemann-Loveland (DPLL) method [25], derived from DP, addressed this problem and became the basis for many of the best solvers known today, which fall into a general class that we will call *backtracking search solvers*.

We will not further consider non-backtracking resolution refutation solvers as a separate class, although contemporary solvers use the present-day abundance of memory to keep resolvents in much the same way. We only wish to point out that DP did not use backtracking but nonetheless contributed key concepts (i.e. CNF formulae, pure literals, and unit propagation) to its successor, DPLL, and most backtracking solvers thereafter.

We will develop the notion of backtracking search carefully, introducing ideas and notation step-by-step. The literature often uses several different terms for a given concept and it is not always readily apparent what the authors intend. Naturally, we wish to avoid this

problem. The treatment here begins with the absolute basics, so readers familiar with the area may wish to skip or skim the early sections.

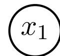
2.4.1 Backtracking Search

2.4.1.1 Naive Backtracking

One obvious, naive algorithm is simply to consider all 2^n possible assignments to the variables and test each one to see if it satisfies the instance. If no assignment satisfies, the instance is unsatisfiable. One way to implement this is via *backtracking search*. The search could be depth-first or breadth-first. However, a breadth-first search would require space exponential in n, whereas depth-first requires only linear space, making it the only practical choice for large instances.

Such a backtrack search need only maintain a single partial assignment. The algorithm for what we will call *naive backtracking* is described recursively (see Algorithm 1) but most practical implementations are iterative. Note that the $variable_select()$ function is unspecified. For now we will simply assume that it returns the index of the first (ordered by index) unassigned variable or $None$ if no variables are unassigned. This is a *lexicographical variable ordering*. We will discuss variable selection in greater detail in Section 2.4.4.

We can view the search diagrammatically using a *search tree*. One possible tree representation of naive backtrack for an unsatisfiable instance of three variables is shown in Figure 2.1. Each circular node



represents a variable under consideration, and the labelled branches downward from a node represent the value choices. The solid, filled squares (e.g. ■) indicate that the algorithm has determined that the partial assignment at that point cannot satisfy the instance.

The numbers down the left side of the diagram show the *assignment levels* in the tree, where the root is at assignment level 1 ($AL = 1$). When discussing the relative positions of variables along a branch of the search tree we will refer to variables of higher assignment level as being *later* or *more recent*. Similarly, variables of lower assignment level are *earlier* or *less recent*. Often we are concerned with the *latest* or *most recent* assignment.

The partial assignment and ordering along a branch extending from the root to any node corresponds to our notion of a *prefix* (see Section 2.1.2.3). We will consider the current partial assignment in a backtrack search tree as a prefix.

There are other ways in which the search tree can be shown diagrammatically but this way suits our purposes here.

2.4.1.2 Conflict Pruning

Backtracking is clearly not the only way to enumerate all possible assignments and is actually a poor way to implement exhaustive search for SAT (testing assignments would be

Algorithm 1 Naive Backtracking Algorithm

NaiveBacktrack()

1. Start with empty assignment: $\rho \leftarrow \{\}$
2. If $NaiveBacktrack_Branch(\rho) = unsatisfiable$, terminate with result *unsatisfiable.*
3. Otherwise, terminate with result *satisfiable.*

NaiveBacktrack_Branch(ρ)

1. Select *choice variable* for assignment: $i \leftarrow variable_select(\rho)$
2. If $i = None$ (all variables are assigned)
   (a) If $satisfying(\rho)$, return *satisfiable*.
   (b) Else, return $unsatisfiable$.
3. $\rho' \leftarrow \rho + [x_i \leftarrow \mathcal{F}]$
4. If $NaiveBacktrack_Branch(\rho') = unsatisfiable$
   (a) $\rho' \leftarrow \rho + [x_i \leftarrow \mathcal{T}]$
   (b) If $NaiveBacktrack_Branch(\rho') = unsatisfiable$, return $unsatisfiable$.
5. Return *satisfiable.*

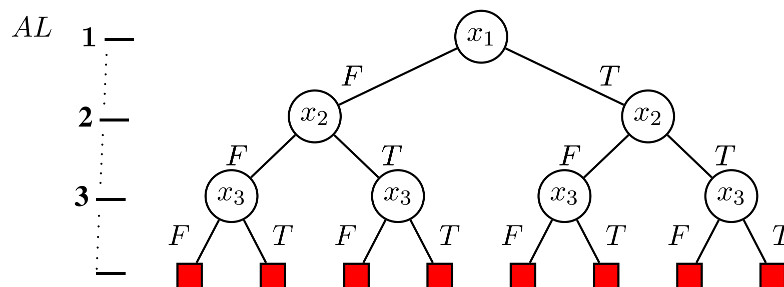


Figure 2.1: Naive Backtrack Search Tree

much faster if the enumeration were a Gray code). However, the choice is motivated by the first in a series of search improvement we will describe. Quite simply, because SAT is a constraint satisfaction problem, we can stop considering a partial assignment as soon as some constraint is violated.

In the SAT case, a violated constraint is a clause that has been reduced to an empty clause. The following simple example demonstrates the idea. Given the clause, $c = (x_1 \vee x_2 \vee x_4)$, and the partial assignment, $\mathbf{a} \leftarrow \{\bar{x}_1, \bar{x}_2, x_3, \bar{x}_4\}$, the result is the reduced clause, $c^{\mathbf{a}} = \emptyset$. Since c is unsatisfied, the instance as a whole can never be satisfied by any partial assignment that is a superset of $\mathbf{a}$ (i.e. $\mathbf{a}$ is not consistent). Therefore, instead of waiting until all variables are assigned, we backtrack as soon as any clause is reduced to empty (see Algorithm 2 – only the branching function is shown, the rest is the same as Algorithm 1).

Algorithm 2 Conflict Pruning Backtracking Algorithm

ConflictPruningBacktrack_Branch(ρ)

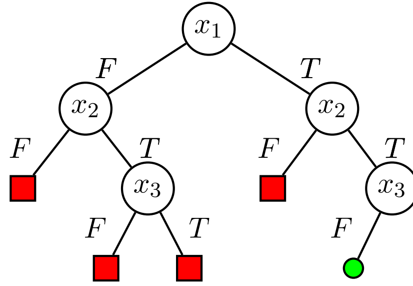
1. If not $consistent(\rho)$, return $unsatisfiable$.
2. Select *choice variable* for assignment: $i \leftarrow variable_select(\rho)$
3. If $i = None$ (all variables are assigned), return *satisfiable*.
4. $\rho' \leftarrow \rho + [x_i \leftarrow \mathcal{F}]$
5. If $ConflictPruningBacktrack_Branch(\rho') = unsatisfiable$
   (a) $\rho' \leftarrow \rho + [x_i \leftarrow \mathcal{T}]$
   (b) If $ConflictPruningBacktrack_Branch(\rho') = unsatisfiable$, return $unsatisfiable$.
6. Return *satisfiable*.

Figure 2.2 shows an example where nodes are pruned by detecting empty clauses within the following satisfiable formula:

$$F = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3) \wedge (x_1 \vee x_3) \tag{2.20}$$

We use a small, filled circle (e.g. ●) to indicate a satisfying assignment in the search tree.[5] When we discover an empty clause, we mark the assignment with a solid, filled square

[5]Strictly speaking, the formula (2.20) is satisfied as soon as $x_1 \leftarrow \mathcal{T}$ and $x_2 \leftarrow \mathcal{T}$, because there are two solutions, corresponding to $x_3 \leftarrow \mathcal{F}$ and $x_3 \leftarrow \mathcal{T}$. In practice, some algorithms will detect satisfying partial assignments while others will only detect full assignments that lead to no conflicts. We will assume the latter form. Furthermore, most implementations stop after finding a single solution, although it is fairly straightforward to find multiple solutions. As our search tree suggests, we will assume that only single solutions are sought.



$$F = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3) \wedge (x_1 \vee x_3)$$

Figure 2.2: Conflict Pruning Backtrack Search Tree for Satisfiable Formula (2.20)

(e.g. ■), as we did for non-satisfying full assignments. We will call the discovery of an empty clause a *conflict*, reflecting the fact that some choice we have made is incompatible with the constraints and that we must revisit an earlier choice in order to resolve the conflict. In general, it is conflicts that force us to backtrack.

2.4.2 Unit Propagation

Unit propagation is a key feature found in all successful backtracking solvers. The more general notion of *constraint propagation* has become a cornerstone of contemporary CSP algorithms [27], so it is well worth understanding the special case presented by SAT.

2.4.2.1 Davis-Putnam-Logemann-Loveland

The Davis-Putnam-Logemann-Loveland (DPLL) method [25] changed DP's resolution-based elimination rule to a *splitting rule*, which is simply the backtrack search procedure with conflict pruning already described. DPLL also has two other notable features, *unit propagation* (UP) and *pure literal* handling (described in Section 2.4.3). We will first discuss unit propagation.

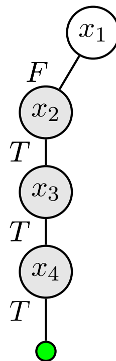
Consider a *unary* or *unit clause* (i.e. an undecided clause with a single undecided literal), like $(\bar{x}_3)$. It is clear that the only way to satisfy a formula containing such a clause is to satisfy the clause's one literal (i.e. assign $x_3 \leftarrow \mathcal{F}$). Since we are constrained to make this assignment, we call it a *forced move* (also referred to as *forced literal* or *implication*), as opposed to a *choice move*, where there are no constraints on the variable or value we may choose.[6]

Such forced moves are valuable because they reduce the number of decisions we must make, reducing the size of the search subtree by a factor of two for each forced move. Clearly, discovering many forced moves offers exponential savings in search time.

[6]Choice moves are also commonly called *decision moves* or *open moves*. We use the term choice to avoid confusion with "decision" in the sense of a decision regarding satisfiability and feel that "open" is too vague.

Now consider a formula $F = (x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee x_4)$. If we make the initial assignment, $x_1 \leftarrow \mathcal{F}$, the first clause is reduced to the unit clause (x_2). If we immediately apply this forced move, the second clause is, in turn, reduced to (x_3) and, continuing the process, the fourth clause to (x_4). Thus, a single choice forced the assignment of all remaining variables by a succession of unit clauses. We call this process *unit propagation*, and it is undoubtedly the most important optimization found in any backtrack solver.

Figure 2.3 shows this unit propagation example as a search tree. We indicate variables whose values are forced by a shaded circle with the variable name (e.g. x_i) and that only a single value is possible by using a single, downward edge instead of branching.



$$F = (x_1 \vee x_2) \wedge (\bar{x}_2 \vee x_3) \wedge (\bar{x}_3 \vee x_4)$$

Figure 2.3: An Example of Unit Propagation

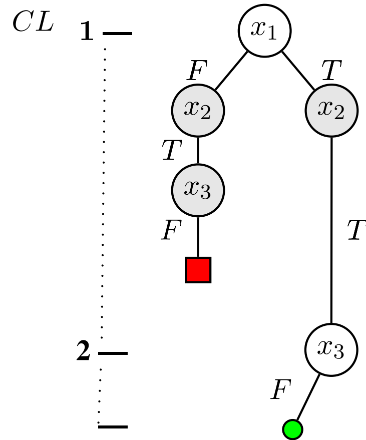
If we consider our earlier example, formula (2.20), setting $x_1 \leftarrow \mathcal{F}$ *implies* three forced moves by reducing the first, third, and fourth clauses to unit clauses, (x_2), $(\bar{x}_3)$, and (x_3). Clearly, the second and third unit clauses form a contradiction, a specific form of conflict.[7] Figure 2.4 shows one possible search tree resulting from unit propagation on formula (2.20).[8]

2.4.2.2 Choice Moves and Choice Levels

With the introduction of UP, we now have two distinct kinds of moves: *forced moves* and *choice moves*. Forced moves are those moves we are constrained to make because of unit

[7] Our earlier description of conflict was loosely defined as between value choices and the constraints. With UP, it makes more sense to identify conflicts as contradictions between implied literals, although this is really only another view of the same thing. Hereafter, we will use *conflict* and *contradiction* interchangeably because we are unlikely to consider scenarios without UP.

[8] Exactly how processing should proceed in our example depends on how these unit clauses are detected and handled, details which are usually omitted in the literature. Readers interested in this issue generally and those who wish to clarify any ambiguity in the algorithms behind some examples may consult Appendix D.



$$F = (x_1 \vee x_2) \wedge (\bar{x}_1 \vee x_2) \wedge (x_1 \vee \bar{x}_3) \wedge (x_1 \vee x_3)$$

Figure 2.4: UP Pruning Backtrack Search Tree

clauses while choice moves are moves where we can freely select a variable and a value for that variable. In the simple algorithms introduced before UP, all moves were choice moves.

Typically, as long as any forced moves are implied by the current assignment, only those moves are executed. It makes little sense to consider another choice move before fully exploring the implications of the last choice move. This leads to a pattern of a choice move followed by zero or more forced moves, repeated over and over.

This suggests a new way of grouping assignments, unlike the assignment levels described earlier. Instead, we consider *choice levels*, where each level starts with a choice move, followed by any implied forced moves. We use CL to designate the current choice level in the search and Figure 2.4 is spaced and labelled so as to make the choice levels clear (they are indicated down the left side). The first choice, the root, is labelled $CL = 1$.

When backtracking, one need not consider forced moves, but only the most recent choice move. Thus, we backtrack to the most recent choice level when we hit a contradiction. All variables in a choice level can be unassigned at the same time.

2.4.2.3 Open and Closed Prefixes

A given partial assignment or prefix may or may not imply forced moves. This leads us to define the notion of a *closed prefix*, a prefix that is “closed” under the UP operation in the sense that applying UP will not change the prefix. Such a prefix implies no unit clauses and consists of a sequence of complete choice levels. An *open prefix* is one that implies one or more unit clauses. Applying UP to an open prefix will result in further assignments. An open prefix consists of a sequence of complete choice levels followed by one incomplete choice level.

By our definition in Section 2.1.3.2, a partial assignment that produces an unsatisfied

clause is *inconsistent*. Note that this definition only refers to variables actually decided by the assignment, not to any implied literals. So, by our definition, $\{\bar{x}_1\}$ is consistent in our last example (Figure 2.4), even though it implies a contradiction. We would have to consider an assignment like $\{\bar{x}_1, x_2, \bar{x}_3\}$ before a clause becomes unsatisfied. We call an open prefix that will be consistent after UP *implied consistent*, and an open prefix that will be inconsistent after UP, *implied inconsistent*.

2.4.2.4 Notation for UP

We define the operator $UP()$, which generates a new prefix, ρ', by applying unit propagation to a prefix, ρ:

$$\rho' \leftarrow UP(\rho)$$

The new prefix is either identical (because ρ is closed) or is extended by the assignment of a single variable. The $UP^*()$ operator applies the closure of $UP()$, possibly assigning multiple variables and resulting in a ρ' that is a closed prefix:

$$\rho' \leftarrow UP^*(\rho)$$

This operation extends the original prefix by zero or more assigned variables (i.e. if applied to a closed prefix, that prefix will not change). Obviously, this operation is simply the repeated application of the single UP operation until the prefix does not change. Both notations may also be applied to partial assignments (e.g. $\mathbf{a}$).

2.4.3 Pure Literals

If, at any point during the search, a variable is found that occurs either only positively or only negatively in the current set of clauses, then we have a *pure literal*. Obviously we can assert a pure literal, satisfying any clauses in which it occurs. This produces a subinstance with the same satisfiability as the original instance (i.e. if the original was (un)satisfiable, the subinstance will be the same). If we examine formula (2.20), we can see that x_1 and x_4 are pure literals, allowing us to satisfy two clauses automatically. The remaining clause can be trivially satisfied.

Pure literals were used in the original DPLL algorithm (see Algorithm 3), but are rarely a key feature in any solver.[9] No contemporary solver expends significant effort to detect pure literals. They are only checked for if the data structures make it inexpensive to do so. Moves made due to pure literals are often called *free moves* because the move cannot increase the cost of subsequent search and may (depending on implementation) decrease the cost. We will avoid this term because of potential confusion with choice moves and instead call them *pure literal moves.*

We use a notation for pure literal moves similar to that used for UP, $\rho' \leftarrow PL(\rho)$, and the corresponding closure, $\rho' \leftarrow PL^*(\rho)$.

[9] In fact, UP and pure literals were features in the non-backtracking DP algorithm as well [26].

Algorithm 3 Davis-Putnam-Logemann-Loveland Algorithm

DPLL_Branch(ρ)

1. While F contains one or more pure literals, make pure literal moves: $\rho' \leftarrow PL^*(\rho)$
2. If ρ' is inconsistent return $unsatisfiable$.
3. While F contains one or more unit clauses, perform unit propagation:$\rho'' \leftarrow UP^*(\rho')$
4. If ρ'' is inconsistent return $unsatisfiable$.
5. Select *choice variable* for assignment: $i \leftarrow variable_select(\rho'')$
6. If $i = None$, return *satisfiable*.
7. $\rho''' \leftarrow \rho'' + [x_i \leftarrow \mathcal{F}]$
8. If $DPLL_Branch(\rho''') = unsatisfiable$
   (a) $\rho''' \leftarrow \rho'' + [x_i \leftarrow \mathcal{T}]$
   (b) If $DPLL_Branch(\rho''') = unsatisfiable$, return $unsatisfiable$.
9. Return *satisfiable.*

2.4.4 Variable Ordering and Value Ordering

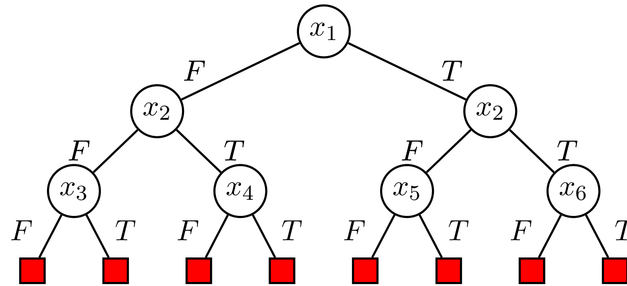
> *In some cases, whether or not the program could actually prove the validity of a given formula (without running out of fast access storage) depended on how one shuffled the punched-data deck before reading it into the assembler!* [25]

The above quote, taken from the original DPLL paper [25], shows how early the question of ordering arose in SAT research. It remains a dominant theme in all forms of backtracking search to this day. The most important ordering decision made by backtrack search solvers is which variable to assign next on a choice move. A secondary question is the value to be assigned to that variable.

2.4.4.1 Search Trees and Resolution Proofs

The search performed by backtracking with UP reflects a resolution proof. Consider the unsatisfiable formula (2.21), formed by adding two clauses to (2.20). Its conflict-pruned search tree is shown in Figure 2.5.

$$F = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_4) \wedge \\ (\bar{x}_1 \vee x_2 \vee x_5) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_5) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_6) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_6) \quad (2.21)$$



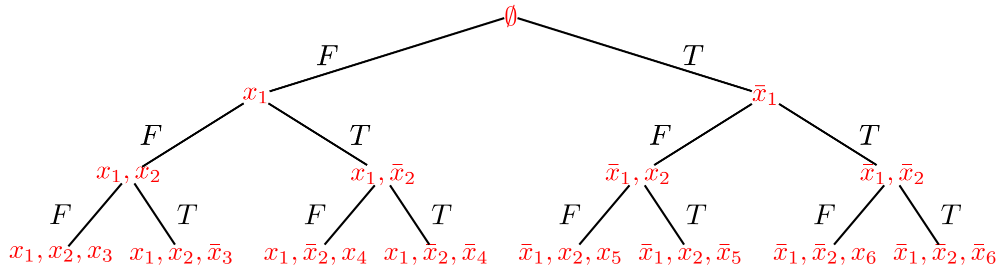
$$F = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_4) \wedge \\ (\bar{x}_1 \vee x_2 \vee x_5) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_5) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_6) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_6)$$

Figure 2.5: Conflict Pruning Backtrack Search Tree for Unsatisfiable Formula (2.21)

The following is a resolution refutation for (2.21), with the clauses listed in the left column and the right column showing the number for each clause and the clauses resolved to obtain it (the first eight are the original clauses).

$$
\begin{array}{lcl}
 & (x_1 \vee x_2 \vee x_3) & [1] \\
 & (x_1 \vee x_2 \vee \bar{x}_3) & [2] \\
 & (x_1 \vee \bar{x}_2 \vee x_4) & [3] \\
 & (x_1 \vee \bar{x}_2 \vee \bar{x}_4) & [4] \\
 & (\bar{x}_1 \vee x_2 \vee x_5) & [5] \\
 & (\bar{x}_1 \vee x_2 \vee \bar{x}_5) & [6] \\
 & (\bar{x}_1 \vee \bar{x}_2 \vee x_6) & [7] \\
 & (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_6) & [8] \\
\text{Step 1} & (x_1 \vee x_2) & [9] = [1\&2] \\
\text{Step 2} & (x_1 \vee \bar{x}_2) & [10] = [3\&4] \\
\text{Step 3} & (x_1) & [11] = [9\&10] \\
\text{Step 4} & (\bar{x}_1 \vee x_2) & [12] = [5\&6] \\
\text{Step 5} & (\bar{x}_1 \vee \bar{x}_2) & [13] = [7\&8] \\
\text{Step 6} & (\bar{x}_1) & [14] = [12\&13] \\
\text{Step 7} & \emptyset & [15] = [11\&14]
\end{array}
$$

If we show it as a tree (Figure 2.6) it mirrors the backtrack search tree shown in Figure 2.5. The order in which the variables are considered can have a profound effect on the size of the resolution refutation and the corresponding search tree.



$$F = (x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (x_1 \vee \bar{x}_2 \vee x_4) \wedge (x_1 \vee \bar{x}_2 \vee \bar{x}_4) \wedge (\bar{x}_1 \vee x_2 \vee x_5) \wedge (\bar{x}_1 \vee x_2 \vee \bar{x}_5) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee x_6) \wedge (\bar{x}_1 \vee \bar{x}_2 \vee \bar{x}_6)$$

Figure 2.6: Resolution Tree for Unsatisfiable Formula (2.21). (Space constraints oblige us to use commas instead of $\vee$ to indicate disjunction in this diagram.)

2.4.4.2 Value Ordering

It is important to note that having chosen a variable to assign, one must also choose a value. Since there are only two values, the only question is which to pick first, hence, *value ordering*. In the algorithms we have considered so far this choice is irrelevant if the instance is unsatisfiable, because we will be forced to explore both values sooner or later. We will see algorithms later where this is not the case (e.g. algorithms that learn new constraints during the search or that avoid exploring some alternative branches). For satisfiable instances there may be a difference, even in simple algorithms, since a poor choice of value for early variables may force a large search before the other value is considered.

Relatively little work has been done on value ordering for SAT (however, see [39, 95] for other constraint satisfaction problems). Some heuristics do both variable and value ordering, but the value part is rarely the prime consideration.

2.4.4.3 MOM Heuristic

The first widely recognized variable ordering heuristic is rooted in the recognition that unit propagation and quickly finding contradictions are both very important. Given that finding clauses of length 1 or 0 is useful, it is worth considering whether small clauses are, in general, desirable. There are two obvious obvious advantages. Firstly, the number of assignments ruled out by a given clause increases exponentially as clause size decreases. Secondly, small clauses, when reduced may lead to unit clauses which may be propagated.

The *maximum occurrences of minimum-length* (MOM) heuristic [28, 37, 89] seeks to generate many small clauses by preferring literals that already occur frequently in smallish clauses. More precisely, the heuristic considers only those clauses of minimal length, k_{min}, in the current formula. Given those clauses, it ranks the literals according to how often they occur amongst those clauses. Those literals that occur most frequently will, if made false, result in the largest number of immediately reduced clauses (ignoring the effects of unit propagation).

This approach is often generalized to consider clauses of all lengths, giving high weight to small clauses and low weight to large ones. We refer to such schemes as *weighted MOM*.

2.4.4.4 Balanced Heuristics

In the case of unsatisfiable instances, both branches of a choice variable will be explored by a backtracking search. This means that two subtrees must be evaluated and the sum of the sizes of those subtrees determine how expensive that variable will be to explore. If we can effectively estimate the sizes of the subtrees, we can use that information to order variables. The MOM heuristic attempts to capture this property by preferring variables that occur frequently in small clauses since they are likely to produce a contradiction quickly, and hence have small subtrees. However, MOM greedily selects its literal, ignoring the potential size of the subtree corresponding to that literal's negation. Balanced heuristics

come from the recognition that two imbalanced subtrees whose depths sum to d will be larger, in total, than two balanced subtrees whose depths also sum to d, or

$$\begin{aligned} 2^{\frac{d}{2}} + 2^{\frac{d}{2}} &\leq 2^{\frac{d}{2}+a} + 2^{\frac{d}{2}-a} \\ 2^{\frac{d}{2}+1} &\leq 2^{\frac{d}{2}+a} + 2^{\frac{d}{2}-a} \end{aligned}$$

where $0 \leq a \leq d/2$.

If the MOM score of a literal is an estimate of the depth of its subtree, then we can further order variables by preferring those variables with similar MOM scores for both literals. Almost all variable ordering heuristics in contemporary SAT solvers are based around this idea. We collectively refer to them as *balanced heuristics*, early examples of which include [89, 28]. The balancing principle is typically combined with some other heuristic (e.g. MOM, weighted MOM, UP heuristics, literal frequency, etc).

2.4.4.5 C-SAT

Dubois makes the shrewd observation that a solver can emphasize satisfiability or unsatisfiability in its design and heuristics [28]. He suggests that if a backtracking solver is expected to solve unsatisfiable instances, it should work to identify contradictions as quickly as possible, and that the idea of balancing subtrees is critical to its success. His solver C-SAT therefore employs a heuristic based on weighted MOM and balancing.[10]

2.4.4.6 Failed Literals

The MOM and similar heuristics attempt to predict the size of subtrees of a variable based on its role in the formula. This prediction is based on the current assignment only, and no search is performed to estimate subtree size. The next major advance in SAT variable ordering heuristics is a form of *forward-checking,* a class of methods in which limited forward search is used to estimate properties of the remaining search tree. *Failed literal detection* is a first obvious step in this direction.

When deciding which variable to assign next, some subset of the variables are assigned first true and then false to see whether either assignment leads to a contradiction. It can be thought of as a breadth-first search of depth one (but typically searching only a small subset of the variables). This check can be quite cheap, simply asserting the literal and propagating. If a literal is found to lead to a contradiction, it is called a *failed literal*. Otherwise, the checked variable is left unassigned and the shallow search simply moves on to the next literal.

When a failed literal is found, its negation is immediately tried (if it hasn't been already). If both fail, then the search must backtrack. Otherwise, the search has uncovered a forced move. Once the move has been taken, the search for failed literals may continue or a variable may be selected by heuristic to continue the search as usual. Thus, several

[10]This C-SAT solver is not to be confused with Gent and Walsh's local search CSAT solver [42].

variables may be assigned based on this shallow speculative search before continuing the search in the regular fashion.

While some solvers employed failed literal detection as a distinct step, its cost when used this way often outweighs its benefits unless it is limited to examining only a few variables [38]. However, failed literal detection is subsumed by stronger forward-checking heuristics that detect failed literals as a side-effect of computing their values.

2.4.4.7 UP Heuristics

Arguably the most successful ordering heuristics for backtracking SAT search (when no other tricks are used), *unit propagation heuristics* or *UP heuristics* extend the forward-checking idea by speculatively assigning variables one by one, propagating units, and then evaluating some heuristic like MOM on the resulting subinstance. This idea occurs quite far back in the literature [90, 121], but received renewed attention in the early to mid 90's with the POSIT [37, 38] and TABLEAU solvers [22], which strongly influenced each other. The UP heuristic essentially peaked with the highly successful SATZ solver [72, 73, 71], which was constructed as the result of a systematic study of UP heuristics, considering such questions as when it should be applied (e.g. choices near the root of the tree are very important compared to choices near the leaves), and to what extent (e.g. how many variables should be evaluated, and how deeply).

2.4.5 Randomization

A relatively recent innovation in backtracking search is adding randomization where possible (e.g. breaking ties in variable ordering heuristics) while preserving the completeness of the solver. This was demonstrated to be highly effective with SATZ-rand [49, 14], which demonstrated that randomization with *restarts* (periodically abandoning the current search and starting again) combats the heavy-tailed distribution over runtimes found in many SAT instances. We discuss the issue of restarts more deeply in the section on local search, specifically Section 2.5.1.

2.4.6 Data Structures

Most published work on SAT solvers provides few details on implementation. Key ideas are presented, but some important tricks are often omitted that can make a critical difference. For example, a naive implementation of the local searcher GSAT, based only on the original published paper, is very slow compared with one that incrementally updates certain values at every step.

Three notable exceptions are Freeman, who discusses implementation details in some detail in his thesis, even including a section with recommendations [37]; the SATO research [122, 125]; and the work on Chaff [85]. These address the question of data structures and memory handling explicitly, leading to important advances in the efficiency of backtrack solvers.

2.4.6.1 SATO

The SATO research gave rise to a series of interesting solvers, focusing for the most part on improving the data structures used in systematic solvers. SATO was originally derived from Discrimination-tree-based Davis-Putnam Prover (DDPP) [107], which uses a *trie* data structure and an expensive trie-merge operation. Early versions of SATO (1.0 and 2.0) use trie data structures to represent the current state of the clauses. While interesting, they are not the contribution that concerns us here and have not found widespread adoption, so we will not discuss the trie-based approaches here, leaving the interested reader to pursue the subject via the published reports [124, 125].

An unnamed algorithm, which we will call HEADTAIL, is described in [122]. It features a very fast UP mechanism which was later added to the SATO solvers. It represents the most significant improvement in backtracking SAT solver data structures in recent years. The algorithm uses *witness lists* (called *head/tail lists* by the authors). The idea of a witness stems from the fact that most of the work of a backtracking solver is devoted to unit propagation, so quickly detecting when clauses become unit is of tremendous value.

In a backtracking solver, we are concerned with three states of clauses: satisfied, unit (undecided with only one undecided literal), and *properly undecided* (undecided with at least two undecided literals). We coin the term *properly undecided* because a unit clause is not really undecided (we will simply force it to take the only satisfying value). Clauses never really become unsatisfied in a backtracking solver so we are not interested in the unsatisfied state. For the purposes of UP, we only need to take action when properly undecided clauses become unit.

If we know that a clause has two undecided literals, we can conclude that it is not unit. Therefore, before any assignments have been made, we select from each clause two *witness literals* (also called *watch literals*). Clearly, if neither of the variables corresponding to the two witnesses are assigned at a given point in the search, we know that the clause still has at least two undecided literals. This means the clause is either satisfied (by one of the non-witness literals) or is properly undecided. Either way, the clause is not particularly interesting at present. We must now consider what happens when assignments are made to witnessing variables.

For each variable, two *witness lists* are maintained, one for the clauses in which the variable occurs as a positive witness literal, and one for the clauses in which the variable occurs as a negative witness literal. When the variable is assigned, only one of its two lists need be traversed. Take the case where the variable is assigned positively. Every clause in the variable's positive witness list is now satisfied. We do not need to visit these clauses. We can determine the number of newly satisfied clauses readily from the size of the list, although even this is not necessary. Every clause in the variable's negative witness list now has an additional unsatisfied literal. We must visit each of these clauses to determine their state. Before the assignment, the clause was either satisfied or properly undecided. When visiting the clause, we examine the other literals in the clause.

- if a satisfied literal is encountered, the clause is satisfied and we can immediately stop

examining literals[11]

- if an undecided non-witness literal is encountered, we make it replace the old witness literal (now falsified) and it becomes a new witness to the clause. We update the lists of the old and new witness variables accordingly, and immediately stop examining literals

- if all literals are examined, and no satisfied or undecided non-witness literals are found, then only one undecided literal remains, the other witness literal. Since the clause has a single undecided literal and the rest are unsatisfied, it is unit so we can force the assignment and propagate. The witnesses are left unchanged.

The use of witness lists makes unit clause detection and propagation very fast and offers a huge performance gain. A secondary, but by no means insignificant, gain is that when backtracking, no changes have to made to the witnesses or witness lists. If variables are unassigned in the reverse order of their assignment, then the witnesses for clauses of any state will still be valid (the unassignments can only leave clauses satisfied or return them to properly undecided states). Typically, the unassignment procedure need only set the variable back to undecided.

It is important to note that the use of witness lists has some interesting side-effects. The solver never visits clauses when they become satisfied (although it may visit them when other variables in the clause are assigned) and only visits undecided clauses when one of the witnesses is assigned. At each assignment, many clauses are thus implicitly satisfied or reduced. Many variable-ordering heuristics use the length of properly undecided clauses, but a witness-based approach means that the only reduction we are guaranteed to observe is the reduction to unit clause. Therefore, many popular heuristics become unworkable. It is precisely because the witness scheme tracks so little information about the state of the clauses that it is so fast, and so it is evident that there are no free lunches to be had here. One possibility, unexplored to the best of our knowledge, would be to use more witnesses per clause. Three witnesses would allow one to detect reductions to binary clauses, which are used by several heuristics. The tradeoff with speed would have to be investigated.

Leaving witnesses in a particular state will clearly impact the future performance of the search. There are many possibilities here, systematically studied by Lynce and Marques-Silva [76]. The results are quite interesting, but the experiments were programmed using Java, which leaves the question of effective compiler optimization uncertain. Our own experimentation with some possibilities and extensive profiling of witness implementations shows that witness operations can account for as much as 80% or more of a solver's computation and that minor perturbations and optimization can have dramatic effects. It would be interesting to see a similar study using a statically compiled language.

While SATO solvers were substantially improved by the technique, it was its use in Chaff (see Section 2.4.8.3) that led to a major breakthrough in backtracking solvers. SATO

[11] We can optionally update the witnesses, making the satisfying literal a witness in place of the newly unsatisfied literal. This usually offers a substantial performance boost.

3.0 [123] used witness lists, along with conflict-directed backjumping, nogood learning with a parameterized maximum nogood length, and a dynamic variable ordering heuristic (earlier versions of SATO used a fixed variable ordering).

2.4.7 Backjumping

Forward-checking techniques improved SAT solvers substantially. Another dimension to their recent improvement is the adoption of *lookback* techniques, such as *backjumping* or *non-chronological backtracking*, which intelligently examine the decisions to determine which of them caused a contradiction and then use this information to backtrack. Naive backtracking always assumes the most recent decision must be changed whereas it may have been an earlier decision that caused the problem. Chen and van Beek have several theoretical results on backjumping for general CSPs [18]. We will confine our current discussion to the *conflict-directed backjumping* (CBJ) found in recent SAT solvers.

In order to diagnose the assignment that led to a conflict it is necessary to track the *reason* for each assignment. Choice moves have no reason because they are, by definition, moves we are not constrained to make. A forced move is made because a unit clause was detected, constraining us to select that literal. The cause is the prefix up to that point and the clause that was reduced to unit by that assignment. Since the solver already tracks the prefix, the only additional information we need to track is the clause that became unit, which we call the *reason* for the forced move. The runtime cost of storing this information is small compared to the cost of detecting the unit in the first place, and the memory requirement is only a constant increase to that required for the prefix.

When a conflict is discovered, two clauses have been reduced to unit clauses containing opposing literals. These two clauses are resolved to produce the conflict clause, c_{con}. We repeatedly apply the following analysis to the conflict clauses generated during the backjump:

- Let v_{max} be a variable in c_{con} whose literal has the largest choice level ("largest" means "most recent" – see Section 2.4.2.2). If multiple variables are in the same choice level, we consider the level's choice variable last. Ties between forced moves can be broken arbitrarily. v_{max} is the most recently assigned variable, so we must examine its role in the conflict:
  - If v_{max} is a choice variable, then v_{max} becomes the backjump point. We undo all assignments until we reach it and then flip its assignment. The c_{con} can be seen as an implicitly added constraint that forces the new assignment, and c_{con} is recorded as the reason for the new assignment to v_{max}. The backjump is now complete and the search continues, propagating any units resulting from the new assignment to v_{max}.
  - If v_{max} is not a choice variable, we resolve c_{con} with v_{max}'s reason, producing a new conflict clause free of v_{max}. We then continue with this new conflict clause as with the original (i.e. find a v_{max} and examine it).

This analysis produces a sequence of implicit constraints that eventually lead to one that forces a new value on a choice variable. If at any point a conflict clause is empty, then the instance has been proved unsatisfiable. Figure 2.7 shows an example of the process.

2.4.8 Clause Learning

The original Davis-Putnam solver constructed a resolution proof, storing all the clauses required. On the machines of the time, memory limited the method to very small instances, prompting the switch to DPLL, a backtracking search. Contemporary machines have plentiful memory and backtracking search has become CPU-bound, with only a small fraction of the memory required for the search. It is natural in this context to return to storing resolvents learned during the search. As observed earlier, backtracking search corresponds to a resolution proof, where each contradiction corresponds to a resolvent. Computing and storing these resolvents, known more generally as *nogoods* has become a common feature in CSP systems. SAT solvers using this approach are frequently described as *clause learning*.

Two key questions arise when clause learning is used; the solver must decide what clauses to store and how long they should be kept. We will first look at important work on deciding the latter question and then return to the former.

2.4.8.1 relsat

Theoretical results for *relevance-bounded learning* in constraint graphs from Bayardo and Schrag [6] led them to modify Tableau to include *nogood learning* and CBJ.[12] The resulting solver was called *relsat* [5].

Like Tableau, relsat uses a UP heuristic, the chief difference lying in the randomness added to various parts of the heuristic (see Algorithm 4). However, the key novel feature of relsat was the use of relevance-bounded learning to control the space complexity of nogood learning. For $relsat(i)$, a nogood is only kept as long as no more than i of the nogood's variables have changed assignment (i.e. become unassigned or switched values).

Controlling the space complexity serves two purposes; it reduces memory consumption and can also save time by preventing the solver from considering many, possibly useless, nogoods during the search.

Before relsat, the authors implemented generators for "exceptionally hard random instances" and tested them on Tableau enhanced with CBJ and *size-bounded learning*, referred to as $sizesat(i)$, where only nogoods with no more than i variables are kept [7]. The subsequent work on relsat clearly shows that relevance-bounded learning is more effective. The authors achieved their best results with $relsat(4)$.

2.4.8.2 GRASP

GRASP was an important step in recent solver history [79]. It uses CBJ and clause learning, nicely formulated as the analysis of a directed graph of variables with their reasons as

[12] A CBJ-enabled version Tableau called *ntab_back* has also been implemented by the Tableau team.

No assigments initially	$c_1 \vee c_2 \vee c_3 \vee c_4 \vee c_5$
	$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge (x_4 \vee x_5)$
$\{x_1 \leftarrow \mathcal{F}\}$ (no reason)	
	$(x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge (x_4 \vee x_5)$
$\{x_4 \leftarrow \mathcal{F}\}$ (no reason)	
	$(x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge (x_5)$
$\{x_5 \leftarrow \mathcal{T}\}\ (x_4 \vee x_5)$	
	$(x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge \mathcal{T}$
$\{x_2 \leftarrow \mathcal{F}\}$ (no reason)	
	$(x_3) \wedge (\bar{x}_3) \wedge \mathcal{T} \wedge \mathcal{T} \wedge \mathcal{T}$
contradiction	
resolve	
$(x_1 \vee x_2 \vee x_3) \wedge (x_1 \vee x_2 \vee \bar{x}_3)$	
conflict clause $c_6 = (x_1 \vee x_2)$	
$v_{max} = x_2,$ a choice	
backtrack to x_2 choice	$(x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3) \wedge \mathcal{T} \wedge (x_2)$
$\{x_2 \leftarrow \mathcal{T}\}\ (x_1 \vee x_2)$	
	$\mathcal{T} \wedge \mathcal{T} \wedge (\bar{x}_3) \wedge (x_3) \wedge \mathcal{T} \wedge \mathcal{T}$
contradiction	
resolve $(\bar{x}_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee x_3)$	
conflict clause $c_7 = (\bar{x}_2)$	
$v_{max} = x_2$ is not a choice	
resolve $(\bar{x}_2) \wedge (x_1 \vee x_2)$	
$c_8 = (x_1)$	
$v_{max} = x_1,$ a choice	
backtrack to x_1 choice	$(x_1 \vee x_2 \vee x_3) \wedge (x_2 \vee \bar{x}_3) \wedge (\bar{x}_2 \vee \bar{x}_3) \wedge$ $(\bar{x}_2 \vee x_3) \wedge \mathcal{T} \wedge (x_1 \vee x_2) \wedge (\bar{x}_2) \wedge (x_1)$
$\{x_1 \leftarrow \mathcal{T}\}\ (x_1)$	we have backjumped to x_1 skipping x_4
	...

Figure 2.7: An example of backjumping. The right column shows the formula reduced by the current assignment at each step. The left column explains each step. Variable assignments are shown along with their reason.

Algorithm 4 relsat Variable Selection Heuristic

relsat_var_select()

1. If all variables are assigned, return $None$.
2. If there are no binary clauses, return a variable at random
3. For each variable, $x_i \in \mathbf{x}$, compute $b(x_i) = binpos(x_i)binneg(x_i) + binpos(x_i) + binneg(x_i)$
4. Gather all variables within 20% of the highest $b(x_i)$ into a candidate set, $\mathbf{C}_1$.
5. If there are more than 10 candidates in $\mathbf{C}_1$ then randomly select 10 without replacement to form $\mathbf{C}_2$, otherwise, $\mathbf{C}_2 \leftarrow \mathbf{C}_1$.
6. If $\mathbf{C}_2$ contains only one candidate, return it.
7. For each variable, $x_j \in \mathbf{C}_2$,
   (a) Compute $u(x_i)$ as the total number of variables assigned by unit propagating x_i and $\bar{x}_i$.
   (b) If one of the two is a failed literal, it should be propagated immediately.
   (c) If both fail (contradiction), return $unsatisfiable$.
   (d) If neither fail, add x_j to $\mathbf{C}_3$.
8. Gather all variables in $\mathbf{C}_3$ that are within 10% of the highest $u(x_j)$ into a candidate set, $\mathbf{C}_4$.
9. Select a variable randomly from $\mathbf{C}_4$ and return it.

edges. In these *implication graph*s they identify *unique implication points* (*UIP*s), single variables that can single-handedly account for a conflict (there may be several UIPs for a given conflict). Backjumping consists of backtracking to a UIP.

One way to think of backjumping is that it temporarily adds a clause which forces the UIP to change value. Such a clause would prevent the same conflict from arising again. Clause learning in this context simply consists of adding such clauses to the formula instead of just changing the value. A detailed discussion of implication graphs and UIPs are beyond the scope of this thesis, but the analysis closely reflects the CBJ algorithm described above. To control the size of the formula, clauses greater than some fixed length, specified by a parameter, are discarded as soon as they become properly undecided.

2.4.8.3 Chaff

Chaff, often called *zChaff*, is a solver built by refining the witness scheme from SATO, combining it with the clause learning/CBJ from GRASP plus relevance-bounded learning, and carefully engineering the solver for high performance [85, 126]. Chaff also features restarting and, optionally, randomization in its ordering heuristic. The randomization can be optional (it is disabled by default) because Chaff's heuristic changes as clauses are learned, so restarting while keeping learned clauses results in a different search.

Without contributing radically new ideas, Chaff's fusion of some of the best recent methods produced a new level of performance, solving large structured instances and dominating its contemporaries on all but random 3-SAT (where Chaff performs quite poorly).

The Chaff researchers studied the costs and benefits of searching the various possible UIPs in the implication graph in the hopes of obtaining better resolvents. They concluded that the *1-UIP cut*, basing the resolvent around the UIP closest to the conflict (and therefore the first one found) gave the best performance. Chaff uses the 1-UIP cut.

Recent derivatives of Chaff, such as BerkMin [45] adjust the variable heuristics and deletion policies to improve the performance, but what remains interesting about this family of solvers is that they use relatively weak variable heuristics based on literal counts. All the expensive UP heuristics are replaced by fast heuristics, only occasionally updated. Indeed, UP heuristics are not feasible under the witness scheme since the reduced clauses are not explicitly tracked. Clause learning and fast unit propagation compensate for these weak heuristics, but it seems likely that CBJ is also an important factor, since it is no longer true that both subtrees of every choice variable must be explored. This means that unbalanced subtrees, or poor estimates of their size, are less likely to be of critical importance.

2.4.9 Summary

While progress has been made beyond Chaff in a variety of novel directions, we have explained enough of the backtrack search research to support our own work and so leave the subject for now. The table in Figure 2.8 may serve as a useful guide to the features of the solvers we have discussed.

Solver	Lookahead				Lookback		Preprocess		Data		Year
	Bal	UPH	FLD	PLD	CBJ	CL	LR	LVE	Wit	Tri	
DPLL				Y							’62
TABLEAU	Y	Y	I								’93-’95
C-SAT	Y										’93
POSIT	Y	Y	Y	Y				Y			’93-95
SATO 2.x									Y	Y	’96
SATO 3.0	Y				Y	Y			Y	Y	’97
SATZ	Y	Y	I				Y				’97
relsat	Y	Y	I		Y	Y					’96
GRASP					Y	Y					’99
CHAFF					Y	Y			Y		’01
BerkMin					Y	Y			Y		’02

Y = has feature, I = has feature implicitly (as a result of some other feature)

Bal Balanced variable ordering heuristic.

UPH UP heuristic.

FLD Failed literal detection.

PLD Pure literal detection.

CBJ Conflict-directed backjumping.

CL Clause (nogood) learning.

LR Limited resolution which adds resolvents to the formula at beginning of search .

LVE Limited DP-style variable elimination at the beginning of search.

Wit Witness mechanism.

Tri Trie data structure.

Figure 2.8: Summary of Backtracking Solvers and their features

2.5 Local Search Solvers

The other major approach to solving SAT instances uses *local search* solvers. These are model finding solvers that typically use *full assignments* (i.e. assignments to all of the variables). At step t with a current assignment, $\mathbf{a}^{(t)}$, they explore a set of neighbouring assignments generated by a *neighbourhood function*, $N(\mathbf{a}^{(t)})$. The neighbours are evaluated according to some *objective function*, $g(\mathbf{a})$, and a neighbour is selected to become the assignment for the next step in the search, $\mathbf{a}^{(t+1)}$.

The procedure runs until a satisfying assignment is found or some *search termination condition* is met (e.g. exceeded maximum time or exceeded maximum steps). A generic template for local search solvers is given in Algorithm 5.

Algorithm 5 Generic Local Search Algorithm

localSearch()

1. Pick initial full assignment, $\mathbf{a}^{(0)}$
2. Set step counter, $t \leftarrow 0$
3. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success (satisfiable)
4. If a *search termination condition* has been met, exit with failure (undecided)
5. Construct the set of neighbouring assignments, $N(\mathbf{a}^{(t)})$.
6. Evaluate each neighbouring assignment, $\mathbf{a}$, according to the objective function, $g(\mathbf{a})$
7. Select a neighbouring assignment, $\mathbf{a}^{(t+1)}$, based on the evaluation.
8. Advance the step counter, $t \leftarrow t + 1$
9. Goto step 3

Since the search is not systematic, local search solvers cannot determine when they have considered all assignments and thus cannot decide that an instance is unsatisfiable. Such algorithms are said to be *incomplete*.

2.5.0.1 Local Minima vs. Plateaus

In the interests of clarity, we will carefully define a few notions about an assignment's relationship to its neighbourhood that are often used with different meanings in the literature. We will adopt the convention of minimizing objective functions throughout this work.

- The search is said to be at a *plateau* when the current objective value is equal to that of all neighbours, $g(\mathbf{a}) = g(\mathbf{a}'), \forall \mathbf{a}' \in N(\mathbf{a})$.

- The search is said to be at a *strict local minimum* when the current objective value is less than that of all neighbours, $g(\mathbf{a}) < g(\mathbf{a}'), \forall \mathbf{a}' \in N(\mathbf{a})$.

- The search is said to be at a *non-strict local minimum* when the current objective value is less than or equal to that of all neighbours, $g(\mathbf{a}) \leq g(\mathbf{a}'), \forall \mathbf{a}' \in N(\mathbf{a})$.

2.5.1 Random Restarts

Most of these algorithms have stochastic components and generally use a random starting point. One common improvement to a local search algorithm is the use of *restarts* (also called *tries*). Under this simple mechanism, if a search termination condition is met without finding a solution, the algorithm selects a new starting point and starts again (see Algorithm 6). Some *restart termination condition* is fixed for the procedure (e.g. exceeded maximum number of restarts or exceeded maximum time).

This means that any local search algorithm typically has a restart parameter (usually a maximum number of search steps) and tunings for this parameter are algorithm dependent. An extra parameter to tune makes empirical studies of local searchers arduous, but since any local search algorithm may be improved by restarting, it is necessary to consider this issue.

Algorithm 6 Restart Strategy

---

restart()

1. Initialize restart counter, $r \leftarrow 0$
2. If localSearch() succeeds, exit with success (satisfiable)
3. $r \leftarrow r + 1$
4. If a *restart termination condition* has been met, exit with failure (undecided)
5. Goto step 2

2.5.1.1 Optimal Restarting

Parkes and Walser address this problem directly with their *Retrospective Variation of MAXFLIPS* (RPV) [87]. In brief, they note that by running many experiments with some fixed cutoff, t_c, we can estimate the effects of using smaller cutoffs, $t < t_c$. In particular, we can compute the *estimated expected search steps* (EESS) for a given cutoff, t:

$$EESS_t = E(T_t) = \frac{t}{P(T \leq t)} - [t - E(T|T \leq t)] \tag{2.22}$$

where

- T is the number of search steps for a successful run
- T_t is the number of steps required given a cutoff of t
- $E(T_t)$ is the expectation of T_t.
- $P(T \leq t)$ is the probability that $T \leq t$.
- $E(T|T \leq t)$ is the expectation of T given that $T \leq t$.

We are particularly interested in the "optimal" cutoff value, t^*, that minimizes the expected number of steps:

$$t^* = \arg \min_{t:0<t\leq\infty} E(T_t) \tag{2.23}$$

and the corresponding optimal EESS, $E(T_{t*})$. This value is the *estimated expected search steps under an optimal restart policy* (OEESS) and is computed and reported in our experiments in lieu of searching the parameter space of cutoffs. Typically, we report the average search steps along with OEESS, but regard the latter as the more informative value.

Considerable research has been done on the distribution of runtimes for stochastic search algorithms on various SAT and CSP problems and on the principles underlying restarting stochastic search algorithms (e.g. [75]). In particular, Gomes et al. have noted that these distributions are frequently "heavy-tailed" and may have an infinite mean [49, 48]. This means that, on a given instance and using a given stochastic algorithm, runs of many different length, from short to extremely long, will be encountered. They show that randomizing search algorithms and restarting can offer substantial performance improvements by virtue of avoiding the extreme values in these heavy tails, in hopes of hitting more reasonable runs.

Other research examines features of the problems and the related distribution of runtimes more closely in order to predict hardness [61, 65][69]. Finally, Boyan and Moore's STAGE [12] algorithm offers an interesting alternative to random restarts by learning an objective function for walking to new starting points and then resuming the greedy descent.

Since any restarting strategy is likely to offer improvements to all algorithms, we do not consider them further and rely on OEESS to make sure comparisons are fair.[13] Note that when runtimes are used, we will often report OEERT, the *estimated expected runtime under an optimal restart policy* . This is simply the OEESS times the average runtime per search step observed during the search. Since variance in the runtime of individual steps is typically low, this measure is quite reasonable.

[13]A number of runs with some high cutoff are used to compute the OEESS. If a large proportion of these runs complete without finding a satisfying assignment then the corresponding estimates will be less reliable. However, this only arises in situations where the algorithm is clearly a very poor performer, so a quick check of its failure rate is sufficient to dismiss a potentially misleading OEESS.

