Novel Value Ordering Heuristics Using Non-Linear Optimization In Boolean Satisfiability

by

Vladimir Pisanov

A thesis
presented to the University of Waterloo
in fulfilment of the
thesis requirement for the degree of
Master of Mathematics
in
Computer Science

Waterloo, Ontario, Canada, 2012

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.

Vladimir Pisanov

Abstract

Boolean Satisfiability (SAT) is a fundamental NP-complete problem of determining whether there exists an assignment of variables which makes a Boolean formula evaluate to True. SAT is a convenient representation for many naturally occurring optimization and decisions problems such as planning and circuit verification. SAT is most commonly solved by a form of backtracking search which systematically explores the space of possible variable assignments. We show that the order in which variable polarities are assigned can have a significant impact on the performance of backtracking algorithms. We present several ways of transforming SAT instances into non-linear objective functions and describe three value-ordering methods based on iterative optimization techniques. We implement and test these heuristics in the widely-recognized MiniSAT framework. The first approach determines polarities by applying Newton's Method to a sparse system of non-linear objective functions whose roots correspond to the satisfying assignments of the propositional formula. The second approach determines polarities by minimizing an objective function corresponding to the number of clauses conflicting with each assignment. The third approach determines preferred polarities by performing stochastic gradient descent on objective functions sampled from a family of continuous potentials. The heuristics are evaluated on a set of standard benchmarks including random, crafted and industrial problems. We compare our results to five existing heuristics, and show that MiniSAT equipped with our heuristics often outperforms state-of-the-art SAT solvers.

Acknowledgments

I would like to thank my supervisor, professor Peter van Beek for encouragement and advice he has provided throughout my research. His guidance and constructive suggestions have played a crucial role in shaping this thesis. I also wish to express my sincere gratitude to the members of my graduate committee, associate professor Pascal Poupart and professor Peter A. Forsyth for their insightful feedback and helpful suggestions for improving my work.

Contents

List of Tables

List of Figures

Chapter 1

Introduction

In this chapter, we informally introduce and motivate the problem of value ordering in backtracking search in the context of Boolean Satisfiability. We also summarize our contributions to this field and outline the organization of the thesis.

1.1 Value Ordering in Boolean Satisfiability

The Boolean Satisfiability Problem (SAT) will always hold a special status in computer science as it was the first problem to be shown to be NP-Complete. SAT poses a deceptively simple question: given a Boolean formula F, is there an assignment of variables which makes F evaluate to $True$? While short formulas can be easily checked by hand or by a simple brute-force algorithm, longer formulas involving hundreds or thousands of variables render such methods prohibitively expensive because a function of n Boolean variables has 2^n possible assignments. Over the past fifty years, many innovative algorithms have been developed with the aim of solving the Satisfiability Problem without having to check all possible assignments. Such algorithms are efficiently implemented in computer programs collectively known as SAT solvers which can be used to solve practical SAT instances.

When one considers the ubiquity and the expressive power of Boolean formulas, it is not difficult to see why efficient SAT solvers are in demand. Boolean logic provides an intuitive and concise way of encoding many kinds of decision problems such as database queries and digital circuits. SAT is of tremendous importance in electronic design automation, artificial intelligence, algorithmics, and many other branches of computer science, engineering, and mathematics. Many real-world problems such as circuit verification, Field Programmable Gate Array routing, and planning are routinely tackled by SAT solvers. It is not uncommon for such problems to contain hundreds of thousands or even millions of variables. Moreover, a direct consequence of the Cook-Levin Theorem ([Coo71], [Lev73]) is that *any* problem in NP can be reduced to an equivalent SAT instance in polynomial time and space, which makes SAT a convenient gateway between complex problems and efficient solvers. For example, any Constraint Satisfaction Problem (CSP) such as scheduling or graph colouring can be encoded and solved as a SAT instance. While this may not always be the best

way to approach CSPs, it has been shown that such transformations can be done efficiently ([Gen02a], [Wal00]).

SAT solvers come in two varieties: complete and incomplete. Incomplete solvers are stochastic in nature, and search for a satisfying assignment by repeatedly refining an initial (typically random) guess. Unlike complete solvers, incomplete solvers cannot guarantee finding a solution, nor can they disprove a formula. Complete solvers, on the other hand, systematically explore the space of all variable assignments and will find a solution if one exists, given enough time. While it may seem like incomplete solvers are an inferior choice, they have been shown to outperform complete solvers on many classes of problems. Incomplete solvers such as those belonging to the `WalkSAT` [SKC93] family can rapidly discover solutions to huge industrial and random problems with millions of variables. Although we focus on complete solvers in this thesis, we make use of several techniques and concepts employed in incomplete solvers.

Early complete SAT solvers, most notably the Davis-Putnam algorithm [DP60], relied on a formula reduction technique known as *clause resolution*. Unfortunately, this approach had an exponential storage complexity in the worst case, and was therefore limited to very short formulas with only a handful of variables. In 1962, this limitation was lifted with the introduction of the Davis-Putnam-Loveland-Logeman (DPLL) procedure [DLL62] which solved SAT using a backtracking depth-first search. At each step, the algorithm would choose a variable and assign it to either $True$ or $False$, thereby reducing the size of the formula. If the assignment falsified the formula, the algorithm would backtrack and try the opposite polarity. This process is repeated until either a satisfying assignment is found, or until all combinations are exhausted, at which point the formula is determined to be unsatisfiable. Unlike clause resolution, this search mechanism has a linear memory complexity and is only limited by CPU time. Despite its simple nature, DPLL is still the basis of most modern complete SAT solvers. Over the years, much effort has gone into refining and improving the baseline DPLL algorithm. This gave rise to a successful class of algorithms known as Conflict-Driven Clause-Learning (CDCL) algorithms. These DPLL-based algorithms incorporate a number of additional techniques designed to speed up the search process. Most prominently, CDCL solvers make extensive use of encountered conflicts to guide the exploration of the search space. This thesis focuses on DPLL-CDCL solvers.

When implementing a DPLL solver, two natural questions arise: in what order should the variables be assigned, and which polarity should be tried first? It did not take long for researchers to observe that these two decisions can have a dramatic impact on the performance of backtracking search. Not surprisingly, finding the optimal answer to both questions is itself an NP-complete problem, which has spurred research into so-called *variable ordering* and *value ordering* heuristics. While variable ordering has received a lot of attention, value ordering heuristics tend to be less popular despite their enormous potential (a perfect value ordering heuristic leads to a solution of a satisfiable formula without any backtracking at all; the same cannot be said about a perfect variable ordering heuristic). Devising a value ordering heuristic that is both accurate and efficient has proved to be a daunting task; for this rea-