2.5.1.2 Effects of Restarts

It can be shown that imposing an ideal restart value t^* on the random search time T yields an improvement in expected time for most natural search distributions. To see this, note that for any random variable T we have

$$\mathrm{E}(T) \;=\; \mathrm{E}(T|T>t)\,\mathrm{P}(T>t) \;+\; \mathrm{E}(T|T\leq t)\,\mathrm{P}(T\leq t) \tag{2.24}$$

From (2.22) we know that the expected search time when using a random restart after t steps is

$$\begin{aligned}\mathrm{E}(T_t) &= \frac{t}{\mathrm{P}(T\leq t)} - [t - \mathrm{E}(T|T\leq t)] \\ &= t\left(\frac{1}{\mathrm{P}(T\leq t)} - 1\right) + \mathrm{E}(T|T\leq t) \\ &= t\,\frac{\mathrm{P}(T>t)}{\mathrm{P}(T\leq t)} + \mathrm{E}(T|T\leq t) \end{aligned}\tag{2.25}$$

To determine whether (2.25) offers an improvement over (2.24) first note that for an exponential random variable T (i.e. such that $\mathrm{P}(T>x) = e^{-x/\mu}$ for all $x > 0$) we actually have $\mathrm{E}(T) = \mathrm{E}(T_t)$ for every cutoff value $t > 0$. This follows from the "memoryless property" of exponential random variables which states that $\mathrm{P}(T>t+x|T>t) = \mathrm{P}(T>x)$ for all $t > 0$ and $x > 0$ [93], and immediately implies that

$$\mathrm{E}(T|T>t) \;=\; t + \mathrm{E}(T) \tag{2.26}$$

for all $t > 0$. Thus

$$\begin{aligned}\mathrm{E}(T) &= \mathrm{E}(T|T>t)\,\mathrm{P}(T>t) \;+\; \mathrm{E}(T|T\leq t)\,\mathrm{P}(T\leq t) \\ &= t\,\mathrm{P}(T>t) \;+\; \mathrm{E}(T)\,\mathrm{P}(T>t) \;+\; \mathrm{E}(T|T\leq t)\,\mathrm{P}(T\leq t)\end{aligned}\tag{2.27}$$

from the memoryless property (2.26), and therefore for exponential random variables we always have

$$\mathrm{E}(T)\,\mathrm{P}(T\leq t) \;=\; t\,\mathrm{P}(T>t) \;+\; \mathrm{E}(T|T\leq t)\,\mathrm{P}(T\leq t)$$

and hence

$$\begin{aligned}\mathrm{E}(T) &= t\frac{\mathrm{P}(T>t)}{\mathrm{P}(T\leq t)} \;+\; \mathrm{E}(T|T\leq t) \\ &= \mathrm{E}(T_t)\end{aligned}\tag{2.28}$$

for all $t > 0$, as stated.

So imposing a random restart after a cutoff value t does not affect the expectation of any exponential random variable T. It might therefore appear that random restarts may not offer a useful improvement in general. However, the equality (2.28) only holds because of

the memoryless property (2.26), and it turns out that this property holds for all $t > 0$ *only* for exponential random variables [93].

However, there is substantial evidence that the runtime distributions for heuristic search algorithms exhibit a heavy-tailed behavior on difficult constraint satisfaction problems [48]. The tail of the runtime distribution $\mathrm{P}(T > x)$ is not exponential, but instead is typically a much slower converging function such as a power law $\mathrm{P}(T > x) = Cx^{-\alpha}$ for $\alpha > 0$, $C > 0$ (where the tails become heavier for smaller α) [48]. A simple example of a power law distribution is a Pareto density $p(x) = \alpha(1 + x)^{-\alpha-1}$, $x > 0$, $\alpha > 0$, which defines a random variable T such that $\mathrm{P}(T > x) = (1 + x)^{-\alpha}$. The kth moments of power law distributions are only defined for $k < \alpha$, and in particular their expected values become infinite for $\alpha \leq 1$.

For heavy-tailed distributions in general it is easy to demonstrate that a random restart strategy will always yield a reduction in expected runtime. To see this, first consider the case of a heavy-tailed distribution with a finite expected runtime; that is, a power law distribution such that $\alpha > 1$. An interesting property of such a distribution is that it is not memoryless. In fact, the additional expected runtime actually *increases* given that an early solution has not been found:

$$\mathrm{E}(T|T > t) \quad > \quad t + \mathrm{E}(T) \tag{2.29}$$

for $t > 0$, which is in direct contrast to (2.26). From this property it immediately follows that a random restart strategy yields an improvement in expected solution time, since

$$\begin{aligned}
\mathrm{E}(T) &= \mathrm{E}(T|T > t)\,\mathrm{P}(T > t) \;+\; \mathrm{E}(T|T \leq t)\,\mathrm{P}(T \leq t) \\
&> t\,\mathrm{P}(T > t) \;+\; \mathrm{E}(T)\,\mathrm{P}(T > t) \;+\; \mathrm{E}(T|T \leq t)\,\mathrm{P}(T \leq t) \\
&= \mathrm{E}(T_t)
\end{aligned}$$

For example, for a Pareto distribution with $\alpha > 1$ we have $\mathrm{E}(T|T > f) = t + \mathrm{E}(T) + t\mathrm{E}(T)$ for all $t > 0$ and hence immediately obtain $\mathrm{E}(T_t) < \mathrm{E}(T)$.

For the more extreme case of a heavy-tailed distribution with an *infinite* expected value (i.e. when $\alpha \leq 1$) it is obvious that a random restart strategy significantly improves runtime, since $\mathrm{E}(T_t)$ is clearly finite (2.25) for any t such that $\mathrm{P}(T \leq t) > 0$ by inspection.

Therefore, in general, using a random restart strategy with an optimal cutoff value like (2.23), should improve expected runtime, since if (2.29) is satisfied for some t we will immediately obtain $\mathrm{E}(T_{t^*}) < \mathrm{E}(T)$, and otherwise, if no such value of t exists, we obtain $\mathrm{E}(T_{t^*}) = \mathrm{E}(T)$ (and hence cause no harm). Therefore, under any circumstance, employing a random restart strategy with an ideal cutoff value t^* is never a losing strategy.[14]

[14]This assumes that restarts are "free" or have negligible cost. Typically, a restart involves a completely new assignment to the variables, so the usual, efficient single step updates used by local searchers do not apply. Initialization to an initial assignment is typically linear in the number of literals in the formula, and restarting to a fresh assignment can usually be done in n or fewer regular local search steps. However, for any problem of interesting size these costs are typically very small compared to the cost of searching up to the optimal cutoff, so we treat them as negligible.

2.5.2 Queens and Satisfiability

The first extensive body of work addressing the use of local search for satisfiability comes from Jun Gu in the late 80's and early 90's [50, 51]. Gu had turned the classic *n*-queens constraint satisfaction problem into an unconstrained optimization problem by using the number of conflicting pairs of queens as an objective function. He found local search was highly effective on the queens instances and adapted the technique for satisfiability using the number of unsatisfied clauses as an objective function.

Gu created a variety of algorithms (the SAT1 family) incorporating many ideas that have been seen in subsequent solvers (e.g. resolution, noise, tabu search and backtracking hybrids). Some analysis of their per step time complexity was done but the analysis is of comparatively naive implementations. This may explain why the subsequent work on GSAT, which has a fast and relatively sophisticated implementation, attracted more attention than the SAT1 solvers.

2.5.3 GSAT

The first prominent local search algorithm for SAT came in 1992 and was called GSAT [106]. The algorithm is extremely simple (see Algorithm 7) but it succeeded in outperforming the runtime of the dominant DPLL method in 1992, especially in the class of hard, random instances (see Section 2.2.4).

Although simple, GSAT has a couple of features critical to its success. One is that it allows "backwards" or "damaging" moves. It will always move to some neighbouring assignment, even if the best available is worse than the current assignment. This allows GSAT to escape some local minima. GSAT's success also relied heavily on the use of restarts, selecting a new random starting point each time it restarted. Figure 2.9 shows the original results from [106] as obtained in 1992 when GSAT and DPLL were compared on HR instances of increasing size.

We have generated similar but more up-to-date results using our own implementation of GSAT and a version of the DPLL algorithm implemented on top of the highly efficient zChaff UP engine. A plot is shown in Figure 2.10. Note that this plot shows log expected time in order to emphasize the advantage GSAT has even now. The plot also includes results for DPLL using the zChaff variable ordering heuristic (instead of default, lexicographical ordering used by DPLL).

2.5.4 HSAT

One of the earliest improvements on GSAT was the result of a systematic study of the features of GSAT [43]. Gent and Walsh nicely factored the various features of GSAT (greediness, randomness, sideways moves, etc.) and tried many combinations on hard random instances. Their conclusions are quite interesting, showing that most of these features offer little or no improvement. Somewhat orthogonally, they propose a new algorithm, HSAT, which is similar to GSAT, but when presented with a choice of variables, it prefers the least

Algorithm 7 GSAT

GSAT()

1. Randomly generate initial full assignment, $\mathbf{a}^{(0)}$
2. Set step counter, $t \leftarrow 0$
3. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success
4. If the termination criteria have been met, exit with failure
5. Neighbourhood, $N(\mathbf{a})$ has n assignments, each obtained by flipping a single variable in $\mathbf{a}^{(t)}$
6. Evaluate assignments in $N(\mathbf{a})$ according to the function,
   $g_{unsat}(\mathbf{a}) = unsat(\mathbf{a})$
   (this can be done more efficiently by computing $loss(\mathbf{a}^{(t)}, \mathbf{a})$)
7. Choose $\mathbf{a}^{(t+1)}$ to be the neighbour with the smallest value. If there are "ties" (several assignments with the same value), choose randomly among the smallest.
8. Advance the step counter, $t \leftarrow t + 1$
9. Goto step 3

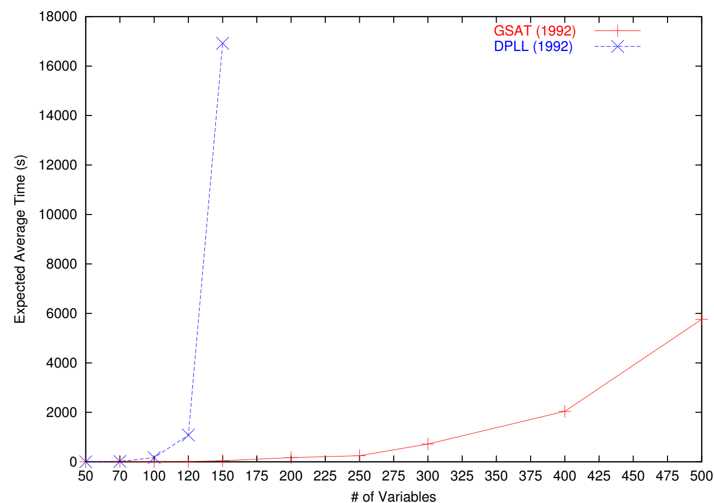


Figure 2.9: Original (1992) GSAT vs. DPLL results [106]

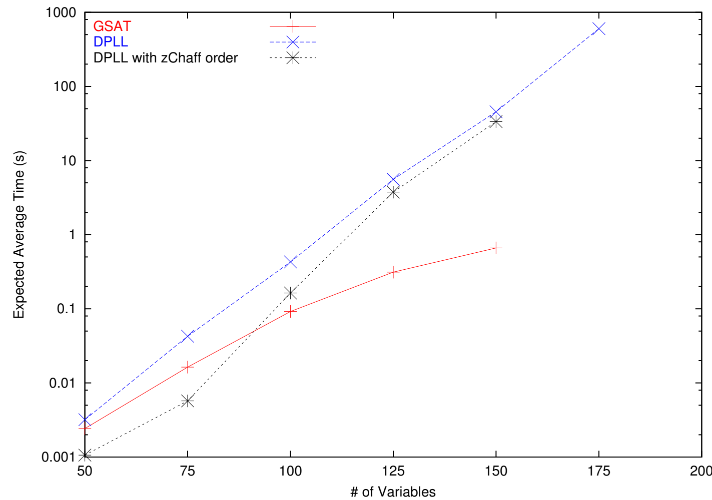


Figure 2.10: Current GSAT vs. DPLL results, log expected time for HR instances with varying # of variables

recently flipped variable. This policy of avoiding revisiting choices is essentially a form of Tabu search [44].

2.5.5 WalkSAT

Following the success of GSAT, the next year produced its more notable successor, WalkSAT [104][105], a solver which is still competitive on many instance classes. WalkSAT partially arose from the observation that a *random walk* provides a provably quadratic-time, randomized algorithm for 2-SAT [86]. However, this alone is insufficient to account for the large improvement in performance over GSAT. Three factors are critical:

1. Random walk as a means of escaping "traps".
2. Limiting neighbourhoods to variables that occur in unsatisfied clauses.
3. Subtle variations on the GSAT objective function.

The first factor has the loose theoretical motivation mentioned above but those results only hold for 2-SAT. Essentially, mixing random walk with greedy search prevents WalkSAT from becoming "trapped" in a region of the search space. Such random walk features appear in many subsequent local search solvers and are frequently referred to as *noisy strategies* (associated parameters controlling the amount of randomness are often called *noise* parameters).

The second factor allows for dramatic speedup over GSAT. Instead of considering all n variables at every search step, WalkSAT explores only the variables in a single clause at each

step. For k-CNF instances, this reduces the complexity from GSAT's $O(n)$ to $O(k)$. Some clever data structures are required to realize these time complexities but the implementation is not arduous and the real speedup is substantial.

The third and final factor that gives WalkSAT its edge is that it does not use the same objective function as GSAT. Where GSAT uses $loss(\mathbf{a}^{(t)}, \mathbf{a})$, WalkSAT uses $break(\mathbf{a}^{(t)}, \mathbf{a})$. This seems counter-intuitive, because minimizing $loss()$ is the obvious greedy choice. However, minimizing $break()$ is more conservative, choosing the step that will do the least damage to the current set of satisfied clauses. See Algorithm 8 for the exact procedure.

Algorithm 8 WalkSAT

WalkSAT()

1. Randomly generate initial full assignment, $\mathbf{a}^{(0)}$
2. Set step counter, $t \leftarrow 0$
3. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success
4. If the termination criteria have been met, exit with failure
5. Randomly select an unsatisfied clause, c_u
6. Neighbourhood $N(\mathbf{a})$ has k_u assignments, each obtained by flipping a single variable in c_u
7. Evaluate assignments in $N(\mathbf{a})$ according to the function,
   $g(\mathbf{a}^{(t)}, \mathbf{a}) = break(\mathbf{a}^{(t)}, \mathbf{a})$
8. If any neighbour creates zero new unsatisfied clauses ($g(\mathbf{a}^{(t)}, \mathbf{a}) = 0$), then goto step 10
9. Decide, with probability p_{walk}, whether to execute a "random walk move":
   a) Randomly select a variable in c_u and flip it to produce $\mathbf{a}^{(t+1)}$
   b) Go to step 11
10. Choose $\mathbf{a}^{(t+1)}$ to be the neighbour with the smallest value. If there are "ties" (several assignments with the same value), choose randomly among the smallest
11. Advance the step counter, $t \leftarrow t + 1$
12. Goto step 3

WalkSAT was not discovered easily and the apparently trivial details really matter. To illustrate, let us examine two close relatives of WalkSAT described in [80], WalkSAT-G and WalkSAT-B.

WalkSAT-B

WalkSAT-B is identical to WalkSAT, but does not automatically take a "zero break" move ($break(\mathbf{a}^{(t)}, \mathbf{a}) = 0$) if one is possible. Instead, it always randomly selects between random walk and a greedy move (see Algorithm 9). Note that this difference means that WalkSAT is more expensive than WalkSAT-B, since it must always evaluate the variables in the unsatisfied clause, whereas WalkSAT-B only needs to evaluate them if it has already decided not to make a random move. Nonetheless, WalkSAT wins in actual runtime because it typically requires fewer steps to find a solution.

Algorithm 9 WalkSAT-B

WalkSAT-B()

1. Randomly generate initial full assignment, $\mathbf{a}^{(0)}$
2. Set step counter, $t \leftarrow 0$
3. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success
4. If the termination criteria have been met, exit with failure
5. Randomly select an unsatisfied clause, c_u
6. Decide, with probability p_{walk}, whether to execute a "random walk move":
   a) Randomly select a variable in c_u and flip it to produce $\mathbf{a}^{(t+1)}$
   b) Go to step 9
7. Neighbourhood $N(\mathbf{a})$ has k_u assignments, each obtained by flipping a single variable in c_u
8. Evaluate assignments in $N(\mathbf{a})$ according to the function,
   $g(\mathbf{a}^{(t)}, \mathbf{a}) = break(\mathbf{a}^{(t)}, \mathbf{a})$
9. Choose $\mathbf{a}^{(t+1)}$ to be the neighbour with the smallest value. If there are "ties" (several assignments with the same value), choose randomly among the smallest
10. Advance the step counter, $t \leftarrow t + 1$
11. Goto step 3

WalkSAT-G

WalkSAT-G is identical to WalkSAT, but uses the $loss()$ objective function instead of $break()$ (see Algorithm 10). As noted above, using $loss()$ seems intuitively sound but is actually much less effective.

Algorithm 10 WalkSAT-G

WalkSAT-G()

1. Randomly generate initial full assignment, $\mathbf{a}^{(0)}$
2. Set step counter, $t \leftarrow 0$
3. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success
4. If the termination criteria have been met, exit with failure
5. Randomly select an unsatisfied clause, c_u
6. Decide, with probability p_{walk}, whether to execute a "random walk move":
   a) Randomly select a variable in c_u and flip it to produce $\mathbf{a}^{(t+1)}$
   b) Go to step 9
7. Neighbourhood $N(\mathbf{a})$ has k_u assignments, each obtained by flipping a single variable in c_u
8. Evaluate assignments in $N(\mathbf{a})$ according to the function,
   $g(\mathbf{a}) = loss(\mathbf{a})$
9. Choose $\mathbf{a}^{(t+1)}$ to be the neighbour with the smallest value. If there are "ties" (several assignments with the same value), choose randomly among the smallest
10. Advance the step counter, $t \leftarrow t + 1$
11. Goto step 3

2.5.6 Novelty and R-Novelty

Novelty and R-Novelty [80] represent the most recent major improvement to the WalkSAT family of solvers. These solvers were the product of extensive search. The authors state that they explored more than 50 variations of WalkSAT, optimally tuning these algorithms by means of two "invariant" metrics based on the run-time behaviour of the objective function (see Chapter 3 for more information on these metrics).

Novelty is similar to WalkSAT. The chief difference is that it tracks the step at which each variable was last flipped. In the event that two flip choices have equal objective value, the least recently flipped variable is preferred. However, if the "best" variable was flipped in the previous step, there is a probability, $p_{secondbest}$, that the second best will be flipped instead (see Algorithm 11). Note that $p_{secondbest}$ is a parameter. It should also be noted that Novelty uses $unsat()$ as an objective function, rather than $break()$.

This is essentially a Tabu strategy designed to prevent Novelty from frequently revisiting recent assignments. R-Novelty is similar but uses a more complex comparison between the best and second best choices when the best was most recently flipped (see Algorithm 12, step 10).

In 1999, Hoos showed that it was possible to trap Novelty in part of the search space of certain instances [58]. He proposed a simple solution to this problem; at each step there is a small probability of flipping a variable at random. This adds a random walk component to the algorithm that prevents it from becoming trapped because there is a guaranteed, non-zero probability of reaching any point in the search space (i.e. it cannot be trapped indefinitely in one region of the search space). Hoos calls this property *probabilistic asymptotic completeness* (PAC) [58][23]. Adding random walk to any solver gives this property trivially. Random restarts also offer a way to add this feature to a solver.

The resulting algorithms are called Novelty+ and R-Novelty+. While they typically exhibit better behaviour by preventing the local search from becoming trapped, the random walk sometimes degrades performance by making bad moves.

2.5.7 Simulated Annealing

The success of GSAT sparked interest in other stochastic local search methods for SAT. A natural choice is *simulated annealing* [67], which was investigated in 1993 by Selman et al. [103]. They were unable to find an annealing schedule that performed better than GSAT (with random walk). Spears later presented evidence that forms of simulated annealing (SA) were comparable or superior to GSAT (without random walk) [108]. Beringer et al. revisited the issue, pointing out that the earlier studies each had problems and performing their own experiments [9]. Their conclusions seem to indicate that whereas SA could offer fewer search steps, the steps would be more expensive because SA could not use the same clever computational tricks as GSAT. Our own brief experiments with SA showed that it never compared favourably with any solver on par with WalkSAT or better.

Algorithm 11 Novelty

Novelty()

1. Randomly generate initial full assignment, $\mathbf{a}^{(0)}$
2. Initialize a vector $\mathbf{lastflip}$: $\mathbf{lastflip}_i \leftarrow 0,\ 1 \leq i \leq n$
3. Set step counter, $t \leftarrow 0$
4. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success
5. If the termination criteria have been met, exit with failure
6. Randomly select an unsatisfied clause, c_u
7. Neighbourhood $N(\mathbf{a})$ has k_u assignments, each obtained by flipping a single variable in c_u
8. Evaluate assignments in $N(\mathbf{a})$ according to the function, $g_{unsat}(\mathbf{a}) = unsat(\mathbf{a})$ (this can be done more efficiently by computing $loss(\mathbf{a}^{(t)}, \mathbf{a})$)
9. Choose $\mathbf{a}'$ and $\mathbf{a}''$ to be the neighbours best and second best values respectively. Where two neighbours have equal objective value, the least recently flipped is preferred ($\mathbf{lastflip}_i < \mathbf{lastflip}_j$). Absolute ties are broken randomly.
10. Let $best = \mathbf{FLIPPED}(\mathbf{a}^{(t)}, \mathbf{a}')$ (the variable flipped for the best neighbour).
11. If $lastflip_{best} = t$ then with probability $p_{secondbest}$ set $\mathbf{a}^{(t+1)} \leftarrow \mathbf{a}''$,
12. Otherwise, set $\mathbf{a}^{(t+1)} \leftarrow \mathbf{a}'$.
13. Let $v = \mathbf{FLIPPED}(\mathbf{a}^{(t)}, \mathbf{a}^{(t+1)})$ and $\mathbf{lastflip}_v \leftarrow t + 1$
14. Advance the step counter, $t \leftarrow t + 1$
15. Goto step 3

Algorithm 12 R-Novelty

R-Novelty()

1. Randomly generate initial full assignment, $\mathbf{a}^{(0)}$
2. Initialize a vector **lastflip**: $\mathbf{lastflip}_i \leftarrow 0,\ 1 \leq i \leq n$
3. Set step counter, $t \leftarrow 0$
4. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success
5. If the termination criteria have been met, exit with failure
6. Randomly select an unsatisfied clause, c_u
7. Neighbourhood $N(\mathbf{a})$ has k_u assignments, each obtained by flipping a single variable in c_u
8. Evaluate assignments in $N(\mathbf{a})$ according to the function,
   $g_{unsat}(\mathbf{a}) = unsat(\mathbf{a})$
   (this can be done more efficiently by computing $loss(\mathbf{a}^{(t)}, \mathbf{a})$)
9. Choose $\mathbf{a}'$ and $\mathbf{a}''$ to be the neighbours best and second best values respectively. Where two neighbours have equal objective value, the least recently flipped is preferred ($\mathbf{lastflip}_i < \mathbf{lastflip}_j$). Absolute ties are broken randomly.
10. Let $best = \mathbf{FLIPPED}(\mathbf{a}^{(t)}, \mathbf{a}')$ (the variable flipped for the best neighbour).
11. If $lastflip_{best} = t$ then one of the following four cases applies:
    (a) Let $g_{diff} = g_{unsat}(\mathbf{a}'') - g_{unsat}(\mathbf{a}')$
    (b) if $p_{secondbest} < 0.5$ and $g_{diff} > 1$, then $\mathbf{a}^{(t+1)} \leftarrow \mathbf{a}'$
    (c) if $p_{secondbest} < 0.5$ and $g_{diff} = 1$, then
        i. with probability $2p_{secondbest}$, set $\mathbf{a}^{(t+1)} \leftarrow \mathbf{a}''$
        ii. otherwise $\mathbf{a}^{(t+1)} \leftarrow \mathbf{a}'$
    (d) if $p_{secondbest} \geq 0.5$ and $g_{diff} = 1$, then $\mathbf{a}^{(t+1)} \leftarrow \mathbf{a}''$
    (e) if $p_{secondbest} \geq 0.5$ and $g_{diff} > 1$, then
        i. with probability $2(p_{secondbest} - 0.5)$, set $\mathbf{a}^{(t+1)} \leftarrow \mathbf{a}''$
        ii. otherwise $\mathbf{a}^{(t+1)} \leftarrow \mathbf{a}'$
12. Otherwise, set $\mathbf{a}^{(t+1)} \leftarrow \mathbf{a}'$
13. Let $v = \mathbf{FLIPPED}(\mathbf{a}^{(t)}, \mathbf{a}^{(t+1)})$ and $\mathbf{lastflip}_v \leftarrow t + 1$
14. Advance the step counter, $t \leftarrow t + 1$
15. Goto step 3

2.5.8 Weighted GSAT and Weighted WalkSAT

2.5.8.1 Weighted GSAT

Weighted GSAT (WGSAT) was independently invented by Selman and Kautz [102][103], and by Morris [84]. Selman and Kautz were investigating the poor performance of GSAT on structured instances and noted that certain clauses represented critical constraints (the particular example is graph colouring problems with so-called *gerrymandered* graphs). They decided to give a *weight* to each clause to represent the relative importance of that clause in the objective function.

We will refer to all such schemes as *clause weighting* approaches. For any such scheme, we maintain a set of weights, $\lambda = (\lambda_1, .., \lambda_m)$, one for each clause in the formula. The weighted GSAT objective, then, is:

$$g_{WGSAT}(\mathbf{a}) = \sum_{i \in \mathbf{UNSAT}(\mathbf{a})} \lambda_i \tag{2.30}$$

Instead of counting the number of unsatisfied clauses, we now allow each clause a different importance. Selman and Kautz used integer weights, reflecting their view of weighting as "multiple counting" of clauses. They increase the weight of all false clauses at the end of each restart by a fixed amount, ω. This means WGSAT is intrinsically bound to a restart strategy (see Algorithm 13). In the reported experiments, $\omega = 1$.

2.5.8.2 Breakout

Morris's *Breakout* algorithm is essentially the same as WGSAT, although Morris used a "physical forces" analogy to justify it and organized it in a slightly different fashion [84]. He decided to view the problem as a set of nogoods, each forbidding a specific assignment to the variables. He viewed local minima as situations in which the nogoods of neighbouring assignments "equally repelled" the local search process. He therefore decided to weight the nogoods to break the balance. Relatively few details of the algorithm are given. The search is apparently not greedy, but instead appears to be first-descent.

The important difference from WGSAT is that Breakout was conceived as a way to escape local minima, rather than as a means for prioritizing clauses. Breakout updates the weights when a local minimum or plateau is reached. This seems a more logical time to update than the end of a restart since Breakout will update the clauses related to the local minimum instead of the more arbitrary set of false clauses found at the end of the restart.

We now have two useful notions of the value of clause weighting:

- a means for prioritizing "important" clauses
- a mechanism for handling local minima (heretofore handled with random walk or restarts)

Algorithm 13 WGSAT

restart()

1. Initialize restart counter, $r \leftarrow 0$
2. Initialize all weights to 1, $\lambda^{(0)} \leftarrow \mathbf{1}$
3. If WGSAT() succeeds, exit with success (satisfiable)
4. $r \leftarrow r + 1$
5. If a *restart termination condition* has been met, exit with failure (undecided)
6. Increase weight of false clauses: $\lambda_i^{(r)} \leftarrow \lambda_i^{(r-1)} + \omega, \forall i \in \mathbf{UNSAT}(\mathbf{a})$
   where, $\mathbf{a}$ is the last assignment of the previous search.
7. Goto step 2

WGSAT()

1. Randomly generate initial full assignment, $\mathbf{a}^{(0)}$
2. Set step counter, $t \leftarrow 0$
3. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success
4. If the termination criteria have been met, exit with failure
5. Neighbourhood $N(\mathbf{a})$ has n assignments, each obtained by flipping a single variable in $\mathbf{a}^{(t)}$
6. Evaluate assignments in $N(\mathbf{a})$ according to the function,
   $g_{WGSAT}(\mathbf{a}) = \sum_{i \in \mathbf{UNSAT}(\mathbf{a})} \lambda_i^{(r)}$
7. Choose $\mathbf{a}^{(t+1)}$ to be the neighbour with the smallest value. If there are "ties" (several assignments with the same value), choose randomly among the smallest.
8. Advance the step counter, $t \leftarrow t + 1$
9. Goto step 3

Morris notes that plateaus are treated the same way as minima in this framework but does not appear to have investigated whether a different, plateau-specific strategy would be better. He also describes a more general, complete clause weighting approach called *Fill* and proves that it is complete. Fill consists of recording every nogood that occurs at a local minimum and increasing its weight whenever it is encountered. While obviously costly in space and time, Fill provides some interesting intuitions about the behaviour of clause weighting in a global sense.

2.5.8.3 Cha and Iwama

The work by Selman and Kautz on WGSAT was extended by Cha and Iwama [15]. This research includes many innovative ideas and is fertile ground for further invention. They verified that weighted GSAT was effective compared to GSAT and its near relatives, and explored several variations:

- adding 1-step Tabu
- adding random walk
- different weight updates
  - fixed and random, integer and real increments of various sizes
  - add exactly enough to escape the current local minimum (min-fill)
  - update single clauses only

Finding no significant advantage to any of these, they performed some interesting analysis on the behaviour of the algorithm, noting that as the clause/variable ratio grows higher the search space topology becomes more favourable and the problems become easier. They concluded that adding clauses may make the problem easier and so generate new clauses by adding resolvents of existing clauses. It is not clear how they select the clauses to resolve but the text suggests that they are selected at random. They noted that the additional clauses typically reduce the number of search steps required but naturally increase the cost per step so there is no net gain. This work is still quite interesting as it foreshadows later solvers that use resolvents to great advantage (e.g. zChaff [85]).

In later work [16], this idea was refined to produce resolvents from unsatisfied clauses encountered at local minima and add them to the formula. The resulting algorithm is called *ANC* (for Adding New Clauses). The intuition is that instead of reweighting clauses involved in a local minimum, one can achieve a similar effect by creating new clauses that are also unsatisfied at the local minimum. Rather than increasing the weight of each unsatisfied clause by 1 (as WGSAT does), a new clause is produced by resolving each unsatisfied clause with some other clause to produce an unsatisfied resolvent.

There is no guarantee that sufficient resolvents can be found, but the authors claimed that this problem never arose in practice. The authors noted that small resolvents are desirable because they rule out more assignments. Finally, they demonstrated substantial reductions in search steps when using ANC (compared to WGSAT) but did not report runtimes, which they acknowledged to be poor. Constructing suitable resolvents is expensive and the computational burden of local search increases as clauses are added.

The connection between weights and redundant constraints that inspired Cha and Iwama does not explain all of the improvement. The new constraints certainly have a similar effect on the objective function but they can affect the objective value of assignments that would be unchanged by simple clause reweighting. Furthermore, there is no reason to suppose that the combination of weighting and redundant constraints might not be even more effective. While these two mechanisms clearly interact with each other in a complex fashion, they are not completely interchangeable.

2.5.8.4 Frank

Frank continued the study of clause weighting approaches based on WGSAT [34]. He tried several variations, all of which update the weights of the unsatisfied clauses after every search step (unlike WGSAT, which is every try). He also characterized an important property of the *weight profile* (the distribution of weight over all the clauses in the instance). He made the observation that if the search takes many steps, the weights become very large and the additive increments to the weights become insignificant. The profile becomes increasingly resistant to change. Since it is the relative weighting, rather than the absolute, that determines the search behaviour, clause weighting eventually becomes irrelevant.

His initial solution to this problem was to change the way weights were used. Instead of taking the sum of the weights of unsatisfied clauses, he used the sum of some power of the weights, in order to emphasize the differences.

$$g_{power}(\mathbf{a}) = \sum_{i \in \mathbf{UNSAT}(\mathbf{a})} \lambda_i^{\alpha} \tag{2.31}$$

where α is a real-valued constant.

The approach improves performance but creates a new parameter to tune, and one that depends on how large the weights are expected to become. Further research [35][36] abandoned this change to the objective function in favour of changing the weight profile itself by introducing *weight decay*. Here, the weight update consists of some fraction of the current weight plus an increment:

$$\lambda_i^{(t+1)} = \rho \lambda_i^{(t)} + \omega \tag{2.32}$$

where ω is the size of the weight increment (or *learning rate,* as Frank characterizes it) and $0 \leq \rho \leq 1$ is the decay factor. He notes that the relevance of weights appears to be highly localized and that, as the search moves away from a given assignment, it is important to "forget" the emphasis learned in that region. This weight decay scheme offers a modest

improvement in the larger problems attempted, but the results are not striking. However, subsequent work on clause weighting methods has shown that decay (or *smoothing*) is crucial to good performance (see Section 2.5.9, and Chapters 4 and 5).

2.5.9 DLM

The *Discrete Lagrangian Method* (DLM) is a collection of solvers developed over a few years and incorporating a variety of methods to produce an effective solver [111, 112, 117, 118, 119]. While DLM is described as a general purpose approach to optimization in discrete spaces (e.g. [116]), the focus is on SAT. We will refer to the DLM formulation for SAT as DLM-SAT.

At its core, DLM-SAT is a clause-weighting method, like WGSAT and its derivatives, but it uses a variety of heuristics to control the search. One of the chief claims of the authors is that the method does not require restarting, unlike GSAT and its relatives. By perturbing the search space via the weights, DLM can escape from local minima and reach other regions of the space. No formal proof of its reachability is provided but the authors did investigate an interesting consequence of this claim. They started the algorithm from the "origin" (presumably some uniform assignment to the variables) instead of a random starting point. While they did not present the results, the authors state that the algorithm still works but performs better when using random starting points.

The authors present a detailed description of their Lagrangian method for integer spaces, along with various proofs and definitions, most of which have little relevance for SAT (largely because it is easy to tell when the problem has been solved). We will omit much of this and instead briefly introduce the ideas behind Lagrangian optimization that are important for understanding both DLM and our own algorithms.

2.5.9.1 Lagrangian Methods

The authors formulate the general DLM procedure in terms of Lagrangian optimization (see [10] for extensive discussion of classical Lagrangian methods). The goal is to optimize (we will minimize) some objective function, $f(\mathbf{x})$, subject to m constraints. For the time being, we will leave the exact form of the constraints unspecified and simply represent them by a vector $\mathbf{v}$ of m functions, where each $v_i(\mathbf{x})$, $1 \leq i \leq m$ is less than or equal to zero when the ith constraint is unviolated, and positive otherwise. Thus, we can formulate what we call the *primal problem*:

$$\min_{\mathbf{x}} f(\mathbf{x}) \quad \text{subject to} \quad \mathbf{v}(\mathbf{x}) \leq \mathbf{0} \tag{2.33}$$

Lagrangian methods turn this constrained optimization problem into an unconstrained optimization by considering violated constraints as incurring some *penalty*. Each constraint has an associated weight, λ_i, and the total penalty is the weighted sum of violations. Combining this penalty term with the original objective, $f(\mathbf{x})$, we obtain an unconstrained opti-

mization version of the original problem in the form of a new objective function, $L()$, which we call the Lagrangian:

$$L(\mathbf{x}, \lambda) = f(\mathbf{x}) + \sum_{i=1}^{m} \lambda_i v_i(\mathbf{x}) \tag{2.34}$$

where $\lambda_i \geq 0$. The weights on each constraint are called *Lagrange multipliers*.

We can now consider trying to optimize this new objective. If we hold the weights constant, the problem becomes the following minimization:

$$D(\lambda) = \min_{\mathbf{x}} L(\mathbf{x}, \lambda) \tag{2.35}$$

$D(\lambda)$ is called the *dual function*. We can now define:

$$\max_{\lambda} D(\lambda) \quad \text{subject to} \quad \lambda \geq \mathbf{0} \tag{2.36}$$

This is called the *dual problem*. While it is not obvious from the definitions, under the right conditions, solving the dual problem can solve, or help to solve, the primal problem. If the value obtained by solving the primal problem (2.33) is P^* (i.e. the value of the optimal solution), the *weak duality theorem* [10] states that the value of the dual function for any λ is a lower bound on P^*, or more formally, that $D(\lambda) \leq P^*$, $\forall \lambda \geq 0$. Lower bounds obtained by computing the dual function can help us solve the problem (they are commonly used in branch-and-bound algorithms).

Now consider the value obtained by solving the dual problem (2.36), D^*. D^* is a lower bound on the value of P^*. This property, $D^* \leq P^*$, is likewise due to the weak duality theorem. A natural concept arising from this property is the *duality gap*, which is the difference, $P^* - D^*$. Clearly, if the duality gap is zero, then the optimal value of the dual problem is the same as the optimal value of the primal problem. This is the case for linear (or convex) programs with real-valued variables. Thus, solving the dual problem may give us the solution to the primal problem. If the gap is non-zero, then D^* at least provides us with a lower bound on the optimal value of our problem. Attempting to solve the dual problem now has a nicely intuitive meaning; in doing so, we are trying to maximize a lower bound on the value of the optimal solution.

2.5.9.2 Primal-Dual Methods

The SAT problem cannot be formulated as a linear or convex program, so we have no guarantees that we can obtain an exact solution by solving a dual form of SAT. Furthermore, the minimization and maximization operations are not straightforward because the functions involved are not differentiable. A common way to overcome such problems is the use of *primal-dual methods*. These consist of alternately minimizing and maximizing $L(\mathbf{x}, \lambda)$ with respect to the two sets of variables, $\mathbf{x}$ and λ, in the hopes that we will iteratively arrive at a solution.

A solution $(\mathbf{x}^*, \lambda^*)$ to the dual problem is called a *saddle point*. A saddle point is a feasible point (i.e. no violated constraints at $\mathbf{x}^*$) where

$$L(\mathbf{x}^*, \lambda) \leq L(\mathbf{x}^*, \lambda^*) \leq L(\mathbf{x}, \lambda^*), \ \forall \mathbf{x} \text{ and } \forall \lambda \geq 0 \tag{2.37}$$

Any such pair, $(\mathbf{x}^*, \lambda^*)$, is an optimal solution to the problem. At a feasible point the constraints are either zero or negative. For the zero constraints, any values for the corresponding multipliers are a maximum of the dual function, whereas the multipliers for the negative constraints must be zero (this makes the penalty term zero at a saddle point). For SAT, any saddle point (indeed, any feasible point) represents a satisfying assignment.

Computing the dual function for some $\hat{\lambda}$ gives us a corresponding $\hat{\mathbf{x}}$. If constraints are violated at $\hat{\mathbf{x}}$ then $\max_\lambda L(\hat{\mathbf{x}}, \lambda)$ has no solution (the multipliers would go to infinity). The problem is that for many problems (e.g. SAT) we cannot simultaneously maximize with respect to λ and minimize with respect to $\mathbf{x}$ as the dual problem requires. One solution is to alternate back and forth between the two sets of variables instead by (i) fixing a λ' and minimizing $L(\lambda', \mathbf{x})$ with respect to $\mathbf{x}$ (this is called a *primal step*) to obtain $\mathbf{x}'$, and then (ii) fixing $\mathbf{x}'$ and applying some finite adjustment to λ' to produce a λ'' such that $L(\lambda'', \mathbf{x}') > L(\lambda', \mathbf{x}')$ (this is called a *dual step*).[15] We then perform another primal step, and so on. A dual step is a finite step towards the maximum in the dual problem and so avoids the aforemention problem with maximizing when constraints are violated. Typically, the dual step is made by adding a fixed amount to some of the multipliers but the details depend on the exact method. The procedure will terminate (i.e. the primal and dual steps will not change $\mathbf{x}$ and λ) if a *saddle point*, $(\mathbf{x}^*, \lambda^*)$ is reached. Algorithm 14 summarizes the primal-dual procedure.

Algorithm 14 Primal-Dual Methods

1. $t \leftarrow 0$
2. Pick an initial $\mathbf{x}^{(0)}$.
3. $\lambda^{(0)} \leftarrow \mathbf{1}$
4. do
   (a) Primal Step: $\mathbf{x}^{(t+1)} = \arg\min_{\mathbf{x}} L(\mathbf{x}, \lambda^{(t)})$
   (b) Dual Step: find $\lambda^{(t+1)}$ such that $L(\lambda'', \mathbf{x}') > L(\lambda', \mathbf{x}')$
   (c) $t \leftarrow t + 1$
5. while $\mathbf{x}^{(t)} \neq \mathbf{x}^{(t-1)}$ and $\lambda^{(t)} \neq \lambda^{(t-1)}$

A primal-dual method can be seen as searching for a saddle point, which, in turn, is

[15]This is a simplification used to convey the idea. Strictly speaking, λ'' need not increase the value of the Lagrangian compared with λ', it need only be closer than λ' to some λ^*.

an optimal solution. Under certain conditions that we will not explore further here (an important one, however, is that the dual steps be sufficiently small), it can be shown that primal-dual methods will converge to the optimal solution for the dual problem [10]. It suffices for us to introduce the idea here because this approach is used by DLM and most other clause-weighting approaches (although their authors may not recognize or claim it). Both DLM and our own ESG algorithm (see Chapter 5) are explicitly derived in this framework but with some notable differences.

It is worth noting that if the problem is infeasible (unsatisfiable in the case of SAT), then a standard primal-dual approach will simply increase the Lagrange multipliers forever. The clause-weighting methods that fall into this framework are all incomplete, so we did not expect to solve these problems in any event.

2.5.9.3 DLM and SAT

The authors motivate their approach by explaining it as a form of Lagrangian optimization in discrete spaces. It has been applied to constrained optimization problems such as MAX-SAT and the design of QMF filter banks [116], but is best known for its application to SAT. The DLM-SAT formulation uses a Lagrangian like (2.34) where the objective function, $f(\mathbf{a}) = unsat(\mathbf{a})$, the number of unsatisfied clauses, and represents the constraints as giving a violation of 1 when the corresponding clause is unsatisfied and 0 when satisfied. This gives the following specific form:

$$L(\mathbf{a}, \lambda) = unsat(\mathbf{a}) + \sum_{i=1}^{m} \lambda_i v_i(\mathbf{a}) \tag{2.38}$$

where $v_i(\mathbf{a}) = \begin{cases} 0 & \text{if } c_i \text{ is satisfied} \\ 1 & \text{if } c_i \text{ is unsatisfied} \end{cases}$

Note that assignments are boolean, hence the discreteness. The weights, however, are real-valued.

If we note that $unsat(\mathbf{a}) = \sum_{i=1}^{m} v_i(\mathbf{a})$, then the objective simply becomes:

$$L(\mathbf{a}, \lambda) = \sum_{i=1}^{m} (1 + \lambda_i) v_i(\mathbf{a}) \tag{2.39}$$

This objective is arguably peculiar because it redundantly enforces the constraints. It minimizes the number of violated clauses plus a weighted penalty for violated clauses. The authors offer a vague assertion that this formulation offers some advantage by searching in “two spaces”, but do not back this up with any argumentation [112]. They further argue that if a large number of clauses become violated, the unweighted part of the objective will force the search back to satisfying clauses. This latter argument is not convincing either, since it is not clear that this is desirable and the weight of any individual clause can become large enough to overwhelm this sum in any event. While not a trivial rescaling of the objective function, it seems unlikely that it has any profound effect on the search. At most,

it delays the effects of early weight increases by requiring them to be higher before they will substantially alter the ordering of variable choices.[16]

In its most basic form, DLM is a primal-dual method, and alternates between minimizing (2.39) in $\mathbf{a}$ and maximizing it in λ. Since it is searching over discrete assignments, there is no straightforward way to minimize with respect to $\mathbf{a}$, so DLM uses a descent via local search. The exact details of this search vary from version to version, but all are similar to members the GSAT/WalkSAT family but searching in the weighted objective.

Naturally, local search may be trapped by a local minimum, which is the case with all clause-weighting techniques. This problem represents the most important deviation from typical Lagrangian optimization, the theory behind which assumes that global minima with respect to either the primal or the dual variables may be readily found. Thus, arguments relating to the duality gap, optimality, and convergence typically do not apply. However, the success of these methods suggests that similar principles may underlie these approximate techniques, giving them their strength.

A dual step is performed whenever the local search reaches a minimum. Again, it varies from one version of DLM to another, but in its simplest forms it is similar to WGSAT, incrementing the weights of unsatisfied clauses. Later versions of DLM introduce weight decay or smoothing, similar in spirit to that proposed by Frank [35].

Around this general framework, a number of elaborate additions have been made over time, resulting in very complex algorithms with many parameters. This may explain why relatively few SAT researchers attempt comparisons with DLM, despite the source code's availability, since the task of determining optimal settings for the algorithm is daunting and it is not always easy to be sure what features are present in a given version. Rather than trying to give all the variants in their formal algorithmic forms, we will simply list their features and differences, referring the interested reader to the various related papers for further details.

2.5.9.4 The Evolution of DLM

The first solvers were called DLM-A_1, DLM-A_2, and DLM-A_3 [112], and progressively include new heuristics, new implementation optimizations, and improved performance.

DLM-A_1

- Weighted-objective
- Selects first variable offering a strict improvement (first-descent).
- Switches between examining all variables and maintaining an unsatisfied clause list depending on the number of unsatisfied clauses (speed tradeoff).

[16]In our own algorithms, SDF and ESG (Sections 4 and 5), we use a zero objective function, rendering the problem in its most natural form, a simple constraint satisfaction problem.

- Additively upweights unsatisfied clauses if no improvement is possible (local minimum or plateau). Satisfied clauses are not modified (in particular, they are not reduced).[17]

DLM-A_2

Same as DLM-A_1 but,

- Switches between examining all variables and maintaining a list of variables that offer improvements (speed tradeoff).

DLM-A_3

Same as DLM-A_2 but,

- If no improvement is possible, but an equally good assignment is, then make a "flat move", if
  - no more than some fixed number of flat moves have been made
  - or, the variable has not been flipped for some specified number of recent steps (Tabu strategy)
- otherwise, perform a reweight.
- Every fixed number of steps, all clauses have their weights reduced by a fixed amount.[18]

Note an important difference between DLM-A_2 and DLM-A_3. In DLM-A_2, when a plateau is reached, a reweight is performed. DLM-A_3 is allowed to explore plateaus without reweighting, subject to some limitations that prevent it from exploring forever and revisiting regions of the plateau. This is closely related to the plateau-related strategies used by GSAT and WalkSAT.

In 1999, the authors introduced a new algorithm, DLM-99-SAT [118], which is the successor to DLM-A_3.

DLM-98-BASIC-SAT

This algorithm is cited to be DLM-A_3 but the descriptions in the two papers ([112] and [118]) do not agree. It is not clear exactly where DLM-98-BASIC-SAT comes from, but it seems to be intermediate between DLM-A_3 and DLM-99-SAT. Differences from DLM-A_3 include:

- Steepest-descent instead of first-descent.

[17] This difference in the treatment of satisfied and unsatisfied clauses is important and will be discussed further in Chapter 5 in relation to different penalty functions.

[18] Note that both satisfied and unsatisfied clauses are reduced. This is the introduction of smoothing in DLM.

- Accepts some uphill-moves.
- Always avoids tabu variables (not only on plateaus).
- Reweights after some number of uphill and/or flat moves have been made (instead of at minima and plateaus).
- Weight reductions are made after a fixed number of reweights (instead of steps).

The authors note that their earlier algorithms can become “trapped” in a region of the search space. Even though reweighting will force the solver away from the region, subsequent reweights will force it back. The key solution they propose is to identify “traps” and keep track of the number of times a clause is found unsatisfied in a trap (we will call this number a “trap count” for the clause). A “trap” is simply a local minimum in the weighted objective function. Any clause with a high trap count is frequently unsatisfied at local minima. Trap counts do not decay over time.

DLM-99-SAT

Same as DLM-98-BASIC-SAT except:

- Propagates any unit clauses before it starts searching.[19]
- Maintain a “trap count” for each clause that is incremented when a “trap” (local minimum) is encountered and the clause is found to be unsatisfied at that point.
- Every reweight, perform a “special increase”. This consists of examining a set of clauses (either all of them or only the unsatisfied clauses, depending on a parameter) and checking their trap counts relative to the total trap count of the set. If a clause has a high relative trap count, its weight is increased by an extra amount.

The “special increase” mechanism is the important change. The argument is that if the solver is repeatedly falling into the same trap, its clauses’ trap counts will become high and eventually the corresponding weights will be increased, thus permanently raising its importance relative to the other clauses. In [119], the authors also mention that it may require several reweight operations to escape a local minimum, which is a costly process. The special increase can reduce the number of reweights required to escape a deep trap.

The authors’ observations bear out these interpretations, but it seems likely that there is a second, more general effect. The trap counts record how often a clause was unsatisfied, regardless of the specific traps involved (the clause may occur in several different traps). Each count can be viewed as measuring the intrinsic difficulty of satisfying the corresponding clause (the Lagrange weights decay over time, and so only provide a recent history). Even if the solver is not repeatedly trapped by specific regions of the search space or wasting time filling in deep traps, this global history may still identify difficult clauses that are

[19]This is a trivial reduction that could be applied by any SAT solver.

unsatisfied in many different local minima. This gives us a new notion, that of *short-term weights* vs. *long-term weights*, which was also identified by Frank [35][36].

DLM-2000-SAT

The most recent version of DLM as of the time of writing was DLM-2000-SAT, described in [119]. This approach returns to a simpler framework and is essentially the same as DLM-98-BASIC-SAT but with a different objective function. This algorithm adds a "distance penalty" to the objective function, penalizing moves that do not increase the Hamming distance of the next point relative to a set of recent points. The exact formulation gives a rather weak preference for points farther away from recent points, but all distances larger than 2 are treated as being equal to 2, so the algorithm really attempts to avoid recent points less than one move away (i.e. all neighbours in the recent move history). The solution times on hard instances are quite good compared to DLM-99-SAT, but since flip counts are only shown for DLM-2000-SAT, it's hard to be sure what's going on. The method is certainly more expensive per step than DLM-98-SAT-BASIC, but the relationship to DLM-99-SAT is not clear.

The more complex DLM solvers have many parameters (the authors list nine for DLM-99-SAT, and there are several more in the parameter files) and thus pose a serious tuning problem. The authors' addressed these issues to some extent in [117]. They constructed five hand-tuned parameter sets for DLM-99-SAT, and then went on to propose a set of metrics and a heuristic scheme that can be used to select a parameter set based on measurements from short runs using each of the five parameter sets.[20] We briefly revisit this research in Chapter 3.

Our own experience with these parameter sets suggests that they are quite effective, although they have radical effects on the run-time per step, which means that even though a set may dramatically decrease the number of search steps, the steps themselves are much slower and so no real gains are realized. It is difficult to predict which set will be effective and the best set sometimes varies with the scale of the instance, even within a single instance class. We have not implemented their selection scheme and in most cases we simply run all five parameter sets to determine the best.[21]

The best DLM solvers succeeded in solving several instances never before solved by local search algorithms, especially the very hard, satisfiable DIMACS instances. The authors systematically attacked these instances, observing the behaviour of the different solvers and adapting them to address difficulties. The result is a very complicated, but very effective local search solver that easily outperforms any of the GSAT and WalkSAT-family solvers on hard instances, and remains one of the best local search SAT solvers available.

[20][117] also describes a new variation of DLM-99-SAT which was used to improve performance on the DIMACS *hanoi* instances. It provides two ways to perform weight decay. We will refer to it as DLM-99-SAT-B.

[21]In all experiments we present, the results are performed using the DLM-99-SAT algorithm although the code base is for DLM-2000-SAT. The parameter sets do not appear to enable DLM-2000-SAT's features by default.

2.5.10 Guided Local Search

Guided Local Search (GLS) is a broad framework (derived from GENET [24]) for controlling local search mechanisms that has been tried on SAT and weighted MAX-SAT [82]. Briefly, GLSSAT uses a local search mechanism similar to HSAT (i.e. GSAT + Tabu), with a weighted objective function. Considerable customization was performed to bring the general GLS framework to bear on SAT, including specialized ideas such as weight smoothing directly inspired by DLM. The only substantial difference from other clause weighting methods is that clauses that are very frequently penalized will automatically stop receiving penalties until other clauses have also received some. This mechanism seems to offer a second form of smoothing. While it performs better than WalkSAT on many SAT problems (with a few catastrophic failures), the authors present no comparison with DLM so it's unclear whether it really offers a performance advantage relative to its contemporaries.

2.6 Survey Propagation

Survey propagation [81, 13] is one of the most significant algorithms to arrive on the SAT research scene in recent years. It is a probabilistic method based on *belief propagation*, a method used for inference in graphical probability models. Based in statistical physics, Mezard *et al.* used the algorithm to characterize the structure of the phase transition of random 3-SAT and computed upper and lower bounds for it. They also developed an incomplete solver, *survey propagation decimation*.

While the details are complicated, the basic idea runs as follows. Suppose we know the set of solutions for a satisfiable instance, $\mathbf{S}$. The solutions can be grouped into *clusters*. In a given cluster, each variable is constrained to be true, false, or unconstrained (*don't care*). The unconstrained variables are what makes a cluster contain multiple solutions. The constraints on variables are imposed by the clauses in which they occur. These clauses can be thought of communicating a *message* to their component variables. Each message is a $\{0, 1\}$ value, 0 if the clause puts no constraint on the variable, and 1 if it constrains it to take a satisfying value. Contradictory messages (i.e. messages from two different clauses constraining the variable to opposing values) are not allowed.

The survey propagation algorithm approximately computes the marginals of a distribution over these messages by an iterative message passing algorithm, starting with randomized messages and, hopefully, converging to a fixed point. This gives a distribution over three values for each variable: true, false, and unconstrained.

Survey propagation decimation computes these probabilities and then assigns some fixed number of variables, greedily selecting those that have the strongest bias toward taking a constraining value (true or false). The instance is then simplified and unit propagation performed. If an empty clause is reached, the algorithm gives up or restarts. Otherwise, the probabilities are computed for the new instance. If an instance is produced that has no biased variables (i.e. all are unconstrained), a local search solver (e.g. WalkSAT) is used to try to find a satisfying assignment.

In the decimation algorithm, survey propagation is essentially used as a variable ordering heuristic. The algorithm has achieved startling results, solving large (tens and even hundreds of thousands of variables) instances in the phase transition (reported experiments are actually at clause-variable ratio, $r = 4.2$, but our own experiments indicate that it frequently works even around 4.26 – see Section 2.2.4 for discussion of the phase transition). This is beyond the capability of existing solvers, even the best of the local searchers. One can imagine inserting survey propagation as a variable ordering in a number of contexts. However, it is very slow compared to currently accepted ordering heuristics.

It is also important to note that it is more or less dedicated to HR instances and typically performs quite poorly on structured instances. For the sake of curious readers, we have included results from our own implementation of survey propagation in the general collection provided in Appendix E. The implementation is somewhat slower than the original (see Appendix B.2), but the results may still be of interest since we are unaware of any study of the method with the breadth offered here.

Chapter 3

Metrics for Local Search

> *The blackness of eternal night encompassed me. [...] The agony of suspense grew at length intolerable, and I cautiously moved forward, with my arms extended. [...] I proceeded for many paces, but still all was blackness and vacancy. [...] My outstretched hands at length encountered some solid obstruction. I followed it up; stepping with all the careful distrust with which certain antique narratives had inspired me. This process, however, afforded me no means of ascertaining the dimensions of my dungeon; as I might make its circuit, and return to the point whence I set out, without being aware of the fact.*
> - **The Pit and the Pendulum**, Edgar Allan Poe

Our first significant work on satisfiability was on local search methods, trying to understand how they work. These algorithms are typically very fragile. Small and seemingly unimportant details can have tremendous impact on their performance (e.g the variants of WalkSAT discussed in Section 2.5.5). This makes it very hard to predict what methods will work well, or to *a priori* design features likely to succeed. Where intuitions seem so unreliable, some objective measures may help to assess algorithms and understand the underlying features.

Informal experimentation with several local search solvers, such as GSAT, WalkSAT, and Novelty+, gave clues that led to the investigation of the following three metrics: *depth*, *mobility*, and *coverage*. We will describe each of these in detail and demonstrate how they can characterize the performance of a solver. We then show dual versions of mobility and coverage that behave very like the originals.

3.1 Depth

A very natural metric when using an objective function is the solver's average objective value over the course of the search. The use of an objective function to evaluate neighbours implicitly assumes that solutions are likely to be found near favourable objective values (low values would be favourable if we are minimizing). Under this assumption, one might

suppose that the more time spent in regions with favourable objective values, the more likely one is to find solutions. A rather weaker, but safer, statement is that spending a lot of time in regions with unfavourable objective values is unlikely to produce solutions.

The simplest measure of what we will call *depth* is simply the average objective value at all points along the search. Practically, this can give rise to a problem. A poor solver may spend a long time searching for a solution before finding one whereas a good solver may find one very quickly. In measuring depth we are really interested in the long term behaviour, how much time is spent in promising regions. In hill-climbing searches with random starting points, the searcher frequently starts in poor regions and rapidly descends to much better regions. However, a particularly effective solver may find a solution during or soon after this initial descent. Thus, its average behaviour will look quite poor compared to a weaker solver that has a similar initial descent but spends more time searching.

There are several possible corrections one can imagine, but we simply ignore some fixed number of initial steps, d_{skip}, to get past the initial descent, and then average over the remaining search. The number of steps to skip can be easily selected by examining several initial descents. In all results reported here, the same skip count was used across all solvers after examining their collective behaviour.

$$depth(d_{skip}) = \frac{1}{T - d_{skip}} \sum_{t=d_{skip}+1}^{T} g(\mathbf{a}^{(t)}) \tag{3.1}$$

where $g()$ is the objective function used by the solver, or some other objective for which we expect the solver to maintain reasonable values.[1]

In order to determine how effective depth is as a predictor of success, an experiment was run using a single instance from SATLIB's [60] uf100 collection, a 100 variable, 430 clause random 3-SAT instance (the class described in Section 2.2.4).[2] Instance uf100-0953 was selected because our experiments showed it to be among the hardest instances in uf100. Five solvers, WalkSAT (WSAT), Novelty, Novelty+, DLM, and SDF (our own algorithm described fully in Chapter 4), were each run 100 times on this instance, measuring the average depth (lower is better), average search steps (lower is better), and OEESS (lower is better - see Section 2.5.1 for details of what this statistic means).[3]

Parameters are indicated in the tables and correspond to the order and, where unspecified, the default values shown in Appendix C. No restarts were used. WalkSAT was run at several different noise (p_{walk}) levels in order to capture the effect that noisy strategies such as random walk can have on depth and performance. Table 3.1 shows the results of this

[1] Comparing solvers with different objective functions may be meaningless under the depth measure. Therefore, in cases where this is necessary, we have measured depth in the same objective function, $g_{UNSAT}()$, for all solvers (i.e. the number of unsatisfied clauses). Thus, a solver may search in one objective while we measure depth in another.

[2] All experiments in Chapter 3 were performed using the hardware environment described in Appendix A.1.

[3] Since the objective here is not to compare the solvers, *per se*, but rather to understand the relationship between the various metrics and solver performance, this methodology is quite appropriate. For brevity, and because the instrumented solvers incur extra runtime penalties, we only report search steps.

study.

Solver	Av Depth	Av Steps	OEESS
WSAT(.9)	12.3	59,042	53,451
WSAT(.8)	10.3	25,543	18,698
WSAT(.7)	8.60	18,042	17,282
WSAT(.5)	5.16	11,757	9,802
Novelty(.5)	3.75	4,916	4,563
Novelty+(.5, .01)	3.68	3,965	3,965
DLM(pars4)	5.40	2,182	1,984
SDF(.00085, .995)	3.03	1,192	1,174

Table 3.1: Average depth (lower is better), along with the average number of steps needed to find a solution and the estimated average number of steps needed under an optimal restart scheme. Results are averaged over 100 runs on problem uf100-0953 from SATLIB.

Notice the effect of increasing noise on WalkSAT's depth and performance. As it makes more random moves, and correspondingly fewer greedy ones, it is more likely to walk to poor regions. Its depth and performance deteriorate together. Typically, as depth improves (grows smaller), the performance in terms of steps and expected steps improves. This is not uniformly the case. DLM has poorer depth than Novelty+ and yet achieves better performance. This is not unexpected, since depth is unlikely to be the sole factor determining performance.

A broader study shows more plainly that depth is not the sole factor governing performance. The results in Table 3.2 were obtained by a similar experiment. All 1000 instances in the uf100 set were used, each run 100 times, and the average depth overall reported for GSAT, HSAT, WSAT-G, WSAT, Novelty, Novelty+, DLM, and SDF (see Section 2.5 for details of these solvers, except for SDF which is described later on in Chapter 4).

Solver	Av Depth	Av Steps	OEESS
GSAT()	1.62	74,595	10,823
HSAT()	1.35	73,510	2,503
WSAT-G(.5)	3.13	7,695	4,431
WSAT(.5)	3.49	3,582	2,828
Novelty(.5)	1.90	2,586	1,473
Novelty+(.5, .01)	1.93	2,224	1,393
DLM(pars4)	6.36	1,020	800
SDF(0.00085, .995)	1.52	870	725

Table 3.2: Same as Table 3.1, but results averaged over 100 runs on all 1000 uf100 problems from SATLIB.

These results clearly demonstrate that depth is no guarantee of good performance. Al-

gorithms like GSAT find deep regions but may get stuck in them, resulting in excellent depth values but very poor performance. The solver spends all its time trapped in a deep local minimum. Still, most of the better solvers demonstrate excellent depth.

These averaged results cannot by themselves clearly establish that good depth and performance tend to go hand-in-hand. The following analysis establishes this relationship more plainly. Four comparable solvers (DLM, Novelty, Novelty+, and WSAT) were run on 2700 instances from SATLIB (uf50, uf75, uf100, uf125, uf150, uf175, uf200, uf225, uf250), all HR instances of sizes from 50 variables up to 250. Each solver was run 100 times and the average depth and OEESS recorded for each instance. The four solvers were then ranked on depth and on OEESS, for each instance.

Now for each instance we can consider <depth rank, OEESS rank> pairs (e.g. the solver ranked 1st on depth and the solver ranked 1st on OEESS, the solver ranked 1st on depth and the one ranked 2nd on OEESS, etc.) Table 3.3 shows a grid with the ranks in terms of depth versus the ranks in terms of OEESS. The figures in each entry are the the proportion of instances on which the ith depth ranked solver was the same as the jth OEESS ranked solver. Therefore, the top-left entry means the proportion of instances where the best solver in terms of OEESS was also the best solver in terms of depth was 74%. The entry to right shows that 9% of all instances had the top OEESS ranked solver only rank second in terms of depth.

If we expect depth to be a good predictor of performance (as measured by OEESS), then we expect large values along the diagonal and small values elsewhere. The table shows this quite nicely, with a particularly strong relationship between the solvers of top rank on both depth and OEESS. A few strong off-diagonal values confirm our conclusion that depth is not the whole story (and may also be due to the close similarity between Novelty and Novelty+), but its predictive value is clearly supported.

OEESS	Depth rank			
rank	best 1	2	3	worst 4
best 1	.74	.09	.13	.05
2	.10	.28	.38	.25
3	.14	.42	.34	.11
worst 4	.03	.22	.16	.60

Table 3.3: Frequencies of (steps rank, depth rank) pairs from among four search procedures, DLM, Novelty, Novelty+, and WSAT. Frequencies measured over 2700 uf instances from SATLIB.

3.2 Primal Mobility

Clearly depth alone is not sufficient. One easy way to achieve good average depth is to find a region of favourable values and simply stay there. A successful search must explore

new regions. Mobility is a another simple measure. We will describe *primal mobility* first, so called because it relates to the primal variables of the search (i.e. the boolean variables of the formula) and will introduce a dual version presently. If we consider a sequence of assignments, primal mobility is simply the average Hamming distance travelled over some fixed subinterval. For each step, t, in the search, we compute the Hamming distance, $H(\mathbf{x}^{(t)}, \mathbf{x}^{(t+w)})$, between the assignment at step t and the assignment w steps later. We then take the average of these distances:

$$primalMobility(w) = \frac{1}{T-w} \sum_{t=1}^{T-w} H(\mathbf{a}^{(t)}, \mathbf{a}^{(t+w)})$$

where T is the total number of steps in the search. This metric has the disadvantage of having a single parameter, w, which is the length of the window. Again, it is not hard to select a value for this parameter by looking at average run lengths for solvers and also the maximum possible Hamming distance. Local search solvers that flip only a single variable at each step cannot move more than $min(w, n)$ steps within a window (recall that n is the number of variables). While no Hamming distance between any two assignments can be greater than n, selecting a window of length greater than n is still reasonable since it is unlikely that any solver will flip all variables in succession.

Given that we have a maximum distance, it makes sense to use a normalized version of this measure:

$$normPrimalMobility(w) = \frac{1}{T-w} \sum_{t=1}^{T-w} \frac{H(\mathbf{a}^{(t)}, \mathbf{a}^{(t+w)})}{min(w, n)}$$

To investigate the mobility of solvers, we used the same experiments that were run for depth, computing $primalMobility(100)$ as well. Table 3.4 shows both the depth and primal mobility for 100 runs on uf100-0953. Note that the mobility is not normalized. These results show a much stronger relationship between mobility and performance than was demonstrated for depth. The ranking in terms of mobility matches the ranking in terms of OEESS.[4]

Table 3.5 shows the $primalMobility(100)$ results over all 1000 uf100 problems. Ranking by mobility matches ranking by OEESS for the most part (HSAT appears to offer better performance than the higher mobility WSAT under some aggressive restart schedule), and SDF slightly outperforms DLM despite having slightly lower mobility.

Although mobility is not the only relevant factor, it does appear to be an excellent predictor of performance. To more clearly demonstrate this, the same rank vs. rank matrix was produced as for depth and is shown in Table 3.6. Again, high diagonal values suggest a strong relationship, and we find a very strong diagonal here, with only slight dilution of results in the middle, probably due again to the similarity between Novelty and Novelty+.

Finally, it is interesting to see how mobility behaves over a variety of window sizes. Figure 3.1 shows a plot of $primalMobility(w)$ for $0 \leq w \leq 150$ averaged over 100 runs

[4] † In some cases, GSAT failed all runs so its OEESS is not reported.

Solver	Av PMob	Av Depth	Av Steps	OEESS
GSAT()	6.0	2.19	500,000	n/a †
HSAT()	9.0	2.06	495,003	16,700
WSAT-G(.5)	10.1	4.20	29,661	15,442
WSAT(.5)	16.1	5.16	11,757	9,802
Novelty(.5)	19.0	3.75	4,916	4,563
Novelty+(.5, .01)	18.7	3.68	3.965	3.965
DLM(pars4)	28.6	5.40	2,182	1,984
SDF(.00085, .995)	29.7	3.03	1,192	1,174

Table 3.4: Average primal mobility over 100 steps (higher is better), along with depth and search step measures. Results are averaged over 100 runs on problem uf100-0953 in SATLIB.

Solver	Av PMob	Av Depth	Av Steps	OEESS
GSAT()	8.8	1.62	74,595	10,823
HSAT()	15.8	1.35	73,510	2,503
WSAT-G(.5)	12.8	3.13	7,695	4,431
WSAT(.5)	18.9	3.49	3,582	2,828
Novelty(.5)	23.4	1.90	2,586	1,473
Novelty+(.5, .01)	23.6	1.93	2,224	1,393
DLM(pars4)	32.4	6.36	1,020	800
SDF(0.00085, .995)	33.4	1.52	870	725

Table 3.5: Same as Table 3.4, but results averaged over 100 runs on all 1000 uf100 problems from SATLIB.

OEESS	Primal mobility rank			
rank	best 1	2	3	worst 4
best 1	.92	.07	.01	.00
2	.06	.72	.21	.01
3	.03	.20	.75	.03
worst 4	.00	.01	.03	.96

Table 3.6: Frequencies of (steps rank, mobility rank) pairs from among four search procedures, DLM, Novelty, Novelty+, and WSAT. Frequencies measured over the entire collection of uf problems in SATLIB (2700 problems in total). Mobility is measured between assignments 75 steps apart in the search, averaged over the length of a search run.

of uf100-0953. The ranking of these lines from top to bottom corresponds closely to their ranking in performance and is consistent over most window lengths. The only odd case in this plot is HSAT, which has erratic behaviour. Figure 3.2 shows the same plot, but with mobility averaged over all 1000 problems in uf100. Again, the ranking is mostly preserved.

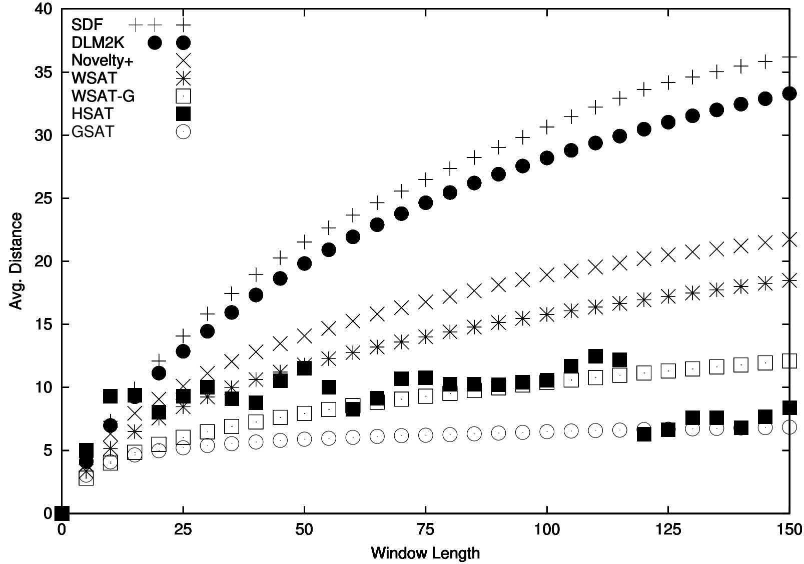


Figure 3.1: Average primal mobility for a range of window lengths, obtained by various search procedures, averaged across 100 repetitions of the uf100-0953 problem in SATLIB.

3.3 Primal Coverage

High primal mobility ensures that a solver does not simply sit in one place but, given the windowed nature of the measurement, it is still possible to be trapped in a large basin in the search space. A solver might follow a large "circle", with a length considerably larger than the window, leading to a false conclusion. One solution is to consider how much of the search space is actually visited by the solver.

This informal notion of *coverage* makes considerable intuitive sense but turning it into a precise measure presents some problems. One possibility is simply to count the number of unique assignments visited. This is a very weak notion for a couple of reasons. First, the number of possible assignments is exponential, so that the actual proportion explored becomes increasingly trivial in any practical application. Second, all visited points could be near each other, so we obtain little information about the global nature of the search.

A more interesting measure can be obtained by considering the Hamming distance between some explored point, $\mathbf{a}_e$, and some other, unexplored point, $\mathbf{a}_u$. In particular, we would like information about the relationship between some $\mathbf{a}_u$ and the nearest explored

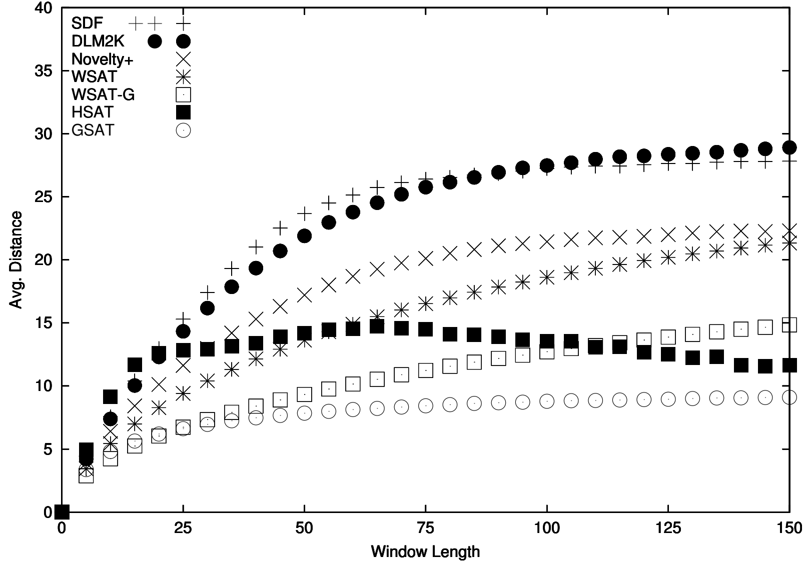


Figure 3.2: Same as 3.1, but results averaged over 100 runs on all 1000 uf100 problems from SATLIB.

point. We normalize this distance by dividing by the maximum distance and call it the *gap* of $\mathbf{a}_u$:

$$gap(\mathbf{a}_u) = \frac{1}{n} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_u)$$

where $\mathbf{A}_e$and $\mathbf{A}_u$ are the sets of explored and unexplored points, respectively, and $\mathbf{a}_u \in \mathbf{A}_u$.

If we now consider all unexplored points, the largest gap gives us a notion of how well our space has been covered by our search. If a large gap exists, then some unexplored point is far from any explored point, and we consider this to be poor coverage:

$$maxgap = \frac{1}{n} \max_{\mathbf{a}_u \in \mathbf{A}_u} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_u)$$

For the intuitive convenience of having a coverage measure that increases as coverage improves, we define *primal coverage* to be the maximum distance (n) minus the maximum gap:

$$primalCoverage = \frac{1}{n}[n - \max_{\mathbf{a}_u \in \mathbf{A}_u} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_u)]$$

Unfortunately, computing this value is NP-complete in the size of the set of explored points. It is equivalent to computing the *covering radius* of a set of codes [32]. In our research, the following approximation was used. The furthest point from any given assignment, **a,** is its complement, $\bar{\mathbf{a}}$. We therefore construct the set, $\bar{\mathbf{A}}_e$, which contains the

complement of every explored point. We use this set instead of $\mathbf{A}_u$ to compute our *approximate coverage*:

$$approxPrimalCoverage = \frac{1}{n}[n - \max_{\mathbf{a}_u \in \bar{\mathbf{A}}_e} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_u)]$$

Computing this measure is only quadratic in $|\mathbf{A}_e|$. The idea is that if we are only going to consider a small set of unexplored points, we will at least consider points known to be at a large (in fact, maximum) distance from at least one explored point. While far from at least one explored point, there is no guarantee that it will not be close to some other explored point and there may be other unvisited points farther from all explored points than any in $\bar{\mathbf{A}}_e$. Thus, our approximate measure gives a lower bound on the largest gap, which corresponds to an upper bound on coverage.

How well the approximation agrees with the exact version has not been evaluated. For our purposes, we seek practical metrics that give some predictive value. Therefore, we have confined our studies to how predictive the approximate metric is, without worrying about the properties of a metric we can't use in general. How predictive the "true" coverage is remains an interesting research question insofar as one might try to construct other approximations that may offer better predictive value than ours.

With coverage, we run into a problem similar to that encountered with depth. A successful solver may require only a short exploration to solve a problem whereas a poor solver may explore many points during its search. Clearly the relative coverage of two solvers that have sampled differing numbers of points is meaningless.

One way to deal with this would be to consider only a small number of explored points for each solver, but this complicates matters. Instead, we measure the *rate of coverage*, in other words, the rate at which the maximum gap diminishes. This is simply the (approximate) coverage at the end of the search, divided by the number of search steps:

$$approxPrimalCoverageRate = \frac{1}{nT}[n - \max_{\mathbf{a}_c \in \bar{\mathbf{A}}_e} \min_{\mathbf{a}_e \in \mathbf{A}_e} H(\mathbf{a}_e, \mathbf{a}_c)]$$

Once again, we present results based on the same experiments used for depth and mobility. Table 3.4 shows the average rate of approximate primal coverage and other measures for uf100-0953 and Table 3.5 shows the result averaged over all 1000 uf100 instances. Since even our approximation requires quadratic time to compute, only the first 10,000 assignments of each run are considered. The closeness of the rankings in terms of coverage and OEESS in both tables suggests that coverage is a good indicator of performance. However, if we look at the rank vs. rank matrix for coverage and OEESS (Table 3.6), we have a much weaker diagonal than that found for mobility. It seems that mobility is our best single indicator, with coverage and depth offering supporting evidence.

Solver	Av PCov	Av PMob	Av Depth	Av Steps	OEESS
GSAT()	.00003	6.0	2.19	500,000	n/a †
HSAT()	.00028	9.0	2.06	495,003	16,700
WSAT-G(.5)	.00011	10.1	4.20	29,661	15,442
WSAT(.5)	.00013	16.1	5.16	11,757	9,802
Novelty(.5)	.00013	19.0	3.75	4,916	4,563
Novelty+(.5, .01)	.00017	18.7	3.68	3.965	3.965
DLM(pars4)	.00047	28.6	5.40	2,182	1,984
SDF(.00085, .995)	.00053	29.7	3.03	1,192	1,174

Table 3.7: Average primal coverage rates (higher is better) obtained over the first 10,000 steps of each search run by various search procedures, along with depth, primal mobility, and search step measures. Results are averaged over 100 runs on problem uf100-0953 in SATLIB.

Solver	Av PCov	Av PMob	Av Depth	Av Steps	OEESS
GSAT()	.0002	8.8	1.62	74,595	10,823
HSAT()	.0007	15.8	1.35	73,510	2,503
WSAT-G(.5)	.0004	12.8	3.13	7,695	4,431
WSAT(.5)	.0005	18.9	3.49	3,582	2,828
Novelty(.5)	.0010	23.4	1.90	2,586	1,473
Novelty+(.5, .01)	.0010	23.6	1.93	2,224	1,393
DLM(pars4)	.0014	32.4	6.36	1,020	800
SDF(.00085, .995)	.0016	33.4	1.52	870	725

Table 3.8: Same as Table 3.7, but results averaged over 100 runs on all 1000 uf100 problems from SATLIB.

OEESS	Primal coverage rank			
rank	best 1	2	3	worst 4
best 1	.67	.22	.10	.01
2	.19	.43	.32	.06
3	.13	.32	.45	.11
worst 4	.00	.04	.14	.82

Table 3.9: Frequencies of (steps rank, coverage rank) pairs from among four search procedures, DLM, Novelty, Novelty+, and WSAT. Frequencies measured over the entire collection of uf100 problems in SATLIB (1000 problems in total).

3.4 Necessary vs. Sufficient

The above results suggest that no single metric of the three proposed can predict good performance, but give good evidence that poor scores on any single metric will lead to poor performance. That is, we have demonstrated that the three metrics probably represent *necessary* conditions for good performance, but have not demonstrated that they are sufficient. The following experiment attempts to establish this fact.

Several solvers (DLM, SDF, Novelty+, Novelty, WSAT, HSAT, and GSAT) were run on 2700 uf problems (the same uf sets described in Section 3.1). Each solver was run with three different parameter sets, giving a total of 21 methods. The three metrics and OEESS performance were recorded in each case. Solver results were averaged over 100 runs per instance. This gives us 56,700 tuples of the form <*depth, primal mobility, primal coverage, OEESS*>.

With the aim of determining the effect of one metric while holding the other two metrics "constant", the tuples were divided into buckets, with each bucket accepting a small range of value for the two metrics. So, to investigate depth, the tuples were split into 25 buckets based on mobility (0-4, 5-8, ..., 97-100 percentiles). These buckets were then further subdivided, each into 25 sub-buckets, based on coverage. The tuples in any one of these 625 buckets would have very similar scores for coverage and mobility. This allowed us to study the effects of depth while holding the other metrics constant. We measured the correlation between depth and performance (in OEESS) for each bucket and averaged over all buckets.

One potential problem with this approach would be under- or over-representation of the various methods in some buckets. If the tuples were spread uniformly over the buckets, we would expect $56,700/625 \approx 91$ tuples per bucket. The hope is that a wide variety of methods are represented in each bucket. To check this property, the entropy of the distribution over methods was measured for each bucket and the average taken over all buckets. Given that there are 21 methods, the maximum entropy for any bucket (i.e. a uniform distribution) would be 4.39 bits. These measurements were similarly performed for mobility (holding depth and coverage constant) and coverage (holding depth and mobility constant). The results are shown in Table 3.10.

	Depth	Primal Mobility	Primal Coverage
Correlation	0.21	-0.31	-0.49
Entropy	2.20	2.07	1.23

Table 3.10: Average correlation of each metric with OEESS holding the other two metrics roughly constant. Numbers were averaged over 625 buckets of controlled tuples obtained by running 21 methods over 2700 uf instances from SATLIB. The average entropy of the method distributions in buckets is also reported.

The results lend support to the notion that these metrics may be sufficient. The correlations in each case have the correct direction (positive with depth, negative with mobility

and coverage). The entropies show that there was, on average, reasonable representation of the various methods in a given bucket.

3.5 Dual Space Metrics

The mobility and coverage metrics described above are *primal space metrics*, characterizing the behaviour of the primal variables during the search. An obvious extension to local search for SAT is applying local search to arbitrary constraint satisfaction problems CSPs. General CSPs typically have a set of discrete variables (often non-boolean) and a set of constraints, each of which rules out some assignment(s) to the variables. SAT is a special case, where the clauses are the constraints and all variables are boolean.

Our metrics often rely on a distance measure between two assignments. The fact that the primal variables in SAT are boolean makes Hamming distance a convenient choice. If we now consider generalizing our metrics to arbitrary CSPs we immediately run into problems because the variable can now have non-boolean domains. While some CSP variable domains have obvious choices for distance measures (e.g. ordinal domains), many cases do not. Some more or less arbitrary choice of distance measure would be required to apply our notions of mobility and coverage. However, the dual space variables of any CSP (i.e. the constraints) are always boolean in a local search environment, since the constraint is either satisfied or not. It therefore seems natural to consider whether our metrics offer good predictive value in the dual space as well as in the primal.

Strictly speaking, we have already presented one *dual space metric*, namely *depth*. Average depth is a very crude summary of the behaviour of the dual variables during the search, condensing all of the activity into a single scalar value. We will now consider some more informative metrics, *dual mobility* and *dual coverage*.

3.5.1 Dual Mobility

The dual mobility metric is almost identical to the primal, but uses vectors of constraint states, $\mathbf{c}^{(t)}$, instead of variable assignments. Here, the maximum Hamming distance for a single step is the number of constraints, m. While not as natural a choice as in the primal case, we nonetheless set w to equal m in all of our experiments.

$$dualMobility(w) = \frac{1}{T-w} \sum_{t=1}^{T-w} \frac{H(\mathbf{c}^{(t)}, \mathbf{c}^{(t+w)})}{min(w,m)}$$

The intuitive interpretation is similar as well. If the search persistently revisits a small set of constraint states, it is likely to be trapped. In fact, it may have considerable mobility in the primal space but still be oscillating among a small number of constraint states, since there may be many variables irrelevant to the constraints available to be flipped.

3.5.2 Dual Coverage

Again, the *dual coverage* metric is virtually identical to the primal version, substituting the sets of visited constraint configurations, $\mathbf{C}_e$, and the set of complements, $\bar{\mathbf{C}}_e$.

$$approxDualCoverageRate = \frac{1}{mT}[m - \max_{\mathbf{c}_c \in \bar{\mathbf{C}}_e} \min_{\mathbf{c}_e \in \mathbf{C}_e} H(\mathbf{c}_e, \mathbf{c}_c)]$$

Effective local search solvers typically have only a small number of unsatisfied constraints at any given time (i.e. good *depth*). However, the search frequently explores many distinct sets before discovering a solution. While the solver is unlikely to visit states where many constraints are unsatisfied (i.e. gaps will be predominantly large), it seems likely that coverage of the small sets will be a good indicator of success. Therefore, we expect the dual coverage rates to be small but still informative.

3.5.3 Dual Metrics: Experimental Results

To establish the value of dual space metrics, we compare them directly with the primal versions in a fashion similar to the experiments already presented. We consider GSAT, HSAT, WalkSAT, Novelty+, and ESGint.[5] Each problem instance was run 100 times and the results averaged over all runs.[6] The solvers were allowed 500,000 steps and no restarts. All solvers were run with default parameters (see Appendix C).[7] We report all five metrics, the percentage of runs that failed to find a satisfying solution (Fail% - lower is better), average steps, and OEESS.[8] Results are given in Tables 3.11-3.20.

One clear conclusion from these results is that if the failure rate is high, then typically depth is quite small, and mobility and coverage (primal or dual) are likewise small. This indicates that the solver spent most of its time trapped in some local minimum. GSAT falls prey to this often, and HSAT, which employs a Tabu-like strategy of preferring the least recently flipped variable when breaking ties, suffers from it only slightly less.

Generally speaking, ranking the methods by the primal and the dual versions of a metric gives the same results. Similarly, high values of primal or dual mobility and coverage are good predictors of performance, in terms of average numbers of steps, and also in terms of OEESS (when high failure rates do not render the estimate meaningless). This seems to correspond to the previous study's view that good depth, mobility and coverage are necessary, but it now appears we can consider either the primal or the dual versions.

Unlike the earlier study, only one data set shown here corresponds to random instances (Table 3.20). The rest are structured instances. The broader nature of this study strongly

[5]See Chapter 5, more specifically, Section 5.7.2, for details of this algorithm.

[6]Except for primal and dual coverage results. Coverage is expensive to compute so it was only averaged over the first 10,000 steps in each of 20 runs.

[7]Except Novelty which was run with $p_{secondbest} = 0.5$. This setting explains some disappointing Novelty results, but the purpose here is to evaluate the metrics, not to compare the solvers against each other.

[8]There are some very low estimated expected steps in the results for solvers which have performed very poorly. The estimates are very unreliable when failure rates are high. We distinguish these cases with the following symbol:†.

suggests that the predictive value of all the metrics is quite general, and not limited to HR instances.

In more detail we note the following interest specifics:

- Tables 3.11 and 3.18 both show the dual metrics giving a better general prediction of performance.
- Table 3.19 is the only example where the dual version (of mobility in this case) is significantly less predictive. However, performance appears to be more related to coverage in this instance, and the dual coverage is nicely predictive.
- Table 3.20 shows HSAT with high primal coverage, but poor performance. The dual coverage much better reflects performance. Mobility is similar.

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	94	470002	829†	0.0022	0.0502	0.014	0.00041	0.00027
HSAT	98	490000	1499†	0.0023	0.0504	0.009	0.00078	0.00049
WalkSAT	0	1072	997	0.0050	0.2235	0.156	0.00062	0.00023
Novelty+	0	1932	1761	0.0047	0.1899	0.110	0.00058	0.00024
ESGint	0	595	595	0.0034	0.2282	0.173	0.00219	0.00096

Table 3.11: Problem set "ais06" : 61v, 518c, All-Interval Series (SATLIB)

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	100	500000	500000	0.001178	0.025	0.0017	2.05e-05	9.6e-06
HSAT	100	500000	500000	0.001181	0.026	0.0012	2.10e-05	10.0e-06
WalkSAT	0	29362	10800	0.002816	0.170	0.0447	3.66e-05	12.9e-06
Novelty+	0	44385	35131	0.002712	0.149	0.0322	4.95e-05	18.7e-06
ESGint	0	4985	4954	0.001764	0.178	0.1189	18.15e-05	64.7e-06

Table 3.12: Problem set "ais08" : 113v, 1520c, All-Interval Series (SATLIB)

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	79	395007	240†	0.011	0.07	0.048	0.0009	0.0004
HSAT	79	395006	200†	0.009	0.06	0.045	0.0014	0.0006
WalkSAT	0	403	249	0.014	0.22	0.168	0.0023	0.0009
Novelty+	0	598	401	0.016	0.20	0.160	0.0024	0.0011
ESGint	0	96	96	0.006	0.32	0.206	0.0049	0.0018

Table 3.13: Problem set "anomaly" : 48v, 261c, Blocks World 3 Blocks, 3 Steps (SATLIB - Kautz and Selman)

† OEESS results are unreliable when failure rates are high.

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	90	450009	1308†	0.0059	0.04	0.022	0.00015	0.00007
HSAT	88	440007	622†	0.0060	0.05	0.026	0.00037	0.00017
WalkSAT	0	1085	1085	0.0080	0.19	0.181	0.00091	0.00035
Novelty+	0	1362	847	0.0086	0.20	0.174	0.00046	0.00018
ESGint	0	273	271	0.0042	0.31	0.213	0.00163	0.00060

Table 3.14: Problem set "medium" : 116v, 953c, Blocks World 5 Blocks, 4 Steps (SATLIB - Kautz and Selman)

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	100	500000	500000	0.00241	0.012	0.0039	2.42e-05	1.06e-05
HSAT	100	500000	500000	0.00211	0.014	0.0037	5.74e-05	2.27e-05
WalkSAT	0	17976	17389	0.00345	0.092	0.1023	4.24e-05	1.69e-05
Novelty+	0	18569	18478	0.00353	0.090	0.0875	4.12e-05	1.67e-05
ESGint	0	2652	2629	0.00396	0.234	0.1999	20.36e-05	8.10e-05

Table 3.15: Problem set "bw_large.a" : 459v, 4675c, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman)

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	100	500000	500000	0.00292	0.008	0.0034	1.95e-05	8.81e-06
HSAT	100	500000	500000	0.00285	0.010	0.0032	1.99e-05	9.08e-06
WalkSAT	3	133271	132120	0.00134	0.072	0.0225	2.86e-05	9.03e-06
Novelty+	18	253471	253471	0.00130	0.067	0.0148	2.93e-05	9.31e-06
ESGint	0	13213	9068	0.00198	0.157	0.1365	5.10e-05	16.14e-06

Table 3.16: Problem set "logistics.a" : 828v, 6718c, Logistics, 8 Packages, 11 Steps (SATLIB - Kautz and Selman)

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	99	495003	39500†	0.00210	0.0184	0.007	2.39e-05	1.10e-05
HSAT	100	500000	500000	0.00206	0.0181	0.005	2.41e-05	1.12e-05
WalkSAT	0	21831	20852	0.00251	0.0926	0.127	3.65e-05	1.50e-05
Novelty+	0	19395	19155	0.00261	0.0934	0.130	3.27e-05	1.33e-05
ESGint	0	2971	2971	0.00300	0.2297	0.216	16.42e-05	6.57e-05

Table 3.17: Problem set "huge" : 459v, 7054c, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman)

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	35	186597	120342	0.00138	0.257	0.103	0.0007	0.00032
HSAT	49	243184	148152	0.00144	0.224	0.108	0.0014	0.00044
WalkSAT	0	530	527	0.01212	0.230	0.186	0.0016	0.00095
Novelty+	0	1268	1268	0.00604	0.199	0.213	0.0012	0.00078
ESGint	0	344	316	0.00268	0.180	0.205	0.0014	0.00100

Table 3.18: Problem set "ii08" : 14 instances, Inductive Inference (DIMACS - Resende)

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	96.0	485223	472390	0.0128	0.29008	0.016	8.10e-05	2.42e-05
HSAT	99.4	497004	406017	0.0136	0.36490	0.009	7.85e-05	1.48e-05
WalkSAT	9.8	163143	83621	0.0321	0.13042	0.063	7.28e-05	2.50e-05
Novelty+	0.2	59040	36949	0.0215	0.10570	0.047	14.91e-05	5.20e-05
ESGint	0.0	9429	8774	0.0139	0.13048	0.053	37.07e-05	10.53e-05

Table 3.19: Problem set "par08" : 5 instances, Unsimplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford)

Solver	Fail%	Av Steps	OEESS	Av Depth	Av PMob	Av DMob	Av PCov	Av DCov
GSAT	90.0	456413	276154	0.0024	0.068	0.004	3.54e-05	0.75e-05
HSAT	88.3	441720	58028	0.0024	0.099	0.013	10.35e-05	1.90e-05
WalkSAT	1.5	40660	34723	0.0075	0.133	0.034	9.54e-05	2.40e-05
Novelty+	3.1	59397	36017	0.0061	0.101	0.031	8.11e-05	2.23e-05
ESGint	0.1	18581	13108	0.0066	0.255	0.054	20.24e-05	4.04e-05

Table 3.20: Problem set "uf250" : 100 instances, 250v, 1065c, hard, random 3-SAT (SATLIB)

3.6 Other Metrics Research

Attempts to evaluate local search performance are not wholly new, although they are typically formulated in order to automatically tune search parameters.

Cha and Iwama use a notion similar to mobility in analysing the behaviour of weighted GSAT [15][16] by explicitly counting how many previously unvisited assignments are reached in a given period of the search. They were working with small instances where such a measure is not unreasonable. They also plot the Hamming distance between assignments and the eventual solution.

In an attempt to tune the noise parameters for algorithms such as WalkSAT and Novelty, both of which contain a random walk component, McAllester et al. [80] use two metrics which they demonstrate to give similar values across six different algorithms (WalkSAT, Novelty and variations of these) when each algorithm is optimally tuned. That is, there is a value for these metrics which corresponds to optimal performance, regardless of the method. The two metrics are:

- average number of unsatisfied clauses (noise level invariant)
- mean over variance of number of unsatisfied clauses (optimality invariant)

The first is just like our depth measurement. Its invariant property means that if one knows the average depth for an algorithm when optimally tuned, one could obtain optimal tunings for some other algorithm by trying to obtain the same average depth. The second metric is a more complex view of the behaviour of the objective function that manages to capture some notion of mobility. If the variance in objective value is low, it is likely that the search is trapped. The evidence presented suggests that nearly minimizing this ratio will provide an optimal noise tuning. While this research focuses on the issue of how to tune noise parameters in particular, it is similar in spirit to our own by attempting to characterize behaviour over several different algorithms.

In work on *reactive search* for MAX-SAT [4], Batitti notes that there is a tradeoff between *bias* (essentially the same as depth) and *diversification* (similar to mobility[9]). He examines how various features found in local search methods for SAT and MAX-SAT tradeoff these two metrics in an effort to understand what features are effective. Additionally, he uses the diversity metric to automatically tune the list length parameter of a Tabu search method.

Something like mobility and depth are proposed in [117], along with several other metrics. The goal in this case was to tune the many parameters that control DLM-99 automatically. Among the metrics they considered for this purpose were:

- least number of unsatisfied clauses (LUC)

[9] The diversification measure used is the average over all explored assignments of the assignments' Hamming distances from the initial assignment. This is an interesting measure but clearly has some problems since a search trapped at some distance from the initial assignment will appear to have high diversity.

- average number of unsatisfied clauses (AUC)
- Hamming distances computed over a history (DIS)

There were several other metrics relating to clause weights and other features of DLM. These metrics are not independent of the solver, so we do not consider them in the same light.[10] Clearly, AUC is just like our average depth measure (although without the initial skipped steps). DIS, although not clearly explained, appears to be similar to mobility, and is motivated in a similar manner by the authors. Trial and error experimentation led the authors to conclude that LUC and DIS were valuable measures (along with two relating to clause weights). They use these measures after a short run to decide which of five parameters sets to use for DLM-99, according to a hand-crafted heuristic. They did not apply these measures to other solvers.

It is interesting that they found LUC more informative than AUC, and it might be worth exploring the predictive value of LUC. The merits of DIS agree well with our own results.

3.7 Conclusion

Local search is poorly understood, despite more than a decade of intensive exploration of the approach as it relates to SAT. Our work on metrics sheds some light on the subject, capturing notions intuitively appreciated by many researchers, but systematically exploring them for the first time with a view to understanding solvers in general rather than tuning or analyzing a single body of research. The similarity between primal and dual metrics is intriguing, and work must be done to understand their relationship better. In particular, it is important to examine cases where they do not agree, since this disagreement can potentially be used to detect some failing in a solver. Beyond that, theoretical results for solvers in terms of these metrics may help establish the potential, or limitations, of local search.

Practically, these metrics can assist in the analysis of existing algorithms and the development of new ones by allowing the researcher to summarize behaviour and quickly characterize difficulties on particular classes on instances. Furthermore, some of these metrics are cheap to compute and can be applied directly in solvers to make them adaptive.

The study of these metrics has substantially influenced our understanding and research agenda. The observations made directly influenced the work we will present next, motivating features that led to high-performance solvers and providing tools for evaluating our progress along the way.

[10] It is worth noting that some of these DLM-related measures have some relationship to counting the number of times the search reached a local minimum, something we have not examined systematically but which seems particularly interesting to clause weighting methods (e.g. DLM, SDF, and ESG). In particular, they consider the ratio of weight updates to search steps (dual steps over primal steps in ESG parlance), a measure that our own informal experimentation has shown to have a clear relationship to optimal parameter settings for ESG. However, our own limited investigation has not led to any reliable way to exploit this metric.

Chapter 4

Local Search: SDF

I had little object – certainly no hope – in these researches, but a vague curiosity prompted me to continue them.
- **The Pit and the Pendulum**, Edgar Allan Poe

The metrics research inevitably suggested approaches to local search that led us to explore an interesting collection of features, explicitly addressing various aspects of solver behaviour. We constructed a local search method called *SDF* (for *smoothed descent and flood*) [99, 100]. SDF is a primal-dual Lagrangian (or clause-reweighting) method, similar to DLM, but substantially simpler. SDF was based on observations of existing local search methods and intuitions about what makes them succeed or fail. The basic goals for the algorithm were:

- rapid descent in the objective function to reach regions likely to hold solutions
- maintaining good values in the objective function (i.e. depth)
- rapid, widespread exploration of the search space (i.e. mobility and coverage)

As stated, the basic architecture is a primal-dual Lagrangian method (see Section 2.5.9.1) using local search in the objective function and updating weights whenever a local minimum is encountered. SDF has several features, some innovative and others found in existing solvers:

- primal-dual Lagrangian method
- smoothed objective function for primal search
- multiplicative weight updates
- flooding (min-fill) weight update step sizes
- weight smoothing

These features will now be examined in detail.

4.1 Smoothed Objective Function

Let us start by considering how we might improve depth. We noted that the objective functions used by GSAT and WalkSAT are based on counting unsatisfied clauses, and that other solvers go no further than weighted counts in differentiating clauses. No consideration is given to how a clause is satisfied; it is simply satisfied or not.

A clause of length *k* is satisfied if 1 or more of its literals are satisfied. This means that there are $2^k - 1$ ways in which that clause may be satisfied. Clearly, there are many other possible objective functions that consider more information about the satisfying assignment. SDF makes use of one such alternative objective function by considering the number of literals that satisfy each clause. This provides a form of tie-breaking amongst alternatives that appear identical under the GSAT objective.

All things being equal, SDF prefers moves that will result in clauses being satisfied by multiple literals. There are two intuitions behind this. The first is that, by differentiating assignments that the GSAT objective treats as identical, we can reduce plateau walking and improve our chances for descent. The second is that redundantly satisfied clauses will be more robust to subsequent variable flips. If a length 3 clause is satisfied by all three of its literals, it will take at least three flips to unsatisfy it. Thus we hope to preserve our satisfied clauses and thus preserve our depth in the face of changing assignments.

Let $satlit(\mathbf{c}, \mathbf{a})$ be the number of literals in clause $\mathbf{c}$ satisfied by assignment $\mathbf{a}$. Then, given a *k*-CNF formula with m clauses, the *smoothed* objective function is given by:

$$g_{smooth}(\mathbf{a}) = \sum_{i=1}^{m} \sum_{j=satlit(\mathbf{c}_i,\mathbf{a})}^{k} m^{k-j} \tag{4.1}$$

recalling from Section 2.1.3 that $satlit(\mathbf{c}_i, \mathbf{a})$ is the number of literals satisfying the *i*th clause under assignment $\mathbf{a}$ and that k is the number of literals in the longest clause.

This function is designed so that improvements to less satisfied clauses always outrank improvements to more satisfied clauses. Thus, if one choice will satisfy a previously unsatisfied clause (a $0 \rightarrow 1$ satisfying literal change) and another choice will improve all other clauses with a $1 \rightarrow 2$ literal change, the former choice will still be preferred. Similarly, if one choice gives a $1 \rightarrow 2$ literal improvement for a single clause and another choice offers a $2 \rightarrow 3$ literal improvement to all other clauses, the former will be preferred.

> For example, suppose we have $k = 3$ and $m = 4$, with the following clauses:
>
> $$(x_1 \vee x_2 \vee x_3)$$
> $$(x_4 \vee x_5 \vee x_6)$$
> $$(x_4 \vee x_5 \vee x_7)$$
> $$(x_4 \vee x_5 \vee x_8)$$

The current assignment, $\mathbf{a} \leftarrow \{\bar{x}_1, \bar{x}_2, \bar{x}_3, x_4, \bar{x}_5, \bar{x}_6, \bar{x}_7, \bar{x}_8\}$, has one clause unsatisfied and the other three are satisfied by exactly one literal each. The objective value of this assignment is $g_{smooth}(\mathbf{a}) = 85 + 21 + 21 + 21 = 148$.

An assignment that satisfies the first clause but does not change the others, $\mathbf{a}^{'} \leftarrow \{x_1, \bar{x}_2, \bar{x}_3, x_4, \bar{x}_5, \bar{x}_6, \bar{x}_7, \bar{x}_8\}$, has a value $g_{smooth}(\mathbf{a}^{'}) = 21 + 21 + 21 + 21 = 84$, an improvement of 64 over $\mathbf{a}$.

An assignment that leaves the first clause unsatisfied but makes the others satisfied by two literals each, $\mathbf{a}^{''} \leftarrow \{\bar{x}_1, \bar{x}_2, \bar{x}_3, x_4, x_5, \bar{x}_6, \bar{x}_7, \bar{x}_8\}$, has a value $g_{smooth}(\mathbf{a}^{''}) = 85 + 5 + 5 + 5 = 100$, an improvement of 48 over $\mathbf{a}$.

The smoothed objective function contains many fewer plateaus than the GSAT/WalkSAT objective (hence the term *smoothed*). Where a plateau would have required a random walk, g_{smooth} usually offers a direction. SDF, descending in this richer objective function, tends to achieve lower numbers of unsatisfied clauses before hitting a local minimum in g_{smooth}. Figure 4.1 shows the result of an experiment demonstrating that descending in g_{smooth} results in better average depth (as measured by g_{unsat}) than simply descending in g_{unsat} itself.

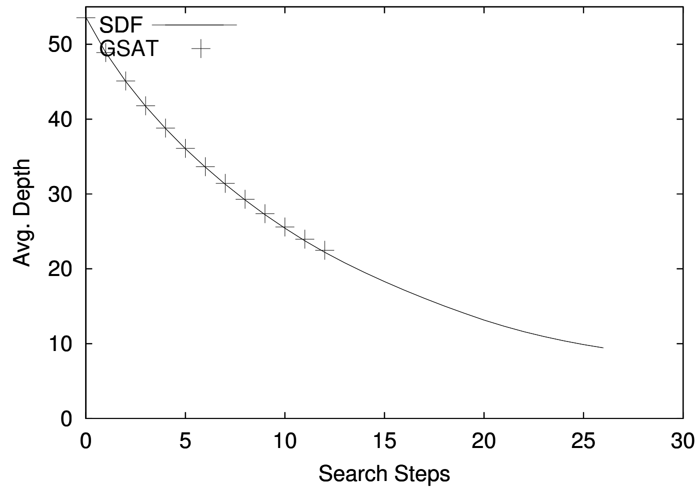


Figure 4.1: Average number of unsatisfied clauses (depth in g_{unsat}) achieved before reaching a local optimum or plateau. Results are obtained by running initial descents in g_{unsat} and g_{smooth} on uf100-0953 until the first strict local minimum or plateau point is reached, and then reporting the average g_{unsat} value at a given time point if at least 95 of 100 runs have successfully descended that many steps.

An inconvenient property of this objective is the growth in objective values as instances become larger (exponential in k and potentially high-polynomial in m.). In integer implementations this rapidly leads to overflow. Floating-point implementations are, naturally,

slower. A useful generalization of this objective function limits the number of literals we consider relevant. We can set a literal cap, κ, beyond which we no longer differentiate assignments. Thus, if $\kappa = 3$, clauses with 3 or more satisfied literals are considered to have equal value. The resulting κ-smoothed objective function looks like this:

$$g_{\kappa-smooth}(\mathbf{a}) = \sum_{i=1}^{m} \sum_{j=cappedsatlit(\mathbf{c}_i,\mathbf{a})}^{\kappa} m^{\kappa-j} \tag{4.2}$$

where $cappedsatlit(\mathbf{c}_i, \mathbf{a}) = \begin{cases} \kappa & \text{if } satlit(\mathbf{c}_i, \mathbf{a}) > \kappa \\ satlit(\mathbf{c}_i, \mathbf{a}) & \text{otherwise} \end{cases}$

This objective function allows us to parameterize the importance of redundant satisfaction and control the overflow problem. Since many natural and transformed SAT instances are comprised mostly of very short clauses with just a few long ones, we can maintain the smoothing quality over most clauses without suffering the numerical issues introduced by the few long clauses. Note that when $\kappa = 1$, $g_{\kappa-smooth}$ is effectively identical to the GSAT heuristic, g_{unsat}.

While SDF uses a weighted version of g_{smooth}, subsequent study has revealed that it usually offers no improvement in terms of search steps and often increases the number of steps required. It appears that any benefit that may accrue from the more robust assignments to variables or the improved depth is offset by the extra steps taken during descents in order to build the redundancy. SDF typically executes more search steps before hitting a local minimum. Interestingly, Gent and Walsh observed that very greedy descent procedures were not particularly important to the success of the local searchers they explored [43], a notion supported by this aspect of the SDF research. Our later reformulation of SDF, ESG (see Chapter 5) drops the smoothed objective function.

4.2 Weighted Objective

In our metric experiments, DLM exhibits excellent mobility. This seems natural since Lagrangian, indeed clause weighting approaches generally, explicitly try to escape local minima and hence improve mobility. Similarly, SDF is a primal-dual Lagrangian method, with Lagrange multipliers, applied to each clause. This approach is combined with the g_{smooth} objective function already discussed to produce the complete SDF objective function:[1]

$$g_{SDF}(\mathbf{a}, \lambda) = \sum_{i=1}^{m} \lambda_i \sum_{j=satlit(\mathbf{c}_i,\mathbf{a})}^{k} m^{k-j} \tag{4.3}$$

Note that when viewed as a Lagrangian, g_{SDF} has a zero objective function and consists only of the penalty term (unlike DLM). We will elaborate on this point in the next chapter.

[1] We only consider the uncapped version here.

4.3 Multiplicative Updates

SDF increases the weight of all unsatisfied clauses when it reaches a local minimum ($g_{SDF}(\mathbf{a}) \leq g_{SDF}(\mathbf{a}'), \forall \mathbf{a}' \in N(\mathbf{a}), \mathbf{a}' \neq \mathbf{a}$). An important, possibly the most important, difference between SDF and earlier clause weighting methods is the use of *multiplicative updates*. The earlier methods used *additive updates*, typically of the form:

$$\lambda_i^{(t+1)} = \lambda_i^{(t)} + \omega$$

where ω is a parameter determining the size of the update.

As Frank observed (see Section 2.5.8.4 and [34]), when weights become large, additive updates become increasingly ineffective because the increments are small compared with the size of the weights involved. It takes a great many increments to change the relative importance of two clauses. The mobility advantages of such methods are compromised as a result.

SDF uses multiplicative updates of the form:

$$\lambda_i^{(t+1)} = \alpha \lambda_i^{(t)} \tag{4.4}$$

where $\alpha > 1$ (one could select α as a parameter but see the following Section 4.4 for the approach used in SDF).

Inspired by machine learning algorithms such as *weighted majority* [74], the idea is that multiplicative updates allow the weight profile to adapt more quickly because the updates scale with the weights themselves. This faster adaptation may explain SDF's superior mobility (see the results from Chapter 3).

At each reweighting, SDF renormalizes the weights of all clauses so they sum to 1. This is simply a technical convenience.

4.4 Flooding

Flooding is a specific implementation of clause-weighting that adapts to local conditions. Most clause reweighting schemes (e.g. WGSAT, DLM) use a fixed step size. If the steps are too small, several reweights may be required to escape a local minimum. Excessively large steps may exaggerate the importance of the currently unsatisfied clauses, pushing the solver into poor regions. Flooding computes a step size to achieve a "minimum fill", increasing the weights just enough to create a new search direction, with an objective value smaller than the current assignment by some fixed factor, δ, which is specified as a parameter.

$$\begin{aligned} &\min \alpha \\ s.t. \quad &\exists \mathbf{a}' \neq \mathbf{a} \wedge g_{SDF}(\mathbf{a}, \lambda^{(t+1)}) \quad \geq \alpha g_{SDF}(\mathbf{a}', \lambda^{(t+1)}) \end{aligned}$$

In practice, this is implemented by looping through all variables that occur in currently unsatisfied clauses, computing for each variable x_i the minimum α_{x_i} required to make flipping x_i an improvement, and then using the smallest such α_{x_i}.

It should be noted that Cha and Iwama tried a "minimum-fill" reweighting scheme but found no substantial differences between this and several other schemes [15].

4.5 Weight Smoothing

Since the weight updates performed at local minima never reduce clause weights (at least, not relative clause weights), there is no way for a clause to diminish in importance once its weight has been raised. Frank introduced weight decays (see Section 2.5.8.4) initially to deal with the shortcomings of additive updates in large weight profiles (a problem better handled by SDF's multiplicative updates), but subsequently observed that clause weights seem to have a chiefly "local" relevance. If the search moves to a new region, the weights may no longer be appropriate. Thus, weight decays serve to let the solver "forget" priorities learned elsewhere in the search space.

After the weight updates described above, SDF uses weight smoothing, where the satisfied clauses are shrunk towards the mean of their weights. This has numerical properties attractive for implementing the weight updates (namely that the total weight of all clauses remains normalized). The smoothing update is:

$$\begin{aligned} \bar{\sigma} &= \frac{1}{|\mathbf{SAT}(\mathbf{a})|} \sum_{i \in \mathbf{SAT}(\mathbf{a})} \lambda_i \\ \lambda_i' &= (1-\rho)\bar{\sigma} + \rho\lambda_i, \forall i \in \mathbf{SAT}(\mathbf{a}) \end{aligned}$$

where ρ is the smoothing parameter controlling the rate at which weights are decayed toward the mean (e.g. if $\rho = 0$, the weights would instantly be set to the mean, whereas if $\rho = 1$, the weights are not smoothed at all).

Smoothing is a double-edged sword with respect to mobility. Too much smoothing will allow the solver to slip back into local minima that are forgotten too quickly. Too little will make the solver systematically avoid some assignments, potentially damaging both mobility and coverage. Nonetheless, smoothing is almost always important to maintain performance.

The full SDF procedure is shown in Algorithm 15.

4.6 SDF Performance

SDF offers excellent performance in terms of search steps, but its actual runtime performance is quite poor, often as much as six times slower than DLM. This is due to several causes: the use of floating-point weights, the complicated objective function, smoothing to the mean of the weights, and correction loops run periodically to combat the effects of numerical drift.[2] We report runtimes for SDF but focus on the step performance. The

[2]The correction loops are not a feature of the algorithm but were used to correct for numerical drift in certain cached values. After a fixed number of reweightings, certain cached values would be recomputed from scratch,

Algorithm 15 SDF

SDF()

1. Randomly generate initial full assignment, $\mathbf{a}^{(0)}$
2. Set step counter, $t \leftarrow 0$
3. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success
4. If the termination criteria have been met, exit with failure
5. Neighbourhood, $N(\mathbf{a})$ has n assignments, each obtained by flipping a single variable in $\mathbf{a}^{(t)}$
6. Evaluate assignments in $N(\mathbf{a})$ according to the function, $g_{SDF}(\mathbf{a}, \lambda) = \sum_{i=1}^{m} \lambda_i \sum_{j=satlit(\mathbf{c}_i,\mathbf{a})}^{k} m^{k-j}$
7. If no neighbour offers an improvement over the current objective value (non-strict local minimum), then we make a *dual step* as follows
   (a) reweight and normalize
   (b) smooth
8. Choose $\mathbf{a}^{(t+1)}$ to be the neighbour with the smallest value. If there are "ties" (several assignments with the same value), choose randomly among the smallest.
9. Advance the step counter, $t \leftarrow t + 1$
10. Goto step 3

main intent of the research was to demonstrate that our metrics led to the development of a solver that effectively reduced search steps. ESG, the successor to SDF, addresses the runtime issue and we defer comprehensive runtime comparisons for now. However, it is worth noting that ESG, which discards some features of SDF to improve runtime, typically requires slightly more search steps than SDF. Indeed, SDF exhibits the best overall search step performance on a wide variety of instances.

We here show the results reported in [100]. Tables 4.1 to 4.8 show average steps, OEESS, percentage of runs that failed (undecided), and average runtime per search step in milliseconds ("steps" refer to primal steps in the case of primal-dual solvers). Table captions show the name of the problem set along and a brief description, the step limit, and the number of repetitions for each instance. Results are averaged over all repetitions of each instance in the problem set and are run for WalkSAT, Novelty, Novelty+, DLM, and SDF. Results are listed in a fixed order by algorithm to make it easier to read the tables. Parameters are indicated for each solver in parentheses after their names and any unspecified parameters used default values. These parameters are specified in the same order as they are listed in Appendix C. Tunings were typically obtained by hand-tuning on a subset of instances, where multiple instances from a class were available.[3]

4.7 Conclusion

SDF has fulfilled its role as a testbed for a variety of ideas inspired by the metrics research. Some of these ideas survive the test of experimentation and some clearly must be reconsidered or discarded. Exhibiting excellent search step performance, SDF also corroborates the predictive value of the metrics, as can be seen by the results in Chapter 3.

However, SDF is far too inefficient to compete with other solvers. The next phase of the research demonstrates that many features and properties of SDF can be preserved while offering state-of-the-art runtime performance, and that some of the mechanisms, heretofore motivated only by intuitions, actually have more principled underpinnings.

an expensive operation. The number of loops was never varied in experiments. Our more recent local search algorithms no longer require such correction loops.

[3]Our later results, for ESG and other algorithms, dispense with per class tunings as unrealistic, and instead use a default setting for all classes, obtained by tuning a small selection of instances from a handful of classes.

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	42,795	34,814	.28	2.7
Novelty(.6)	17,979	13,946	.14	2.8
Novelty+(.6, .01)	17,134	12,540	.02	2.8
DLM(pars4)	9,571	8,314	0	4.8
SDF(.0002)	7,175	6,474	0	20.9

flat100 : 100 instances, 300v, 1117c, satisfiable, Flat 3-Colourable Graphs, 100 vert, 239 edges, (SATLIB) [Max Steps: 50,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	75,135	66,232	1.6	2.9
Novelty(.6)	32,727	24,616	.75	3.0
Novelty+(.5, .01)	36,739	26,066	.78	2.8
DLM(pars4)	26,182	21,506	0	5.1
SDF(.0002)	15,169	13,293	0	21.4

flat125 : 100 instances, 375v, 1403c, satisfiable, Flat 3-Colourable Graphs, 125 vert, 301 edges, (SATLIB) [Max Steps: 100,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	146,163	129,407	8.3	2.9
Novelty(.5)	92,859	60,827	3.6	2.9
Novelty+(.5, .01)	81,276	57,698	2.7	2.9
DLM(pars4)	69,779	54,637	.98	5.5
SDF(.0001)	36,304	29,327	.22	25.3

flat150 : 100 instances, 450v, 1680c, satisfiable, Flat 3-Colourable Graphs, 150 vert, 360 edges, (SATLIB) [Max Steps: 200,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	299,847	298,091	36	3.1
Novelty(.5)	235,679	204,691	26	3.0
Novelty+(.6, .01)	238,738	224,526	27	3.3
DLM(pars4)	280,401	242,439	31	6.2
SDF(.0001)	140,005	112,020	7	26.1

flat200 : 100 instances, 600v, 2237c, satisfiable, Flat 3-Colourable Graphs, 200 vert, 479 edges, (SATLIB) [Max Steps: 500,000 Reps: 100]

Table 4.1: SDF comparison for flat* instances

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	656	496	0	3.6
Novelty(.7)	633	230	.07	3.3
Novelty+(.7, .01)	259	235	0	3.8
DLM(pars4)	186	160	0	6.6
SDF(.003)	156	140	0	11.6

1000 instances, 50v, 218c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 1000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	1818	1365	0	3.7
Novelty(.6)	1493	635	.15	3.3
Novelty+(.7, .01)	661	605	0	3.8
DLM(pars4)	515	404	0	6.1
SDF(.0015)	435	389	0	13.4

100 instances, 75v, 325c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 2000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	3622	2813	0	3.8
Novelty(.6)	2788	1273	.23	3.4
Novelty+(.6, .01)	1581	1261	0	3.7
DLM(pars4)	1020	800	0	6.3
SDF(.00085)	864	725	0	15.9

uf100 : 1000 instances, 100v, 430c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 5000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	8966	5452	.01	3.9
Novelty(.7)	4892	2785	.32	3.7
Novelty+(.7, .01)	3314	2606	0	4.0
DLM(pars4)	2096	1573	0	6.1
SDF(.0006)	1879	1505	0	17.6

uf125: 100 instances, 125v, 538c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 10,000 Reps: 100]

Table 4.2: SDF comparison for small uf instances

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	14,353	8027	.3	4.1
Novelty(.6)	7551	4344	.3	3.8
Novelty+(.6, .01)	6075	4817	0	4.0
DLM(pars4)	3263	2455	0	6.3
SDF(.00065)	3312	2533	0	18.5

uf150 : 100 instances, 150v, 645c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 20,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	27,493	19,289	.66	4.2
Novelty(.6)	14,342	8554	.31	4.1
Novelty+(.6, .01)	13,820	8453	.17	4.2
DLM(pars4)	6819	4923	0	6.4
SDF(.0005)	7228	5491	0	20.3

uf175 : 100 instances, 175v, 753c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 50,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	40,323	30,859	2.7	4.3
Novelty(.6)	26,463	20,437	1.9	4.3
Novelty+(.6, .01)	26,479	21,503	1.9	4.3
DLM(pars4)	13,316	9,020	.08	6.6
SDF(.0003)	14,962	8,467	.44	22.6

uf200 : 100 instances, 200v, 860c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 100,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	44,808	34,481	2.5	4.4
Novelty(.6)	31,634	22,909	2.5	4.4
Novelty+(.6, .01)	31,412	20,823	2.3	4.4
DLM(pars4)	16,098	10,072	.04	6.7
SDF(.00025)	17,505	10,366	.22	24.0

uf225: 100 instances, 225v, 960c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 100,000 Reps: 100]

Table 4.3: SDF comparison for larger uf instances

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	41,287	36,310	1.4	4.5
Novelty(.6)	27,677	24,453	1.7	4.5
Novelty+(.6, .01)	27,639	25,954	1.8	4.9
DLM(pars4)	22,635	12,387	.25	6.9
SDF(.0002)	18,905	13,433	.26	25.4

Table 4.4: SDF comparison for uf250: 100 instances, 250v, 1065c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 100,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	4706	3776	0	7.8
Novelty(.5)	3523	2689	.06	7.2
Novelty+(.5, .01)	3135	2576	0	7.3
DLM(pars4)	879	723	0	12.1
SDF(.0005)	917	877	0	48.4

jnh-sat : 16 instances, satisfiable, Constant Density Random (DIMACS - Hooker) [Max Steps: 5000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	252,616	252,469	50	2.0
Novelty(.6)	251,230	251,002	50	1.7
Novelty+(.7, .01)	251,587	250,494	50	1.7
DLM(pars5)	2,655	2,463	0	4.4
SDF(.0005)	105,105	102,503	16	12.9

aim-sat-100 : 16 instances, 100v, satisfiable, AIM Random Generator (DIMACS - Asahiro/Iwama/Miyano) [Max Steps: 500,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	268,238	258,620	50	2.1
Novelty(.5)	270,069	257,488	51	1.8
Novelty+(.6, .01)	263,989	257,927	50	1.8
DLM(pars1)	51,671	41,464	1.4	3.3
SDF(.00025)	268,238	258,620	50	21.3

aim-sat-200 : 16 instances, 200v, satisfiable, AIM Random Generator (DIMACS - Asahiro/Iwama/Miyano) [Max Steps: 500,000 Reps: 100]

Table 4.5: SDF comparisons for jnh and aim instances

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.7)	1,332	1,061	0	5.7
Novelty(.4)	455,008	1,047	91	5.0
Novelty+(.7, .01)	8,563	1,083	0	5.5
DLM(pars4)	410	406	0	9.5
SDF(.0015)	441	441	0	19.3

ais06 : 61v, 518c, satisfiable, All-Interval Series (SATLIB) [Max Steps: 500,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.3)	29,706	8,000	0	7.9
Novelty(.7)	495,002	13,300	99	8.5
Novelty+(.4, .01)	155,992	8,791	7	8.0
DLM(pars1)	4,678	4,460	0	18.1
SDF(.0004)	4,748	4,641	0	36.2

ais08 : 113v, 1520c, satisfiable, All-Interval Series (SATLIB) [Max Steps: 500,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.3)	190,325	65,600	13	11.4
Novelty(.5)	500,000	n/a	100	12.2
Novelty+(.3, .01)	433,702	433,702	72	11.3
DLM(pars4)	18,420	14,306	0	16.5
SDF(.00013)	20,464	16,320	0	55.8

ais10 : 181v, 3151c, satisfiable, All-Interval Series (SATLIB) [Max Steps: 500,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.6)	487,542	487,542	96	16.1
Novelty(.5)	500,000	n/a	100	15.7
Novelty+(.2, .01)	491,714	491,714	97	15.1
DLM(pars1)	165,904	165,904	3	30.0
SDF(.0001)	156,253	132,342	5	81.1

ais12 : 265v, 5666c, satisfiable, All-Interval Series (SATLIB) [Max Steps: 500,000 Reps: 100]

Table 4.6: SDF comparison for ais (All-Interval series) instances (1 instance each)

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	20,753	17,340	0	7.3
Novelty(.4)	9,028	9,022	0	7.1
Novelty+(.4, .01)	10,553	10,546	0	7.0
DLM(pars4)	3,712	3,701	0	15.7
SDF(.0001)	2,906	2,902	0	39.8

bw_large.a : 459v, 4675c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Steps: 50,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.4)	336,855	336,855	42	9.3
Novelty(.4)	195,126	170,137	4	9.4
Novelty+(.4, .01)	147,370	130,004	4	9.4
DLM(pars4)	44,361	39,216	0	19.2
SDF(.00005)	37,122	36,728	0	79.8

bw_large.b : 1087v, 13772c, satisfiable, Blocks World 11 Blocks, 9 Steps (SATLIB - Kautz and Selman) [Max Steps: 500,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	1,000,000	n/a	100	18.1
Novelty(.5)	1,000,000	n/a	100	18.8
Novelty+(.2, .01)	924,451	924,451	83	12.9
DLM(pars4)	895,213	895,213	77	37.9
SDF(.00002)	939,975	938,975	89	171

bw_large.c : 3016v, 50457c, satisfiable, Blocks World 15 Blocks, 14 Steps (SATLIB - Kautz and Selman) [Max Steps: 1,000,000 Reps: 100]

Table 4.7: SDF comparison for bw_large instances (1 instance each)

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	1038	961	0	5.1
Novelty(.5)	496	493	0	5.6
Novelty+(.6, .01)	482	482	0	5.5
DLM(pars4)	265	265	0	13.0
SDF(.0005)	297	293	0	17.9

medium : 116v, 953c, satisfiable, Blocks World 5 Blocks, 4 Steps (SATLIB - Kautz and Selman) [Max Steps: 50,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.5)	20,154	19,502	0	10.1
Novelty(.4)	9,988	9,065	0	9.7
Novelty+(.4, .01)	10,032	9,759	0	9.8
DLM(pars4)	3,658	3,532	0	21.8
SDF(.0001)	3,018	2,960	0	44.0

huge : 459v, 7054c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Steps: 50,000 Reps: 100]

Solver	Av Steps	OEESS	Fail%	ms/Step
WSAT(.2)	168,895	168,895	2	5.8
Novelty(.4)	115,654	115,120	1	6.3
Novelty+(.3, .01)	109,575	108,141	1	6.5
DLM(pars4)	12,101	11,805	0	24.7
SDF(8.76×10^{-6})	16,849	16,688	0	170

logistics.c : 1141v, 10719c, satisfiable, Logistics, 7 Packages, 13 Steps (SATLIB - Kautz and Selman) [Max Steps: 500,000 Reps: 100]

Table 4.8: SDF comparison for other structured instances (1 instance each)

Chapter 5

Local Search: ESG

> *Quitting the wall, I resolved to cross the area of the enclosure. [...] I took courage and did not hesitate to step firmly – endeavouring to cross in as direct a line as possible. [...] the torn hem of my robe became entangled between my legs. I stepped on it, and fell violently on my face. [...] I put forward my arm, and shuddered to find that I had fallen at the very brink of a circular pit, whose extent of course I had no means of ascertaining at the moment. [...] Shaking in every limb, I groped my way back to the wall – resolving there to perish rather than risk the terrors of the wells, of which my imagination now pictured many in various positions about the dungeon.*
> - **The Pit and the Pendulum**, Edgar Allan Poe

SDF explored several directions that lead to good performance in terms of search steps. The next phase of research distilled these ideas, working to achieve a cleaner, simpler algorithm with competitive runtime performance. At the same time, we tried to understand the method from a broader perspective rather than as a collection of features that work well for SAT. The work culminated in ESG, a state-of-the-art local search SAT solver that naturally extends to other domains.

The Exponentiated Subgradient algorithm (ESG) [101] is an informed reconstruction of SDF that brings three key features:

- a simple algorithm
- greatly improved performance
- generalization to boolean (0-1 integer) linear programming

It is based on the realization that most primal-dual Lagrangian SAT solvers are chiefly rediscoveries of an old idea from operations research (OR) called *subgradient optimization* [30], with a few special tricks and modifications attached. This realization led us to cast SAT as one of a broader class of problems, namely *boolean linear programming* (BLP) problems, also known as *0-1 integer programming* problems [10].

5.1 Boolean Linear Programming

Boolean linear programming instances are a special case of integer linear programming (ILP), where all variables have binary domains (e.g. $\{0,1\}$ or $\{-1,1\}$). The general form is:

$$\begin{aligned} & \min_{\mathbf{x}} \mathbf{d} \cdot \mathbf{x} \\ \text{subject to} \quad & C\mathbf{x} \leq \mathbf{b} \\ \text{and} \quad & \mathbf{x} \in \{-1,1\}^n \end{aligned} \tag{5.1}$$

where n is the number of variables, $\mathbf{d} \in I\!R^n$, $\mathbf{b} \in I\!R^n$, and C is an m by n matrix representing m constraints.

Predictably, we will take a primal-dual Lagrangian approach to solving this problem. Recalling our earlier discussion from Section 2.5.9.1 where we used arbitrary constraints, we can see that this is the *primal BLP problem*, having both a linear objective function and linear constraints. From this, and the general Lagrangian formulation (2.34), we derive our *Lagrangian BLP formulation*:

$$L(\mathbf{x}, \lambda) = \mathbf{d} \cdot \mathbf{x} + \sum_{i=1}^{m} \lambda_i (c_i \cdot \mathbf{x} - b_i) \tag{5.2}$$

where $\lambda_i \geq 0$.

We can now think in terms of the *dual function* (5.3) and the corresponding *dual problem* (5.4):

$$D(\lambda) = \min_{\mathbf{x}} L(\mathbf{x}, \lambda) \tag{5.3}$$

$$\max_{\lambda} D(\lambda) \quad \text{subject to} \quad \lambda \geq \mathbf{0} \tag{5.4}$$

Again recalling Section 2.5.9.1, if P^* is the minimum value of (5.1) (i.e. the value of the optimal solution) and D^* is the maximum value of (5.4), then the *weak duality theorem* states that $D(\lambda) \leq P^*$, $\forall \lambda \geq 0$ and consequently, $D^* \leq P^*$ [10]. Thus, the value of the dual function for any λ is a lower bound on the optimal solution, and the value of a solution to (5.4), D^*, is the best possible lower bound. The dual problem can now be interpreted as maximizing a lower bound on the optimal value.

The difference $P^* - D^*$ is called the *duality gap*. For linear (or convex) programs, this gap is zero, so the primal problem can sometimes be solved by finding a solution to the dual [10]. However, in ILPs the duality gap can be non-zero. Nevertheless, attacking a problem by maximizing the lower bound is the basis of branch-and-bound techniques commonly used for ILPs. While not systematically searching, we pursue the same objective in ESG.

5.2 BLP Formulation of SAT

Formulating SAT as a BLP is quite straightforward. SAT is purely a constraint satisfaction problem. There is no objective to optimize. Therefore, we simply make the objective function the zero function ($\mathbf{d} = \mathbf{0}$). The boolean variables take the domain $\{1, -1\}$ (1 means $true$), and the clauses can be represented in the constraint matrix as follows.

$$C = \begin{bmatrix} c_{11} & \ldots & c_{1m} \\ \vdots & & \vdots \\ c_{n1} & \ldots & c_{nm} \end{bmatrix}$$

where the ith row represents the ith clause, with the entries

$$c_{ij} = \begin{cases} 1 & \text{if } x_j \text{ occurs as a negative literal in } c_i \\ -1 & \text{if } x_j \text{ occurs as a positive literal in } c_i \\ 0 & \text{otherwise} \end{cases}$$

For example, given a SAT instance with four variables and two clauses, $(x_1 \vee x_2 \vee \bar{x}_3) \wedge (\bar{x}_1 \vee x_3 \vee x_4)$, the matrix would like this:

$$C = \begin{bmatrix} -1 & -1 & 1 & 0 \\ 1 & 0 & -1 & -1 \end{bmatrix}$$

The constants are, $b_i = k_i - 2$ (recall that k_i is the number of literals in the ith clause). For our example here, $\mathbf{b}$ is the column vector $[1, 1]$.

If the assignments to the variables are the same as all the corresponding non-zero entries for a row in the constraint matrix, none of the literals in that clause are satisfied by the assignment. The inner product for that constraint will be, $\mathbf{c_i} \cdot \mathbf{x} = k_i > b_i$, so that constraint will be violated. If a single literal is satisfied, the inner product will now contain a single negative term (-2). This gives, $\mathbf{c_i} \cdot \mathbf{x} = k_i - 2 \leq b_i$, so the constraint will be satisfied. Each additional satisfied literal drops the inner product by 2.

Using our example matrix C from above, consider the assignment $\{x_1 = \mathcal{F}, x_2 = \mathcal{F}, x_3 = \mathcal{T}, x_4 = \mathcal{T}\}$. In the BLP formulation, this corresponds to a column vector, $\mathbf{x} = [-1, -1, 1, 1]$. This gives:

$$\begin{aligned} C\mathbf{x} &\leq \mathbf{b} \\ \begin{bmatrix} -1 & -1 & 1 & 0 \\ 1 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} -1 \\ -1 \\ 1 \\ 1 \end{bmatrix} &\leq \begin{bmatrix} 1 \\ 1 \end{bmatrix} \\ \begin{bmatrix} 3 \\ -3 \end{bmatrix} &\leq \begin{bmatrix} 1 \\ 1 \end{bmatrix} \end{aligned}$$

So the first clause is violated ($3 > 1$) and the second is satisfied ($-3 \leq 1$). If we now flip x_1, we get:

$$\begin{bmatrix} -1 & -1 & 1 & 0 \\ 1 & 0 & -1 & -1 \end{bmatrix} \begin{bmatrix} 1 \\ -1 \\ 1 \\ 1 \end{bmatrix} \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

$$\begin{bmatrix} 1 \\ -1 \end{bmatrix} \leq \begin{bmatrix} 1 \\ 1 \end{bmatrix}$$

We have now satisfied the first clause ($1 \leq 1$) and our second clause is still satisfied ($-1 \leq 1$).

Our final general formulation of SAT as a BLP is consequently very simple:

$$\begin{array}{rl} & \min_{\mathbf{x}} \mathbf{0} \cdot \mathbf{x} \\ \text{subject to} & C\mathbf{x} \leq \mathbf{b} \\ \text{and} & \mathbf{x} \in \{-1, 1\}^n \end{array} \tag{5.5}$$

5.3 Subgradient Optimization

Subgradient optimization is a well-established idea in operations research, typically credited as originating from Everett [30]. Everett never uses the term "subgradient" but rather describes a generalization of continuous-space Lagrangian optimization to cases where the variable domains are integer. The approach was later used with great success by Held and Karp on symmetric travelling salesman problems [53], for which they discuss a rich variety of formulations and associated algorithms, and more recently by Larsson et al. [68].

In order to use primal-dual Lagrangian optimization, we must have a means to minimize the Lagrangian with respect to the primal variables and to maximize the dual function with respect to the dual variables. If we cannot directly solve for maxima with respect to the dual variables, we turn to searching for maxima. Gradient ascent is an obvious choice but if the dual function is non-differentiable we must seek an alternative.

While the dual function may be non-differentiable, it is nonetheless always concave [10]. This can be appreciated intuitively by considering Figure 5.1, where plot (i) shows the Lagrangian plotted vs. λ for three different values of $\mathbf{x}$. Clearly, for any fixed $\mathbf{x}$, $L()$ is linear in λ, so we get three lines. The dual function takes the minimum of these lines, producing the concave function shown in (ii).

In an excellent overview of Lagrangian relaxation methods in OR, Fisher states, somewhat colourfully, that the subgradient method is "a brazen adaptation of the gradient method in which gradients are replaced by subgradients" [31]. The *subgradient* is a generalization of the concept of a gradient that allows for non-differentiable functions. While a gradient is obtained by considering a plane tangent to a differentiable function, a subgradient

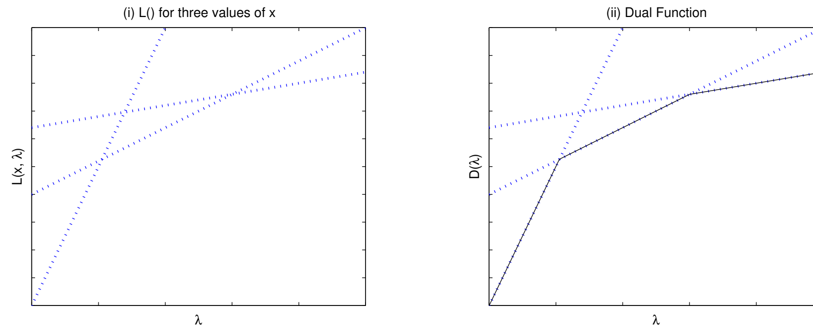


Figure 5.1: (i) The Lagrangian, $L(\mathbf{x}, \lambda)$, plotted for three different values of $\mathbf{x}$. (ii) The resulting concave dual function, $D(\lambda)$, (min of the three lines).

is obtained by considering any plane which touches it at the desired point and lies above the function everywhere else. For concave functions such planes can be obtained even at non-differentiable points.

In Figure 5.2, plot (i) shows an example of a gradient $\mathbf{u}$, obtained for some differentiable function $f()$, while plot (ii) illustrates a subgradient $\mathbf{v}$ obtained for a non-differentiable point in our function. Note that the subgradient is not uniquely defined at such a point, but will vary depending on which tangent plane we consider (two possible such planes are shown in plot (ii)). However, any subgradient at the point gives us a direction in which the function increases, allowing us to search for maxima. Because $D(\lambda)$ is always concave, a subgradient always exists. The only question remaining is whether we can find it readily.

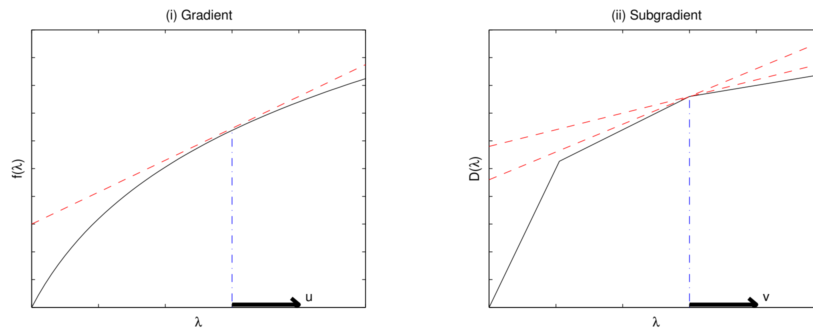


Figure 5.2: (i) A differentiable function, $f()$, its tangent plane, and corresponding gradient $\mathbf{u}$ for a single point. (ii) Two possible tangent planes for a point on $D(\lambda)$ and one of the corresponding subgradients, $\mathbf{v}$.

The happy answer to this question is part of the attraction of subgradient optimization. It turns out the penalty term of the Lagrangian at any given point is a subgradient at that

point, and is thus obtained for "free" [10]. Given a value for λ, and a corresponding $\mathbf{x}_\lambda = \arg\min_{\mathbf{x}} L(\mathbf{x}, \lambda)$, we define the *violation*:

$$\mathbf{v}_\lambda = C\mathbf{x}_\lambda - \mathbf{b} \tag{5.6}$$

To avoid cumbersome subscripts, we define $\mathbf{v}^{(t)} = \mathbf{v}_{\lambda^{(t)}}$. Since the violation is a subgradient, we now have a simple update rules for the weights.

$$\lambda_i^{(t+1)} = \alpha v_i^{(t)} + \lambda_i^{(t)} \tag{5.7}$$

where α is a step size parameter. If α is sufficiently small, this procedure is guaranteed to converge to the dual optimal value, D^* [10].

5.4 ESG: Three Enhancements for Subgradient Optimization

5.4.1 Multiplicative Weight Updates

As a reconstruction of SDF, ESG keeps the multiplicative weight updates, leading to the name "Exponentiated Subgradient".

$$\lambda_i^{(t+1)} = \alpha^{v_i^{(t)}} \lambda_i^{(t)} \tag{5.8}$$

where $\alpha > 1$.

However, ESG drops the "flooding" approach of SDF where a minimal α value is computed every dual step to guarantee a new search direction. Instead, α is specified directly as a parameter. Computing a suitable α is quite expensive and a fixed α value has been experimentally found to work well across a wide variety of SAT instances.

5.4.2 Nonlinear Penalty

The standard BLP Lagrangian uses a penalty term that is linear in the violation. However, we can generalize this formulation to include penalties that are some arbitrary function of the violation. We do so by introducing a penalty function $\theta()$.

$$L(\mathbf{x}, \lambda) = \mathbf{d} \cdot \mathbf{x} + \sum_{i=1}^{m} \lambda_i \theta(c_i \cdot \mathbf{x} - b_i) \tag{5.9}$$

This generalization still gives us a concave dual function and our new weight update rule is still straightforward.

$$\lambda_i^{(t+1)} = \alpha^{\theta(v_i^{(t)})} \lambda_i^{(t)} \tag{5.10}$$

Linear Penalty In this general form, making θ the identity function gives the standard BLP Lagrangian (see Figure 5.3 (i)).

$$\theta_l(\mathbf{v}) = \mathbf{v} \tag{5.11}$$

In this case, the penalty term is the weighted sum of the violations. However, since the violations can be negative (i.e. the constraint is not merely satisfied but is actually "overly satisfied"), overly-satisfying one constraint allows us to violate some other constraint and still achieve the same penalty. Violations can thus be "traded off" between constraints.

Hinge Penalty In a local search approach, such as that used by ESG, the tradeoff offered by linear penalties can lead to wasted time as the search flops back and forth between alternative constraints (one can think of such tradeoffs as plateaus in the search space). Generalizing the Lagrangian to allow nonlinear penalties offers an alternative where we penalize violated constraints, but do not reward overly-satisfied constraints. A "hinge" function, such as the one shown in Figure 5.3 (ii), is the obvious candidate for this, and was used implicitly in both SDF and DLM. Here we explicitly acknowledge its importance.

$$\theta_h(v_i) = \begin{cases} 0 & \text{if } v_i \leq 0 \\ v_i & \text{if } v_i > 0 \end{cases} \tag{5.12}$$

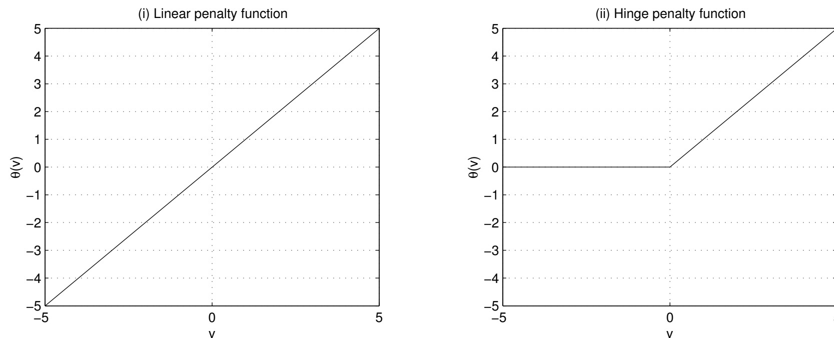


Figure 5.3: (i) Linear and (ii) Hinge Penalty functions

5.4.3 Smoothing

The smoothing in the original ESG [101] is similar to that in SDF, with the same rationale. Every dual step, the weights for all unsatisfied clauses are first increased and then the weights of all clauses are smoothed. SDF smooths using the following formula:

$$\lambda_i^{(t+1)} = (1-\rho)\bar{\lambda} + \rho\lambda_i^{(t)} \tag{5.13}$$

where $\bar{\lambda} = \frac{1}{m}\sum_{j=1}^{m}\lambda_j$ (the mean of the weights).

ESG was made considerably faster by changing this smoothing slightly. Computing the mean of the weights is expensive and the mean is really an arbitrary, although not unreasonable, target. By smoothing toward the unit weight ($\lambda_i = 1$), the implementation becomes substantially faster in terms of runtime, provides a form of automatic normalization for the weights, and behaves almost identically in terms of search steps. The smoothing used in ESG is therefore:

$$\lambda_i^{(t+1)} = (1 - \rho) + \rho\lambda_i^{(t)} \tag{5.14}$$

5.5 Primal Search

ESG operates by attempting to solve the dual problem, alternately minimizing in the primal variables and maximizing in the duals. The primal part of this search computes $\min_{\mathbf{x}} L(\mathbf{x}, \lambda)$. Unfortunately this minimization is difficult because of the integrality constraint. We might relax this constraint in our search, but the nonlinear hinge penalty still renders the problem nondifferentiable with respect to $\mathbf{x}$ and we have little reason to expect LP relaxations will be very effective in any event (see the performance of CPLEX in Section 5.8.1). Therefore, local search remains the most convenient approach to this minimization. Because the search is subject to local minima, it only approximates the dual problem.

The objective of this primal search has already been established as

$$g_{ESG}(\mathbf{a}, \lambda) = \mathbf{d} \cdot \mathbf{a} + \sum_{i=1}^{m} \lambda_i \theta(c_i \cdot \mathbf{x} - b_i) \tag{5.15}$$

where, for SAT, $\mathbf{d}$ is the zero vector and the constraints are as described earlier.

ESG searches greedily in this objective until it reaches a non-strict local minimum at which point it executes a dual step, reweighting and smoothing as described earlier.

Algorithm 16 shows the final combination of these features.

5.6 Related Work

5.6.1 DLM

The most obvious difference between DLM and ESG is that DLM uses additive weight updates instead of multiplicative. As noted in Section 2.5.9.4 in the discussion of DLM-99-SAT, some versions of DLM have a secondary weighting mechanism. A “trap count” is kept for each clause of the number of times it is unsatisfied at a local minimum. If the trap count for a clause exceeds a threshold parameters, a “special increase” to its weight is applied whenever it is updated. This is an extra fixed amount, specified by a parameter, and added beyond the standard weight increment. This can be viewed as a very crude approximation of multiplicative updates, since high weight clauses will likely have high trap counts and thus receive larger updates (this is similar to multiplicative updates for large weights, which

Algorithm 16 ESG

ESG(α,ρ)

1. Randomly generate initial full assignment, $\mathbf{a}^{(0)}$
2. Set primal step counter, $t \leftarrow 0$, and dual step counter, $s \leftarrow 0$
3. If $\mathbf{a}^{(t)}$ is a satisfying assignment, exit with success
4. If the termination criteria have been met, exit with failure
5. Neighbourhood, $N(\mathbf{a})$ has n assignments, each obtained by flipping a single variable in $\mathbf{a}^{(t)}$
6. Evaluate assignments in $N(\mathbf{a})$ according to the function, $g_{ESG}(\mathbf{a}, \lambda) = \mathbf{d} \cdot \mathbf{a} + \sum_{i=1}^{m} \lambda_i^{(s)} \theta(c_i \cdot \mathbf{x} - b_i)$
7. If no neighbour offers an improvement over the current objective value (non-strict local minimum), then make a *dual step* as follows
   (a) reweight: $\lambda_i^{(s+1)} = \alpha^{\theta(v_i^{(s)})} \lambda_i^{(s)}$, $i \in \mathbf{UNSAT}(\mathbf{a})$
   (b) smooth: $\lambda_i^{(s+1)} = (1 - \rho) + \rho \lambda_i^{(s)}$, $1 \leq i \leq m$
   (c) Advance the dual step counter, $s \leftarrow s + 1$
   (d) Goto step 4
8. Otherwise, choose $\mathbf{a}^{(t+1)}$ to be the neighbour with the smallest value. If there are "ties" (several assignments with the same value), choose randomly among the smallest.
9. Advance the primal step counter, $t \leftarrow t + 1$
10. Goto step 3

will be larger than for small weights). However, since trap counts do not decay like the weights, they can also be viewed as a long term memory of important constraints. It would be interesting to test these two possibilities (pseudo-multiplicative vs. long term memory) to see if either effect, or both, explain the good performance of DLM.

A more subtle difference between the two algorithms is that DLM has a non-zero objective function. Instead, it minimizes the number of violated constraints, subject to the constraints. This second copy of the constraints has already been discussed in Section 2.5.9.3, and we only note here that casting CSPs as BLPs with zero objective functions seems more straightforward.

5.6.2 SAPS

The SAPS research [63] improved the implementation of ESG by decoupling the smoothing from the upweighting. The original ESG implementation performed both operations every dual step. SAPS made smoothing probabilistic, with a parameter specifying the probability of smoothing at each dual step. This simplifies the implementation and offers substantial performance improvements, at the cost of an additional parameter. The research also examined the behaviour of weights over time and concluded that the only significant parameter is the smoothing parameter (i.e. the weighting factor and walk probability can be left at certain default values without hurting performance). In later research [109], they removed the randomness they introduced into the dual steps, concluding that the probabilistic smoothing is unnecessary and that smoothing after a fixed number of dual steps has elapsed is effective. Our own experiments confirm a moderate improvement for many instances, but we have found that performance on some structured instances is noticeably damaged by fixed rate smoothing in our recent implementations of ESG.

5.7 Implementation

ESG works very well on SAT problems with the BLP encoding. The implementation for SAT is specialized somewhat, but is still the same algorithm used for other BLP problems. The specializations are natural consequences of the specific problem (e.g. the zero objective) and do not include any additional heuristics. Furthermore, the general version of ESG works directly on the same formulation, albeit with inferior runtime performance. During the course of this research, there have been several minor variants of ESG. We identify two, used in the experiments presented in different parts of this thesis, corresponding to the earliest published results and our most recent implementation.

5.7.1 ESGflt

The original ESG used floating-point weights and had substantial code devoted to combating numerical drift. ESGflt's primal search also differs slightly. When a minimum is

reached, there is a small probability, ω, of taking a random step by flipping a variable selected uniformly. This adds a random walk component and offers the possibility of escaping local minima without reweighting. The walk probability is set to 0.01 in all reported experiments. This feature has been discarded in the most recent implementations since it rarely offers any advantage and occasionally damages performance slightly [63, 109]. The very earliest versions also smoothed to the mean of the weights like SDF, but results from that version only appear in [101].

5.7.2 ESGint

The results presented in [101] were competitive with or superior to any competitors available at that time. Since then, substantial improvements have been made, some due to the SAPS research of Hutter et al. [63], and some improvements of our own. Although still the same algorithm, the implementation differences and consequent runtime improvement are large enough to be distinguished. We have dubbed the latest version, *ESGint*. The major differences are:

- A pure integer implementation (fixed point) (hence the name ESGint), dropping the floating point weight values from earlier versions. Aside from the improved performance offered by integers, the new implementation no longer drifts numerically, so no correction loops are required. The implementation was streamlined substantially in terms of the information tracked because of the numerical stability.

- We use the upweight/smoothing decoupling that is the main features of SAPS. SAPS does this probabilistically, setting a probability for smoothing at each dual step. The alternative is to smooth only every fixed number of dual steps, deterministically. Tompkins and Hoos have since reported that probabilistic smoothing offers no advantage over fixed rate smoothing [109]. Our more recent results confirm this in the main, showing that for many problems there is actually a small advantage to fixed rate smoothing. However, it appears that without probabilistic smoothing, ESG (the integer version at least) can become trapped, typically on specific structured instances (e.g. ii16, par16, aim-sat-200, beijing_jobshop). It is possible that the randomness in earlier techniques and/or the inevitable numerical drift in floating-point weights conspire to avoid certain deterministic traps. It is difficult to confirm this suspicion as it only occurs rarely in a very few instances of non-trivial size, and manual investigations have not discovered any trivial behavioural problems (e.g. oscillation between a small number of assignments). We therefore use probabilistic smoothing in ESGint.

- A list is kept of clauses with non-unit weights. This means smoothing operations can be restricted to clauses in this list. Typically, large numbers of clauses remain satisfied for much of the search, so this list can be quite small and offers significant savings.

- For clauses satisfied by a single literal, the satisfying literal is tracked in a separate data structure. This was loosely inspired by the witness literals in Chaff.

Results presented here will refer to ESGflt or ESGint. Parameter settings for the two implementations may appear quite different because the original ESG attempted to automatically adjust the α parameter according to the problem size. Therefore, the "α" parameter values reported in [101] for the original ESG are not the actual factor used for upweighting. If we call the rescaled parameter $\tilde{\alpha}$, then the true factor is obtained by $\alpha = 1 + \tilde{\alpha}\frac{n}{m}$. This odd scaling is more trouble than it is worth, especially since a single value for the raw parameters, $\alpha = 1.3$, turns out to be effective for a wide variety of problems. ESGint has only been implemented for the specialized SAT case, but most of its features apply equally well to a general ESG implementation.

5.8 ESG Performance

We present results on ESG's performance in two parts. The first compares ESG with CPLEX, an industry standard LP/ILP solver, to establish that ESG's specializations are not simply dominated by off-the-shelf solutions. This study also explores the importance of the hinge vs. linear penalty function. The second section compares ESG with other local search SAT methods.

5.8.1 Comparison with Standard ILP Methods

BLPs are simply a special case of integer linear programming, for which commercial, "off-the-shelf" solvers exist. Among the most popular of these is ILOG's CPLEX,[1] which includes a highly optimized mixed ILP solver. It is natural to consider how such a package compares with ESG on the BLP problems we consider. It is important to note that a solver like CPLEX is different in several respects. In broad terms, CPLEX uses a branch-and-bound search with LP relaxations of the ILP to solve the problem. It is a complete method, so that if no feasible solution exists to the ILP, CPLEX will eventually report this fact. ESG is, of course, incomplete.

Here, we compare CPLEX with ESG and some variations, in order to demonstrate that ESG is highly effective, and that details of its implementation are, indeed, important. A set of hard, random problems from SATLIB were encoded as ILPs and solved using the following approaches:

- CPLEX
- ESG$_h(\alpha, 1-\rho, \omega)$ is the algorithm described above. The subscript "h" indicates that the hinge penalty is used.
- ESG$_l(\alpha, 1-\rho, \omega)$ is the ESG algorithm but with a linear penalty indicated by the subscript "l".

[1] See *http://www.ilog.com*

- ASG$_h(\alpha, 1-\rho, \omega)$ and ASG$_l(\alpha, 1-\rho, \omega)$ are the ESG algorithm using additive instead of multiplicative weight updates, and the hinge and linear penalties respectively. Note that ASG$_l$ with no smoothing and a zero walk probability represents standard subgradient optimization.

Note that CPLEX was only tested on a few instances due to limited availability of our CPLEX facility. The various implementations of ESG were based on ESGflt. We have reported average steps, OEESS, percentage of failed runs, and OEERT. Note that the "steps" for CPLEX do not correspond to variable flips as they do in the ESG variants, but are actually node expansions in its branch-and-bound search and provided simply so CPLEX's scaling behaviour can be better understood. Because CPLEX is a complete method, it never fails a run. CPLEX was run using the tunings reported in [2]. The ESG and ASG variants were tuned by running them on a selection of 10 instances chosen at random from each class and limited to running for 500,000 primal steps. The experiments were run on the configuration shown in Appendix A.1. For the subgradient methods, averages were taken over 100 runs of each instance. CPLEX results are averaged over single runs on each instance. The results are shown in Table 5.1.

	Av Steps	OEESS	Fail%	OEERT(s)
uf50		(100 instances)		
CPLEX (10 instances)	49	na	0	.186
ASG$_\ell(.12, 0, .02)$	500,000	na	100	na
ESG$_\ell(2.0, .05, .125)$	178,900	143,030	25	43.3
ASG$_h(.12, 0, .02)$	1,194	359	0.3	.027
ESG$_h(1.2, .99, .0005)$	215	187	0	.0009
uf100		(1000 instances)		
CPLEX (10 instances)	727	na	0	6.68
ASG$_h(.2, 0, .02)$	3,670	2,170	1.3	0.26
ESG$_h(1.15, .01, .002)$	952	839	0	.004
uf150		(100 instances)		
CPLEX (10 instances)	13,808	na	0	275.2
ASG$_h(.5, 0, .02)$	14,290	6,975	1.1	.071
ESG$_h(1.15, .01, .001)$	2,625	2,221	0	.011

Table 5.1: Comparison of general BLP methods on SAT

These results show that CPLEX performs very poorly in terms of runtime on even relatively small BLP-SAT instances. Turning to the ASG and ESG variants, we can see that linear penalties behave very poorly, regardless of the weight updates (results were not even collected for the larger instances). Given a hinge penalty, it is clear that the multiplicative weight updates perform much better than the additive. Oddly, smoothing did not help ASG whereas it substantially assists ESG. SAT is certainly one domain in which ESG offers a considerable advantage over traditional subgradient optimization.

5.8.2 Comparison with Local Search SAT Methods

We have conducted extensive experiments to compare ESGint with other local search methods, including GSAT, HSAT, WalkSAT, Novelty+, DLM, and SAPS.[2] Experiments were conducted on hardware described in Appendix A.3 and with the software from Appendix B.1. All solvers were run with default parameters, a single try, and the time limit and number of repeats indicated in the table captions. The tables may be found at the end of this chapter.

In most cases, ESGint is the fastest in both actual average and expected optimal runtime. In steps, it is frequently the best or at least close to its nearest competitors.[3] Failure rates are almost invariably as good as, or better than, all competitors, suggesting that it rarely gets stuck. Interesting exceptions include *beijing-vlsi-2bit-sat* and *bw_large.c*, where ESGint performs comparatively poorly, although the reason for this is not clear.

5.9 Conclusion

ESG offers excellent performance on the BLP formulation of SAT. It is a state-of-the-art solver, simpler than its nearest competitors, and with at least some theoretical grounding. The associated studies reveal interesting properties of the penalty functions, suggesting that related OR techniques might benefit from similar adaptations.

However, its chief interest lies in its generality, as it is readily applicable to other BLP problems. By adopting a broad and established framework in which to view SAT, our methods can be understood by the light of existing research and also extended in a natural way. One general lesson that may be carried away from this is that it is important to connect one's problem to other problems and other research domains. In the next chapter, we will explore the generality aspect of ESG, applying it to a BLP formulation for combinatorial auctions.

[2] Some sets of instances do not have results for a particular solver. In some cases the solver was omitted because it was highly likely to fail completely based on observations of smaller instances in similar classes (e.g. GSAT on large HR instances). In other cases, it was due to a crash or out of memory problem. In particular, some bridged solvers (code not written by us) had occasional trouble running in the experimental environments or produced incorrect results.

[3] ESGint and SAPS are essentially the same algorithm, but show radical differences in performance in some cases. We believe this is due to the peculiarities of each implementation. The integer implementation of ESGint may get trapped in certain cases, while we believe that numerical drift in SAPS, which uses floating-point, may account for its occasional poor performance.

Local Search Comparison Results

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	15689	10526	0.0124	0.00832
SAPS	0	23059	14668	0.0243	0.01549
DLM	0	13164	8994	0.0120	0.00822
Novelty	1.5	137931	74753	0.1728	0.24832
WalkSAT	0.2	71565	39241	0.0830	0.04550
HSAT	80.6	2240338	173464	1.6171	40.32730
GSAT	70.0	1275482	560535	1.5214	11.69950

Table 5.2: BMS : 500 instances, 100v, satisfiable, Backbone-Minimal Subinstances from RTI (SATLIB - Singer) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	976	813	0.0009	0.0007
SAPS	0	1254	1055	0.0018	0.0015
DLM	0	1019	796	0.0013	0.0010
Novelty	0.01	9507	3229	0.0135	0.0046
WalkSAT	0	3600	2854	0.0043	0.0034
HSAT	74.16	960450	7365	0.7459	0.3531
GSAT	72.71	635464	17511	0.7392	0.3788

Table 5.3: RTI : 500 instances, 100v, 426c, satisfiable, Random 3-SAT (SATLIB - Singer) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	49314	46140	0.0417	0.039
SAPS	0.05	86515	79299	0.1264	0.116
DLM	99.19	615343	612310	1.0085	3.323
Novelty	99.16	828000	828000	1.0105	2.100
WalkSAT	100.00	784856	784856	1.0114	2.234
HSAT	3.14	16073	1937	0.0358	0.004
GSAT	0	21934	21911	0.0597	0.060

Table 5.4: SW100-8-8 : 100 instances, 500v, 3100c, satisfiable, Morphed 5-Colourable Graphs, ratio 2^8 (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	29237	14630	0.020	0.010
SAPS	13.00	634985	18459	0.682	0.020
DLM	0	85006	49567	0.053	0.031
Novelty	49.88	2474046	2470606	2.510	2.506
WalkSAT	49.81	2460364	2153586	2.504	2.192
HSAT	87.56	5586723	2309235	4.383	46.426
GSAT	88.56	3722399	2045878	4.441	8.752

Table 5.5: aim-sat-100 : 16 instances, 100v, satisfiable, AIM Random Generator (DIMACS - Asahiro/Iwama/Miyano) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	20	19544851	99256804	64	328
SAPS	62	183777445	153358667	186	1255
DLM	25	12269543	118761815	76	732
Novelty	50	149179936	148546061	151	1497
WalkSAT	50	135264920	135253759	150	19317
HSAT	89	219536866	157958488	264	8703
GSAT	95	164432760	129083125	285	10911

Table 5.6: aim-sat-200 : 16 instances, 200v, satisfiable, AIM Random Generator (DIMACS - Asahiro/Iwama/Miyano) [Max Time: 300s Reps: 10]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	478	475	0.0007	0.00066
SAPS	0	811	391	0.0014	0.00067
DLM	0	410	405	0.0008	0.00080
Novelty	0	2167	1941	0.0034	0.00306
WalkSAT	0	1091	1066	0.0016	0.00156
HSAT	97	1141296	833	0.9776	0.00071
GSAT	97	903567	2100	0.9780	0.00227

Table 5.7: ais06 : 61v, 518c, satisfiable, All-Interval Series (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	4959	3128	0.007	0.005
SAPS	0	7388	6301	0.015	0.013
DLM	0	5375	5325	0.012	0.012
Novelty	0	40791	24693	0.070	0.042
WalkSAT	0	38917	38116	0.063	0.062
HSAT	100	841528	841528	1.006	1.006
GSAT	100	694731	694731	1.011	1.011

Table 5.8: ais08 : 113v, 1520c, satisfiable, All-Interval Series (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	27165	21545	0.048	0.038
SAPS	0	29462	8828	0.078	0.023
DLM	0	18419	14305	0.054	0.042
Novelty	0	363661	215973	0.709	0.421
WalkSAT	0	480734	216742	0.912	0.411
HSAT	100	6510618	6510618	10.006	10.006
GSAT	100	5327770	5327770	10.015	10.015

Table 5.9: ais10 : 181v, 3151c, satisfiable, All-Interval Series (SATLIB) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	186704	170874	0.366	0.3
SAPS	0	285930	234366	0.940	0.8
DLM	0	167855	133523	0.658	0.5
Novelty	0	7435290	1297200	16.533	2.9
WalkSAT	0	11106073	738900	22.804	1.5
HSAT	100	13132164	99031510	200.003	1508.3
GSAT	100	564686	86464032	200.006	30624.6

Table 5.10: ais12 : 265v, 5666c, satisfiable, All-Interval Series (SATLIB) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	94	94	0.00014	0.000138
SAPS	0	145	145	0.00025	0.000245
DLM	0	111	111	0.00023	0.000226
Novelty	0	734	334	0.00105	0.000478
WalkSAT	0	465	210	0.00056	0.000253
HSAT	82	1658966	215	0.82352	0.000107
GSAT	83	1022581	268	0.83771	0.000220

Table 5.11: anomaly : 48v, 261c, satisfiable, Blocks World 3 Blocks, 3 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	52728	37767	0.2	0.1
SAPS	0	74284	66673	0.6	0.5
DLM	0	261790	132072	1.8	0.9
Novelty	0	604946	491645	2.5	2.1
WalkSAT	16	14544194	14097399	53.3	3302.0
HSAT	100	26657564	69607237	196.4	1108.0
GSAT	100	19875216	62824889	200.0	2260.5

Table 5.12: beijing-bw : 3 instances, satisfiable, Beijing Blocks World - Sussman anomalies (Crawford) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	52443	51313	0.162	0.158
SAPS	0.2	86079	82763	0.916	0.881
DLM	0.2	56633	56080	1.758	1.741
Novelty	1.2	497549	491382	1.398	1.381
WalkSAT	0.5	277779	260722	0.593	0.556
HSAT	100.0	112279	112279	10.006	10.006
GSAT	100.0	92005	92005	10.014	10.014

Table 5.13: beijing-jobshop : 6 instances, satisfiable, Beijing Job Shop (Crawford) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	38.5	4765358	4064	4.0727	0.0035
SAPS	30.3	2720980	5119	3.5944	0.0068
Novelty	0	977	970	0.0016	0.0016
WalkSAT	0	597	597	0.0009	0.0009
HSAT	0.8	18466	1097	0.0791	0.0047
GSAT	0	18038	17998	0.0862	0.0860

Table 5.14: beijing-vlsi-2bit-sat : 4 instances, satisfiable, Beijing 2-bit VLSI design (Crawford) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	421343986	421343986	999.993	2039.450
SAPS	100	95906593	95906593	999.988	999.988
Novelty	0	57579	54338	0.173	0.163
WalkSAT	0	19601	19601	0.053	0.053
HSAT	25	8506130	299342	254.783	8.966
GSAT	100	28342637	28342637	999.992	999.992

Table 5.15: beijing-vlsi-3bit-sat : 2 instances, satisfiable, Beijing 3-bit VLSI design (Crawford) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	1054235450	1054235450	3599.960	3599.960
SAPS	100	38348080	38348080	3598.380	3598.380
DLM	100	31330532	31330532	3599.970	3599.970
Novelty	100	1574194279	1574194279	3599.960	3599.960
WalkSAT	100	1339661473	1339661473	3599.950	3599.950
HSAT	100	17481099	17481099	3599.960	3599.960

Table 5.16: bmc-galileo-8 : 58074v, 294821c, satisfiable, BMC on Galileo FIFO #1, 35 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	1244855540	1244855540	3599.960	3599.960
SAPS	100	44897934	44897934	3599.700	3599.700
DLM	100	32168904	32168904	3599.970	3599.970
Novelty	100	1310220805	1310220805	3599.950	3599.950
WalkSAT	100	1289924670	1289924670	3599.950	3599.950
HSAT	100	16009941	16009941	3599.960	3599.960

Table 5.17: bmc-galileo-9 : 63624v, 326999c, satisfiable, BMC on Galileo FIFO #2, 38 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	2337456763	2337456763	2888	2888
SAPS	100	1240111832	1240111832	3600	3600
DLM	100	867083241	867083241	3600	3600
Novelty	100	2143127606	2143127606	3600	3600
WalkSAT	100	2093634131	2093634131	3600	3600
HSAT	100	80777162	80777162	3600	3600

Table 5.18: bmc-ibm-1 : 9685v, 55870c, satisfiable, BMC on IBM CPU Part 1, 18 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	1593363747	1593363747	3599.96	3599.96
SAPS	100	43417808	43417808	3594.12	3594.12
DLM	100	36924159	36924159	3599.98	3599.98
Novelty	100	1541544262	1541544262	3599.96	3599.96
WalkSAT	100	1483537425	1483537425	3599.96	3599.96
HSAT	100	14577495	14577495	3599.96	3599.96

Table 5.19: bmc-ibm-10 : 61088v, 334861c, satisfiable, BMC on IBM PowerPC BIU #2, 13 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	1543519827	1543519827	3600.0	3600.0
SAPS	100	75227171	75227171	3599.1	3599.1
DLM	100	61572464	61572464	3600.0	3600.0
Novelty	100	1816579032	1816579032	3600.0	3600.0
WalkSAT	100	1813166489	1813166489	3600.0	3600.0
HSAT	100	26068397	26068397	3600.0	3600.0

Table 5.20: bmc-ibm-11 : 32109v, 150027c, satisfiable, BMC on IBM Cache Control #1, 32 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	1204058869	1204058869	3599.960	3599.960
SAPS	100	74112794	74112794	3599.960	3599.960
DLM	100	43939683	43939683	3599.970	3599.970
Novelty	100	1415210948	1415210948	3599.950	3599.950
WalkSAT	100	1244833207	1244833207	3599.950	3599.950
HSAT	100	24952535	24952535	3599.960	3599.960

Table 5.21: bmc-ibm-12 : 39598v, 19477c, satisfiable, BMC on IBM PowerPC BIU #3, 31 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	3081768985	3081768985	3599.960	3599.960
SAPS	100	138206487	138206487	3597.840	3597.840
DLM	100	197132395	197132395	3599.970	3599.970
Novelty	100	2551801962	2551801962	3599.960	3599.960
WalkSAT	100	2074958745	2074958745	3599.950	3599.950
HSAT	100	69581383	69581383	3599.960	3599.960

Table 5.22: bmc-ibm-13 : 13215v, 6572c, satisfiable, BMC on IBM Cache Control #2, 14 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	58360	58360	0.05	0.05
SAPS	100	422858995	422858995	3599.96	3599.96
DLM	0	116643	116643	0.37	0.37
Novelty	0	5302275	5302275	5.98	5.98
WalkSAT	0	199931000	199931000	222.87	222.87
HSAT	100	234317443	234317443	3599.96	3599.96

Table 5.23: bmc-ibm-2 : 3628v, 14468c, satisfiable, BMC on IBM CPU Part 2, 5 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	2090007126	2090007126	3600.0	3600.0
SAPS	100	73447487	73447487	3599.1	3599.1
DLM	100	133348695	133348695	3600.0	3600.0
Novelty	100	2498746510	2498746510	3600.0	3600.0
WalkSAT	100	2040920247	2040920247	3600.0	3600.0
HSAT	100	67087067	67087067	3600.0	3600.0

Table 5.24: bmc-ibm-3 : 14930v, 72106c, satisfiable, BMC on IBM BIU 1996, 14 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	2454418981	2454418981	3599.960	3599.960
SAPS	100	54766977	54766977	3593.830	3593.830
DLM	100	132928872	132928872	3599.970	3599.970
Novelty	100	2252741233	2252741233	3599.950	3599.950
WalkSAT	100	2211673682	2211673682	3599.950	3599.950
HSAT	100	27496215	27496215	3599.960	3599.960

Table 5.25: bmc-ibm-4 : 28161v, 139716c, satisfiable, BMC on IBM PowerPC BIU #1, 24 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	3387884	3387884	3.28	3.28
SAPS	100	312597535	312597535	3599.96	3599.96
DLM	100	1134595691	1134595691	3599.96	3599.96
Novelty	100	2566055443	2566055443	3599.96	3599.96
WalkSAT	100	2696760217	2696760217	3599.95	3599.95
HSAT	100	78162559	78162559	3600.04	3600.04

Table 5.26: bmc-ibm-5 : 9396v, 41207c, satisfiable, BMC on IBM Arbiter #1, 12 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	1033771043	1033771043	3599.960	3599.960
SAPS	100	31844055	31844055	3599.970	3599.970
DLM	100	80154047	80154047	3599.980	3599.980
Novelty	100	1219548425	1219548425	3599.960	3599.960
WalkSAT	100	1242334149	1242334149	3599.950	3599.950
HSAT	100	14195438	14195438	3599.960	3599.960

Table 5.27: bmc-ibm-6 : 51654v, 368367c, satisfiable, BMC on IBM LSU 1997, 22 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	16677	16677	0.03	0.03
SAPS	100	284204742	284204742	3599.07	3599.07
DLM	0	79560	79560	0.72	0.72
Novelty	0	1053136	1053136	1.47	1.47
WalkSAT	0	122479	122479	0.15	0.15
HSAT	100	82245583	82245583	3599.97	3599.97

Table 5.28: bmc-ibm-7 : 8710v, 39774c, satisfiable, BMC on IBM Arbiter #2, 9 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	2998	2992	0.004	0.004
SAPS	0	3429	3418	0.008	0.008
DLM	0	3712	3701	0.011	0.011
Novelty	0	23408	22659	0.038	0.037
WalkSAT	0	18800	16354	0.027	0.024
HSAT	99	507775	41300	0.997	0.081
GSAT	99	434513	124800	1.002	0.288

Table 5.29: bw_large.a : 459v, 4675c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	40857	38537	0.058	0.055
SAPS	0	49619	47732	0.130	0.125
DLM	0	44360	39216	0.149	0.132
Novelty	38	358919	358919	0.622	0.622
WalkSAT	46	479915	479915	0.741	0.741
HSAT	100	265594	265594	1.005	1.005
GSAT	100	228342	228342	1.012	1.012

Table 5.30: bw_large.b : 1087v, 13772c, satisfiable, Blocks World 11 Blocks, 9 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	14740392	12779046	26.9	23.3
SAPS	1	11296613	6720437	43.2	25.7
DLM	0	5506186	1075847	31.9	6.2
Novelty	27	12737120	31451900	118.2	291.8
WalkSAT	96	15796829	101696175	195.4	1257.6
HSAT	100	21689630	21689630	200.0	200.0
GSAT	100	18991533	18991533	199.8	199.8

Table 5.31: bw_large.c : 3016v, 50457c, satisfiable, Blocks World 15 Blocks, 14 Steps (SATLIB - Kautz and Selman) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	30	309865108	140281180	966.42	437.5
SAPS	30	236312751	236312751	1534.63	1534.6
DLM	10	66877193	37810590	657.36	371.7
Novelty	80	243448890	672945620	1959.54	5416.6
WalkSAT	100	314000596	743497326	1999.98	4735.6
HSAT	100	109134539	109134539	1999.82	1999.8
GSAT	100	95658873	95658873	1999.95	2000.0

Table 5.32: bw_large.d : 6325v, 131973c, satisfiable, Blocks World 19 Blocks, 18 Steps (SATLIB - Kautz and Selman) [Max Time: 2000s Reps: 10]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	244142	151675	0.23	0.14
SAPS	0	314071	306047	0.52	0.50
DLM	4	595271	423576	1.14	0.81
Novelty	32	1696326	642242	2.37	0.90
WalkSAT	0	155948	146918	0.21	0.20
HSAT	98	1828208	1828208	4.91	4.91
GSAT	97	1352781	1352781	4.90	4.90

Table 5.33: f0600 : 600v, 2550c, satisfiable, Large Random 3-SAT (DIMACS - Selman) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	34622359	26491155	35.37	27.06
SAPS	0	77152697	29509720	145.17	55.53
DLM	100	285093597	285093597	999.83	999.83
Novelty	50	350441992	10839289	520.15	16.09
WalkSAT	0	539409	467422	0.76	0.66
HSAT	100	242287390	242287390	999.94	999.94
GSAT	100	182090609	182090609	999.99	999.99

Table 5.34: f1000 : 1000v, 4250c, satisfiable, Large Random 3-SAT (DIMACS - Selman) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	8363	7320	0.006	0.005
SAPS	0	9723	8791	0.011	0.010
DLM	0	9570	8313	0.010	0.008
Novelty	5.70	268311	38368	0.301	0.043
WalkSAT	0.02	44233	34880	0.053	0.041
HSAT	99.85	1349141	1192518	2.002	39.370
GSAT	99.83	1056572	904208	2.009	9.194

Table 5.35: flat100 : 100 instances, 300v, 1117c, satisfiable, Flat 3-Colourable Graphs, 100 vert, 239 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	19550	15931	0.013	0.0108
SAPS	0	22375	19218	0.026	0.0223
DLM	0	25282	20542	0.027	0.0221
Novelty	12.94	464793	75814	0.520	1.1176
WalkSAT	0.12	81670	65007	0.094	0.0748
HSAT	99.92	1162781	1073024	2.004	7.0772
GSAT	99.96	927520	895414	2.012	26.2947

Table 5.36: flat125 : 100 instances, 375v, 1403c, satisfiable, Flat 3-Colourable Graphs, 125 vert, 301 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	54673	39267	0.037	0.027
SAPS	0.010	62797	47810	0.074	0.056
DLM	0	71992	53952	0.082	0.061
Novelty	30.257	804080	203116	0.907	0.487
WalkSAT	0.980	188033	147366	0.218	0.171
HSAT	100.000	996122	996122	2.006	13.720
GSAT	100.000	819500	819500	2.013	4.183

Table 5.37: flat150 : 100 instances, 450v, 1680c, satisfiable, Flat 3-Colourable Graphs, 150 vert, 360 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	154588	101313	0.105	0.0685
SAPS	0.01	177436	121477	0.211	0.1445
DLM	0.02	186840	119427	0.225	0.1440
Novelty	32.71	2073066	459812	2.422	3.1366
WalkSAT	0.96	385912	307911	0.428	0.3417
HSAT	100.00	2236852	2236852	5.006	125.2980
GSAT	100.00	1851581	1851581	5.014	69.4973

Table 5.38: flat175 : 100 instances, 525v, 1951c, satisfiable, Flat 3-Colourable Graphs, 175 vert, 417 edges, (SATLIB) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0.18	394386	256640	0.265	0.2
SAPS	0.77	435538	273727	0.527	23.9
DLM	0.83	495093	318561	0.642	3.1
Novelty	48.61	2716058	837383	3.110	18.7
WalkSAT	3.17	686342	508049	0.786	1.6
HSAT	100.00	2032160	2032160	5.006	32.4
GSAT	100.00	1692739	1692739	5.014	21.0

Table 5.39: flat200 : 100 instances, 600v, 2237c, satisfiable, Flat 3-Colourable Graphs, 200 vert, 479 edges, (SATLIB) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	103070212	103070212	3599.96	3599.96
DLM	100	7385456	7385456	3600.01	3600.01
Novelty	100	5600413	5600413	3599.96	3599.96
WalkSAT	100	14781220	14781220	3599.95	3599.95
HSAT	100	52739939	52739939	3599.96	3599.96

Table 5.40: fvp-sat-2 : 1 instance, satisfiable, Formal Verification Buggy 7 Pipeline SS Processor(Velev) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	50	167026683	167026683	775.904	775.904
SAPS	100	130445110	130445110	999.992	999.992
DLM	100	106473438	106473438	999.828	999.828
Novelty	0	11395791	1806660	42.469	6.733
WalkSAT	100	258987844	258987844	999.993	999.993
HSAT	100	120651719	120651719	999.993	999.993

Table 5.41: g125.17 : 2125v, 66272c, satisfiable, Graph Colouring, 17 colours (DIMACS - Johnson) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	22937	22937	0.104	0.104
SAPS	0	35380	34431	0.248	0.242
DLM	2	248264	141270	2.516	1.432
Novelty	0	28808	28438	0.113	0.111
WalkSAT	98	2866657	2866657	9.914	9.914
HSAT	80	827989	27416	8.015	0.265
GSAT	43	497058	98153	4.908	0.969

Table 5.42: g125.18 : 2250v, 70163c, satisfiable, Graph Colouring, 18 colours (DIMACS - Johnson) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	2196	2196	0.067	0.067
SAPS	0	2222	2222	0.095	0.095
DLM	0	2478	2478	1.962	1.961
Novelty	0	4998	4998	0.091	0.091
WalkSAT	0	21259	21259	0.297	0.297
HSAT	0	2252	2252	0.065	0.065
GSAT	0	3063	2993	0.082	0.080

Table 5.43: g250.15 : 3750v, 233965c, satisfiable, Graph Colouring, 15 colours (DIMACS - Johnson) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	38316750	38316750	999.992	999.992
SAPS	100	26799511	26799511	999.445	999.445
DLM	100	22879930	22879930	999.762	999.762
Novelty	0	21451320	15705430	304.770	223.135
WalkSAT	100	60082268	60082268	999.992	999.992
HSAT	100	34755535	34755535	999.988	999.988

Table 5.44: g250.29 : 7250v, 454622c, satisfiable, Graph Colouring, 29 colours (DIMACS - Johnson) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	124324252	983317711	999.992	7909
SAPS	100	265981061	1124974521	999.986	4229
DLM	100	133686839	563183568	999.803	4212
Novelty	100	262969415	692466145	999.970	2633
WalkSAT	100	272961571	702458301	999.993	2573
HSAT	100	284984039	284984039	999.945	1000

Table 5.45: hanoi4 : 718v, 4934c, satisfiable, Towers of Hanoi, 4 discs (DIMACS - Selman) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	100	3296137284	3296137284	3599.96	3599.96
SAPS	100	2147483647	2147483647	2066.26	2066.26
DLM	100	1482162860	1482162860	3599.95	3599.95
Novelty	100	2337616114	2337616114	3599.95	3599.95
WalkSAT	100	2290007697	2290007697	3599.95	3599.95
HSAT	100	374420859	374420859	3599.95	3599.95

Table 5.46: hanoi5 : 1931v, 14468c, satisfiable, Towers of Hanoi, 5 discs (DIMACS - Selman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	59019	36580	0.053	0.033
SAPS	0	69592	39488	0.105	0.060
DLM	0	73039	37522	0.108	0.056
Novelty	2	2583624	170791	7.692	197.706
WalkSAT	0	186449	109885	0.266	0.157

Table 5.47: hrs300 : 100 instances, 300v, 1278c, satisfiable, Hard Random 3-SAT (Southey) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	223232	116516	0.20	0.11
SAPS	0	247128	128124	0.39	0.20
DLM	0	331902	203192	0.56	0.34
WalkSAT	0	419925	188214	0.61	0.27

Table 5.48: hrs400 : 100 instances, 400v, 1704c, satisfiable, Hard Random 3-SAT (Southey) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	1521840	781969	1.5	0.7
SAPS	0	1878886	1165048	3.1	1.9
DLM	0	3534856	2249519	6.8	46.5
Novelty	5	8415356	31940201	97.8	1557.3
WalkSAT	0	1720098	1065062	2.4	1.5

Table 5.49: hrs500 : 134 instances, 500v, 2130c, satisfiable, Hard Random 3-SAT (Southey) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	2629	2610	0.004	0.004
SAPS	0	3634	3563	0.010	0.010
DLM	0	3657	3531	0.014	0.014
Novelty	0	22509	20683	0.043	0.039
WalkSAT	0	19443	18602	0.032	0.030
HSAT	100	466369	466369	1.006	1.006
GSAT	100	398754	398754	1.012	1.012

Table 5.50: huge : 459v, 7054c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	372	321	0.00088	0.00076
SAPS	0	373	350	0.00115	0.00107
DLM	0	311	297	0.00286	0.00272
Novelty	0	1796	1795	0.00433	0.00432
WalkSAT	0	507	497	0.00107	0.00105
HSAT	48	118307	69107	0.48865	0.28543
GSAT	37	74585	50885	0.41207	0.28113

Table 5.51: ii08 : 14 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	7086	6507	0.012	0.0106
SAPS	0	5369	5336	0.017	0.0170
DLM	0.3	8058	2534	0.068	0.0212
Novelty	0	6300	6300	0.029	0.0286
WalkSAT	0	6742	5973	0.025	0.0221
HSAT	71.9	1111708	490023	7.204	3.1753
GSAT	70.3	886078	666126	7.160	5.3829

Table 5.52: ii16 : 10 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	2464	2338	0.0080	0.008
SAPS	0	2172	2082	0.0127	0.012
DLM	0	1698	1430	0.0188	0.016
Novelty	0	1813	1807	0.0092	0.009
WalkSAT	0	1797	1544	0.0121	0.010
HSAT	97	2207161	1618471	4.8588	3.563
GSAT	34	740340	443557	2.4202	1.450

Table 5.53: ii32 : 17 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	903	818	0.0013	0.0012
SAPS	0	1219	1063	0.0028	0.0024
DLM	0	856	771	0.0021	0.0019
Novelty	0	7937	4555	0.0149	0.0085
WalkSAT	0	4544	3589	0.0077	0.0061
HSAT	72	606204	55277	0.7210	0.0657
GSAT	71	457743	51993	0.7206	0.0818

Table 5.54: jnh-sat : 16 instances, satisfiable, Constant Density Random (DIMACS - Hooker) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	10909	8724	0.013	0.011
SAPS	0	9333	8474	0.019	0.017
DLM	0	8020	6568	0.027	0.022
Novelty	3	183030	182610	0.272	0.271
WalkSAT	1	112115	109255	0.151	0.147
HSAT	100	351611	351611	1.006	1.006
GSAT	100	296223	296223	1.010	1.010

Table 5.55: logistics.a : 828v, 6718c, satisfiable, Logistics, 8 Packages, 11 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	6294	6288	0.008	0.008
SAPS	0	7901	7898	0.016	0.016
DLM	0	8380	8364	0.030	0.030
Novelty	0	257993	257993	0.435	0.435
WalkSAT	0	238438	238438	0.342	0.342
HSAT	100	702644	702644	2.005	2.005
GSAT	100	603221	603221	2.010	2.010

Table 5.56: logistics.b : 843v, 7301c, satisfiable, Logistics, 5 Packages, 13 Steps (SATLIB - Kautz and Selman) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	9655	9655	0.01	0.01
SAPS	0	12261	12185	0.03	0.03
DLM	0	12101	11805	0.05	0.05
Novelty	0	491330	486455	0.79	0.78
WalkSAT	0	626399	622267	0.87	0.86
HSAT	100	2697287	2697287	10.00	10.00
GSAT	100	2299466	2299466	10.01	10.01

Table 5.57: logistics.c : 1141v, 10719c, satisfiable, Logistics, 7 Packages, 13 Steps (SATLIB - Kautz and Selman) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	57616	57616	0.077	0.077
SAPS	0	64759	63962	0.153	0.151
DLM	0	36826	36826	0.282	0.282
Novelty	0	368847	368617	0.574	0.574
WalkSAT	0	1082225	1076011	1.507	1.498
HSAT	100	718465	718465	10.007	10.007
GSAT	100	612901	612901	10.015	10.015

Table 5.58: logistics.d : 4713v, 21991c, satisfiable, Logistics, 9 Packages, 14 Steps (SATLIB - Kautz and Selman) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	295	295	0.0004	0.000425
SAPS	0	307	306	0.0006	0.000634
DLM	0	264	264	0.0007	0.000740
Novelty	0	1685	1029	0.0026	0.001604
WalkSAT	0	1363	839	0.0018	0.001108
HSAT	89	1063188	891	0.8935	0.000749
GSAT	92	789398	1240	0.9333	0.001466

Table 5.59: medium : 116v, 953c, satisfiable, Blocks World 5 Blocks, 4 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	9688	8704	0.006	0.005
SAPS	4.4	246266	174883	0.279	0.198
DLM	0	10341	8844	0.010	0.009
Novelty	0	69915	38988	0.080	0.045
WalkSAT	2.4	206023	74103	0.225	0.081
HSAT	97.4	456550	116846	0.981	0.251
GSAT	96.6	347839	347839	0.986	0.986

Table 5.60: par08 : 5 instances, satisfiable, Unsimplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	2348	1851	0.0017	0.0013
SAPS	0	2805	2355	0.0032	0.0027
DLM	0	2131	1689	0.0020	0.0016
Novelty	0	8478	7248	0.0120	0.0103
WalkSAT	0	26153	15378	0.0289	0.0170
HSAT	96	1574456	4751	0.9671	0.0029
GSAT	94	1043777	7654	0.9550	0.0070

Table 5.61: par08c : 5 instances, satisfiable, Simplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	83426432	39690808	51.396	24
SAPS	94	237221537	211362467	285.597	923
DLM	0	22477451	13349478	30.985	18
Novelty	74	204219561	190118013	265.981	1560
WalkSAT	100	248334233	248334233	300.002	973
HSAT	100	53579208	53579208	300.001	300
GSAT	100	40658740	40658740	300.001	300

Table 5.62: par16 : 5 instances, satisfiable, Unsimplified Learning Parity Function, 16 orig. vars (DIMACS - Crawford) [Max Time: 300s Reps: 10]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	7453296	3516626	4.972	2
SAPS	0	9163962	5893089	11.020	113
DLM	0	6301159	3348791	8.484	5
Novelty	54	15662512	65089825	146.811	1351
WalkSAT	100	37156678	166005697	199.896	11864
HSAT	100	7242941	136091960	199.598	3750
GSAT	100	14449455	100348801	199.602	3418

Table 5.63: par16c : 5 instances, satisfiable, Simplified Learning Parity Function, 16 orig. vars (DIMACS - Crawford) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	850273	520475	5	3.0
SAPS	25	2881751	2718848	63	59.7
DLM	5	748581	533718	25	17.9
Novelty	15	9377355	6300449	43	344.4
WalkSAT	32	7646872	16203966	71	344.6

Table 5.64: qg-sat : 10 instances, satisfiable, Quasigroup (SATLIB - Zhang) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	3254	3229	0.0035	0.0034
SAPS	0	4071	3922	0.0066	0.0064
DLM	0	3157	3045	0.0112	0.0108
Novelty	46	440386	172858	0.9924	0.3895
WalkSAT	0	28995	27047	0.0311	0.0290
HSAT	100	346924	346924	2.0073	2.0073
GSAT	100	269911	269911	2.0080	2.0080

Table 5.65: ssa-sat : 4 instances, satisfiable, Circuit Single-Stuck-At Fault Analysis (DIMACS - van Gelder) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	2000	1583	0.0019	0.0015
SAPS	0	2442	2090	0.0034	0.0029
DLM	0	2074	1572	0.0027	0.0020
Novelty	0.25	37393	7986	0.0538	0.0115
WalkSAT	0	9097	6110	0.0118	0.0079
HSAT	79.02	1803987	4061	1.5849	0.0748
GSAT	78.93	1235662	42619	1.5968	0.1807

Table 5.66: uf125 : 100 instances, 125v, 538c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	2941	2339	0.0029	0.0023
SAPS	0	3534	2831	0.0050	0.0040
DLM	0	3300	2235	0.0044	0.0030
Novelty	0.2	84979	11488	0.1193	0.0161
WalkSAT	0	15137	8792	0.0201	0.0117
HSAT	79.0	7520830	310085	7.9104	11.1807
GSAT	78.2	5579970	272122	7.8727	5.0288

Table 5.67: uf150 : 100 instances, 150v, 645c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	6057	4768	0.006	0.004
SAPS	0	7310	5685	0.010	0.008
DLM	0	6667	4544	0.009	0.006
Novelty	0.3	175889	33515	0.244	0.046
WalkSAT	0	29673	17522	0.039	0.023
HSAT	84.5	7542404	353014	8.461	12.395
GSAT	84.0	5427843	413742	8.451	12.767

Table 5.68: uf175 : 100 instances, 175v, 753c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	12490	7349	0.011	0.007
SAPS	0	14725	11064	0.021	0.016
DLM	0	13209	8664	0.018	0.012
Novelty	2.3	560780	77850	0.775	0.460
WalkSAT	0.3	155891	121417	0.199	0.155
HSAT	89.1	14785617	2369082	17.823	231.079
GSAT	88.6	10582225	1595770	17.756	103.296

Table 5.69: uf200 : 100 instances, 200v, 860c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 20s Reps: 100]

Solver	Fail%	Av Steps	OEESS	Av Time(s)	OEERT(s)
ESGint	0	19110	12306	0.0170	0.011
SAPS	0	21803	15266	0.0320	0.022
DLM	0	23660	13836	0.0325	0.019
Novelty	1.3	515978	197095	0.7230	1.648
WalkSAT	0	61357	48831	0.0858	0.068
HSAT	88.3	12583352	757743	17.6723	104.660
GSAT	88.2	9220229	1763709	17.7269	155.703

Table 5.70: uf250 : 100 instances, 250v, 1065c, satisfiable, r=4.26, hard, random 3-SAT (SATLIB) [Max Time: 20s Reps: 100]

Chapter 6

ESG for Combinatorial Auctions

As explained in the previous chapter, ESG can be applied to arbitrary BLPs. In this chapter, we explore the effectiveness of this generality by applying ESG to *combinatorial auction winner determination* (CA). In doing so, we move away from pure constraint satisfaction into constrained optimization. We first discuss combinatorial auctions and their BLP formulation before describing other approaches to this problem. We conclude by empirically evaluating the approach.

6.1 Combinatorial Auctions

In a combinatorial auction, a variety of *goods* are made available for sale. Buyers submit *bids* on some subset of the available goods, specifying a price they are willing to pay for their desired subset (it is assumed that bid is "all or nothing", all desired goods must be sold to meet the bid). The winner determination problem is to determine the set of bids the seller should accept such that their earnings (the sum of accepted bid prices) are maximized, given the constraint that no good can be sold in greater quantity than is available. This is a constrained optimization problem suitable to subgradient optimization.

An auction can be easily expressed as a BLP. Consider a set of m *goods* $\mathbf{g} = \{g_1, \ldots, g_m\}$ available in quantities $\mathbf{q} = \{q_1, \ldots, q_m\}$, and a set of n *bid*s, each requesting a subset of the goods and offering a corresponding price, $\mathbf{v} = \{v_1, \ldots, v_n\}$. We can now define a set of boolean variables, $\mathbf{z} = \{z_1, \ldots, z_n\}$, corresponding to whether or not we accept each bid. The problem is most simply stated as:

$$\begin{aligned} & \max_{\mathbf{z}} \mathbf{v} \cdot \mathbf{z} \\ \text{subject to} \quad & C\mathbf{z} \le \mathbf{q} \\ \text{and} \quad & \mathbf{z} \in \{0,1\}^n \end{aligned} \tag{6.1}$$

where the entries in the constraint matrix are

$$c_{ij} = \begin{cases} 1 & \text{if the } j\text{th bid requests good } g_i \\ 0 & \text{otherwise} \end{cases}$$

To express this in our canonical form (5.1) where boolean variables take the $\{-1, 1\}$ domain and we are minimizing, we substitute $\mathbf{x} = 2\mathbf{z} - 1$, $\mathbf{d} = -\mathbf{v}/2$, and $\mathbf{b} = 2\mathbf{q} - C\mathbf{1}$. The Lagrangian values computed in this form can be transformed back into the original space to get the final solution for the original CA.

6.2 Other CA Solvers

At the time of this research [101], considerable attention was paid to the problem and various solvers were available. Of particular interest is the work of Holte, who showed that even very simple hill-climbers can achieve optimal or near-optimal results on a variety of randomly generated problems [57]. This suggests that more sophisticated local search-based methods might provide good performance. Some work had already been done in this direction by Hoos and Boutilier with the Casanova algorithm [59]. Casanova is loosely based on the Novelty+ algorithm for SAT, but is specialized to work on *single unit auctions*, and does not generalize to other problems. Single unit auctions are auctions in which only a single instance of each good is available for sale ($q_i = 1, \forall i$), and in which goods are not substitutable for each other. The results presented here consider only single-unit auctions to accommodate Casanova. CPLEX and ESG/ASG can be applied to multi-unit auctions without modification.

Another notable CA solver is the CASS solver [40], a CA-specific systematic approach that yielded good results. Sandholm developed similar algorithms [96, 97], eventually resulting in CABOB [98]. However, Andersson et al. [2] demonstrated that several existing random instance generators [96, 40] tended to generate trivial problems, and furthermore, that CPLEX, a general-purpose solver, offered performance competitive with that of specialized solvers.

6.3 CA Instances

The state of random CA instance generators improved with the introduction of the CATS generators [70]. There are five generators that attempt to model "real-world" CA scenarios. We will refer to these as CATS-regions, CATS-arbitrary, CATS-matching, CATS-paths, and CATS-scheduling. One older distribution was used, Decay.

Since even these generators tend to produce fairly easy instances [57], we create a still harder class of instances by transforming hard, random 3-SAT instances into single-unit CAs. We refer to this distribution as $SAT(class) \rightarrow CA$, where $class$ is the source of the transformed SAT instances. Where possible, we use CPLEX to determine the optimal value for the generated instances. The transformation constructs a CA instance that solves

MAX-SAT for the transformed SAT instance (see Algorithm 17 for details). The resulting CA instance is quadratic in the size of the SAT instance.

Algorithm 17 SAT-CA Reduction: A transformation encoding a MAX-SAT problem as a single unit combinatorial auction.

- Let a $Bid(v, t, c)$ represent the action of setting variable v to truth-value t in order to satisfy clause c.

- Thus the number of bids is linear in the length of the SAT problem (one bid for each literal in each clause).

- The *goods* that are being bid on are simply dummy (boolean) tokens used to enforce two kinds of mutual exclusion constraints:

  1. $Bid(v, t_1, ?)$ and $Bid(v, t_2, ?)$ are mutually exclusive if $t_1 \neq t_2$ because v cannot be assigned both truth values simultaneously. If variable v is positive in o^+ clauses and negative in o^- clauses there will be o^+o^- of these goods. For 3-SAT with a clause-variable ratio of r, there will be $9rn$ of these goods (roughly $38n$ at the phase transition).
  2. $Bid(?, ?, c)$ and $Bid(?, ?, c)$ are mutually exclusive because we only want to give credit once for satisfying the clause. For 3-SAT there will be exactly $3m$ of this type of good.

- Each bid is valued at 1 and the optimal value for a satisfiable instance is m (exactly one bid associated with each clause must be accepted and no more because of constraint (2)).

6.4 Comparison of Subgradient Optimization Methods

We compare our four variants of subgradient optimization (i.e. additive vs. multiplicative and linear vs. hinge penalty) on instances from the CATS-regions generator, reporting average runtime and average search steps to find the optimal value (subgradient optimization is incomplete so this is not a proof, simply the earliest point at which the optimal value was discovered), the percentage of runs that failed to find the optimal solution within 10,000 steps, and the average percentage of the optimal value achieved by the end of each run. Results were averaged over 100 runs of each instance and are shown in 6.1. The experiments were run on the environment in Appendix A.1.

As with the SAT instances, we see that the linear penalty function is markedly less effective than the hinge. However, the advantage of the multiplicative reweighting is less significant than in the SAT case.

	Av Time(s)	Av Steps	Fail%	Opt%
CATS-regions	(100 instances)			
$\text{ESG}_h(1.9, .1, .01)$	7.2	1416	4.1	99.93
$\text{ASG}_h(.045, 0, .01)$	12.7	2457	7.9	99.86
$\text{ESG}_\ell(1.3, .9, .01)$	64.7	7948	77	88.11
$\text{ASG}_\ell(.01, 0, .01)$	48.1	9305	90	93.96

Table 6.1: Comparison of subgradient optimization methods

6.5 Comparison of Available Methods

We compared CPLEX and Casanova (kindly provided by Hoos and Boutilier) with ESG and ASG, both using the hinge penalty. Casanova has the same parameters as Novelty+ (the first is the "pick second best" probability and the second is the random walk probability). ESG, ASG, and Casanova's results were averaged over 100 runs on each instance with steps limits (10,000 for regions, arbitrary, matching, and paths; 25,000 for scheduling and Decay; 10,000,000 for $SAT \rightarrow CA$) . CPLEX's results were averaged over single runs for each instance and, like the incomplete methods, the time shown is the time to first discover the optimal value rather than the time to prove its optimality. The results on the five CATS classes and the Decay class are shown in Table 6.2. The results on $SAT \rightarrow CA$ encoded instances are shown in Table 6.3. The experiments were run on the environment in Appendix A.1.

The overall results are fairly mixed, with no one solver dominating the others. CPLEX is arguably the best overall, especially given that no parameters had to be tuned. Casanova and ESG are basically tied in CATS-regions and CATS-matching, with Casanova notably better on CATS-arbitrary, CATS-paths, and CATS-scheduling, and Casanova completely dominating on the Decay instances. ASG never performs particularly well. The $SAT \rightarrow CA$ results show ASG is completely ineffective. ESG performs well compared with Casanova, but CPLEX is the clear winner, given the step limits on ESG (note that CPLEX executed for substantially longer than ESG).

While ESG is not the clear champion in these experiments, they nonetheless demonstrate that it is often competitive with other methods, including one specialized to the CA problem. The fact that general methods like ESG and CPLEX work well here is an encouraging sign that customized methods may not be necessary for many problems. It is also worth noting that the ESG implementation used here is the original and completely general version. It incorporates no implementation tricks specific to the domain and none of the many improvements developed subsequently (see the section on ESGint 5.7.2). We expect its runtime performance would improve dramatically if the implementation were updated.

	Av Time(s)	Av Steps	Fail%	Opt%
CATS-regions	(100 instances)			
CPLEX	6.7	64117	0	100
Casanova$(.5, .17)$	4.2	1404	3.4	99.95
ESG$_h(1.9, .1, .01)$	7.2	1416	4.1	99.93
ASG$_h(.045, 0, .01)$	12.7	2457	7.9	99.86
CATS-arbitrary	(100 instances)			
CPLEX	22	9510	0	100
Casanova$(.04, .12)$	9	2902	0.47	99.98
ESG$_h(2.1, .05, .5)$	33	7506	4.21	99.95
ASG$_h(.04, 0, .01)$	30	6492	4.87	99.87
CATS-matching	(100 instances)			
CPLEX	1.30	499	0	100
Casanova$(.3, .17)$	.17	109	0	100
ESG$_h(1.7, .05, 0)$	.16	215	0	100
ASG$_h(.9, 0, .8)$	.73	1248	0	100
CATS-paths	(100 instances)			
CPLEX	25	1	0	100
Casanova$(.5, .15)$	26	49	0	100
ESG$_h(1.3, .05, .4)$	28	2679	2.5	99.99
ASG$_h(.01, 0, .15)$	75	5501	6.8	99.96
CATS-scheduling	(100 instances)			
CPLEX	15	1426	0	100
Casanova$(.5, .04)$	44	7017	19.9	99.87
ESG$_h(1.35, .05, .015)$	65	11737	41	99.68
ASG$_h(.015, 0, .125)$	58	12925	44.2	99.51
Decay-200-200-.75	(100 instances)			
CPLEX	1.1	2014	0	100
Casanova$(.5, .17)$	0.5	2899	2	99.97
ESG$_h(14.5, .045, .45)$	1.8	24466	96.3	96.91
ASG$_h(.04, 0, .5)$	1.7	24939	99.6	91.49

Table 6.2: Results on CATS and synthetic instances

	Av Time(s)	Av Steps	Fail%	Opt%
SAT(uf50)→CA	(10 problems)			
CPLEX	42	754	0	100
Casanova$(.5, .11)$	468	9.6×10^6	90	99.36
ESG$_h(30, .05, .1)$	31	1.7×10^6	10	99.95
ASG$_h(5, 0, .5)$	173	8.6×10^6	80	99.40
SAT(uf75)→CA	(10 problems)			
CPLEX	666	5614	0	100
Casanova$(.5, .11)$	800	1.0×10^7	100	98.46
ESG$_h(41, .05, .1)$	165	6.2×10^6	60	99.81
ASG$_h(10, 0, .5)$	291	1.0×10^7	100	99.17

Table 6.3: Results on hard SAT→CA encoded problems

6.6 Related Work

ESG is not the first attempt at a general purpose BLP solver based on a SAT solver. DLM falls into this category (with some caveats about the nature of the objective function). We will briefly describe two other bodies of research, the first based on SAT local search, and the second based on DPLL solvers.

6.6.1 WSAT(OIP)

Walser has used a derivative of WalkSAT to solve BLP problems by converting the problem to an *over-constrained integer program* (OIP)[113]. An OIP consists of a set of *hard constraints*, corresponding to the constraints in the original BLP, and a set of *soft constraints* which take the place of the objective function. The WSAT(OIP) algorithm then operates something like the original WalkSAT.

Its objective is to minimize the number of violated soft constraints plus the weighted-sum of hard constraint violations. The soft constraints are unweighted so that a solution to OIP has the same value as the original BLP. The weights on the hard constraints were typically unit in Walser's experiments and were "statically set" on one problem. They do not change during the course of the search.

If both hard and soft constraints are violated, with probability p_{hard} it picks a hard constraint at random to satisfy, otherwise it picks a soft one. If only hard or only soft constraints are violated, it simply picks a constraint at random. Once a constraint has been violated, the variables are scored according to the objective just stated. Some are discarded on a tabu basis, and ties are broken in a manner similar to HSAT. If no improving variable is available, a random move is made with probability, ω, otherwise the least damaging move is selected.

This algorithm is fairly simple, and one can imagine variations and extensions to the search drawn from other local search methods. However, each BLP to be solved must

be converted, by hand, to the OIP form. Furthermore, it seems likely that the conversion selected will impact the performance of the search. Because of this translation, we do not regard WSAT(OIP) as falling into the same class as ESG, which requires no transformation of the original problem. Walser reports impressive results on several problems, compared with CPLEX.

6.6.2 Backtracking SAT-based ILP Methods

Barth formulated a 0-1 ILP method called *opbdp* [3] that iteratively selects a target value for the objective function and constructs a CNF instance for each successive value. The target values are examined in decreasing order until the instances are no longer satisfiable. Any complete SAT solver can serve as the underlying engine for this method.

Manquinho *et al.* [78, 77] describe an ILP solver called *bsolo* that combines branch-and-bound search with GRASP [79], following along the lines of opbdp. They show good results compared to CPLEX and opbdp.

6.7 Conclusions

The work presented here demonstrates that ESG is a viable option on at least one BLP other than SAT, and a constrained optimization problem at that. The results also reaffirm our feature choices, with multiplicative updates and hinge penalty functions still the best option. Comparison with both a completely specialized solver and a high-performance general-purpose solver nicely frames the ESG results. It would be interesting to test ESG on other BLP domains but we leave this for future research.

However, it is worth noting that Elidan *et al.* used an optimization technique directly inspired by ESG [29] to improve optimization for machine learning. In this context it was dubbed *adversarial optimization*. Although the optimization problem in this case was unconstrained, it nonetheless produced good results. While not a direct application of ESG, this nonetheless provides encouraging evidence for the generality of ESG's overall approach.

We will now leave general BLPs behind to return to SAT, exploring other directions in which local search may be augmented for SAT.

Chapter 7

Extending Local Search Via Hybrids

That which is static and repetitive is boring. That which is dynamic and random is confusing. In between lies art.
- John A. Locke (1632-1704)

While both backtrack and local search methods form active research areas, there is surprisingly little exchange of ideas between them. The different strengths and weaknesses of each approach offer compelling reasons to consider how their strengths can be combined. Backtrack searchers are particularly strong on structured instances and are currently the only practical option for unsatisfiable instances. Local search methods offer good performance on satisfiable instances, particularly the class of hard, random instances found at the phase transition [83] where backtrack methods run into serious difficulties.

This research seeks to augment local search by investigating hybrids of ESG with various features found in high-performance backtracking search. Constructing and observing such hybrids has produced interesting results, and leads us to believe that the ideas underlying these two schools can be united to produce solvers that are robust to various problem classes.

7.1 Previous Work On Hybrids

Attempts have been made at combining local and backtrack search in the past and, recently, interest in this area seems to be growing. In an early hybrid, Crawford ran a clause-weighting local search method to learn "difficult" clauses offline and then used them to order variable choices in a backtracking search [21]. The learn-sat algorithm repeatedly "probes" a problem by creating random partial assignments and recording a nogood clause to avoid revisiting that assignment [91]. Prestwich's IDB performs a local search through consistent partial assignments on SAT and other CSPs [88]. The UnitWalk algorithm employs the unit-propagation mechanism found in backtracking solvers to drive a local searcher [54, 55, 56]. WalkSATZ [52] augments the backtrack solver SATZ [72] by

applying the WalkSAT [105] local search method to solving subproblems based on equivalence classes. The CSP literature also contains several studies of interest [120, 64]

7.2 Hybrid: DualRes

One direction for producing hybrids is to incorporate clause learning strategies into local search. As described previously in Section 2.5.8.3, Cha and Iwama developed an algorithm called ANC that generates resolvents from unsatisfied clauses encountered at local minima and adds these to the formula [16]. While not the motivation that drove their research, it is clear that adding one or more clauses guaranteed to be false at a given local minimum will penalize local searchers that explore the area, discouraging them from revisiting it. We refined this basic algorithm by considering what resolvents could be efficiently produced at a local minimum, producing a new algorithm that we call *DualRes*.

7.2.1 Resolution Strategy

Recall (from Section 2.1.4) that in order to resolve two clauses they must have complementary literals, a positive and negative occurrence of one, and only one, variable. In order to produce a resolvent, a variable must be selected, along with two associated clauses containing the literals to be resolved (the resolvands). If we are trying to penalize assignments at a local minimum, it makes sense to focus on the clauses that are unsatisfied at that point. Note that it is impossible for two unsatisfied clauses to be resolvands (if they share a common variable it must have the same sign). Therefore, if we wish to resolve against a set of false clauses, we must select among the satisfied clauses.

Given an assignment reached at time t, $\mathbf{a}^{(t)}$, let the sets of satisfied and unsatisfied clauses at that assignment be $\mathbf{SAT}(\mathbf{a}^{(t)})$ and $\mathbf{UNSAT}(\mathbf{a}^{(t)})$, respectively. Suppose we reach a local minimum at time t. We could examine all the clauses $\mathbf{UNSAT}(\mathbf{a}^{(t)})$ and try to find suitable resolvands amongst $\mathbf{SAT}(\mathbf{a}^{(t)})$. Even if the number of unsatisfied clauses is small (which is typically true), the number of candidate pairs is likely to be too large for any efficient selection other than at random. We would like to restrict the possible candidates, ideally in a manner that prefers resolvands of interest at the local minimum. One strategy for limiting is to consider only clauses nearly related to the local minimum, such as those changed by the most recent step in the search,[1] the step that led to the local minimum. Therefore, we consider the state of clauses at the previous assignment, $\mathbf{a}^{(t-1)}$.

Along these lines, we partition the clauses into four sets:

- Still Satisfied: $\mathbf{S}_{still} = \mathbf{SAT}(\mathbf{a}^{(t-1)}) \cap \mathbf{SAT}(\mathbf{a}^{(t)})$
- Still Unsatisfied: $\mathbf{U}_{still} = \mathbf{UNSAT}(\mathbf{a}^{(t-1)}) \cap \mathbf{UNSAT}(\mathbf{a}^{(t)})$
- Newly Satisfied: $\mathbf{S}_{new} = \mathbf{UNSAT}(\mathbf{a}^{(t-1)}) \cap \mathbf{SAT}(\mathbf{a}^{(t)})$

[1] As usual in local search methods, we take a step to be the flipping of a single variable.

- Newly Unsatisfied: $\mathbf{U}_{new} = \mathbf{SAT}(\mathbf{a}^{(t-1)}) \cap \mathbf{UNSAT}(\mathbf{a}^{(t)})$

The latter two sets represent the clauses changed by the most recent step, and it is these that we examine when considering possible resolvands. Note that every clause in $\mathbf{S}_{new}$ is resolvable with every clause $\mathbf{U}_{new}$, and that only a single variable is suitable for resolution, namely, the variable that was flipped in the last step. The resolvents generated this way are always false at the local minimum (because the resolvands are a newly satisfied and new unsatisfied clause, all literals other than the resolving literals must be unsatisfied), and are thus guaranteed to increase the penalty for visiting the local minimum. This nicely restricts the candidate resolvands and it is reasonably inexpensive to compute $\mathbf{S}_{new}$ and $\mathbf{U}_{new}$, either by tracking clause state changes as they occur or by checking the effects of reversing the most recent step when a minimum is reached (our implementation uses the former approach).

There are endless possible strategies for deciding how many and which resolvents to add. After experimenting with a few options, we chose to add, at most, one new clause per clause in $\mathbf{U}_{new}$, because the size of $\mathbf{U}_{new}$ is typically small (note, however, that we may replace any number of existing clauses with resolvents; we are only concerned with trying to keep the total number of clauses manageable). This strategy at least attempts to address all violated constraints in the local minimum (adding huge numbers of resolvents is counter-productive). In order to prevent the formula from growing too quickly, we decided to check for subsumption[2] of the resolvands ($c_s \in \mathbf{S}_{new}$) and $c_u \in \mathbf{U}_{new}$) by the resolvent, r. If r subsumes either of its resolvands, it replaces the parent and the formula keeps the same number of clauses. If it subsumes both resolvands, it replaces both and the formula actually shrinks. If it replaces neither, the formula grows by at most one clause.

We consider several possible resolvents for each clause in $\mathbf{U}_{new}$, but always accept resolvents that subsume either or both of their resolvands (clearly, if we subsume the current c_u we do not need to consider any more resolvents for it). In the event that no resolvent subsumes the current c_u, and that more than one non-c_s-subsuming resolvent is generated, we must decide which of these new resolvents to add, since we are only going to accept one. We requires some "score" by which to compare resolvents.

One obvious basis for a score is the length of the resolvent. Large clauses do not constrain the search space much and so shorter clauses are preferable. However, we usually have other criteria in the form of the objective function used for our local search algorithm. In our case, DualRes uses ESG as its local search mechanism (hence the name, short for "dual step resolution"), although the general idea applies to any local searcher. The most natural scoring for us to consider in ESG is the weight of a clause.

When adding a new resolvent in DualRes, we must give it an initial weight. One choice would be to assign it a unit weight. However, the resolvents generated by DualRes are always false at the local minimum, and are meant to penalize the minimum, so we would like a less trivial weight. We weight the new resolvent, r, by combining the weights of its resolvands via the following formula: $\lambda_r \leftarrow (\lambda_{c_s} + \lambda_{c_u})/3$. This ensures that the resolvent

[2]See Section 2.1.5 for background on subsumption.

has a weight comparable to its resolvands, but not likely to exceed either of them (unless one is much larger than the other), a conservative strategy that errs on the side of undervaluing the new constraint. If the resolvent deserves higher weight, it will eventually receive it in the natural course of the ESG search, whereas if it is overweight to start with, it may needlessly perturb the search space.

Finally, we come to the scoring. We combine weight and the length of the resolvent relative to its resolvands to score a resolvent: $score_r \leftarrow \lambda_r/(k_r - min(k_{c_s}, k_{c_u}))$. This scoring prefers resolvents with high weight and that are not much longer than their shortest resolvand. Algorithm 18 shows the entire process in pseudo-code. The local search part is simply the ESG algorithm. Every p_{res} dual steps, the resolution procedure is run at the beginning of the dual step, before any weight changes (p_{res} is a parameter for the algorithm and was set to 5 in our experiments).

There are a few other details. If r is the empty clause, we can simply return unsatisfiable as the decision. If r is a unit clause, we perform unit propagation, deleting satisfied clauses and reducing the rest. We also perform some extra subsumption checks, checking if any clause in $\mathbf{S}_{new}$ subsumes any other clause in the same set. We know that they all contain a common literal at the very least so it's an interesting set to check for subsumptions.

7.2.2 DualRes Experiments

A large set of experiments were conducted with DualRes, comparing with ESG (specifically ESGflt), zChaff, and SATZ on both satisfiable and unsatisfiable instances (see the backtrack search comparisons for a discussion of DualRes and unsatisfiable instances). ESG and the ESG-portion of DualRes used the parameter tunings from [101] and were allowed up to 500,000 steps on the largest instances. zChaff and SATZ were run using their default parameters and with no step limit. Some entries in this study were not completed when the overall outcome became clear but there are more than enough results to draw strong conclusions. Experiments were run in the environment described in Appendix A.2 and with the software from Appendix B.1. The results are provided in the tables at the end of the section 7.2.2, and are discussed in the following two subsections. The average steps reported refers to primal steps for ESG and DualRes, and choice moves for SATZ and zChaff. We also reported OEESS, the fraction of runs that failed, and the number of clauses added and deleted by the various solvers.

7.2.2.1 Local Search Comparison: ESG vs. DualRes

The resolution portion of DualRes is quite expensive. We conducted several experiments on satisfiable instances to determine its effectiveness vs. ESG. In particular, we expected runtimes to suffer substantially and were chiefly curious to discover whether the actual number of search steps was substantially reduced. We found the following set of outcomes:

- Substantial reductions in search steps and runtime on: aim-sat-100, aim-sat-200, bw_large.c, bw_large.d, flat100, flat200, logistics.d, par08, and par08c.

Algorithm 18 DualRes Resolution Algorithm

dualResolutionStep()

1. for each $c_u \in \mathbf{U}_{new}$
   (a) $score_{best} \leftarrow 0$
   (b) for each $c_s \in \mathbf{S}_{new}$
      i. resolve c_s and c_u to produce r
      ii. if $r = \emptyset$, return unsatisfiable
      iii. if r is a unit clause, perform unit propagation, goto step 1b and continue with next c_s
      iv. $\lambda_r \leftarrow (\lambda_{c_s} + \lambda_{c_u})/3$
      v. if r subsumes both c_s and c_u
         A. delete c_s and c_u
         B. add r
         C. goto step 1 and continue with next c_u
      vi. else if r subsumes c_u only
         A. delete c_u
         B. add r
         C. goto step 1 and continue with next c_u
      vii. else if r subsumes c_s only
         A. delete c_s
         B. add r
         C. goto step 1b and continue with next c_s
      viii. else r subsumes neither c_s nor c_u
         A. $score_r \leftarrow \lambda_r/(k_r - min(k_{c_s}, k_{c_u}))$
         B. if $score_r > score_{best}$ then $score_{best} \leftarrow score_r$;$r_{best} \leftarrow r$
   (c) add r_{best} (the highest scored resolvent for the current c_u)

- Substantial reductions in search steps but inferior runtime on: ais*, bw_large.b, hrs275, hrs300, huge, medium, jnh, and ssa.
- Better success rates on: bmc-galileo*, bmc_ibm*, hanoi4, par16, par16c (ESG never solved some of these problems within the allotted limits)
- Strictly worse: logistics.c
- Mixed results on random 3-SAT instances: uf* and hrs*.

7.2.2.2 Backtrack Search Comparison: DualRes vs. Chaff

Unlike regular local search methods, it is possible for DualRes to prove unsatisfiability by deriving the empty clause through resolution. DualRes never discards learned clauses, so a proof can eventually be constructed if sufficient storage is available. Unfortunately, proof sizes can be exponential in the size of the original problem and only by constructing a complete proof can DualRes draw such a conclusion, whereas a systematic searcher tracks what assignments have been explored and so need not store the entire proof (although it must, in effect, construct it). DualRes may therefore run out of memory before completing the proof.

The second caveat is that we have no proof that the algorithm described will eventually generate the entire refutation. No provision is made to explicitly avoid creating duplicate resolvents (although the subsumption mechanism will delete the duplicates), so it is possible that the algorithm could fall into some cycle, repeated regenerating a subsection of the proof. Therefore, we can only say that DualRes is *potentially* complete. We ran several experiments with unsatisfiable instances and, in practice, we found that, if sufficient time was allowed, DualRes did prove unsatisfiability. We have no evidence that it becomes stuck. Obviously, ESG is not appropriate for these experiments.

The results for the backtracking solvers and DualRes include the number of clauses added and deleted. SATZ does not have clause learning, but performs some limited resolution before starting the search and adds the resulting clauses.[3] DualRes deletes clauses via the subsumption mechanism and zChaff deletes clauses by bounded relevance. The overall observations of DualRes relative to zChaff and SATZ were:

- DualRes is generally in last place in terms of time.
- Fewer clauses added by DualRes than zChaff for some instances: aim-no, ais*, bf, bw_large.c and bw_large.d.
- On random 3-SAT, DualRes outperforms zChaff but not SATZ.
- DualRes notably outperforms SATZ on logistics.c.

[3] SATZ reports 0 steps on some instances (aim*). The limited resolution it performs actually solves the aim instances directly so this is not an error.

- DualRes can clearly solve the smaller unsatisfiable instances but does not scale well in this regard (running out of steps, not memory).

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.00419	0.00419	1573	1570	0	0
DualRes	0	0.00390	0.00390	430	430	22	26
SATZ	0	0.00017	0.00017	1	1	128	0
zChaff	0	0.00141	0.00141	115	115	39	0

Table 7.1: aim-sat-100 : 16 instances, 100v, satisfiable, AIM Random Generator (DIMACS - Asahiro / Iwama / Miyano) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0.0031	0.1112	0.098	44271	38912	0	0
DualRes	0	0.0813	0.0812	12681	12665	582	273
SATZ	0	0.0003	0.0003	1	1	0	0
zChaff	0	0.0108	0.0108	522	522	196	0

Table 7.2: aim-sat-200 : 16 instances, 200v, satisfiable, AIM Random Generator (DIMACS - Asahiro / Iwama / Miyano) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	0.000914846	0.000914846	70.135	70.135	13	14
SATZ	0	3.6e-05	3.6e-05	0.375	0.375	67	0
zChaff	0	0.000337875	0.000337875	53.5	53.5	14	0

Table 7.3: aim-unsat-050 : 8 instances, 50v, unsatisfiable, AIM Random Generator (DIMACS - Asahiro / Iwama / Miyano) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	0.00194	0.00194	122	122	19	16
SATZ	0	6.35e-05	6.35e-05	1	1	71	0
zChaff	0	0.00065	0.00065	111	111	23	0

Table 7.4: aim-unsat-100 : 8 instances, 100v, unsatisfiable, AIM Random Generator (DIMACS - Asahiro / Iwama / Miyano) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	0.0045	0.0045	227	227	28	20
SATZ	0	0.0001	0.0001	1	1	158	0
zChaff	0	0.0019	0.0019	416	416	40	0

Table 7.5: aim-unsat-200 : 8 instances, 200v, unsatisfiable, AIM Random Generator (DIMACS - Asahiro / Iwama / Miyano) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.00271	0.00266	496	485	0	0
DualRes	0	0.00279	0.00279	96	96	12	0
SATZ	0	0.00164	0.00164	6	6	0	0
zChaff	0	0.00077	0.00077	37	37	16	0

Table 7.6: ais06 : 61v, 518c, satisfiable, All-Interval Series (SATLIB) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.042	0.030	5410	3843	0	0
DualRes	0	0.051	0.035	890	601	183	10
SATZ	0	0.006	0.006	10	10	0	0
zChaff	0	0.025	0.025	520	520	370	0

Table 7.7: ais08 : 113v, 1520c, satisfiable, All-Interval Series (SATLIB) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.19	0.19	18644	18203	0	0
DualRes	0	0.58	0.58	47366	4736	1033	55
SATZ	0	0.02	0.02	25	25	0	0
zChaff	0	1.18	1.18	7778	7778	6972	0

Table 7.8: ais10 : 181v, 3151c, satisfiable, All-Interval Series (SATLIB) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	1.72	1.32	123327	95026	0	0
DualRes	0	10.71	7.11	22848	15171	5749	366
SATZ	0	0.09	0.09	58	58	0	0
zChaff	0	26.20	26.20	50995	50995	46236	0

Table 7.9: ais12 : 265v, 5666c, satisfiable, All-Interval Series (SATLIB) [Max Steps: 500,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	0.13	0.13	4495	4495	106	5
SATZ	0	0.99	0.99	264	264	0	0
zChaff	0	0.03	0.03	495	495	149	0

Table 7.10: bf : 4 instances, unsatisfiable, Circuit Bridge Fault Analysis (DIMACS - van Gelder) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	1	198	198	5.0e+06	5.0e+06	0	0
DualRes	0.52	556	556	4.2e+06	4.2e+06	3813	3357
SATZ	0	60649	60649	230814	230814	0	0
zChaff	0	212	212	245053	245053	43957	0

Table 7.11: bmc-galileo : 2 instances, satisfiable, BMC on Galileo FIFO (IBM - Shtrichman) [Max Steps: 5,000,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0.64	24	24	356145	356145	503	73
zChaff	0	81	81	132227	132227	18262	0

Table 7.12: bmc-ibm : 11 instances, satisfiable, BMC on various IBM designs (IBM - Shtrichman) [Max Steps: 500,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.020	0.012	3004	3004	0	0
DualRes	0	0.210	0.210	3103	3103	199	131
SATZ	0	0.016	0.016	1	1	392	0
zChaff	0	0.004	0.004	42	42	15	0

Table 7.13: bw_large.a : 459v, 4675c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.40	0.35	41110	36192	0	0
DualRes	0	3.78	3.78	31353	31353	2255	1495
SATZ	0	0.04	0.04	2	2	1236	0
zChaff	0	0.03	0.03	255	255	88	0

Table 7.14: bw_large.b : 1087v, 13772c, satisfiable, Blocks World 11 Blocks, 9 Steps (SATLIB - Kautz and Selman) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0.11	48.5	21.8	1.5e+06	692683	0	0
DualRes	0	4.3	4.3	28523	28523	1349	5747
SATZ	0	1.1	1.1	4	4	4106	0
zChaff	0	1.3	1.3	3218	3218	1890	0

Table 7.15: bw_large.c : 3016v, 50457c, satisfiable, Blocks World 15 Blocks, 14 Steps (SATLIB - Kautz and Selman) [Max Steps: 5,000,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	93	93	1.9e+06	1.9e+06	0	0
DualRes	0	77	61	201693	161560	8323	12554
SATZ	0	87	87	326	326	9448	0
zChaff	0	18	18	23409	23409	15954	0

Table 7.16: bw_large.d : 6325v, 131973c, satisfiable, Blocks World 19 Blocks, 18 Steps (SATLIB - Kautz and Selman) [Max Steps: 5,000,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0.077	0.013	0.013	802	802	142	85
zChaff	0	0.002	0.002	116	116	53	0

Table 7.17: dubois: 13 instances, unsatisfiable, Randomly Generated (DIMACS - Dubois) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.021	0.019	8119	7254	0	0
DualRes	0	0.013	0.013	1095	1090	31	0
SATZ	0	0.004	0.004	22	22	0	0
zChaff	0	0.002	0.002	55	55	19	0

Table 7.18: flat100 : 100 instances, 300v, 1117c, satisfiable, Flat 3-Colourable Graphs, 100 vert, 239 edges, (SATLIB) [Max Steps: 500,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0.057	0.68	0.49	236824	171451	0	0
DualRes	0	0.46	0.45	26234	25619	650	9
SATZ	0	0.10	0.10	183	183	0	0
zChaff	0	0.44	0.44	3231	3231	2026	0

Table 7.19: flat200 : 100 instances, 600v, 2237c, satisfiable, Flat 3-Colourable Graphs, 200 vert, 479 edges, (SATLIB) [Max Steps: 500,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	199	199	218526	218264	20161	924
SATZ	0	101	101	124974	124974	0	0
zChaff	0	4	4	17755	17755	13782	0

Table 7.20: hanoi4 : 718v, 4934c, satisfiable, Towers of Hanoi, 4 discs (DIMACS - Selman) [Max Steps: 500,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.10	0.07	25492	17910	0	0
DualRes	0	2.45	1.63	25729	17081	1435	14
SATZ	0	1.58	1.58	3012	3012	91	0
zChaff	0	443.20	443	294009	294009	222278	0

Table 7.21: hrs275 : 100 instances, 275v, 1172c, satisfiable, Hard Random 3-SAT (Waterloo) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.15	0.11	39393	28920	0	0
DualRes	0	6.78	4.37	43407	28014	2226	18
SATZ	0	3.64	3.64	6918	6918	91	0
zChaff	0	304.96	304.96	320474	320474	239568	0

Table 7.22: hrs300 : 100 instances, 300v, 1278c, satisfiable, Hard Random 3-SAT (Waterloo) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	0.102	0.102	5600	5600	492	320
SATZ	0	0.009	0.009	27	27	126	0
zChaff	0	0.033	0.033	758	758	614	0

Table 7.23: hru100 : 100 instances, 100v, 426c, unsatisfiable, Hard Random 3-SAT (Waterloo) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	4.55	4.55	70471	70470	4522	2698
SATZ	0	0.06	0.06	136	136	105	0
zChaff	0	0.64	0.64	7524	7524	5910	0

Table 7.24: hru150 : 100 instances, 150v, 639c, unsatisfiable, Hard Random 3-SAT (Waterloo) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0.69	109.6	109.6	391189	391189	19995	5909
SATZ	0	0.3	0.3	699.5	699	96	0
zChaff	0	16.1	16.1	60523	60523	46728	0

Table 7.25: hru200 : 100 instances, 200v, 852c, unsatisfiable, Hard Random 3-SAT (Waterloo) [Max Steps: 500,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	1	111.3	111.3	443717	443717	21281	996
SATZ	0	1.8	1.8	3537	3537	95	0
zChaff	0	347.7	347.7	379950	379950	290275	0

Table 7.26: hru250 : 100 instances, 250v, 1065c, unsatisfiable, Hard Random 3-SAT (Waterloo) [Max Steps: 500,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.060	0.055	7650	7006	0	0
DualRes	0	0.373	0.373	3229	3229	271	412
SATZ	0	0.015	0.0153	1	1	0	0
zChaff	0	0.004	0.004	51	51	16	0

Table 7.27: huge : 459v, 7054c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	0.3	0.3	17449	16700	102	22
zChaff	0	8.6	8.6	12477	12477	7248	0

Table 7.28: ii16 : 10 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	0.06	0.02	2356	876	19	2
zChaff	0	0.04	0.04	991	991	130	0

Table 7.29: ii32 : 17 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	0.014	0.014	461	461	57	126
zChaff	0	0.005	0.005	113	113	77	0

Table 7.30: jnh-unsat : 34 instances, unsatisfiable, Constant Density Random (DIMACS - Hooker) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0.0006	0.0222	0.0075	3688	1241	0	0
DualRes	0	0.0106	0.0106	319	319	41	64
SATZ	0	0.0039	0.0039	6	6	0	0
zChaff	0	0.0035	0.0035	101	101	49	0

Table 7.31: jnh-sat : 16 instances, satisfiable, Constant Density Random (DIMACS - Hooker) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.10	0.10	8276	8276	0	0
DualRes	0	2.49	2.49	13336	13336	771	52
SATZ	0	0.31	0.31	779	779	909	0
zChaff	0	0.09	0.09	12351	12351	380	0

Table 7.32: logistics.c : 1141v, 10719c, satisfiable, Logistics, 7 Packages, 13 Steps (SATLIB - Kautz and Selman) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.70	0.70	46441	46441	0	0
DualRes	0	0.67	0.67	8565	8565	97	63
SATZ	0	563.27	563.27	58824	58824	0	0
zChaff	0	0.04	0.04	603	603	45	0

Table 7.33: logistics.d : 4713v, 21991c, satisfiable, Logistics, 9 Packages, 14 Steps (SATLIB - Kautz and Selman) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.0019	0.0019	379	379	0	0
DualRes	0	0.0114	0.0114	286	286	23	1
SATZ	0	0.0013	0.0013	1	1	0	0
zChaff	0	0.0004	0.0004	8	8	1	0

Table 7.34: medium : 116v, 953c, satisfiable, Blocks World 5 Blocks, 4 Steps (SATLIB - Kautz and Selman) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0.174	0.534	0.409	211284	162043	0	0
DualRes	0	0.008	0.008	836	836	8	0
SATZ	0	0.003	0.003	6	6	0	0
zChaff	0	0.001	0.001	12	12	8	0

Table 7.35: par-08 : 5 instances, satisfiable, Unsimplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.00606	0.00531	2501	2194	0	0
DualRes	0	0.00222	0.00222	270	270	9	22
SATZ	0	0.00065	0.00065	3	3	0	0
zChaff	0	0.00061	0.00061	12	12	8	0

Table 7.36: par-08c : 5 instances, satisfiable, Simplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	1	2.6	2.6	947629	947629	0	0
DualRes	0.13	40.5	38.1	436242	410733	9053	156
SATZ	0	5.8	5.8	1358	1358	0	0
zChaff	0	1.1	1.1	4355	4355	4223	0

Table 7.37: par-16 : 5 instances, satisfiable, Unsimplified Learning Parity Function, 16 orig. vars (DIMACS - Crawford) [Max Steps: 1,000,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0.89	2.4	2.4	873778	873778	0	0
DualRes	0.11	18.0	18.0	429083	423117	8659	3562
SATZ	0	1.1	1.1	1060	1060	84	0
zChaff	0	0.7	0.7	4444	4444	4340	0

Table 7.38: par-16c : 5 instances, satisfiable, Simplified Learning Parity Function, 16 orig. vars (DIMACS - Crawford) [Max Steps: 1,000,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
DualRes	0	0.10	0.10	3321	3321	134	20
SATZ	0	73.35	73.35	74513	74513	2513	0
zChaff	0	0.02	0.02	415	415	169	0

Table 7.39: ssa-unsat : 4 instances, unsatisfiable, Circuit Singe-Stuck-At Fault Analysis (DIMACS - van Gelder) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.020	0.016	4453	3523	0	0
DualRes	0	0.024	0.024	1437	1437	3	0
SATZ	0	0.145	0.145	72	72	0	0
zChaff	0	0.003	0.003	108	108	2	0

Table 7.40: ssa-sat : 4 instances, satisfiable, Circuit Singe-Stuck-At Fault Analysis (DIMACS - van Gelder) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.00076	0.0007	212	186	0	0
DualRes	0	0.00141	0.0013	188	173	15	0
SATZ	0	0.00147	0.0015	4	4	406	0
zChaff	0	0.00049	0.0005	30	30	12	0

Table 7.41: uf050 : 1000 instances, 50v, 218c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.003	0.003	984	858	0	0
DualRes	0	0.010	0.009	529	512	53	4
SATZ	0	0.006	0.006	15	15	123	0
zChaff	0	0.008	0.008	228	228	156	0

Table 7.42: uf100 : 1000 instances, 100v, 430c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0	0.01	0.008	2899	2236	0	0
DualRes	0	0.04	0.028	3829	3011	194	0
SATZ	0	0.03	0.024	50	50	102	0
zChaff	0	0.10	0.103	1548	1548	1157	0

Table 7.43: uf150 : 100 instances, 150v, 645c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0.0022	0.04	0.03	11660	8213	0	0
DualRes	0.0001	0.36	0.31	7525	6371	526	17
SATZ	0	0.12	0.12	241	241	101	0
zChaff	0	2.40	2.40	14307	14307	10912	0

Table 7.44: uf200 : 100 instances, 200v, 860c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 50,000 Reps: 100]

Solver	FailFrac	Av Time(s)	OEERT(s)	Av Steps	OEESS	Cl+Avg	Cl-Avg
ESGflt	0.0001	0.06	0.050	16783	13201	0	0
DualRes	0.0002	0.91	0.667	14118	10364	858	11
SATZ	0	0.70	0.704	1344	1344	94	0
zChaff	0	62.49	62.49	103669	103669	78602	0

Table 7.45: uf250 : 100 instances, 250v, 1065c, satisfiable, hard, random 3-SAT (SATLIB) [Max Steps: 50,000 Reps: 100]

7.2.3 Comments on DualRes

The results from DualRes are interesting, despite its overall poor performance compared to the systematic solvers. The improvement it offers as an augmentation of ESG can be quite dramatic on structured problems, and it does not appear to seriously damage ESG's performance on random instances. On the other hand, it clearly cannot compete with the systematic solvers on most structured instances. On these, it typically adds many more clauses than zChaff, suggesting that it is finding poor resolvents. Runtime performance is its biggest downfall. The resolvent selection and subsumption mechanisms are very expensive.

It is possible the runtime could be improved, but we believe there is a deeper problem here. As noted before, every backtrack search corresponds to a resolution proof. However, the reverse is not true. Not every resolution proof corresponds to a backtrack search. A variable ordering, even a dynamic variable ordering, imposes structure on the proof by performing resolution in a systematic manner. While one ordering may offer better proof sizes than another, it seems likely that almost any ordering is superior to simply generating resolvents at random. If we compare the space of possible assignments, 2^n, with the space of possible resolvents, 3^n, a lack of guidance would seem to be fatal.[4]

DualRes lacks this guidance because there is no explicit variable ordering behind it, so it tends to construct "multiple" proofs corresponding to several orderings, each containing resolvents of no use to the other proofs. This key insight made the DualRes exercise very worthwhile and helped determine the next direction for research.

7.3 Hybrid: RESG

We will only comment briefly on this intermediate hybrid, built by linking ESG to the original zChaff implementation. The algorithm, inspired by the results of DualRes, similarly

[4]The space of possible resolvents is 3^n in the worst possible case. If we consider a clause, it can be at most length n and each variable may appear positively, negatively, or not at all. A formula containing 2^n clauses corresponding to each possible assignment to the variables, can produce all 3^n resolvents.

performs resolution at local minima. However, the *RESG* algorithm (for *resolving ESG*) uses zChaff's mechanism to supply a variable ordering. The value ordering, on the other hand, is taken from ESG's full assignment at the local minimum.

A single *probe* of zChaff is executed at the minimum, building up a partial assignment in the order that zChaff dictates, but using ESG's value choices, until a conflict (or solution) is detected. The 1-UIP resolvent used by zChaff is computed and added back into the ESG side of the solver and the search continues. This operation is performed every fixed number of dual steps.

This solver, really a combination of two distinct solvers, is quite slow, but offers much smaller proof sizes than DualRes and is overall quite encouraging. It was abandoned in favour of developing backtrack search mechanisms from scratch that could coexist with local search mechanisms, resulting in the software described in Appendix B.2. The basic concept behind RESG is subsumed by ESGProbe, which we will now discuss in detail.

7.4 Hybrid: ESGProbe

Our final algorithm, *ESGProbe*, is essentially a refinement of the RESG approach. As in RESG, ESGProbe runs ESGint and executes a probe when a local minimum is reached. The values used are taken from the current full assignment, giving a set of literals to probe. These literals are ordered by their frequency of occurrence in the formula, with more frequent literals preferred. The probe then asserts them in order, propagating literals, until a conflict is reached. At that point, a 1-UIP resolvent is computed, added to the formula, and given unit weight.

We also consider a new feature. Any disagreement between the partial assignment constructed by the probe, and the full assignment of ESGint is reconciled by altering the assignment in ESG (similar to the mechanism used by UnitWalk [54, 55, 56]). We call this *assignment repair*, and it allows the probe to more immediately impact the local search, rather than waiting for the effects of added clauses to kick in (see Algorithm 19 for an outline).

7.5 Experiments

We constructed ESGProbe using several state-of-the-art solver components. On the backtrack side, we adopt the fast unit propagation data structures (clause witnesses) used in SATO [125] and Chaff [126]. For clause learning, we employ the single UIP cut approach used by Chaff. On the local search side we use ESGint (see Section 5.7.2). The variable ordering is fairly naive, similar to that used by Chaff.

7.5.1 Three Variations

Experiments were run on some variations of this basic framework to establish which of the features are effective:

Algorithm 19 ESGProbe

1. randomly select initial assignment
2. repeat until satisfied, proved unsat, or timed out
   (a) run ESGint until a dual step is encountered
   (b) perform the usual dual step
   (c) order literals
   (d) run probe with UP
      i. clause learning: if contradiction is found, compute conflict clause and add it to the clause database
      ii. Optionally perform *assignment repair*: update full assignment with assignment changes that have been forced by UP during the probe

- ESGProbe: the basic algorithm
- ESGProbe-CL: the basic algorithm with clause learning disabled
- ESGProbe+AR: the basic algorithm plus assignment repair

We conducted a small study, comparing these three variations. The study reports the percentage of failed runs, the average time and averages, the average number of choices made during probes, the average number of unit propagations during probes, and the average number of clauses learned. Since some of these values are not applicable to some solvers (e.g. ESGint does not do unit propagation), an asterisk is used to mark the omission.

As the results shown at the end of this section demonstrate, clause learning offers a substantial improvement in performance, especially on structured instances. The relationship between ESGProbe and ESGProbe+AR is more complicated. While ESGProbe+AR frequently leads to more search steps for the local solver, it conversely makes the probes more efficient, with fewer decisions and unit propagations. Assignment repair appears to damage ESG's local search by altering the primal variables dramatically but leaving the duals unchanged. Since there is good evidence (e.g. the effectiveness of smoothing) that primal and dual variables have only a local relationship, this seems quite probable. On the other hand, assignment repair seems to lead to fewer dual steps (note that the number of dual steps is equal to the number of clauses added show in the results), and correspondingly less work on probes. Probing is computationally quite expensive, so the net effect is to improve runtimes.

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.0030	269	631	2903	105
ESGPrb+AR	0	0.0023	350	454	2092	76
ESGPrb-CL	0	0.0100	1352	4566	15172	0

Table 7.46: RTI : 500 instances, 100v, 426c, satisfiable, Random 3-SAT (SATLIB - Singer) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.0025	570	296	3674	23
ESGPrb+AR	0	0.0023	580	251	3190	19
ESGPrb-CL	0	0.0074	1169	980	19347	0

Table 7.47: SW100-8-8 : 100 instances, 500v, 3100c, satisfiable, Morphed 5-Colourable Graphs, ratio 2≙8 (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.00069	176	199	610	28
ESGPrb+AR	0	0.00068	203	196	595	26
ESGPrb-CL	6	0.38271	71180	575700	490802	0

Table 7.48: aim-sat-100 : 16 instances, 100v, satisfiable, AIM Random Generator (DIMACS - Asahiro/Iwama/Miyano) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.00032	46	100	318	9
ESGPrb+AR	0	0.00028	55	85	253	8
ESGPrb-CL	0	0.00076	122	410	1136	0

Table 7.49: ais06 : 61v, 518c, satisfiable, All-Interval Series (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.0045	242	1803	6700	134
ESGPrb+AR	0	0.0041	448	1450	5931	118
ESGPrb-CL	0	0.0131	1525	8657	25905	0

Table 7.50: ais08 : 113v, 1520c, satisfiable, All-Interval Series (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.000121	39	3	75	1
ESGPrb+AR	0	0.000277	37	2	69	1
ESGPrb-CL	0	0.000118	38	3	74	0

Table 7.51: anomaly : 48v, 261c, satisfiable, Blocks World 3 Blocks, 3 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.00237	365	91	2971	13
ESGPrb+AR	0	0.00228	378	84	2706	12
ESGPrb-CL	0	0.01089	865	744	23702	0

Table 7.52: bw_large.a : 459v, 4675c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.018333	1206	874	30702	83
ESGPrb+AR	0	0.018327	1505	859	30497	85
ESGPrb-CL	5	0.313181	19371	28272	704601	0

Table 7.53: bw_large.b : 1087v, 13772c, satisfiable, Blocks World 11 Blocks, 9 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	65	3.9	112039	497798	2193513	17264
ESGPrb+AR	81	4.4	241042	469767	2204030	17583
ESGPrb-CL	41	3.3	376143	1487998	4261052	0

Table 7.54: f0600 : 600v, 2550c, satisfiable, Large Random 3-SAT (DIMACS - Selman) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.0024	416	250	4734	38
ESGPrb+AR	0	0.0019	536	180	3376	26
ESGPrb-CL	0	0.0069	1254	1004	17348	0

Table 7.55: flat100 : 100 instances, 300v, 1117c, satisfiable, Flat 3-Colourable Graphs, 100 vert, 239 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.007	867	596	12950	95
ESGPrb+AR	0	0.005	1189	417	9005	64
ESGPrb-CL	0	0.019	3455	2793	49304	0

Table 7.56: flat125 : 100 instances, 375v, 1403c, satisfiable, Flat 3-Colourable Graphs, 125 vert, 301 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.018	1855	1296	32206	209
ESGPrb+AR	0	0.014	2835	984	24399	159
ESGPrb-CL	0	0.049	8741	6737	124277	0

Table 7.57: flat150 : 100 instances, 450v, 1680c, satisfiable, Flat 3-Colourable Graphs, 150 vert, 360 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.040	3528	2508	68146	389
ESGPrb+AR	0	0.032	5573	1925	52144	301
ESGPrb-CL	0	0.103	17806	14017	260059	0

Table 7.58: flat175 : 100 instances, 525v, 1951c, satisfiable, Flat 3-Colourable Graphs, 175 vert, 417 edges, (SATLIB) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.11	8160	5820	167870	857
ESGPrb+AR	0.02	0.10	13613	4846	137723	718
ESGPrb-CL	0.26	0.27	47682	37061	672593	0

Table 7.59: flat200 : 100 instances, 600v, 2237c, satisfiable, Flat 3-Colourable Graphs, 200 vert, 479 edges, (SATLIB) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.084	2198	2089	27023	11
ESGPrb+AR	0	0.132	4025	3699	48275	20
ESGPrb-CL	0	0.130	3969	3567	46924	0

Table 7.60: g250.15 : 3750v, 233965c, satisfiable, Graph Colouring, 15 colours (DIMACS - Johnson) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.00252	358	76	2672	11
ESGPrb+AR	0	0.00251	373	77	2547	11
ESGPrb-CL	0	0.00903	728	556	16966	0

Table 7.61: huge : 459v, 7054c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.00178	194	1063	767	10
ESGPrb+AR	0	0.00187	200	1072	776	10
ESGPrb-CL	0	0.00921	347	6725	13759	0

Table 7.62: ii08 : 14 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.0028	333	174	1686	8
ESGPrb+AR	0	0.0025	291	95	984	4
ESGPrb-CL	0	0.0061	677	187	4001	0

Table 7.63: ii32 : 17 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.0020	154	277	1706	49
ESGPrb+AR	0	0.0015	190	213	1224	35
ESGPrb-CL	0	0.0143	1414	4368	17591	0

Table 7.64: jnh-sat : 16 instances, satisfiable, Constant Density Random (DIMACS - Hooker) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.05	2479	21362	64252	419
ESGPrb+AR	0	0.03	1831	18674	40731	273
ESGPrb-CL	11	0.41	11267	730364	415190	0

Table 7.65: logistics.a : 828v, 6718c, satisfiable, Logistics, 8 Packages, 11 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.05	2001	20986	68224	367
ESGPrb+AR	0	0.04	1691	20014	49868	262
ESGPrb-CL	32	1.26	36472	1695664	1477146	0

Table 7.66: logistics.b : 843v, 7301c, satisfiable, Logistics, 5 Packages, 13 Steps (SATLIB - Kautz and Selman) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.000301	82	10	181	1
ESGPrb+AR	0	0.000323	86	11	209	1
ESGPrb-CL	0	0.000294	83	12	185	0

Table 7.67: medium : 116v, 953c, satisfiable, Blocks World 5 Blocks, 4 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.00076	191	24	967	10
ESGPrb+AR	0	0.00078	210	25	935	10
ESGPrb-CL	0	0.02348	3336	1490	48340	0

Table 7.68: par08 : 5 instances, satisfiable, Unsimplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.000315	82	19	356	8
ESGPrb+AR	0	0.000312	94	19	321	7
ESGPrb-CL	0	0.001914	626	241	3222	0

Table 7.69: par08c : 5 instances, satisfiable, Simplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.00336	958	188	3823	3
ESGPrb+AR	0	0.00326	917	180	3509	3
ESGPrb-CL	2	0.07096	2786	4488	152132	0

Table 7.70: ssa-sat : 4 instances, satisfiable, Circuit Single-Stuck-At Fault Analysis (DIMACS - van Gelder) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGProbe	0	0.009	642	1693	8207	251
ESGPrb+AR	0	0.007	965	1288	6426	199
ESGPrb-CL	0.03	0.036	4286	16631	55064	0

Table 7.71: uf125 : 100 instances, 125v, 538c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 2s Reps: 100]

7.5.2 Broader Comparisons

Beyond this study, comparisons were also made with solvers (from the Appendix B.2 collection) including Chaffish,[5] plain ESGint, and SATZ[6] (we drop ESGProbe-CL since it is clearly an overall poor choice). Default parameters were used in all cases and we report the same statistics as in the previous study. The results for stochastic solvers were averaged over repeated runs (specified in the table captions) and all solvers were limited in time (also described by captions). Experiments were run in the environment described in Appendix A.3. The tables for this study can be found at the end of this section.

Overall, ESGProbe and ESGProbe-AR exhibit some positive results in this wider comparison. While they rarely dominate on any set of instances, they sometimes beat the *best portfolio* solution. Portfolios are collections of solvers run in some interleaved fashion [62, 46]. If we take a pure local searcher and a pure backtrack searcher to make a portfolio, running them interleaved with equal time slices, then this composite method takes double the time of the best of the two (double the minimum). Any hybrid must beat the portfolio solver to be worthwhile, which ESGProbe and ESGProbe-AR do in some cases.

The hybrid dramatically improves ESGint's performance on several structured instances. Unfortunately, performance on random 3-SAT instances is damaged more than we expected. Clause learning has very few benefits for random 3-SAT so their addition to the instance simply increases the computation burden. We speculate that the additional of bounded relevance deletion (see Section 2.4.8.1) for clauses might substantially mitigate this effect without damaging structured performance.

[5]Chaffish is our own implementation of zChaff. It has all the features of zChaff but does not reclaim memory from deleted clauses (see Appendix B.2). We used Chaffish because it is usually faster when memory is not an issue and we were unable to make zChaff run correctly in the environment described in Appendix A.3.

[6]This is a "bridged" version of SATZ 2.14 (see Appendix B.2).

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.01240	15689	*	*	*
ESGProbe	0	0.01798	881	3095	15317	517
ESGPrb+AR	0	0.01246	1474	2143	11043	376
Chaffish	0	0.01251	*	856	15262	646
SATZ	0	0.00480	*	47	*	63

Table 7.72: BMS : 500 instances, 100v, satisfiable, Backbone-Minimal Subinstances from RTI (SATLIB - Singer) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00090	976	*	*	*
ESGProbe	0	0.00314	269	631	2903	105
ESGPrb+AR	0	0.00236	350	454	2092	76
Chaffish	0	0.00240	*	209	3138	142
SATZ	0	0.00228	*	15	*	125

Table 7.73: RTI : 500 instances, 100v, 426c, satisfiable, Random 3-SAT (SATLIB - Singer) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0417	49314	*	*	*
ESGProbe	0	0.0028	570	296	3674	23
ESGPrb+AR	0	0.0025	580	251	3190	19
Chaffish	0	0.0016	*	64	6321	33
SATZ	0	0.0082	*	4	*	0

Table 7.74: SW100-8-8 : 100 instances, 500v, 3100c, satisfiable, Morphed 5-Colourable Graphs, ratio 2ˆ8 (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.01955	29237	*	*	*
ESGProbe	0	0.00070	176	199	610	28
ESGPrb+AR	0	0.00069	203	196	595	26
Chaffish	0	0.00052	*	124	731	46
SATZ	0	0.00020	*	1	*	256

Table 7.75: aim-sat-100 : 16 instances, 100v, satisfiable, AIM Random Generator (DIMACS - Asahiro/Iwama/Miyano) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	20	64.4935	19544851	*	*	*
ESGProbe	0	0.0037	676	1032	3047	83
ESGPrb+AR	0	0.0032	784	965	2757	72
Chaffish	0	0.0024	*	459	3747	141
SATZ	0	0.0002	*	1	*	325

Table 7.76: aim-sat-200 : 16 instances, 200v, satisfiable, AIM Random Generator (DIMACS - Asahiro/Iwama/Miyano) [Max Time: 300s Reps: 10]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00066	478	*	*	*
ESGProbe	0	0.00034	46	100	318	9
ESGPrb+AR	0	0.00029	55	85	253	8
Chaffish	0	0.00046	*	36	281	16
SATZ	0	0.00094	*	6	*	0

Table 7.77: ais06 : 61v, 518c, satisfiable, All-Interval Series (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0073	4959	*	*	*
ESGProbe	0	0.0046	242	1803	6700	134
ESGPrb+AR	0	0.0042	448	1450	5931	118
Chaffish	0	0.0080	*	588	8828	412
SATZ	0	0.0024	*	10	*	0

Table 7.78: ais08 : 113v, 1520c, satisfiable, All-Interval Series (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.048	27165	*	*	*
ESGProbe	0	0.035	975	7392	48622	540
ESGPrb+AR	0	0.024	1752	5193	36130	409
Chaffish	0	0.038	*	1560	41544	1331
SATZ	0	0.009	*	25	*	0

Table 7.79: ais10 : 181v, 3151c, satisfiable, All-Interval Series (SATLIB) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.37	186704	*	*	*
ESGProbe	0	0.47	6720	31646	531235	3342
ESGPrb+AR	0	0.78	22785	29662	602946	3699
Chaffish	0	3.81	*	29984	1096828	27518
SATZ	0	0.03	*	58	*	0

Table 7.80: ais12 : 265v, 5666c, satisfiable, All-Interval Series (SATLIB) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.000139	94	*	*	*
ESGProbe	0	0.000129	39	3	75	1
ESGPrb+AR	0	0.000121	37	2	69	1
Chaffish	0	0.000288	*	3	76	1
SATZ	0	0.000426	*	1	*	0

Table 7.81: anomaly : 48v, 261c, satisfiable, Blocks World 3 Blocks, 3 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.16	52728	*	*	*
ESGProbe	0	1.46	11088	32180	665218	2716
ESGPrb+AR	0	0.68	13224	21358	483729	1770
Chaffish	0	0.31	*	4455	179626	3008
SATZ	0	0.02	*	18	*	0

Table 7.82: beijing-bw : 3 instances, satisfiable, Beijing Blocks World - Sussman anomalies (Crawford) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.16	52443	*	*	*
ESGProbe	0.2	1.02	16743	17811	965449	233
ESGPrb+AR	0	0.98	20177	17574	1022008	266
Chaffish	0	0.59	*	11746	869129	1356
SATZ	83.3	11.80	*	8	*	0

Table 7.83: beijing-jobshop : 6 instances, satisfiable, Beijing Job Shop (Crawford) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	39	4.0727	4765358	*	*	*
ESGProbe	0	0.0031	255	989	4515	33
ESGPrb+AR	0	0.0028	270	797	3801	28
Chaffish	0	0.5176	*	4769	162040	3775
SATZ	0	0.0093	*	50	*	0

Table 7.84: beijing-vlsi-2bit-sat : 4 instances, satisfiable, Beijing 2-bit VLSI design (Crawford) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	1000	421343986	*	*	*
ESGProbe	0	5	62461	283388	7180564	3001
ESGPrb+AR	0	2	28664	121866	2649360	1114
SATZ	50	616	*	1066462	*	0

Table 7.85: beijing-vlsi-3bit-sat : 2 instances, satisfiable, Beijing 3-bit VLSI design (Crawford) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3600	1054235450	*	*	*
ESGProbe	0	31	654925	137821	21478486	2000
ESGPrb+AR	0	30	450982	87497	28612648	1954
Chaffish	0	44	*	207492	63514166	41224
SATZ	100	3599	*	46586	*	0

Table 7.86: bmc-galileo-8 : 58074v, 294821c, satisfiable, BMC on Galileo FIFO #1, 35 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3600	1244855540	*	*	*
ESGProbe	0	64	1032783	159224	45302516	3838
ESGPrb+AR	0	51	774138	128433	42832022	3001
Chaffish	0	53	*	248159	69653529	46249
SATZ	100	3581	*	39274	*	0

Table 7.87: bmc-galileo-9 : 63624v, 326999c, satisfiable, BMC on Galileo FIFO #2, 38 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	2887.6	2337456763	*	*	*
ESGProbe	0	5.8	98228	30019	4945966	2787
ESGPrb+AR	0	4.6	100082	26083	3732897	2256
Chaffish	0	0.8	*	44091	1246677	3205
SATZ	100	3600.0	*	1740087	*	0

Table 7.88: bmc-ibm-1 : 9685v, 55870c, satisfiable, BMC on IBM CPU Part 1, 18 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3600	1593363747	*	*	*
ESGProbe	0	128	1639151	451221	68249395	9319
ESGPrb+AR	0	77	671565	335586	47851484	5511
Chaffish	0	139	*	1010901	91237968	86579
SATZ	0	1001	*	5514	*	0

Table 7.89: bmc-ibm-10 : 61088v, 334861c, satisfiable, BMC on IBM PowerPC BIU #2, 13 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3600	1543519827	*	*	*
ESGProbe	0	53	1333529	98626	32791812	6221
ESGPrb+AR	0	23	455110	57266	15359391	3294
Chaffish	0	16	*	146594	22227682	27635
SATZ	0	1508	*	32369	*	0

Table 7.90: bmc-ibm-11 : 32109v, 150027c, satisfiable, BMC on IBM Cache Control #1, 32 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.96	1204058869	*	*	*
ESGProbe	0	433.78	7031836	250386	288352800	27007
ESGPrb+AR	0	96.12	1257214	115481	84415004	8387
Chaffish	0	61.51	*	257673	77809342	61634
SATZ	100	3599.98	*	22672	*	0

Table 7.91: bmc-ibm-12 : 39598v, 19477c, satisfiable, BMC on IBM PowerPC BIU #3, 31 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3600	3081768985	*	*	*
ESGProbe	0	25	498693	103487	23417782	7225
ESGPrb+AR	0	70	1630833	182224	57055661	19288
Chaffish	0	5	*	39264	8211406	13634
SATZ	100	3564	*	151627	*	0

Table 7.92: bmc-ibm-13 : 13215v, 6572c, satisfiable, BMC on IBM Cache Control #2, 14 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.048	58360	*	*	*
ESGProbe	0	0.027	1981	665	19082	64
ESGPrb+AR	0	0.015	1420	428	10815	28
Chaffish	0	0.003	*	289	7275	27
SATZ	0	0.031	*	17	*	0

Table 7.93: bmc-ibm-2 : 3628v, 14468c, satisfiable, BMC on IBM CPU Part 2, 5 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3600.0	2090007126	*	*	*
ESGProbe	0	6.3	203820	13416	4661693	2118
ESGPrb+AR	0	1.6	55179	6805	1818814	516
Chaffish	0	0.5	*	2892	1125731	1596
SATZ	0	25.9	*	413	*	0

Table 7.94: bmc-ibm-3 : 14930v, 72106c, satisfiable, BMC on IBM BIU 1996, 14 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3600	2454418981	*	*	*
ESGProbe	0	8	145055	21769	4363206	1517
ESGPrb+AR	0	7	113695	21633	3783923	1285
Chaffish	0	1	*	15272	1900785	2729
SATZ	0	23	*	940	*	0

Table 7.95: bmc-ibm-4 : 28161v, 139716c, satisfiable, BMC on IBM PowerPC BIU #1, 24 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	3.28	3387884	*	*	*
ESGProbe	0	0.41	12302	4909	161295	280
ESGPrb+AR	0	0.11	6253	2747	60004	54
Chaffish	0	0.05	*	3248	124940	365
SATZ	0	0.52	*	248	*	0

Table 7.96: bmc-ibm-5 : 9396v, 41207c, satisfiable, BMC on IBM Arbiter #1, 12 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3600	1033771043	*	*	*
ESGProbe	0	56	282422	46490	9037676	6353
ESGPrb+AR	0	35	196472	42236	6743319	3605
Chaffish	0	5	*	20893	4347790	9427
SATZ	100	3600	*	1479165	*	0

Table 7.97: bmc-ibm-6 : 51654v, 368367c, satisfiable, BMC on IBM LSU 1997, 22 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.028	16677	*	*	*
ESGProbe	0	0.128	6923	3068	87112	65
ESGPrb+AR	0	0.201	7166	3550	118807	140
Chaffish	0	0.018	*	1176	36825	68
SATZ	0	0.386	*	49	*	0

Table 7.98: bmc-ibm-7 : 8710v, 39774c, satisfiable, BMC on IBM Arbiter #2, 9 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0038	2998	*	*	*
ESGProbe	0	0.0025	365	91	2971	13
ESGPrb+AR	0	0.0024	378	84	2706	12
Chaffish	0	0.0013	*	50	2430	13
SATZ	0	0.0059	*	1	*	392

Table 7.99: bw_large.a : 459v, 4675c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0580	40857	*	*	*
ESGProbe	0	0.0207	1206	874	30702	83
ESGPrb+AR	0	0.0204	1505	859	30497	85
Chaffish	0	0.0138	*	309	38814	141
SATZ	0	0.0159	*	2	*	1236

Table 7.100: bw_large.b : 1087v, 13772c, satisfiable, Blocks World 11 Blocks, 9 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	26.87	14740392	*	*	*
ESGProbe	0	0.93	15900	11626	1156618	1265
ESGPrb+AR	0	0.97	23617	10456	1200907	1305
Chaffish	0	0.29	*	4255	725986	1718
SATZ	0	0.39	*	4	*	4106

Table 7.101: bw_large.c : 3016v, 50457c, satisfiable, Blocks World 15 Blocks, 14 Steps (SATLIB - Kautz and Selman) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	30	966	309865108	*	*	*
ESGProbe	0	10	102705	63478	10003989	5403
ESGPrb+AR	0	18	205569	74452	16318415	9005
Chaffish	0	6	*	30264	8832616	15646
SATZ	0	34	*	326	*	9448

Table 7.102: bw_large.d : 6325v, 131973c, satisfiable, Blocks World 19 Blocks, 18 Steps (SATLIB - Kautz and Selman) [Max Time: 2000s Reps: 10]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.230	244142	*	*	*
ESGProbe	68	4.006	101952	454388	1991519	15746
ESGPrb+AR	89	4.754	257107	499132	2354597	18789
Chaffish	100	5.004	*	42507	2157328	28478
SATZ	100	5.009	*	16601	*	79

Table 7.103: f0600 : 600v, 2550c, satisfiable, Large Random 3-SAT (DIMACS - Selman) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0059	8363	*	*	*
ESGProbe	0	0.0025	416	250	4734	38
ESGPrb+AR	0	0.0019	536	180	3376	26
Chaffish	0	0.0005	*	53	1758	19
SATZ	0	0.0015	*	22	*	0

Table 7.104: flat100 : 100 instances, 300v, 1117c, satisfiable, Flat 3-Colourable Graphs, 100 vert, 239 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0133	19550	*	*	*
ESGProbe	0	0.0071	867	596	12950	95
ESGPrb+AR	0	0.0050	1189	417	9005	64
Chaffish	0	0.0019	*	130	6397	67
SATZ	0	0.0028	*	34	*	0

Table 7.105: flat125 : 100 instances, 375v, 1403c, satisfiable, Flat 3-Colourable Graphs, 125 vert, 301 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.037	54673	*	*	*
ESGProbe	0	0.018	1855	1296	32206	209
ESGPrb+AR	0	0.015	2835	984	24399	159
Chaffish	0	0.009	*	362	24316	232
SATZ	0	0.006	*	51	*	0

Table 7.106: flat150 : 100 instances, 450v, 1680c, satisfiable, Flat 3-Colourable Graphs, 150 vert, 360 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.1045	154588	*	*	*
ESGProbe	0	0.0421	3528	2508	68146	389
ESGPrb+AR	0	0.0319	5573	1925	52144	301
Chaffish	0	0.0324	*	1004	73471	643
SATZ	0	0.0124	*	88	*	0

Table 7.107: flat175 : 100 instances, 525v, 1951c, satisfiable, Flat 3-Colourable Graphs, 175 vert, 417 edges, (SATLIB) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0.18	0.27	394386	*	*	*
ESGProbe	0	0.12	8160	5820	167870	857
ESGPrb+AR	0.04	0.10	13750	4895	139419	726
Chaffish	0	0.14	*	3099	243225	1957
SATZ	0	0.03	*	183	*	0

Table 7.108: flat200 : 100 instances, 600v, 2237c, satisfiable, Flat 3-Colourable Graphs, 200 vert, 479 edges, (SATLIB) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.9600	103070212	*	*	*
ESGProbe	0	768.4400	125264	3137551	555425080	55305
SATZ	0	0.0005	*	0	*	2531

Table 7.109: fvp-sat-2 : 1 instance, satisfiable, Formal Verification Buggy 7 Pipeline SS Processor(Velev) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.104	22937	*	*	*
ESGProbe	56	6.926	45044	424529	9583647	6776
ESGPrb+AR	90	9.478	126306	562711	11886539	9268
Chaffish	100	10.008	*	31623	1474744	20226
SATZ	100	10.005	*	10728	*	0

Table 7.110: g125.18 : 2250v, 70163c, satisfiable, Graph Colouring, 18 colours (DIMACS - Johnson) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0666	2196	*	*	*
ESGProbe	0	0.1007	2198	2089	27023	11
ESGPrb+AR	0	0.1399	4025	3699	48275	20
Chaffish	100	5.0096	*	21412	2327494	12435
SATZ	100	5.0098	*	365	*	0

Table 7.111: g250.15 : 3750v, 233965c, satisfiable, Graph Colouring, 15 colours (DIMACS - Johnson) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	100	1000	124324252	*	*	*
ESGProbe	0	2	41112	52790	1393541	8253
ESGPrb+AR	0	44	398098	188437	5283535	34046
Chaffish	0	1	*	14156	783529	11245
SATZ	0	38	*	124974	*	0

Table 7.112: hanoi4 : 718v, 4934c, satisfiable, Towers of Hanoi, 4 discs (DIMACS - Selman) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.05	59019	*	*	*
ESGProbe	0.6	5.43	42516	130688	677371	10153
ESGPrb+AR	2.8	12.43	124511	177939	973447	14825
Chaffish	52.0	118.43	*	174025	5751835	130236
SATZ	0	1.36	*	6918	*	91

Table 7.113: hrs300 : 100 instances, 300v, 1278c, satisfiable, Hard Random 3-SAT (Southey) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.2	223232	*	*	*
ESGProbe	7	24.0	145735	460290	2295994	26982
ESGPrb+AR	12	37.7	383815	589687	3096395	36581
Chaffish	84	169.5	*	230719	9097437	167921
SATZ	0	43.9	*	202600	*	86

Table 7.114: hrs400 : 100 instances, 400v, 1704c, satisfiable, Hard Random 3-SAT (Southey) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0042	2629	*	*	*
ESGProbe	0	0.0028	358	76	2672	11
ESGPrb+AR	0	0.0026	373	77	2547	11
Chaffish	0	0.0015	*	46	2579	12
SATZ	0	0.0056	*	1	*	0

Table 7.115: huge : 459v, 7054c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00088	372	*	*	*
ESGProbe	0	0.00191	194	1063	767	10
ESGPrb+AR	0	0.00197	200	1072	776	10
Chaffish	0	0.00105	*	456	2135	27
SATZ	0	0.00589	*	10	*	0

Table 7.116: ii08 : 14 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.012	7086	*	*	*
ESGProbe	0	0.023	2366	8186	18028	75
ESGPrb+AR	0	0.018	2016	6063	11734	50
Chaffish	10	1.096	*	6864	578132	3906
SATZ	0	0.094	*	36	*	0

Table 7.117: ii16 : 10 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0080	2464	*	*	*
ESGProbe	0	0.0031	333	174	1686	8
ESGPrb+AR	0	0.0025	291	95	984	4
Chaffish	0	0.0090	*	958	11014	89
SATZ	6	0.6231	*	92	*	0

Table 7.118: ii32 : 17 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.001340	903	*	*	*
ESGProbe	0	0.002110	154	277	1706	49
ESGPrb+AR	0	0.001564	190	213	1224	35
Chaffish	0	0.001340	*	102	1193	49
SATZ	0	0.001958	*	6	*	0

Table 7.119: jnh-sat : 16 instances, satisfiable, Constant Density Random (DIMACS - Hooker) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.013	10909	*	*	*
ESGProbe	0	0.055	2479	21362	64252	419
ESGPrb+AR	0	0.034	1831	18674	40731	273
Chaffish	0	0.010	*	6263	19711	206
SATZ	100	1.003	*	8575	*	616

Table 7.120: logistics.a : 828v, 6718c, satisfiable, Logistics, 8 Packages, 11 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0082	6294	*	*	*
ESGProbe	0	0.0537	2001	20986	68224	367
ESGPrb+AR	0	0.0367	1691	20014	49868	262
Chaffish	0	0.0130	*	7518	27153	236
SATZ	0	0.0085	*	64	*	615

Table 7.121: logistics.b : 843v, 7301c, satisfiable, Logistics, 5 Packages, 13 Steps (SATLIB - Kautz and Selman) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.01	9655	*	*	*
ESGProbe	0	0.11	3467	38054	117833	603
ESGPrb+AR	0	0.07	2757	35858	81546	429
Chaffish	0	0.03	*	15267	40329	303
SATZ	0	0.13	*	779	*	909

Table 7.122: logistics.c : 1141v, 10719c, satisfiable, Logistics, 7 Packages, 13 Steps (SATLIB - Kautz and Selman) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.08	57616	*	*	*
ESGProbe	0	0.19	5239	3596	362459	159
ESGPrb+AR	0	0.10	4543	2107	210387	90
Chaffish	0	0.04	*	1657	129590	231
SATZ	100	10.01	*	2455	*	0

Table 7.123: logistics.d : 4713v, 21991c, satisfiable, Logistics, 9 Packages, 14 Steps (SATLIB - Kautz and Selman) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00043	295	*	*	*
ESGProbe	0	0.00032	82	10	181	1
ESGPrb+AR	0	0.00033	86	11	209	1
Chaffish	0	0.00984	*	7	151	1
SATZ	0	0.00080	*	1	*	0

Table 7.124: medium : 116v, 953c, satisfiable, Blocks World 5 Blocks, 4 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00585	9688	*	*	*
ESGProbe	0	0.00082	191	24	967	10
ESGPrb+AR	0	0.00087	210	25	935	10
Chaffish	0	0.00037	*	12	731	9
SATZ	0	0.00122	*	6	*	0

Table 7.125: par08 : 5 instances, satisfiable, Unsimplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00166	2348	*	*	*
ESGProbe	0	0.00033	82	19	356	8
ESGPrb+AR	0	0.00032	94	19	321	7
Chaffish	0	0.00026	*	11	302	8
SATZ	0	0.00039	*	2	*	0

Table 7.126: par08c : 5 instances, satisfiable, Simplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	51.4	83426432	*	*	*
ESGProbe	0	7.4	160422	66581	3167715	13061
ESGPrb+AR	0	2.3	233586	34012	1519084	5815
Chaffish	0	1.1	*	9320	1828442	9145
SATZ	0	2.0	*	1358	*	0

Table 7.127: par16 : 5 instances, satisfiable, Unsimplified Learning Parity Function, 16 orig. vars (DIMACS - Crawford) [Max Time: 300s Reps: 10]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	4.97	7453296	*	*	*
ESGProbe	0	2.12	85712	45767	1075106	8047
ESGPrb+AR	0	1.35	161092	37340	863509	6365
Chaffish	0	0.41	*	5787	527033	5549
SATZ	0	0.38	*	1059	*	84

Table 7.128: par16c : 5 instances, satisfiable, Simplified Learning Parity Function, 16 orig. vars (DIMACS - Crawford) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	4.9	850273	*	*	*
ESGProbe	0.5	5.8	48697	44334	856466	5107
ESGPrb+AR	3.8	19.0	151771	60545	1405335	9452
Chaffish	0	2.0	*	6239	683943	5591
SATZ	0	3.9	*	1219	*	0

Table 7.129: qg-sat : 10 instances, satisfiable, Quasigroup (SATLIB - Zhang) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00345	3254	*	*	*
ESGProbe	0	0.00387	958	188	3823	3
ESGPrb+AR	0	0.00340	917	180	3509	3
Chaffish	0	0.00082	*	120	1609	4
SATZ	0	0.04812	*	72	*	0

Table 7.130: ssa-sat : 4 instances, satisfiable, Circuit Single-Stuck-At Fault Analysis (DIMACS - van Gelder) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.002	2000	*	*	*
ESGProbe	0	0.009	642	1693	8207	251
ESGPrb+AR	0	0.007	965	1288	6426	199
Chaffish	0	0.012	*	722	13749	536
SATZ	0	0.005	*	30	*	112

Table 7.131: uf125 : 100 instances, 125v, 538c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.003	2941	*	*	*
ESGProbe	0	0.017	1063	2932	14513	383
ESGPrb+AR	0	0.014	1613	2202	11290	300
Chaffish	0	0.044	*	1629	35808	1230
SATZ	0	0.009	*	50	*	102

Table 7.132: uf150 : 100 instances, 150v, 645c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0056	6057	*	*	*
ESGProbe	0	0.0536	2474	7242	36245	850
ESGPrb+AR	0	0.0539	4557	6301	33134	790
Chaffish	0	0.2192	*	4905	118028	3744
SATZ	0	0.0207	*	111	*	102

Table 7.133: uf175 : 100 instances, 175v, 753c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.01	12490	*	*	*
ESGProbe	0.06	0.19	5307	15150	78944	1675
ESGPrb+AR	0.14	0.28	11132	14883	81587	1757
Chaffish	1.00	1.53	*	13931	367751	10688
SATZ	0	0.05	*	241	*	101

Table 7.134: uf200 : 100 instances, 200v, 860c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 20s Reps: 100]

Solver	Fail%	Av Time(s)	Av Steps	Av Chc	Av UPs	Av Cl+
ESGint	0	0.02	19110	*	*	*
ESGProbe	0.6	0.59	11831	37345	191141	3314
ESGPrb+AR	2.0	1.07	30293	43882	237023	4180
Chaffish	28.0	7.81	*	38929	1175259	29566
SATZ	0	0.26	*	1343	*	94

Table 7.135: uf250 : 100 instances, 250v, 1065c, satisfiable, r=4.26, hard, random 3-SAT (SATLIB) [Max Time: 20s Reps: 100]

7.6 Conclusions

While no breakthrough has been achieved in these hybrid attempts, we nonetheless believe that the results are encouraging. Looking forward, adaptive solvers may be required to shift the emphasis from one solution approach to another at runtime, but we hope that a more principled unification of the full assignment and partial assignment approaches may yield a technique that naturally combines the advantages of the two schools.

Chapter 8

Conclusion

We feel this body of research has, at least partially, illuminated the comparatively poorly understood world of local search for satisfiability. By introducing some metrics, systematically evaluating algorithms, and then applying that knowledge to developing new solvers, we feel that some of the *ad hoc* nature of local search research has been mitigated. By extending these algorithms beyond satisfiability to general boolean linear programs, we underline the fact that local search has a potentially wider field than satisfiability and that more work is required to see how far it can reach. By exploring hybrids with backtracking search methods, we have demonstrated that some cooperative action between these methodologies produces results that promise well for the future of such methods.

The local search field is still very murky. For this reason, some elements of the AI research community are leery of its pursuit, and with some justification, because much of the research has no theoretical underpinning or only a weak, intuitive basis. However, its performance characteristics in some domains remain undeniable. At the risk of speculation, we expect that future theoretical results and empirical exploration will eventually uncover fundamental properties of local search techniques that will push them beyond haphazard heuristics to a set of principled methods applicable to a wide variety of problems.

In the shorter term, we believe the pursuit of relationships between backtracking and local search techniques will soon produce solvers robust to a variety of instances and a deeper understanding of both approaches. The application of our techniques to a wider range of problems also remains as an important extension to the current work. However, for the purposes of this dissertation, we hope we can conclude with the following words:[1]

satis est

[1]Latin: *it is enough*

Appendix A

Experimental Environments

Since this research occurred over several years, it was performed on a variety of hardware. We describe each of the general environments and indicate for all experiments which environment was used (unless timings were irrelevant).

Multi-processor machines were used, but while multiple experiments were simultaneously executed on a single machine, individual experiments were **not** distributed over multiple processors or machines. In general, the machines in question had no significant load other than the running experiments and our "wall-clock" timings are therefore believed to be a good indicator of the real CPU usage.

A.1 Environment 1

- 3 machines, dual-processor Pentium III, 450 MHz, 512 MB RAM
- RedHat Linux operating system (6.x)

A.2 Environment 2

- Cluster of 13 machines, quad-processor Pentium III, 750 MHz, each with RAM $\geq$1 GB.
- RedHat Linux operating system (6.x) or Debian (3.x)

A.3 Environment 3

- 16 processor, IBM P-Series 690, 32 GB RAM
- AIX operating system (5.2.0.0)

Appendix B

Software Suites

Experimental results presented in this paper were achieved either by (i) implementations of algorithms written by us (we call these *native solvers*), or by (ii) implementations written by the originators of an algorithm and then integrated into our testing framework (we call these *bridged solvers).*

During this research, two experimental frameworks were built, the first in C, and its successor, reimplemented almost from scratch in C++.

B.1 ESG-SAT

The results published in [99, 100, 101] were obtained using the first framework, written in C, which we call *ESG-SAT*. Some of these results have been used in this thesis. Solver implemented in this framework include GSAT, HSAT, WalkSAT, WalkSAT-G, WalkSAT-B, Novelty+, simulated annealing, SDF, ESGflt, and a number of experimental hybrids including DualRes. Bridges were built to DLM, zChaff, and SATZ.

B.2 HybSAT

All other results in this thesis were based on the new C++ framework, *HybSAT*. This framework was carefully engineered with efficiency and cache locality in mind, while still allowing for local search and backtrack search algorithms to be implemented on the same clause database and hybridized without loss of performance. Solvers implemented in this framework include:

- GSAT
- HSAT
- WalkSAT, WalkSAT-G, WalkSAT-B
- Novelty+

- survey propagation decimation (spe) (using the clause witness mechanism and with ESGint for the local search part)
- DPLL (with clause witnesses)
- BJ (DPLL with backjumping and clause witnesses)
- Chaffish (a reimplementation of zChaff, except that memory from deleted clauses is not reclaimed).
- ESGint
- Stable-Res

Bridged solvers include:

- DLM
- zChaff
- SATZ (version 2.14)
- SAPS
- survey propagation decimation (sp)

Timings are carefully implemented, using CPU timings when running times are sufficiently long that clock resolution is not an issue, and real-time clocks when very short runs are performed (under Linux, the real-time clock has higher-resolution). Time limits are implemented by means of a thread mechanism, so system calls checking time are kept to a minimum.

Every effort has been made to ensure that reimplementations are fairly constructed and that bridged solvers are efficiently integrated. All native solvers share the same instance loading mechanism which sanitizes the instance to remove vacuous clauses, duplicate literals, and propagates any unit clauses in the original instance. Load times are not included in the timings reported for any solver, bridged or native. The reimplementations typically run as fast, or faster than the original implementations, even with initial UP step which most solvers do not perform. One notable exception is *spe*, our reimplementation of survey propagation, which is a little less than twice as slow as the original, largely due to some redundancy in the problem representation.

Whereas early results with the ESG-SAT solvers were typically tuned quite carefully, our more recent experiments favour using default parameters for all solvers as the fairest mechanism for comparison.

Appendix C

Default Parameters

The default parameters used for solvers from HybSAT (see Appendix B.2):

- WalkSAT, WalkSAT-G, WalkSAT-B: random walk probability, $\omega = 0.5$
- Novelty+: second best probability, $p_{secondbest} = 0.4$; random walk probability, $\omega = 0.01$
- ESGint: upweight factor, $\alpha = 1.3$; smoothing rate, $\rho = 0.8$; smoothing probability, $p_{smooth} = 0.05$
- DLM: This bridged solver is distributed with 5 parameter sets. We use the fourth such set (par4), which provides the best overall performance.

The same parameters were shared for solvers in the ESG-SAT suite (see Appendix B.1). We must also specify:

- SDF: weight factor, $\delta = 0.05/n$; smoothing rate, $\rho = 0.995$

Appendix D

Details in Implementing Unit Propagation

This short discussion of the details involved in implementing unit propagation are intended to expose some questions rarely addressed in the literature. Furthermore, it serves to clear up any ambiguity in examples, which sometimes depend on exactly how unit clauses are detected and handled.

1. One consideration in determining the effects of reduction is the order in which clauses are reduced. Only clauses containing the *reducing variable* need to be examined. The order is typically arbitrary, depending on how the clauses are stored in memory, and in some solvers the order may change as the search progresses (e.g. SATO - see Section 2.4.6.1). Unless otherwise specified, our examples will assume that clauses are considered left to right, top to bottom. It is conceivable that a choice of ordering could improve search (e.g. there may be some clauses that are more likely to produce contradictions and so should be checked first). To the best of our knowledge, such orderings have not been explored.

2. Another choice is whether unit clauses should be processed breadth-first or depth-first. If one is reducing a set of clauses due to the literal x_i, and a reduction implies a new unit clause, x_j, should we:

   (a) continue reducing clauses containing x_i and queue x_j on a *pending move list* (breadth-first), or

   (b) immediately start reductions with x_j, only returning to reducing with x_i if no contradictions are found (depth-first)?

3. Assuming one is processing unit clauses in a breadth-first fashion, when should one check for contradictions?

(a) Check before adding a unit clause to the pending move list. If the opposing literal is already assigned or on the pending list, signal a contradiction (early contradiction checking).

(b) Check before removing a unit clause from the pending list and committing its assignment. If the opposing literal is already assigned, signal a contradiction (late contradiction checking).

The former choice detects contradictions as early as possible but requires that we be able to quickly determine whether or not a variable is on the pending list. The latter choice allows us a cheaper pending list, but we may process many forced moves before discovering a contradiction that could have been detected earlier.

4. Again assuming breadth-first, should the pending move list be treated as a stack or a FIFO queue? Limited, informal experimentation on our part suggests that it does not make a difference.

5. Should the variables on the pending move list be treated as already assigned or only considered assigned once we remove them from the pending list for reduction (late assignment commit vs. early assignment commit)?

Our earlier example, Figure 2.4, shows one possible search tree resulting from unit propagation on formula (2.20). This algorithm corresponds to breadth-first processing, early contradiction checking, and a FIFO pending list with late assignment commit. Unless otherwise specified, all UP processing in our examples is assumed to be performed this way.

Appendix E

Lots of Numbers

This appendix is provided simply for anyone interested in a broad view of solver behaviour. It combines results from other sections and adds some new ones. All results were obtained by running solvers with their default parameters (see Appendix C). Time limits and repetitions are shown in the table captions. All experiments were run in the environment from Appendix A.3 and with the software suite described in B.2. Missing solvers for any particular set of instances are due to some technical problem (i.e. crashing, running out of memory), deliberate omission on the grounds that it was almost certain to fail, or limitations on the shared computing resources available at the time of writing. Values that do not apply to a particular solver are marked with an asterisk (e.g. ESGint does not do unit propagation).

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.01240	0.00832	15689	10526	*	*	*
ESGProbe	0	0.01798	0.01786	881	876	3095	15317	517
ESGPrb+AR	0	0.01246	0.01194	1474	1414	2143	11043	376
SAPS	0	0.02435	0.01549	23059	14668	*	*	*
DLM	0	0.01203	0.00822	13164	8994	*	*	*
Novelty	1.5	0.17285	0.24832	137931	74753	*	*	*
WalkSAT	0.2	0.08298	0.04550	71565	39241	*	*	*
HSAT	80.6	1.61710	40.32730	2240338	173464	*	*	*
GSAT	70.0	1.52141	11.69950	1275482	560535	*	*	*
SurvProp	0	0.01835	0.01210	15736	10379	0	0	0
Chaffish	0	0.01251	*	*	*	856	15262	646
SATZ	0	0.00480	*	*	*	47	*	63

Table E.1: BMS : 500 instances, 100v, satisfiable, Backbone-Minimal Subinstances from RTI (SATLIB - Singer) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00090	0.0007	976	813	*	*	*
ESGProbe	0	0.00314	0.0031	269	266	631	2903	105
ESGPrb+AR	0	0.00236	0.0022	350	329	454	2092	76
SAPS	0	0.00176	0.0015	1254	1055	*	*	*
DLM	0	0.00131	0.0010	1019	796	*	*	*
Novelty	0.01	0.01346	0.0046	9507	3229	*	*	*
WalkSAT	0	0.00427	0.0034	3600	2854	*	*	*
HSAT	74.16	0.74587	0.3531	960450	7365	*	*	*
GSAT	72.71	0.73925	0.3788	635464	17511	*	*	*
SurvProp	32.02	0.18211	0.0094	43526	2253	0	9	0
Chaffish	0	0.00240	*	*	*	209	3138	142
SATZ	0	0.00228	*	*	*	15	*	125

Table E.2: RTI : 500 instances, 100v, 426c, satisfiable, Random 3-SAT (SATLIB - Singer) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0417	0.0390	49314	46140	*	*	*
ESGProbe	0	0.0028	0.0028	570	570	296	3674	23
ESGPrb+AR	0	0.0025	0.0025	580	580	251	3190	19
SAPS	0.05	0.1264	0.1159	86515	79299	*	*	*
DLM	99.19	1.0085	3.3225	615343	612310	*	*	*
Novelty	99.16	1.0105	2.0996	828000	828000	*	*	*
WalkSAT	100.00	1.0114	2.2337	784856	784856	*	*	*
HSAT	3.14	0.0358	0.0043	16073	1937	*	*	*
GSAT	0	0.0597	0.0596	21934	21911	*	*	*
SurvProp	97.34	0.9902	0	96	0	0	16	0
Chaffish	0	0.0016	*	*	*	64	6321	33
SATZ	0	0.0082	*	*	*	4	*	0

Table E.3: SW100-8-8 : 100 instances, 500v, 3100c, satisfiable, Morphed 5-Colourable Graphs, ratio 2^8 (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.01955	0.00978	29237	14630	*	*	*
ESGProbe	0	0.00070	0.00070	176	176	199	610	28
ESGPrb+AR	0	0.00069	0.00068	203	203	196	595	26
SAPS	13.00	0.68186	0.01982	634985	18459	*	*	*
DLM	0	0.05288	0.03083	85006	49567	*	*	*
Novelty	49.88	2.50980	2.50631	2474046	2470606	*	*	*
WalkSAT	49.81	2.50425	2.19200	2460364	2153586	*	*	*
HSAT	87.56	4.38311	46.42640	5586723	2309235	*	*	*
GSAT	88.56	4.44081	8.75237	3722399	2045878	*	*	*
SurvProp	24.44	0.14158	0.05085	35475	12741	0	22	0
Chaffish	0	0.00052	*	*	*	124	731	46
SATZ	0	0.00020	*	*	*	1	*	256

Table E.4: aim-sat-100 : 16 instances, 100v, satisfiable, AIM Random Generator (DIMACS - Asahiro/Iwama/Miyano) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	20	64.4935	327.5250	19544851	99256804	*	*	*
ESGProbe	0	0.0037	0.0037	676	670	1032	3047	83
ESGPrb+AR	0	0.0032	0.0031	784	758	965	2757	72
SAPS	62	185.9510	1255.3700	183777445	153358667	*	*	*
DLM	25	75.5878	731.6450	12269543	118761815	*	*	*
Novelty	50	150.7380	1496.5400	149179936	148546061	*	*	*
WalkSAT	50	149.5580	19316.700	135264920	135253759	*	*	*
HSAT	89	263.8310	8702.9900	219536866	157958488	*	*	*
GSAT	95	284.9820	10911.000	164432760	129083125	*	*	*
SurvProp	62	67.4437	1182.3200	84673983	72634775	0	14	0
Chaffish	0	0.0024	*	*	*	459	3747	141
SATZ	0	0.0002	*	*	*	1	*	325

Table E.5: aim-sat-200 : 16 instances, 200v, satisfiable, AIM Random Generator (DIMACS - Asahiro/Iwama/Miyano) [Max Time: 300s Reps: 10]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00066	0.00066	478	475	*	*	*
ESGProbe	0	0.00034	0.00033	46	46	100	318	9
ESGPrb+AR	0	0.00029	0.00029	55	55	85	253	8
SAPS	0	0.00139	0.00067	811	391	*	*	*
DLM	0	0.00081	0.00080	410	405	*	*	*
Novelty	0	0.00342	0.00306	2167	1941	*	*	*
WalkSAT	0	0.00159	0.00156	1091	1066	*	*	*
HSAT	97	0.97755	0.00071	1141296	833	*	*	*
GSAT	97	0.97800	0.00227	903567	2100	*	*	*
SurvProp	70	0.26658	0	159924	0	0	29	0
Chaffish	0	0.00046	*	*	*	36	281	16
SATZ	0	0.00094	*	*	*	6	*	0

Table E.6: ais06 : 61v, 518c, satisfiable, All-Interval Series (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0073	0.004582	4959	3128	*	*	*
ESGProbe	0	0.0046	0.004582	242	242	1803	6700	134
ESGPrb+AR	0	0.0042	0.004109	448	444	1450	5931	118
SAPS	0	0.0154	0.013124	7388	6301	*	*	*
DLM	0	0.0120	0.011917	5375	5325	*	*	*
Novelty	0	0.0697	0.042196	40791	24693	*	*	*
WalkSAT	0	0.0631	0.061807	38917	38116	*	*	*
HSAT	100	1.0063	1.0062600	841528	841528	*	*	*
GSAT	100	1.0110	1.0109700	694731	694731	*	*	*
SurvProp	93	0.7279	0	60921	0	0	25	0
Chaffish	0	0.0080	*	*	*	588	8828	412
SATZ	0	0.0024	*	*	*	10	*	0

Table E.7: ais08 : 113v, 1520c, satisfiable, All-Interval Series (SATLIB) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.048	0.038	27165	21545	*	*	*
ESGProbe	0	0.035	0.035	975	973	7392	48622	540
ESGPrb+AR	0	0.024	0.021	1752	1598	5193	36130	409
SAPS	0	0.078	0.023	29462	8828	*	*	*
DLM	0	0.054	0.042	18419	14305	*	*	*
Novelty	0	0.709	0.421	363661	215973	*	*	*
WalkSAT	0	0.912	0.411	480734	216742	*	*	*
HSAT	100	10.006	10.006	6510618	6510618	*	*	*
GSAT	100	10.015	10.015	5327770	5327770	*	*	*
SurvProp	99	2.355	0	97434	0	0	9	0
Chaffish	0	0.038	*	*	*	1560	41544	1331
SATZ	0	0.009	*	*	*	25	*	0

Table E.8: ais10 : 181v, 3151c, satisfiable, All-Interval Series (SATLIB) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.366	0.335	186704	170874	*	*	*
ESGProbe	0	0.475	0.465	6720	6584	31646	531235	3342
ESGPrb+AR	0	0.776	0.339	22785	9945	29662	602946	3699
SAPS	0	0.940	0.771	285930	234366	*	*	*
DLM	0	0.658	0.524	167855	133523	*	*	*
Novelty	0	16.533	2.884	7435290	1297200	*	*	*
WalkSAT	0	22.804	1.517	11106073	738900	*	*	*
HSAT	100	200.003	1508.250	13132164	99031510	*	*	*
GSAT	100	200.006	30624.600	564686	86464032	*	*	*
SurvProp	100	8.530	0	2012485	0	0	8	0
Chaffish	0	3.814	*	*	*	29984	1096828	27518
SATZ	0	0.035	*	*	*	58	*	0

Table E.9: ais12 : 265v, 5666c, satisfiable, All-Interval Series (SATLIB) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.000139	0.000138	94	94	*	*	*
ESGProbe	0	0.000129	0.000128	39	39	3	75	1
ESGPrb+AR	0	0.000121	0.000120	37	37	2	69	1
SAPS	0	0.000247	0.000245	145	145	*	*	*
DLM	0	0.000227	0.000226	111	111	*	*	*
Novelty	0	0.001050	0.000478	734	334	*	*	*
WalkSAT	0	0.000560	0.000253	465	210	*	*	*
HSAT	82	0.823520	0.000107	1658966	215	*	*	*
GSAT	83	0.837705	0.000220	1022581	268	*	*	*
SurvProp	47	0.002660	0	0	0	0	38	0
Chaffish	0	0.000288	*	*	*	3	76	1
SATZ	0	0.000426	*	*	*	1	*	0

Table E.10: anomaly : 48v, 261c, satisfiable, Blocks World 3 Blocks, 3 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.16	0.1	52728	37767	*	*	*
ESGProbe	0	1.46	1.5	11088	11088	32180	665218	2716
ESGPrb+AR	0	0.68	0.7	13224	13206	21358	483729	1770
SAPS	0	0.61	0.5	74284	66673	*	*	*
DLM	0	1.83	0.9	261790	132072	*	*	*
Novelty	0	2.53	2.1	604946	491645	*	*	*
WalkSAT	16	53.32	3302.0	14544194	14097399	*	*	*
HSAT	100	196.44	1108.0	26657564	69607237	*	*	*
GSAT	100	200.00	2260.5	19875216	62824889	*	*	*
Chaffish	0	0.31	*	*	*	4455	179626	3008
SATZ	0	0.02	*	*	*	18	*	0

Table E.11: beijing-bw : 3 instances, satisfiable, Beijing Blocks World - Sussman anomalies (Crawford) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.162	0.158	52443	51313	*	*	*
ESGProbe	0.2	1.015	1.015	16743	16743	17811	965449	233
ESGPrb+AR	0	0.979	0.979	20177	20177	17574	1022008	266
SAPS	0.2	0.916	0.881	86079	82763	*	*	*
DLM	0.2	1.758	1.741	56633	56080	*	*	*
Novelty	1.2	1.398	1.381	497549	491382	*	*	*
WalkSAT	0.5	0.593	0.556	277779	260722	*	*	*
HSAT	100.0	10.006	10.006	112279	112279	*	*	*
GSAT	100.0	10.014	10.014	92005	92005	*	*	*
SurvProp	100.0	10.053	NaNQ	0	0	0	237	0
Chaffish	0	0.591	*	*	*	11746	869129	1356
SATZ	83.3	11.802	*	*	*	8	*	0

Table E.12: beijing-jobshop : 6 instances, satisfiable, Beijing Job Shop (Crawford) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	38.5	4.0727	0.0035	4765358	4064	*	*	*
ESGProbe	0	0.0031	0.0031	255	255	989	4515	33
ESGPrb+AR	0	0.0028	0.0027	270	269	797	3801	28
SAPS	30.3	3.5944	0.0068	2720980	5119	*	*	*
Novelty	0	0.0016	0.0016	977	970	*	*	*
WalkSAT	0	0.0009	0.0009	597	597	*	*	*
HSAT	0.8	0.0791	0.0047	18466	1097	*	*	*
GSAT	0	0.0862	0.0860	18038	17998	*	*	*
SurvProp	31.8	3.6880	0.0064	3228700	5625	0	0	0
Chaffish	0	0.5176	*	*	*	4769	162040	3775
SATZ	0	0.0093	*	*	*	50	*	0

Table E.13: beijing-vlsi-2bit-sat : 4 instances, satisfiable, Beijing 2-bit VLSI design (Crawford) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	999.9930	2039.450	421343986	421343986	*	*	*
ESGProbe	0	5.4986	5.499	62461	62461	283388	7180564	3001
ESGPrb+AR	0	2.0983	2.098	28664	28664	121866	2649360	1114
SAPS	100	999.9880	999.988	95906593	95906593	*	*	*
Novelty	0	0.1727	0.163	57579	54338	*	*	*
WalkSAT	0	0.0525	0.053	19601	19601	*	*	*
HSAT	25	254.7830	8.966	8506130	299342	*	*	*
GSAT	100	999.9920	999.992	28342637	28342637	*	*	*
SurvProp	100	999.9910	2280.870	382404213	382404213	0	0	0
SATZ	50	616.4750	*	*	*	1066462	*	0

Table E.14: beijing-vlsi-3bit-sat : 2 instances, satisfiable, Beijing 3-bit VLSI design (Crawford) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.960	3599.960	1054235450	1054235450	*	*	*
ESGProbe	0	31.288	31.288	654925	654925	137821	21478486	2000
ESGPrb+AR	0	30.103	30.103	450982	450982	87497	28612648	1954
SAPS	100	3598.380	3598.380	38348080	38348080	*	*	*
DLM	100	3599.970	3599.970	31330532	31330532	*	*	*
Novelty	100	3599.960	3599.960	1574194279	1574194279	*	*	*
WalkSAT	100	3599.950	3599.950	1339661473	1339661473	*	*	*
HSAT	100	3599.960	3599.960	17481099	17481099	*	*	*
SurvProp	100	248.022	0	0	0	0	0	0
Chaffish	0	43.712	*	*	*	207492	63514166	41224
SATZ	100	3598.630	*	*	*	46586	*	0

Table E.15: bmc-galileo-8 : 58074v, 294821c, satisfiable, BMC on Galileo FIFO #1, 35 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.960	3599.960	1244855540	1244855540	*	*	*
ESGProbe	0	64.180	64.180	1032783	1032783	159224	45302516	3838
ESGPrb+AR	0	50.726	50.726	774138	774138	128433	42832022	3001
SAPS	100	3599.700	3599.700	44897934	44897934	*	*	*
DLM	100	3599.970	3599.970	32168904	32168904	*	*	*
Novelty	100	3599.950	3599.950	1310220805	1310220805	*	*	*
WalkSAT	100	3599.950	3599.950	1289924670	1289924670	*	*	*
HSAT	100	3599.960	3599.960	16009941	16009941	*	*	*
SurvProp	100	276.836	0	0	0	0	0	0
Chaffish	0	52.568	*	*	*	248159	69653529	46249
SATZ	100	3580.890	*	*	*	39274	*	0

Table E.16: bmc-galileo-9 : 63624v, 326999c, satisfiable, BMC on Galileo FIFO #2, 38 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	2887.6	2888	2337456763	2337456763	*	*	*
ESGProbe	0	5.8	6	98228	98228	30019	4945966	2787
ESGPrb+AR	0	4.6	5	100082	100082	26083	3732897	2256
SAPS	100	3600.0	3600	1240111832	1240111832	*	*	*
DLM	100	3600.0	3600	867083241	867083241	*	*	*
Novelty	100	3600.0	3600	2143127606	2143127606	*	*	*
WalkSAT	100	3600.0	3600	2093634131	2093634131	*	*	*
HSAT	100	3600.0	3600	80777162	80777162	*	*	*
SurvProp	100	37.7	0	0	0	0	0	0
Chaffish	0	0.8	*	*	*	44091	1246677	3205
SATZ	100	3600.0	*	*	*	1740087	*	0

Table E.17: bmc-ibm-1 : 9685v, 55870c, satisfiable, BMC on IBM CPU Part 1, 18 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.96	3599.96	1593363747	1593363747	*	*	*
ESGProbe	0	128.28	128.28	1639151	1639151	451221	68249395	9319
ESGPrb+AR	0	76.94	76.94	671565	671565	335586	47851484	5511
SAPS	100	3594.12	3594.12	43417808	43417808	*	*	*
DLM	100	3599.98	3599.98	36924159	36924159	*	*	*
Novelty	100	3599.96	3599.96	1541544262	1541544262	*	*	*
WalkSAT	100	3599.96	3599.96	1483537425	1483537425	*	*	*
HSAT	100	3599.96	3599.96	14577495	14577495	*	*	*
SurvProp	100	227.33	0	0	0	0	0	0
Chaffish	0	138.68	*	*	*	1010901	91237968	86579
SATZ	0	1001.06	*	*	*	5514	*	0

Table E.18: bmc-ibm-10 : 61088v, 334861c, satisfiable, BMC on IBM PowerPC BIU #2, 13 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3600.0	3600.0	1543519827	1543519827	*	*	*
ESGProbe	0	52.9	52.9	1333529	1333529	98626	32791812	6221
ESGPrb+AR	0	22.9	22.9	455110	455110	57266	15359391	3294
SAPS	100	3599.1	3599.1	75227171	75227171	*	*	*
DLM	100	3600.0	3600.0	61572464	61572464	*	*	*
Novelty	100	3600.0	3600.0	1816579032	1816579032	*	*	*
WalkSAT	100	3600.0	3600.0	1813166489	1813166489	*	*	*
HSAT	100	3600.0	3600.0	26068397	26068397	*	*	*
SurvProp	100	123.1	0	0	0	0	0	0
Chaffish	0	16.0	*	*	*	146594	22227682	27635
SATZ	0	1508.3	*	*	*	32369	*	0

Table E.19: bmc-ibm-11 : 32109v, 150027c, satisfiable, BMC on IBM Cache Control #1, 32 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.960	3599.960	1204058869	1204058869	*	*	*
ESGProbe	0	433.776	433.776	7031836	7031836	250386	288352800	27007
ESGPrb+AR	0	96.120	96.120	1257214	1257214	115481	84415004	8387
SAPS	100	3599.960	3599.960	74112794	74112794	*	*	*
DLM	100	3599.970	3599.970	43939683	43939683	*	*	*
Novelty	100	3599.950	3599.950	1415210948	1415210948	*	*	*
WalkSAT	100	3599.950	3599.950	1244833207	1244833207	*	*	*
HSAT	100	3599.960	3599.960	24952535	24952535	*	*	*
SurvProp	100	193.453	0	0	0	0	0	0
Chaffish	0	61.510	*	*	*	257673	77809342	61634
SATZ	100	3599.980	*	*	*	22672	*	0

Table E.20: bmc-ibm-12 : 39598v, 19477c, satisfiable, BMC on IBM PowerPC BIU #3, 31 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.960	3599.960	3081768985	3081768985	*	*	*
ESGProbe	0	24.935	24.935	498693	498693	103487	23417782	7225
ESGPrb+AR	0	70.350	70.350	1630833	1630833	182224	57055661	19288
SAPS	100	3597.840	3597.840	138206487	138206487	*	*	*
DLM	100	3599.970	3599.970	197132395	197132395	*	*	*
Novelty	100	3599.960	3599.960	2551801962	2551801962	*	*	*
WalkSAT	100	3599.950	3599.950	2074958745	2074958745	*	*	*
HSAT	100	3599.960	3599.960	69581383	69581383	*	*	*
SurvProp	100	50.837	0	0	0	0	0	0
Chaffish	0	5.014	*	*	*	39264	8211406	13634
SATZ	100	3564.380	*	*	*	151627	*	0

Table E.21: bmc-ibm-13 : 13215v, 6572c, satisfiable, BMC on IBM Cache Control #2, 14 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.048	0.05	58360	58360	*	*	*
ESGProbe	0	0.027	0.03	1981	1981	665	19082	64
ESGPrb+AR	0	0.015	0.02	1420	1420	428	10815	28
SAPS	100	3599.960	3599.96	422858995	422858995	*	*	*
DLM	0	0.371	0.37	116643	116643	*	*	*
Novelty	0	5.976	5.98	5302275	5302275	*	*	*
WalkSAT	0	222.870	222.87	199931000	199931000	*	*	*
HSAT	100	3599.960	3599.96	234317443	234317443	*	*	*
SurvProp	100	2.044	0	0	0	0	0	0
Chaffish	0	0.003	*	*	*	289	7275	27
SATZ	0	0.031	*	*	*	17	*	0

Table E.22: bmc-ibm-2 : 3628v, 14468c, satisfiable, BMC on IBM CPU Part 2, 5 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3600.0	3600.0	2090007126	2090007126	*	*	*
ESGProbe	0	6.3	6.3	203820	203820	13416	4661693	2118
ESGPrb+AR	0	1.6	1.6	55179	55179	6805	1818814	516
SAPS	100	3599.1	3599.1	73447487	73447487	*	*	*
DLM	100	3600.0	3600.0	133348695	133348695	*	*	*
Novelty	100	3600.0	3600.0	2498746510	2498746510	*	*	*
WalkSAT	100	3600.0	3600.0	2040920247	2040920247	*	*	*
HSAT	100	3600.0	3600.0	67087067	67087067	*	*	*
SurvProp	100	54.7	0	0	0	0	0	0
Chaffish	0	0.5	*	*	*	2892	1125731	1596
SATZ	0	25.9	*	*	*	413	*	0

Table E.23: bmc-ibm-3 : 14930v, 72106c, satisfiable, BMC on IBM BIU 1996, 14 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.960	3599.960	2454418981	2454418981	*	*	*
ESGProbe	0	7.899	7.899	145055	145055	21769	4363206	1517
ESGPrb+AR	0	6.555	6.555	113695	113695	21633	3783923	1285
SAPS	100	3593.830	3593.830	54766977	54766977	*	*	*
DLM	100	3599.970	3599.970	132928872	132928872	*	*	*
Novelty	100	3599.950	3599.950	2252741233	2252741233	*	*	*
WalkSAT	100	3599.950	3599.950	2211673682	2211673682	*	*	*
HSAT	100	3599.960	3599.960	27496215	27496215	*	*	*
SurvProp	100	58.299	0	0	0	0	0	0
Chaffish	0	1.083	*	*	*	15272	1900785	2729
SATZ	0	22.519	*	*	*	940	*	0

Table E.24: bmc-ibm-4 : 28161v, 139716c, satisfiable, BMC on IBM PowerPC BIU #1, 24 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	3.28	3.28	3387884	3387884	*	*	*
ESGProbe	0	0.41	0.41	12302	12302	4909	161295	280
ESGPrb+AR	0	0.11	0.11	6253	6253	2747	60004	54
SAPS	100	3599.96	3599.96	312597535	312597535	*	*	*
DLM	100	3599.96	3599.96	1134595691	1134595691	*	*	*
Novelty	100	3599.96	3599.96	2566055443	2566055443	*	*	*
WalkSAT	100	3599.95	3599.95	2696760217	2696760217	*	*	*
HSAT	100	3600.04	3600.04	78162559	78162559	*	*	*
SurvProp	100	14.97	0	0	0	0	0	0
Chaffish	0	0.05	*	*	*	3248	124940	365
SATZ	0	0.52	*	*	*	248	*	0

Table E.25: bmc-ibm-5 : 9396v, 41207c, satisfiable, BMC on IBM Arbiter #1, 12 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.960	3599.960	1033771043	1033771043	*	*	*
ESGProbe	0	56.132	56.132	282422	282422	46490	9037676	6353
ESGPrb+AR	0	34.860	34.860	196472	196472	42236	6743319	3605
SAPS	100	3599.970	3599.970	31844055	31844055	*	*	*
DLM	100	3599.980	3599.980	80154047	80154047	*	*	*
Novelty	100	3599.960	3599.960	1219548425	1219548425	*	*	*
WalkSAT	100	3599.950	3599.950	1242334149	1242334149	*	*	*
HSAT	100	3599.960	3599.960	14195438	14195438	*	*	*
SurvProp	100	209.544	0	0	0	0	0	0
Chaffish	0	4.593	*	*	*	20893	4347790	9427
SATZ	100	3599.960	*	*	*	1479165	*	0

Table E.26: bmc-ibm-6 : 51654v, 368367c, satisfiable, BMC on IBM LSU 1997, 22 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.028	0.03	16677	16677	*	*	*
ESGProbe	0	0.128	0.13	6923	6923	3068	87112	65
ESGPrb+AR	0	0.201	0.20	7166	7166	3550	118807	140
SAPS	100	3599.070	3599.07	284204742	284204742	*	*	*
DLM	0	0.719	0.72	79560	79560	*	*	*
Novelty	0	1.471	1.47	1053136	1053136	*	*	*
WalkSAT	0	0.152	0.15	122479	122479	*	*	*
HSAT	100	3599.970	3599.97	82245583	82245583	*	*	*
SurvProp	100	11.390	0	0	0	0	0	0
Chaffish	0	0.018	*	*	*	1176	36825	68
SATZ	0	0.386	*	*	*	49	*	0

Table E.27: bmc-ibm-7 : 8710v, 39774c, satisfiable, BMC on IBM Arbiter #2, 9 cycles (IBM - Shtrichman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0038	0.0038	2998	2992	*	*	*
ESGProbe	0	0.0025	0.0025	365	365	91	2971	13
ESGPrb+AR	0	0.0024	0.0024	378	378	84	2706	12
SAPS	0	0.0078	0.0077	3429	3418	*	*	*
DLM	0	0.0111	0.0111	3712	3701	*	*	*
Novelty	0	0.0383	0.0371	23408	22659	*	*	*
WalkSAT	0	0.0271	0.0236	18800	16354	*	*	*
HSAT	99	0.9966	0.0811	507775	41300	*	*	*
GSAT	99	1.0020	0.2878	434513	124800	*	*	*
SurvProp	89	0.5146	0	0	0	0	182	0
Chaffish	0	0.0013	*	*	*	50	2430	13
SATZ	0	0.0059	*	*	*	1	*	392

Table E.28: bw_large.a : 459v, 4675c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0580	0.0547	40857	38537	*	*	*
ESGProbe	0	0.0207	0.0207	1206	1206	874	30702	83
ESGPrb+AR	0	0.0204	0.0204	1505	1505	859	30497	85
SAPS	0	0.1295	0.1246	49619	47732	*	*	*
DLM	0	0.1490	0.1317	44360	39216	*	*	*
Novelty	38	0.6217	0.6217	358919	358919	*	*	*
WalkSAT	46	0.7413	0.7413	479915	479915	*	*	*
HSAT	100	1.0054	1.0054	265594	265594	*	*	*
GSAT	100	1.0119	1.0119	228342	228342	*	*	*
SurvProp	99	1.0001	0	0	0	0	142	0
Chaffish	0	0.0138	*	*	*	309	38814	141
SATZ	0	0.0159	*	*	*	2	*	1236

Table E.29: bw_large.b : 1087v, 13772c, satisfiable, Blocks World 11 Blocks, 9 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	26.87	23.30	14740392	12779046	*	*	*
ESGProbe	0	0.93	0.93	15900	15900	11626	1156618	1265
ESGPrb+AR	0	0.97	0.97	23617	23617	10456	1200907	1305
SAPS	1	43.16	25.67	11296613	6720437	*	*	*
DLM	0	31.88	6.23	5506186	1075847	*	*	*
Novelty	27	118.19	291.85	12737120	31451900	*	*	*
WalkSAT	96	195.35	1257.64	15796829	101696175	*	*	*
HSAT	100	200.00	200.00	21689630	21689630	*	*	*
GSAT	100	199.81	199.81	18991533	18991533	*	*	*
SurvProp	100	31.55	NaNQ	0	0	0	735	0
Chaffish	0	0.29	*	*	*	4255	725986	1718
SATZ	0	0.39	*	*	*	4	*	4106

Table E.30: bw_large.c : 3016v, 50457c, satisfiable, Blocks World 15 Blocks, 14 Steps (SATLIB - Kautz and Selman) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	30	966.42	437.5	309865108	140281180	*	*	*
ESGProbe	0	9.78	9.8	102705	102705	63478	10003989	5403
ESGPrb+AR	0	18.06	17.9	205569	204224	74452	16318415	9005
SAPS	30	1534.63	1534.6	236312751	236312751	*	*	*
DLM	10	657.36	371.7	66877193	37810590	*	*	*
Novelty	80	1959.54	5416.6	243448890	672945620	*	*	*
WalkSAT	100	1999.98	4735.6	314000596	743497326	*	*	*
HSAT	100	1999.82	1999.8	109134539	109134539	*	*	*
GSAT	100	1999.95	2000.0	95658873	95658873	*	*	*
SurvProp	100	155.20	0	0	0	0	930	0
Chaffish	0	5.59	*	*	*	30264	8832616	15646
SATZ	0	33.98	*	*	*	326	*	9448

Table E.31: bw_large.d : 6325v, 131973c, satisfiable, Blocks World 19 Blocks, 18 Steps (SATLIB - Kautz and Selman) [Max Time: 2000s Reps: 10]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.230	0.14	244142	151675	*	*	*
ESGProbe	68	4.006	4.01	101952	101952	454388	1991519	15746
ESGPrb+AR	89	4.754	4.75	257107	257107	499132	2354597	18789
SAPS	0	0.517	0.50	314071	306047	*	*	*
DLM	4	1.137	0.81	595271	423576	*	*	*
Novelty	32	2.369	0.90	1696326	642242	*	*	*
WalkSAT	0	0.209	0.20	155948	146918	*	*	*
HSAT	98	4.915	4.91	1828208	1828208	*	*	*
GSAT	97	4.896	4.90	1352781	1352781	*	*	*
SurvProp	100	3.417	NaNQ	0	0	0	1	0
Chaffish	100	5.004	*	*	*	42507	2157328	28478
SATZ	100	5.009	*	*	*	16601	*	79

Table E.32: f0600 : 600v, 2550c, satisfiable, Large Random 3-SAT (DIMACS - Selman) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	35.37	27.06	34622359	26491155	*	*	*
SAPS	0	145.17	55.53	77152697	29509720	*	*	*
DLM	100	999.83	999.83	285093597	285093597	*	*	*
Novelty	50	520.15	16.09	350441992	10839289	*	*	*
WalkSAT	0	0.76	0.66	539409	467422	*	*	*
HSAT	100	999.94	999.94	242287390	242287390	*	*	*
GSAT	100	999.99	999.99	182090609	182090609	*	*	*
SurvProp	100	7.67	0	0	0	0	12	0
SATZ	100	999.89	*	*	*	2339885	*	78

Table E.33: f1000 : 1000v, 4250c, satisfiable, Large Random 3-SAT (DIMACS - Selman) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0059	0.0052	8363	7320	*	*	*
ESGProbe	0	0.0025	0.0025	416	416	250	4734	38
ESGPrb+AR	0	0.0019	0.0019	536	535	180	3376	26
SAPS	0	0.0111	0.0100	9723	8791	*	*	*
DLM	0	0.0098	0.0085	9570	8313	*	*	*
Novelty	5.70	0.3007	0.0430	268311	38368	*	*	*
WalkSAT	0.02	0.0525	0.0414	44233	34880	*	*	*
HSAT	99.85	2.0023	39.3696	1349141	1192518	*	*	*
GSAT	99.83	2.0090	9.1941	1056572	904208	*	*	*
SurvProp	95.30	0.0978	0	193	0	0	140	0
Chaffish	0	0.0005	*	*	*	53	1758	19
SATZ	0	0.0015	*	*	*	22	*	0

Table E.34: flat100 : 100 instances, 300v, 1117c, satisfiable, Flat 3-Colourable Graphs, 100 vert, 239 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0133	0.0108	19550	15931	*	*	*
ESGProbe	0	0.0071	0.0070	867	864	596	12950	95
ESGPrb+AR	0	0.0050	0.0050	1189	1175	417	9005	64
SAPS	0	0.0260	0.0223	22375	19218	*	*	*
DLM	0	0.0272	0.0221	25282	20542	*	*	*
Novelty	12.94	0.5203	1.1176	464793	75814	*	*	*
WalkSAT	0.12	0.0939	0.0748	81670	65007	*	*	*
HSAT	99.92	2.0036	7.0772	1162781	1073024	*	*	*
GSAT	99.96	2.0124	26.2947	927520	895414	*	*	*
SurvProp	98.20	0.1279	0	192	0	0	158	0
Chaffish	0	0.0019	*	*	*	130	6397	67
SATZ	0	0.0028	*	*	*	34	*	0

Table E.35: flat125 : 100 instances, 375v, 1403c, satisfiable, Flat 3-Colourable Graphs, 125 vert, 301 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.037	0.027	54673	39267	*	*	*
ESGProbe	0	0.018	0.018	1855	1832	1296	32206	209
ESGPrb+AR	0	0.015	0.014	2835	2756	984	24399	159
SAPS	0.010	0.074	0.056	62797	47810	*	*	*
DLM	0	0.082	0.061	71992	53952	*	*	*
Novelty	30.257	0.907	0.487	804080	203116	*	*	*
WalkSAT	0.980	0.218	0.171	188033	147366	*	*	*
HSAT	100.000	2.006	13.720	996122	996122	*	*	*
GSAT	100.000	2.013	4.183	819500	819500	*	*	*
SurvProp	99.238	0.165	0	0	0	0	183	0
Chaffish	0	0.009	*	*	*	362	24316	232
SATZ	0	0.006	*	*	*	51	*	0

Table E.36: flat150 : 100 instances, 450v, 1680c, satisfiable, Flat 3-Colourable Graphs, 150 vert, 360 edges, (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.1045	0.0685	154588	101313	*	*	*
ESGProbe	0	0.0421	0.0415	3528	3482	2508	68146	389
ESGPrb+AR	0	0.0319	0.0312	5573	5444	1925	52144	301
SAPS	0.01	0.2111	0.1445	177436	121477	*	*	*
DLM	0.02	0.2253	0.1440	186840	119427	*	*	*
Novelty	32.71	2.4222	3.1366	2073066	459812	*	*	*
WalkSAT	0.96	0.4283	0.3417	385912	307911	*	*	*
HSAT	100.00	5.0061	125.2980	2236852	2236852	*	*	*
GSAT	100.00	5.0144	69.4973	1851581	1851581	*	*	*
SurvProp	99.78	0.2075	0	0	0	0	204	0
Chaffish	0	0.0324	*	*	*	1004	73471	643
SATZ	0	0.0124	*	*	*	88	*	0

Table E.37: flat175 : 100 instances, 525v, 1951c, satisfiable, Flat 3-Colourable Graphs, 175 vert, 417 edges, (SATLIB) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0.18	0.265	0.17	394386	256640	*	*	*
ESGProbe	0	0.118	0.12	8160	8031	5820	167870	857
ESGPrb+AR	0.04	0.103	0.10	13750	12787	4895	139419	726
SAPS	0.77	0.527	23.88	435538	273727	*	*	*
DLM	0.83	0.642	3.12	495093	318561	*	*	*
Novelty	48.61	3.110	18.73	2716058	837383	*	*	*
WalkSAT	3.17	0.786	1.55	686342	508049	*	*	*
HSAT	100.00	5.006	32.38	2032160	2032160	*	*	*
GSAT	100.00	5.014	20.99	1692739	1692739	*	*	*
SurvProp	99.91	0.224	0	0	0	0	227	0
Chaffish	0	0.137	*	*	*	3099	243225	1957
SATZ	0	0.034	*	*	*	183	*	0

Table E.38: flat200 : 100 instances, 600v, 2237c, satisfiable, Flat 3-Colourable Graphs, 200 vert, 479 edges, (SATLIB) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.9600	3599.96	103070212	103070212	*	*	*
ESGProbe	0	768.4400	768.44	125264	125264	3137551	555425080	55305
DLM	100	3600.0100	3600.01	7385456	7385456	*	*	*
Novelty	100	3599.9600	3599.96	5600413	5600413	*	*	*
WalkSAT	100	3599.9500	3599.95	14781220	14781220	*	*	*
HSAT	100	3599.9600	3599.96	52739939	52739939	*	*	*
SurvProp	100	3599.9600	3599.96	22002591	22002591	0	16622	0
SATZ	0	0.0005	*	*	*	0	*	2531

Table E.39: fvp-sat-2 : 1 instance, satisfiable, Formal Verification Buggy 7 Pipeline SS Processor(Velev) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	50	775.904	775.904	167026683	167026683	*	*	*
SAPS	100	999.992	999.992	130445110	130445110	*	*	*
DLM	100	999.828	999.828	106473438	106473438	*	*	*
Novelty	0	42.469	6.733	11395791	1806660	*	*	*
WalkSAT	100	999.993	999.993	258987844	258987844	*	*	*
HSAT	100	999.993	999.993	120651719	120651719	*	*	*
SurvProp	60	781.355	781.355	188162293	188162293	0	0	0
SATZ	100	999.989	*	*	*	886583	*	0

Table E.40: g125.17 : 2125v, 66272c, satisfiable, Graph Colouring, 17 colours (DIMACS - Johnson) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.104	0.104	22937	22937	*	*	*
ESGProbe	56	6.926	6.926	45044	45044	424529	9583647	6776
ESGPrb+AR	90	9.478	9.478	126306	126306	562711	11886539	9268
SAPS	0	0.248	0.242	35380	34431	*	*	*
DLM	2	2.516	1.432	248264	141270	*	*	*
Novelty	0	0.113	0.111	28808	28438	*	*	*
WalkSAT	98	9.914	9.914	2866657	2866657	*	*	*
HSAT	80	8.015	0.265	827989	27416	*	*	*
GSAT	43	4.908	0.969	497058	98153	*	*	*
SurvProp	0	1.710	1.698	25371	25191	0	0	0
Chaffish	100	10.008	*	*	*	31623	1474744	20226
SATZ	100	10.005	*	*	*	10728	*	0

Table E.41: g125.18 : 2250v, 70163c, satisfiable, Graph Colouring, 18 colours (DIMACS - Johnson) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0666	0.067	2196	2196	*	*	*
ESGProbe	0	0.1007	0.101	2198	2198	2089	27023	11
ESGPrb+AR	0	0.1399	0.140	4025	4025	3699	48275	20
SAPS	0	0.0951	0.095	2222	2222	*	*	*
DLM	0	1.9617	1.961	2478	2478	*	*	*
Novelty	0	0.0906	0.091	4998	4998	*	*	*
WalkSAT	0	0.2965	0.297	21259	21259	*	*	*
HSAT	0	0.0646	0.065	2252	2252	*	*	*
GSAT	0	0.0822	0.080	3063	2993	*	*	*
SurvProp	100	5.1474	NaNQ	0	0	0	0	0
Chaffish	100	5.0096	*	*	*	21412	2327494	12435
SATZ	100	5.0098	*	*	*	365	*	0

Table E.42: g250.15 : 3750v, 233965c, satisfiable, Graph Colouring, 15 colours (DIMACS - Johnson) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	999.9920	999.992	38316750	38316750	*	*	*
SAPS	100	999.4450	999.445	26799511	26799511	*	*	*
DLM	100	999.7620	999.762	22879930	22879930	*	*	*
Novelty	0	304.7700	223.135	21451320	15705430	*	*	*
WalkSAT	100	999.9920	999.992	60082268	60082268	*	*	*
HSAT	100	999.9880	999.988	34755535	34755535	*	*	*
SurvProp	100	999.9920	999.992	47089750	47089750	0	0	0
SATZ	0	0.0003	*	*	*	0	*	0

Table E.43: g250.29 : 7250v, 454622c, satisfiable, Graph Colouring, 29 colours (DIMACS - Johnson) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	999.992	7909	124324252	983317711	*	*	*
ESGProbe	0	2.361	2	41112	41112	52790	1393541	8253
ESGPrb+AR	0	44.098	25	398098	223650	188437	5283535	34046
SAPS	100	999.986	4229	265981061	1124974521	*	*	*
DLM	100	999.803	4212	133686839	563183568	*	*	*
Novelty	100	999.970	2633	262969415	692466145	*	*	*
WalkSAT	100	999.993	2573	272961571	702458301	*	*	*
HSAT	100	999.945	1000	284984039	284984039	*	*	*
SurvProp	100	2.100	0	0	0	0	101	0
Chaffish	0	1.063	*	*	*	14156	783529	11245
SATZ	0	37.835	*	*	*	124974	*	0

Table E.44: hanoi4 : 718v, 4934c, satisfiable, Towers of Hanoi, 4 discs (DIMACS - Selman) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	100	3599.96	3599.96	3296137284	3296137284	*	*	*
SAPS	100	2066.26	2066.26	2147483647	2147483647	*	*	*
DLM	100	3599.95	3599.95	1482162860	1482162860	*	*	*
Novelty	100	3599.95	3599.95	2337616114	2337616114	*	*	*
WalkSAT	100	3599.95	3599.95	2290007697	2290007697	*	*	*
HSAT	100	3599.95	3599.95	374420859	374420859	*	*	*
SurvProp	100	8.81	0	0	0	0	0	0
SATZ	100	3579.89	*	*	*	5240695	*	0

Table E.45: hanoi5 : 1931v, 14468c, satisfiable, Towers of Hanoi, 5 discs (DIMACS - Selman) [Max Time: 3600s Reps: 1]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.053	0.033	59019	36580	*	*	*
ESGProbe	0.6	5.426	3.408	42516	26702	130688	677371	10153
ESGPrb+AR	2.8	12.432	7.158	124511	71684	177939	973447	14825
SAPS	0	0.105	0.060	69592	39488	*	*	*
DLM	0	0.108	0.056	73039	37522	*	*	*
Novelty	1.9	7.692	197.706	2583624	170791	*	*	*
WalkSAT	0	0.266	0.157	186449	109885	*	*	*
SurvProp	63.8	6.433	47.978	1564335	2057195	0	8	0
Chaffish	52.0	118.428	*	*	*	174025	5751835	130236
SATZ	0	1.362	*	*	*	6918	*	91

Table E.46: hrs300 : 100 instances, 300v, 1278c, satisfiable, Hard Random 3-SAT (Southey) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.20	0.11	223232	116516	*	*	*
ESGProbe	7	23.97	15.09	145735	91765	460290	2295994	26982
ESGPrb+AR	12	37.69	26.80	383815	272910	589687	3096395	36581
SAPS	0	0.39	0.20	247128	128124	*	*	*
DLM	0	0.56	0.34	331902	203192	*	*	*
WalkSAT	0	0.61	0.27	419925	188214	*	*	*
SurvProp	53	7.76	144.20	1408275	2224784	0	7	0
Chaffish	84	169.49	*	*	*	230719	9097437	167921
SATZ	0	43.94	*	*	*	202600	*	86

Table E.47: hrs400 : 100 instances, 400v, 1704c, satisfiable, Hard Random 3-SAT (Southey) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	1.5	0.7	1521840	781969	*	*	*
SAPS	0	3.1	1.9	1878886	1165048	*	*	*
DLM	0	6.8	46.5	3534856	2249519	*	*	*
Novelty	5	97.8	1557.3	8415356	31940201	*	*	*
WalkSAT	0	2.4	1.5	1720098	1065062	*	*	*

Table E.48: hrs500 : 134 instances, 500v, 2130c, satisfiable, Hard Random 3-SAT (Southey) [Max Time: 1000s Reps: 10]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0042	0.0042	2629	2610	*	*	*
ESGProbe	0	0.0028	0.0027	358	358	76	2672	11
ESGPrb+AR	0	0.0026	0.0026	373	373	77	2547	11
SAPS	0	0.0099	0.0098	3634	3563	*	*	*
DLM	0	0.0145	0.0140	3657	3531	*	*	*
Novelty	0	0.0429	0.0394	22509	20683	*	*	*
WalkSAT	0	0.0317	0.0303	19443	18602	*	*	*
HSAT	100	1.0057	1.0057	466369	466369	*	*	*
GSAT	100	1.0123	1.0123	398754	398754	*	*	*
SurvProp	93	0.5157	0	0	0	0	171	0
Chaffish	0	0.0015	*	*	*	46	2579	12
SATZ	0	0.0056	*	*	*	1	*	0

Table E.49: huge : 459v, 7054c, satisfiable, Blocks World 9 Blocks, 6 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00088	0.00076	372	321	*	*	*
ESGProbe	0	0.00191	0.00183	194	187	1063	767	10
ESGPrb+AR	0	0.00197	0.00187	200	191	1072	776	10
SAPS	0	0.00115	0.00107	373	350	*	*	*
DLM	0	0.00286	0.00272	311	297	*	*	*
Novelty	0	0.00433	0.00432	1796	1795	*	*	*
WalkSAT	0	0.00107	0.00105	507	497	*	*	*
HSAT	48.4	0.48865	0.28543	118307	69107	*	*	*
GSAT	37.4	0.41207	0.28113	74585	50885	*	*	*
SurvProp	0.3	0.02647	0.01777	382	257	0	0	0
Chaffish	0	0.00105	*	*	*	456	2135	27
SATZ	0	0.00589	*	*	*	10	*	0

Table E.50: ii08 : 14 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.012	0.01059	7086	6507	*	*	*
ESGProbe	0	0.023	0.02198	2366	2301	8186	18028	75
ESGPrb+AR	0	0.018	0.01807	2016	1989	6063	11734	50
SAPS	0	0.017	0.01696	5369	5336	*	*	*
DLM	0.3	0.068	0.02125	8058	2534	*	*	*
Novelty	0	0.029	0.02865	6300	6300	*	*	*
WalkSAT	0	0.025	0.02206	6742	5973	*	*	*
HSAT	71.9	7.204	3.17529	1111708	490023	*	*	*
GSAT	70.3	7.160	5.38285	886078	666126	*	*	*
SurvProp	0	0.183	0.08033	6874	3024	0	10	0
Chaffish	10.0	1.096	*	*	*	6864	578132	3906
SATZ	0	0.094	*	*	*	36	*	0

Table E.51: ii16 : 10 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0080	0.0075	2464	2338	*	*	*
ESGProbe	0	0.0031	0.0031	333	332	174	1686	8
ESGPrb+AR	0	0.0025	0.0025	291	291	95	984	4
SAPS	0	0.0127	0.0122	2172	2082	*	*	*
DLM	0	0.0188	0.0158	1698	1430	*	*	*
Novelty	0	0.0092	0.0092	1813	1807	*	*	*
WalkSAT	0	0.0121	0.0104	1797	1544	*	*	*
HSAT	97.1	4.8588	3.5629	2207161	1618471	*	*	*
GSAT	34.0	2.4202	1.4500	740340	443557	*	*	*
SurvProp	0.2	0.1183	0.0207	4458	779	0	2	0
Chaffish	0	0.0090	*	*	*	958	11014	89
SATZ	5.9	0.6231	*	*	*	92	*	0

Table E.52: ii32 : 17 instances, satisfiable, Inductive Inference (DIMACS - Resende) [Max Time: 5s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.001340	0.0012	903	818	*	*	*
ESGProbe	0	0.002110	0.0021	154	154	277	1706	49
ESGPrb+AR	0	0.001564	0.0015	190	189	213	1224	35
SAPS	0	0.002767	0.0024	1219	1063	*	*	*
DLM	0	0.002090	0.0019	856	771	*	*	*
Novelty	0	0.014862	0.0085	7937	4555	*	*	*
WalkSAT	0	0.007673	0.0061	4544	3589	*	*	*
HSAT	72	0.720978	0.0657	606204	55277	*	*	*
GSAT	71	0.720559	0.0818	457743	51993	*	*	*
SurvProp	18	0.185607	0.0063	4765	163	0	8	0
Chaffish	0	0.001340	*	*	*	102	1193	49
SATZ	0	0.001958	*	*	*	6	*	0

Table E.53: jnh-sat : 16 instances, satisfiable, Constant Density Random (DIMACS - Hooker) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.013	0.011	10909	8724	*	*	*
ESGProbe	0	0.055	0.055	2479	2479	21362	64252	419
ESGPrb+AR	0	0.034	0.034	1831	1831	18674	40731	273
SAPS	0	0.019	0.017	9333	8474	*	*	*
DLM	0	0.027	0.022	8020	6568	*	*	*
Novelty	3	0.272	0.271	183030	182610	*	*	*
WalkSAT	1	0.151	0.147	112115	109255	*	*	*
HSAT	100	1.006	1.006	351611	351611	*	*	*
GSAT	100	1.010	1.010	296223	296223	*	*	*
SurvProp	100	1.007	0	1121	0	0	48	0
Chaffish	0	0.010	*	*	*	6263	19711	206
SATZ	100	1.003	*	*	*	8575	*	616

Table E.54: logistics.a : 828v, 6718c, satisfiable, Logistics, 8 Packages, 11 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0082	0.008	6294	6288	*	*	*
ESGProbe	0	0.0537	0.054	2001	2001	20986	68224	367
ESGPrb+AR	0	0.0367	0.037	1691	1691	20014	49868	262
SAPS	0	0.0160	0.016	7901	7898	*	*	*
DLM	0	0.0299	0.030	8380	8364	*	*	*
Novelty	0	0.4347	0.435	257993	257993	*	*	*
WalkSAT	0	0.3417	0.342	238438	238438	*	*	*
HSAT	100	2.0052	2.005	702644	702644	*	*	*
GSAT	100	2.0101	2.010	603221	603221	*	*	*
SurvProp	100	2.0120	0	97597	0	0	275	0
Chaffish	0	0.0130	*	*	*	7518	27153	236
SATZ	0	0.0085	*	*	*	64	*	615

Table E.55: logistics.b : 843v, 7301c, satisfiable, Logistics, 5 Packages, 13 Steps (SATLIB - Kautz and Selman) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0135	0.01	9655	9655	*	*	*
ESGProbe	0	0.1128	0.11	3467	3467	38054	117833	603
ESGPrb+AR	0	0.0723	0.07	2757	2757	35858	81546	429
SAPS	0	0.0275	0.03	12261	12185	*	*	*
DLM	0	0.0542	0.05	12101	11805	*	*	*
Novelty	0	0.7901	0.78	491330	486455	*	*	*
WalkSAT	0	0.8705	0.86	626399	622267	*	*	*
HSAT	100	10.0043	10.00	2697287	2697287	*	*	*
GSAT	100	10.0145	10.01	2299466	2299466	*	*	*
SurvProp	100	9.5251	0	3737362	0	0	463	0
Chaffish	0	0.0270	*	*	*	15267	40329	303
SATZ	0	0.1304	*	*	*	779	*	909

Table E.56: logistics.c : 1141v, 10719c, satisfiable, Logistics, 7 Packages, 13 Steps (SATLIB - Kautz and Selman) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.077	0.077	57616	57616	*	*	*
ESGProbe	0	0.191	0.191	5239	5239	3596	362459	159
ESGPrb+AR	0	0.098	0.098	4543	4543	2107	210387	90
SAPS	0	0.153	0.151	64759	63962	*	*	*
DLM	0	0.282	0.282	36826	36826	*	*	*
Novelty	0	0.574	0.574	368847	368617	*	*	*
WalkSAT	0	1.507	1.498	1082225	1076011	*	*	*
HSAT	100	10.007	10.007	718465	718465	*	*	*
GSAT	100	10.015	10.015	612901	612901	*	*	*
SurvProp	100	9.616	NaNQ	0	0	0	231	0
Chaffish	0	0.038	*	*	*	1657	129590	231
SATZ	100	10.009	*	*	*	2455	*	0

Table E.57: logistics.d : 4713v, 21991c, satisfiable, Logistics, 9 Packages, 14 Steps (SATLIB - Kautz and Selman) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00043	0.000425	295	295	*	*	*
ESGProbe	0	0.00032	0.000315	82	82	10	181	1
ESGPrb+AR	0	0.00033	0.000334	86	86	11	209	1
SAPS	0	0.00064	0.000634	307	306	*	*	*
DLM	0	0.00074	0.000740	264	264	*	*	*
Novelty	0	0.00263	0.001604	1685	1029	*	*	*
WalkSAT	0	0.00180	0.001108	1363	839	*	*	*
HSAT	89	0.89352	0.000749	1063188	891	*	*	*
GSAT	92	0.93333	0.001466	789398	1240	*	*	*
SurvProp	59	0.02024	0	0	0	0	88	0
Chaffish	0	0.00984	*	*	*	7	151	1
SATZ	0	0.00080	*	*	*	1	*	0

Table E.58: medium : 116v, 953c, satisfiable, Blocks World 5 Blocks, 4 Steps (SATLIB - Kautz and Selman) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00585	0.00526	9688	8704	*	*	*
ESGProbe	0	0.00082	0.00082	191	191	24	967	10
ESGPrb+AR	0	0.00087	0.00087	210	210	25	935	10
SAPS	4.4	0.27890	0.19806	246266	174883	*	*	*
DLM	0	0.01048	0.00896	10341	8844	*	*	*
Novelty	0	0.08012	0.04468	69915	38988	*	*	*
WalkSAT	2.4	0.22474	0.08084	206023	74103	*	*	*
HSAT	97.4	0.98082	0.25102	456550	116846	*	*	*
GSAT	96.6	0.98559	0.98558	347839	347839	*	*	*
SurvProp	95.8	0.01830	0	0	0	0	103	0
Chaffish	0	0.00037	*	*	*	12	731	9
SATZ	0	0.00122	*	*	*	6	*	0

Table E.59: par08 : 5 instances, satisfiable, Unsimplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00166	0.00131	2348	1851	*	*	*
ESGProbe	0	0.00033	0.00033	82	82	19	356	8
ESGPrb+AR	0	0.00032	0.00031	94	94	19	321	7
SAPS	0	0.00321	0.00270	2805	2355	*	*	*
DLM	0	0.00204	0.00162	2131	1689	*	*	*
Novelty	0	0.01203	0.01028	8478	7248	*	*	*
WalkSAT	0	0.02887	0.01698	26153	15378	*	*	*
HSAT	96	0.96708	0.00292	1574456	4751	*	*	*
GSAT	94	0.95503	0.00700	1043777	7654	*	*	*
SurvProp	93	0.02070	0	10703	0	0	34	0
Chaffish	0	0.00026	*	*	*	11	302	8
SATZ	0	0.00039	*	*	*	2	*	0

Table E.60: par08c : 5 instances, satisfiable, Simplified Learning Parity Function, 8 orig. vars (DIMACS - Crawford) [Max Time: 1s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	51.396	24	83426432	39690808	*	*	*
ESGProbe	0	7.376	7	160422	158557	66581	3167715	13061
ESGPrb+AR	0	2.266	2	233586	186615	34012	1519084	5815
SAPS	94	285.597	923	237221537	211362467	*	*	*
DLM	0	30.985	18	22477451	13349478	*	*	*
Novelty	74	265.981	1560	204219561	190118013	*	*	*
WalkSAT	100	300.002	973	248334233	248334233	*	*	*
HSAT	100	300.001	300	53579208	53579208	*	*	*
GSAT	100	300.001	300	40658740	40658740	*	*	*
SurvProp	100	0.223	0	0	0	0	432	0
Chaffish	0	1.055	*	*	*	9320	1828442	9145
SATZ	0	1.978	*	*	*	1358	*	0

Table E.61: par16 : 5 instances, satisfiable, Unsimplified Learning Parity Function, 16 orig. vars (DIMACS - Crawford) [Max Time: 300s Reps: 10]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	4.972	2.3	7453296	3516626	*	*	*
ESGProbe	0	2.121	1.8	85712	72074	45767	1075106	8047
ESGPrb+AR	0	1.354	1.4	161092	161092	37340	863509	6365
SAPS	0	11.020	113.1	9163962	5893089	*	*	*
DLM	0	8.484	4.5	6301159	3348791	*	*	*
Novelty	54	146.811	1351.1	15662512	65089825	*	*	*
WalkSAT	100	199.896	11864.4	37156678	166005697	*	*	*
HSAT	100	199.598	3750.4	7242941	136091960	*	*	*
GSAT	100	199.602	3418.3	14449455	100348801	*	*	*
SurvProp	100	1.091	0	950583	0	0	158	0
Chaffish	0	0.412	*	*	*	5787	527033	5549
SATZ	0	0.379	*	*	*	1059	*	84

Table E.62: par16c : 5 instances, satisfiable, Simplified Learning Parity Function, 16 orig. vars (DIMACS - Crawford) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	4.9	3.0	850273	520475	*	*	*
ESGProbe	0.5	5.8	5.8	48697	48392	44334	856466	5107
ESGPrb+AR	3.8	19.0	17.6	151771	140819	60545	1405335	9452
SAPS	25.3	63.3	59.7	2881751	2718848	*	*	*
DLM	4.9	25.1	17.9	748581	533718	*	*	*
Novelty	15.0	43.0	344.4	9377355	6300449	*	*	*
WalkSAT	31.8	71.3	344.6	7646872	16203966	*	*	*
SurvProp	84.2	12.1	0	0	0	0	64	0
Chaffish	0	2.0	*	*	*	6239	683943	5591
SATZ	0	3.9	*	*	*	1219	*	0

Table E.63: qg-sat : 10 instances, satisfiable, Quasigroup (SATLIB - Zhang) [Max Time: 200s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.00345	0.00342	3254	3229	*	*	*
ESGProbe	0	0.00387	0.00386	958	958	188	3823	3
ESGPrb+AR	0	0.00340	0.00339	917	917	180	3509	3
SAPS	0	0.00665	0.00640	4071	3922	*	*	*
DLM	0	0.01119	0.01079	3157	3045	*	*	*
Novelty	46	0.99243	0.38954	440386	172858	*	*	*
WalkSAT	0	0.03109	0.02900	28995	27047	*	*	*
HSAT	100	2.00728	2.00728	346924	346924	*	*	*
GSAT	100	2.00803	2.00803	269911	269911	*	*	*
SurvProp	76	0.54805	0	0	0	0	434	0
Chaffish	0	0.00082	*	*	*	120	1609	4
SATZ	0	0.04812	*	*	*	72	*	0

Table E.64: ssa-sat : 4 instances, satisfiable, Circuit Single-Stuck-At Fault Analysis (DIMACS - van Gelder) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0019	0.0015	2000	1583	*	*	*
ESGProbe	0	0.0092	0.0087	642	607	1693	8207	251
ESGPrb+AR	0	0.0074	0.0064	965	829	1288	6426	199
SAPS	0	0.0034	0.0029	2442	2090	*	*	*
DLM	0	0.0027	0.0020	2074	1572	*	*	*
Novelty	0.25	0.0538	0.0115	37393	7986	*	*	*
WalkSAT	0	0.0118	0.0079	9097	6110	*	*	*
HSAT	79.02	1.5849	0.0748	1803987	4061	*	*	*
GSAT	78.93	1.5968	0.1807	1235662	42619	*	*	*
SurvProp	36.66	0.2922	0.0561	98982	19016	0	14	0
Chaffish	0	0.0121	*	*	*	722	13749	536
SATZ	0	0.0050	*	*	*	30	*	112

Table E.65: uf125 : 100 instances, 125v, 538c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 2s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0029	0.0023	2941	2339	*	*	*
ESGProbe	0	0.0171	0.0150	1063	928	2932	14513	383
ESGPrb+AR	0	0.0135	0.0107	1613	1276	2202	11290	300
SAPS	0	0.0050	0.0040	3534	2831	*	*	*
DLM	0	0.0044	0.0030	3300	2235	*	*	*
Novelty	0.2	0.1193	0.0161	84979	11488	*	*	*
WalkSAT	0	0.0201	0.0117	15137	8792	*	*	*
HSAT	79.0	7.9104	11.1807	7520830	310085	*	*	*
GSAT	78.2	7.8727	5.0288	5579970	272122	*	*	*
SurvProp	41.3	0.7105	3.7426	468870	207406	0	11	0
Chaffish	0	0.0438	*	*	*	1629	35808	1230
SATZ	0	0.0089	*	*	*	50	*	102

Table E.66: uf150 : 100 instances, 150v, 645c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0056	0.0044	6057	4768	*	*	*
ESGProbe	0	0.0536	0.0471	2474	2177	7242	36245	850
ESGPrb+AR	0	0.0539	0.0384	4557	3245	6301	33134	790
SAPS	0	0.0104	0.0081	7310	5685	*	*	*
DLM	0	0.0086	0.0059	6667	4544	*	*	*
Novelty	0.3	0.2439	0.0465	175889	33515	*	*	*
WalkSAT	0	0.0386	0.0228	29673	17522	*	*	*
HSAT	84.5	8.4607	12.3951	7542404	353014	*	*	*
GSAT	84.0	8.4511	12.7667	5427843	413742	*	*	*
SurvProp	48.7	0.9085	3.9532	521599	400784	0	11	0
Chaffish	0	0.2192	*	*	*	4905	118028	3744
SATZ	0	0.0207	*	*	*	111	*	102

Table E.67: uf175 : 100 instances, 175v, 753c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 10s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.011	0.007	12490	7349	*	*	*
ESGProbe	0.06	0.191	0.150	5307	4161	15150	78944	1675
ESGPrb+AR	0.14	0.281	0.190	11132	7537	14883	81587	1757
SAPS	0	0.021	0.016	14725	11064	*	*	*
DLM	0	0.018	0.012	13209	8664	*	*	*
Novelty	2.32	0.775	0.460	560780	77850	*	*	*
WalkSAT	0.27	0.199	0.155	155891	121417	*	*	*
HSAT	89.06	17.823	231.079	14785617	2369082	*	*	*
GSAT	88.56	17.756	103.296	10582225	1595770	*	*	*
SurvProp	61.60	1.381	1.282	840003	381053	0	9	0
Chaffish	1.00	1.530	*	*	*	13931	367751	10688
SATZ	0	0.046	*	*	*	241	*	101

Table E.68: uf200 : 100 instances, 200v, 860c, satisfiable, r=4.3, hard, random 3-SAT (SATLIB) [Max Time: 20s Reps: 100]

Solver	Fail%	Av Time(s)	OEERT(s)	Av Steps	OEESS	Av Chc	Av UPs	Av Cl+
ESGint	0	0.0170	0.011	19110	12306	*	*	*
ESGProbe	0.6	0.5897	0.385	11831	7716	37345	191141	3314
ESGPrb+AR	2.0	1.0690	0.548	30293	15533	43882	237023	4180
SAPS	0	0.0320	0.022	21803	15266	*	*	*
DLM	0	0.0325	0.019	23660	13836	*	*	*
Novelty	1.3	0.7230	1.648	515978	197095	*	*	*
WalkSAT	0	0.0858	0.068	61357	48831	*	*	*
HSAT	88.3	17.6723	104.660	12583352	757743	*	*	*
GSAT	88.2	17.7269	155.703	9220229	1763709	*	*	*
SurvProp	51.0	1.2998	0.001	705022	254	0	11	0
Chaffish	28.0	7.8136	*	*	*	38929	1175259	29566
SATZ	0	0.2643	*	*	*	1343	*	94

Table E.69: uf250 : 100 instances, 250v, 1065c, satisfiable, r=4.26, hard, random 3-SAT (SATLIB) [Max Time: 20s Reps: 100]

Bibliography

[1] *2nd DIMACS Challenge on Cliques, Coloring, and Satisfiability*, October 1993.

[2] A. Andersson, M. Tenhunen, and F. Ygge. Integer programming for combinatorial auction winner determination. In *Proceedings of the Fourth International Conference on Multiagent Systems (ICMAS-00)*, Boston, 2000.

[3] Peter Barth. A Davis-Putnam based enumeration algorithm for linear pseudo-Boolean optimization. Research Report MPI-I-95-2-003, Max-Planck-Institut für Informatik, Im Stadtwald, D-66123 Saarbrücken, Germany, January 1995.

[4] Roberto Battiti. Reactive search: Toward self–tuning heuristics. In V. J. Rayward-Smith, I. H. Osman, C. R. Reeves, and G. D. Smith, editors, *Modern Heuristic Search Methods*, pages 61–83. John Wiley & Sons Ltd., Chichester, 1996.

[5] Roberto Bayardo and Robert Schrag. Using CSP Look-Back Techniques to Solve Real-World SAT Instances. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 203–208, 1997.

[6] R. J. Bayardo Jr. and D. P. Miranker. A complexity analysis of space-bounded learning algorithms for the constraint satisfaction problem. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 298–304, 1996.

[7] Roberto J. Bayardo Jr. and Robert Schrag. Using CSP look-back techniques to solve exceptionally hard SAT instances. In *Proceedings of the 2nd International Conference on the Principles and Practice of Constraint Programming (CP-96)*, pages 46–60, 1996.

[8] Adam Beacham, Xinguang Chen, Jonathan Sillito, and Peter van Beek. Constraint programming lessons learned from crossword puzzles. In *Proceedings of the 14th Canadian Conference on Artificial Intelligence (AI-2001)*, pages 78–87, June 2001.

[9] A. Beringer, G. Aschemann, H. Hoos, M. Metzger, and A. Weiß. GSAT versus simulated annealing. In A. G. Cohn, editor, *Proceedings of the 11th European Conference on Artificial Intelligence (ECAI-94)*, pages 130–134. John Wiley & Sons, 1994.

[10] Dimitri P. Bertsekas. *Nonlinear Programming*. Athena Scientific, Belmont, Massachusetts, 2nd edition edition, 1999.

[11] A. Biere and W. Kunz. SAT and ATPG: Boolean engines for formal hardware verification. In *Proceedings of 20th IEEE/ACM International Conference on Computer Aided Design (ICCAD-02)*, San Jose, CA, USA, November 2002. IEEE.

[12] Justin A. Boyan and Andrew W. Moore. Learning evaluation functions for global optimization and boolean satisfiability. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, pages 3–10, 1998.

[13] A. Braunstein, M. Mezard, and R. Zecchina. Survey propagation: an algorithm for satisfiability. 2002.

[14] Carla P. Gomes. *Constraint and Integer Programming: Toward a Unified Methodology*, chapter Complete Randomized Backtrack Search, pages 233–283. Kluwer, 2003.

[15] Byungki Cha and Kazuo Iwama. Performance Test of Local Search Algorithms Using New Types of Random CNF Formulas. In *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 304–311. Kyushu University, 1995.

[16] Byungki Cha and Kazuo Iwama. Adding New Clauses for Faster Local Search. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 332–337, 1996.

[17] Peter Cheeseman, Bob Kanefsky, and William M. Taylor. Where the Really Hard Problems Are. In *Proceedings of 12th International Joint Conference on Artificial Intelligence (IJCAI-91)*, pages 331–337, 1991.

[18] Xinguang Chen and Peter van Beek. Conflict-directed backjumping revisited. *Journal of Artificial Intelligence Research*, 14:53–81, 2001.

[19] Stephen A. Cook. The complexity of theorem-proving procedures. In *Conference Record of Third Annual ACM Symposium on Theory of Computing*, pages 151–158, Shaker Heights, Ohio, 1971.

[20] Stephen A. Cook and David G. Mitchell. *Satisfiability Problem: Theory and Applications*, volume 35 of *DIMACS Series in Discrete Mathamatics and Theoretical Computer Science*, chapter Finding Hard Instances of the Satisfiability Problem. 1997.

[21] James M. Crawford. Solving Satisfiability Problems Using a Combination of Systematic and Local Search. In *Second DIMACS Challenge: Cliques, Coloring, and Satisfiability*, Rutgers University, NJ, October 1993.

[22] James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3-SAT. *Artificial Intelligence*, 81(1-2):31–57, 1996.

[23] Joseph Culberson, Ian P. Gent, and Holger Hoos. On the Probabilistic Approximate Completeness of WalkSAT for 2-SAT. Technical Report APES-15a-2000, APES Research Group, 2000.

[24] Andrew J. Davenport, Edward P. K. Tsang, Chang J. Wang, and Kangmin Zhu. GENET: A connectionist architecture for solving constraint satisfaction problems by iterative improvement. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 325–330, 1994.

[25] Martin Davis, George Logemann, and Donald Loveland. A Machine Program for Theorem-Proving. *Communications of the ACM*, 5:394–397, 1962.

[26] Martin Davis and Hilary Putnam. A Computing Procedure for Quantification Theory. *Journal of the ACM*, 7:201–215, 1960.

[27] Rina Dechter. *Constraint Processing*. Morgan Kaufmann, May 2003.

[28] O. Dubois, P. Andre, Y. Boufkhad, and J. Carlier. SAT Versus UNSAT. In D. S. Johnson and M. A. Trick, editor, *Second DIMACS Challenge: Cliques, Coloring and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[29] Gal Elidan, Matan Ninio, Nir Friedman, and Dale Schuurmans. Data Perturbation for Escaping Local Maxima in Learning. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002)*, pages 132–139, 2002.

[30] H. Everett. Generalized Lagrange multiplier method for solving problems of optimum allocation of resources. *Operations Research*, 11(3):399–417, 1963.

[31] M.L. Fisher. The Lagrangian relaxation method for solving integer programming problems. *Management Science*, 27(1):1–18, 1981.

[32] Moti Frances and Ami Litman. On Covering Problems of Codes. In *Theory of Computing Systems*, volume 30, pages 113–119, March 1997.

[33] J. Franco and M. Paull. Probabilistic analysis of the Davis– Putnam procedure for solving the satisfiability problem. *Discrete Applied Mathematics*, 5(1):77–87, 1983.

[34] Jeremy Frank. Weighting for Godot: Learning Heuristics for GSAT. In *Proceedings of the 13th National Conference on Artificial Intelligence (AAAI-96)*, pages 338–343, 1996.

[35] Jeremy Frank. Learning short-term weights for GSAT. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 384–391, 1997.

[36] Jeremy Frank. *Local Search for NP-Hard Problems*. PhD thesis, University of California - Davis, 1997.

[37] Jon W. Freeman. *Improvements to Propositional Satisfiability Search Algorithms*. PhD thesis, University of Pennsylvania, Philadelphia, 1995.

[38] Jon W. Freeman. Hard Random 3-SAT Problems and the Davis-Putnam Procedure. *Artificial Intelligence*, 81(1-2):183–198, 1996.

[39] Daniel Frost and Rina Dechter. Look-ahead value ordering for constraint satisfaction problems. In *Proceedings of the 14th International Joint Conference on Artificial Intelligence (IJCAI-95)*, pages 572–578, Montreal, Canada, 1995.

[40] Y. Fujishima, K. Leyton-Brown, and Y. Shoham. Taming the computational complexity of combinatorial auctions: Optimal and approximate approaches. In *Proceedings of 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 548–553, Stockholm, 1999.

[41] Alex Fukunaga. Automated Discovery of Composite SAT Variable-Selection Heuristics. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002)*, 2002.

[42] Ian Gent and Toby Walsh. Unsatisfied variables in local search. In J. Hallam, editor, *Hybrid Problems, Hybrid Solutions (AISB-95)*, pages 73–85. IOS Press, 1995.

[43] Ian P. Gent and Toby Walsh. Towards an Understanding of Hill-Climbing Procedures for SAT. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-93)*, pages 28–33, 1993.

[44] F. Glover. Tabu Search - Part I. *ORSA Journal of Computing*, 1(3):190–206, 1989.

[45] E. Goldberg and Y. Novikov. Berkmin: A fast and robust sat solver. In *Design Automation and Test in Europe (DATE 2002)*, pages 142–149., 2002.

[46] Carla Gomes and Bart Selman. Algorithm Portfolios. *Artificial Intelligence*, 126:43–62, 2001.

[47] Carla P. Gomes and Bart Selman. Problem structure in the presence of perturbations. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*, pages 221–226, 1997.

[48] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry A. Kautz. Heavy-tailed phenomena in satisfiability and constraint satisfaction problems. *Journal of Automated Reasoning*, 24(1/2):67–100, 2000.

[49] Carla P. Gomes, Bart Selman, and Henry Kautz. Boosting Combinatorial Search Through Randomization. In *Proceedings of the 15th National Conference on Artificial Intelligence (AAAI-98)*, 1998.

[50] Jun Gu. Efficient local search for very large-scale satisfiability problems. *SIGART Bulletin*, 3(1):8–12, 1992.

[51] Jun Gu. Local Search for the Satisfiability (SAT) Problem. *IEEE Transactions on Systems and Cybernetics*, 23(3):1108–1129, July 1993.

[52] Djamal Habet, Chu Min Li, Laure Devendeville, and Michel Vasquez. A Hybrid Approach for SAT. In P. Van Hentenryck, editor, *Proceedings of the 8th International Conference on the Principles and Practice of Constraint Programming (CP-2002)*, volume 2470 of *Lecture Notes in Computer Science*, pages 172–184, September 2002.

[53] Michael Held and Richard M. Karp. The Traveling-Salesman Problem and Minimum Spanning Trees. *Operations Research*, 18(6):1138–1162, 1970.

[54] E. Hirsch and A. Kojevnikov. Unitwalk: A new sat solver that uses local search guided by unit clause elimination. In *Proceedings of the 7th International Conference on the Principles and Practice of Constraint Programming (CP-2001)*, pages 605–609, 2001.

[55] Edward A. Hirsch and Arist Kojevnikov. Solving Boolean satisfiability using local search guided by unit clause elimination. In *Proceedings of the 7th International Conference on the Principles and Practice of Constraint Programming (CP-2001)*, November 2001.

[56] Edward A. Hirsch and Arist Kojevnikov. UnitWalk: A new SAT solver that uses local search guided by unit clause elimination. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*, pages 35–42, May 2002.

[57] Robert C. Holte. Combinatorial auctions, knapsack problems, and hill-climbing search. In *Proceedings of the Canadian Conference on ArtificiaI Intelligence (AI-2001)*, volume 2056 of *Lecture Notes in Computer Science*. Springer, 2001.

[58] Holger Hoos. On the Run-time Behaviour of Stochastic Local Search Algorithms for SAT. In *Proceedings of the 16th National Conference on Artificial Intelligence (AAAI-99)*, pages 661–666. MIT Press, 1999.

[59] Holger H. Hoos and Craig Boutilier. Solving combinatorial auctions using stochastic local search. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 22–29, 2000.

[60] Holger H. Hoos and Thomas Stützle. SATLIB: An Online Resource for Research on SAT. In I. P. Gent, H von Maaren, and T. Walsh, editors, *Proceedings of the 3rd International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2000)*, pages 283–292. IOS Press, 2000.

[61] Eric Horvitz, Yongshao Ruan, Carla P. Gomes, Henry Kautz, Bart Selman, and David Maxwell Chickering. A Bayesian approach to tackling hard computational problems. In *Proceedings of the 17th Conference on Uncertainty and Artificial Intelligence (UAI 2001)*, pages 235–244, August 2001.

[62] Bernardo A. Huberman, Rajan M. Lukose, and Tad Hogg. An economics approach to hard computational problems. *Science*, 275:51–54, January 1997.

[63] Frank Hutter, Dave A. D Tompkins, and Holger H. Hoos. Scaling and Probabilistic Smoothing: Efficient Dynamic Local Search for SAT. In *Proceedings of the 8th International Conference on the Principles and Practice of Constraint Programming (CP-2002)*, 2002.

[64] Narendra Jussien and Olivier Lhomme. Local search with constraint propagation and conflict-based heuristics. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 169–174, Austin, TX, USA, August 2000.

[65] Henry Kautz, Eric Horvitz, Yongshao Ruan, Carla Gomes, and Bart Selman. Dynamic Restart Policies. In *Proceedings of the 18th National Conference on Artificial Intelligence (AAAI-2002)*, 2002.

[66] Henry A. Kautz and Bart Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence (ECAI-92)*, pages 359–363, 1992.

[67] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science, Number 4598, 13 May 1983*, 220, 4598:671–680, 1983.

[68] Torbjörn Larsson, Michael Patriksson, and Strömberg Ann-Brith. Conditional Subgradient Optimization - Theory and Applications. *European Journal of Operational Research*, 88:382–403, 1996.

[69] Kevin Leyton-Brown, Eugene Nudelman, and Yoav Shoham. Learning the empirical hardness of optimization problems: The case of combinatorial auctions. In *Proceedings of the 8th International Conference on the Principles and Practice of Constraint Programming (CP-2002)*, 2002.

[70] Kevin Leyton-Brown, Mark Pearson, and Yoav Shoham. Towards a universal test suite for combinatorial auction algorithms. In *Proceedings of the ACM Conference on Electronic Commerce (EC-00)*, pages 66–76, 2000.

[71] Chu Min Li. A Constraint-Based Approach to Narrow Search Trees for Satisfiability. *Information Processing Letters*, 71:75–80, 1999.

[72] Chu Min Li and Anbulagan. Heuristics Based On Unit Propagation for Satisfiability Problems. In *Proceedings of 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 366–371. LaRIA, Univ. de Picardie Jules Verne, Morgan Kaufmann, 1997.

[73] Chu Min Li and Anbulagan. Look-Ahead Versus Look-Back for Satisfiability Problems. In *Proceedings of the 3rd International Conference on the Principles and Practice of Constraint Programming (CP-97)*, LNCS 1330, pages 342–356. Springer-Verlag, 1997.

[74] Nick Littlestone and Manfred K. Warmuth. The weighted majority algorithm. In *IEEE Symposium on Foundations of Computer Science*, pages 256–261, 1989.

[75] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of Las Vegas Algorithms. In *Israel Symposium on Theory of Computing Systems*, pages 128–133, 1993.

[76] I. Lynce and J. Marques-Silva. Efficient data structures for fast sat solvers. In *Proceedings of the 5th International Symposium on the Theory and Applications of Satisfiability Testing (SAT 2002)*, pages 308–315, 2002.

[77] Vasco Manquinho and João Marques-Silva. Search pruning conditions for boolean optimization. In *Proceedings of the 14th European Conference on Artificial Intelligence (ECAI-2000)*, pages 103–107, August 2000.

[78] Vasco M. Manquinho, João P. Marques Silva, Arlindo L. Oliveira, and Karem A. Sakallah. Satisfiability-based algorithms for 0-1 integer programming. In *IEEE/ACM International Workshop on Logic Synthesis*, June 1998.

[79] João P. Marques-Silva and Karem A. Sakallah. GRASP: A Search Algorithm for Propositional Satisfiability. *IEEE Transactions on Computers*, 48(5):506 –521, 1999.

[80] David McAllester, Bart Selman, and Henry Kautz. Evidence for Invariants in Local Search. In *Proceedings of the 14th National Conference on Artificial Intelligence (AAAI-97)*. AT&T Laboratories, 1997.

[81] Marc Mezard, Giorgio Parisi, and Riccardo Zecchina. Analytic and algorithmic solutions of random satisfiability problems. *Science*, 297(5582):812–815, 27 June 2002.

[82] P. Mills and E.P.K. Tsang. Guided local search for solving SAT and weighted MAX-SAT problems. *Journal of Automated Reasoning*, 24:205–223, 2000.

[83] David Mitchell, Bart Selman, and Hector Levesque. Hard and Easy Distributions of SAT Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 459–465. AT&T Bell Labs, July 1992.

[84] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the 11th National Conference on Artificial Intelligence (AAAI-93)*, pages 40–45, 1993.

[85] Matthew H. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an Efficient SAT Solver. In *39th Design Automation Conference*, June 2001.

[86] C. H. Papadimitriou. On selecting a satisfying truth assignment. In *Proceedings of the 32nd Annual IEEE Symposium on Foundations of Computer Science, FOCS-91*, pages 163–169, 1991.

[87] 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, Portland, OR, 1996.

[88] Steven Prestwich. Local search and backtracking versus non-systematic backtracking. In *Papers from the AAAI 2001 Fall Symposium on Using Uncertainty Within Computation*, pages 109–115, North Falmouth, Cape Cod, MA, November 2-4 2001. The American Association for Artificial Intelligence, The AAAI Press.

[89] D. Pretolani. Efficiency and Stability of Hypergraph SAT Algorithms. In D. S. Johnson and M. A. Trick, editor, *Second DIMACS Challenge: Cliques, Coloring and Satisfiability*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*. American Mathematical Society, 1996.

[90] P. Purdom, C. Brown, and E. Robertson. Backtracking with multi-level dynamic search rearrangement. *Acta Informatica*, 15:99–114, 1981.

[91] E. Thomas Richards and Barry Richards. Nonsystematic search and no-good learning. *Journal of Automated Reasoning*, 24(4):483–533, 2000.

[92] J. A. Robinson. A Machine-Oriented Logic Based on the Resolution Principle. *Journal of the ACM*, 12(1):23–41, January 1965.

[93] Sheldon M. Ross. *Introduction to Probability Models*. Academic Press, San Diego, 6th edition edition, 1997.

[94] Stuart Russell and Peter Norvig. *Artifical Intelligence: A Modern Approach*. Prentice Hall, Upper Saddle River, NJ, 1st edition, 1995.

[95] Norman Sadeh-Koniecpol and Mark S. Fox. Variable and Value Ordering Heuristics for Activity-Based Job-Shop Scheduling. In *Proceedings of the Fourth International Conference on Expert Systems in Production and Operations Management*, 1990.

[96] Tuomas Sandholm. An algorithm for optimal winner determination in combinatorial auctions. In *Proceedings of 16th International Joint Conference on Artificial Intelligence (IJCAI-99)*, pages 542–547, 1999.

[97] Tuomas Sandholm. Algorithm for optimal winner determination in combinatorial auctions. *Artificial Intelligence*, 135(1-2):1–54, 2002.

[98] Tuomas Sandholm, Subhash Suri, Andrew Gilpin, and David Levine. CABOB: A fast optimal algorithm for combinatorial auctions. In *Proceedings of 17th International Joint Conference on Artificial Intelligence (IJCAI-2001)*, pages 1102–1108, 2001.

[99] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. In *Proceedings of the 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 297–302, 2000.

[100] Dale Schuurmans and Finnegan Southey. Local search characteristics of incomplete SAT procedures. *Artificial Intelligence*, 132(2):121–150, 2001.

[101] Dale Schuurmans, Finnegan Southey, and Robert C. Holte. The exponentiated subgradient algorithm for heuristic boolean programming. In *Proceedings of 17th International Joint Conference on Artificial Intelligence (IJCAI-2001)*, volume 1, pages 334–341, 2001.

[102] Bart Selman and Henry Kautz. Domain-Independent Extensions to GSAT: Solving Large Structured Satisfiability Problems. In *Proceedings of 14th International Joint Conference on Artificial Intelligence (IJCAI-93)*. AT&T Laboratories, 1993.

[103] Bart Selman and Henry Kautz. An Empirical Study of Greedy Local Search for Satisfiability Testing. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-93)*, pages 46–51, 1993.

[104] Bart Selman, Henry A. Kautz, and Bram Cohen. Local Search Strategies for Satisfiability Testing. In *2nd DIMACS Challenge on Cliques, Coloring, and Satisfiability*. AT&T Bell Labs, October 1993.

[105] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise Strategies for Improving Local Search. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*. AT&T Bell Labs, July 1994.

[106] Bart Selman, Hector Levesque, and David Mitchell. A New Method for Solving Hard Satisfiability Problems. In *Proceedings of the 10th National Conference on Artificial Intelligence (AAAI-92)*, pages 440–446. AT&T, University of Toronto, Simon Fraser, 1992.

[107] J. Slaney, M. Fujita, and M. Stickel. Automated reasoning and exhaustive search: Quasigroup existence problems. *Computers and Mathematics with Applications*, 1993.

[108] William M Spears. Simulated Annealing for Hard Satisfiability Problems. In David S. Johnson and Michael A. Trick, editors, *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, volume 26 of *DIMACS Series in Discrete Mathematics and Theoretical Computer Science*, pages 533–558. American Mathematical Society, 1996.

[109] 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 (AIMA-04)*, 2004.

[110] Miroslav N. Velev and Randal E. Bryant. Effective use of boolean satisfiability procedures in the formal verification of superscalar and VLIW. In *Proceedings of the 38th Conference on Design Automation Conference 2001*, pages 226–231, New York, NY, USA, 2001. ACM Press.

[111] Benjamin W. Wah and Yi Shang. Discrete Lagrangian-Based Search for Solving MAX-SAT Problems. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI-97)*, pages 378–383, August 1997.

[112] Benjamin W. Wah and Yi Shang. A Discrete Langrangian-Based Global-Search Method for Solving Satisfiability Problems. *Journal of Global Optimization*, 12(1):61–99, January 1998.

[113] J. P. Walser. *Integer Optimization by Local Search - A Domain-Independent Approach*, volume 1637 of *Lecture Notes in Artificial Intelligence*. Springer-Verlag, August 1999.

[114] Daniel S. Weld. Recent advances in AI planning. *AI Magazine*, 20(2):93–123, 1999.

[115] Ryan Williams, Carla Gomes, and Bart Selman. Backdoors To Typical Case Complexity. In *Proceedings of 18th International Joint Conference on Artificial Intelligence (IJCAI-2003)*, Acapulco, Mexico, 2003.

[116] Zhe Wu. The Discrete Lagrangian Theory and its Application to Solve Nonlinear Discrete Constrained Optimization Problems. Master's thesis, University of Illinois at Urbana-Champaign, Department of Computer Science, May 1998.

[117] Zhe Wu and Benjamin W. Wah. Solving Hard Satisfiability Problems: A Unified Algorithm Based On Discrete Lagrange Multipliers. In *Proceedings of 11th IEEE Conference on Tools with Artificial Intelligence*. UIUC, November 1999.

[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 16th National Conference on Artificial Intelligence (AAAI-99)*, 1999.

[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 17th National Conference on Artificial Intelligence (AAAI-2000)*, pages 310–315, July 2000.

[120] Makoto Yokoo. Weak-commitment search for solving constraint satisfaction problems. In *Proceedings of the 12th National Conference on Artificial Intelligence (AAAI-94)*, pages 313–318, Seattle, WA, USA, July 31 - August 4 1994. AAAI Press, 1994.

[121] Ramin Zabih and David A. McAllester. A rearrangement search strategy for determining propositional satisfiability. In *Proceedings of the 7th National Conference on Artificial Intelligence (AAAI-88)*, pages 155–160, St. Paul, Minnesota, 1988. AAAI Press / The MIT Press.

[122] H. Zhang and M. E. Stickel. An efficient algorithm for unit propagation. In *Proceedings of the Fourth International Symposium on Artificial Intelligence and Mathematics (AI-MATH-96)*, Fort Lauderdale (Florida USA), 1996.

[123] Hantao Zhang. SATO: An Efficient Propositional Prover. In *Conference on Automated Deduction*, pages 272–275, 1997.

[124] Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam Algorithm By Tries. Technical report, University of Iowa, 1994.

[125] Hantao Zhang and Mark E. Stickel. Implementing the Davis-Putnam method. *Journal of Automated Reasoning*, 24(1/2):277–296, 2000.

[126] Lintao Zhang, Conor F. Madigan, Matthew H. Moskewicz, and Sharad Malik. Efficient Conflict Driven Learning in a Boolean Satisfiability Solver. In *Proceedings of 19th IEEE/ACM International Conference on Computer Aided Design (ICCAD-01)*, November 2001.

Index

κ-smoothed objective function, 86
$\tilde{\alpha}$, 110
0-1 integer programming, 99
1-UIP cut, 35

additive updates, 87
adversarial optimization, 141
ANC, 53, 144
assignment level, 16
assignments, 4
atoms, 3

backdoors, 13
Backjumping, 31
backtracking search solvers, 15
balanced heuristics, 27
best portfoilio, 163
bids, 135
binary clause, 6
binary clauses, 7
boolean linear programming, 99
breakout, 51
bridged solvers, 185
broken clauses, 8
bsolo, 141

C-SAT, 27
CATS, 136
CBJ, 31
Chaff, 35
Chaffish, 163
choice level, 21
choice move, 19, 20
clause, 6
clause learning, 32
clause weighting, 51
clause-variable ratio, 13
clauses, 3
closed prefix, 21
CNF, 3
combinatorial auction, 135
complete, 14, 148
conflict, 19, 20
conflict-directed backjumping, 31
conjunctive normal form, 3
constraint propagation, 19
constraints, 3
contracting resolution, 9
contradiction, 10, 20
coverage, 71
covering radius, 72
CPLEX, 110, 141

David-Putnam-Logemann-Loveland, 19
Davis-Putnam, 15
Davis-Putnam-Logemann-Loveland, 19
DDPP, 29
decided, 4
decision move, 19
depth, 65, 76
DIMACS, 14
Discrete Lagrange Method, 55
Discrimination-tree-based Davis-Putnam Prover, 29
DLM, 55, 106
DLM-SAT, 55
DPLL, 15, 19
dual coverage, 76, 77
dual function, 56, 100
dual mobility, 76

dual problem, 56, 100
dual space metric, 76
dual step, 57
duality gap, 56, 100
DualRes, 144

EESS, 38
elimination rule, 15, 19
empty clause, 6
ESG, 99, 146
ESGflt, 108
ESGint, 109
ESGProbe, 156
estimated expected runtime under an optimal restart policy, 39
estimated expected search steps, 38
estimated expected search steps under an optimal restart policy, 39
exponentiated subgradient, 99, 104

failed literal detection, 27
false, 4
Fill, 53
flat move, 60
flooding, 87
forced literal, 19
forced move, 19, 20
formal verification, 12
forward checking, 27
free moves, 22
full assignment, 4
full assignments, 37

GLS, 63
GLSSAT, 63
goods, 135
GRASP, 35
GSAT, 42
Guided Local Search, 63

Hard Random Instances, 13
hardness, 10
head/tail lists, 29
HEADTAIL, 29
hinge penalty, 105
HR instances, 13
HSAT, 42
hybrids, 143

implication, 19
implies, 20
incomplete, 14
instance, 4
integer linear programming, 100

k-SAT, 4

Lagrange multipliers, 56
Lagrangian, 55
Lagrangian BLP formulation, 100
Lagrangian optimization, 55
length, 3
level, 6
linear penalty, 105
literals, 3
local search, 37
long-term weights, 62
lookback, 31
Loveland, 15

made clauses, 8
maximum occurrences of minimum-length, 26
metrics, 65
mobility, 68
model finder, 14
MOM, 26
mulitplicative updates, 87
multiplicative updates, 104

native solvers, 185
Natural Instances, 12
negation, 5
negative literal, 3
neighbourhood function, 37
nogoods, 32
noise, 44
noisy strategies, 44

non-chronological backtracking, 31
non-strict local minimum, 38
nonlinear penalty, 104
Novelty, 48
Novelty+, 48

objective function, 37
OEERT, 39
OEESS, 39
OIP, 140
opbdp, 141
open move, 19
open prefix, 21
optimal restart policy, 39
ordering, 16
over-constrained integer program, 140

partial assignment, 4, 16
pending move list, 189
phase transition, 13
plateau, 37
positive literal, 3
prefix, 6, 16
Prefixes, 6
primal BLP problem, 100
primal coverage, 71
primal mobility, 68
primal problem, 55
primal space metrics, 76
primal step, 57
primal-dual methods, 56
probabilistic asymptotic completeness, 48
properly undecided, 29
pure literal, 19, 22
pure literal moves, 22

R-Novelty, 48
R-Novelty+, 48
random walk, 44
reason, 31
reduce, 7
reduced clause, 7
reduction, 7
relevance-bounded learning, 32, 35
rescaled α, 110
RESG, 156
resolution, 9
resolution refutation, 10, 14
resolvands, 9
resolvent, 9
restarts, 28
Retrospective Variation of MAXFLIPS, 38
RPV, 38

SAPS, 108
SAT1, 42
satisfiability problem, 3
satisfied, 4
SATLIB, 14
SATO, 29, 35
SATZ, 146
SDF, 83
search, 16
short-term weights, 62
simulated annealing, 48
single unit auctions, 136
smoothed objective function, 84
smoothing, 55, 60, 105
source problem, 12
splitting rule, 19
strict local minimum, 38
strictly partial assignment, 4
Structured Instances, 12
subgradient, 102
subsume, 10
Subsumption, 10
survey propagation, 13, 63
systematic model finders, 15
systematic searchers, 15

Tabu, 48
tautology, 10
ternary clause, 6
Transformed Instances, 12
trie, 29
true, 4

UIP, 35
unassigned, 4
undecided, 4
unique implication points, 35
unit clause, 6
unit propagation, 19, 20
unit propagation heuristics, 28
unsatisfied, 4
UP heuristics, 28

vacuous clause, 10
vacuous clauses, 3
value ordering, 24, 26
variable ordering, 24

WalkSAT, 44, 140
WalkSAT-B, 45
WalkSAT-G, 46
watch literals, 29
weak duality theorem, 56, 100
weight decay, 54
weight profile, 54
weight smoothing, 88
Weighted GSAT, 51
weighted MOM, 26
weights, 51
WGSAT, 51
winner determination, 135
witness lists, 29
witness literals, 29, 109
WSAT(OIP), 140

zChaff, 35, 146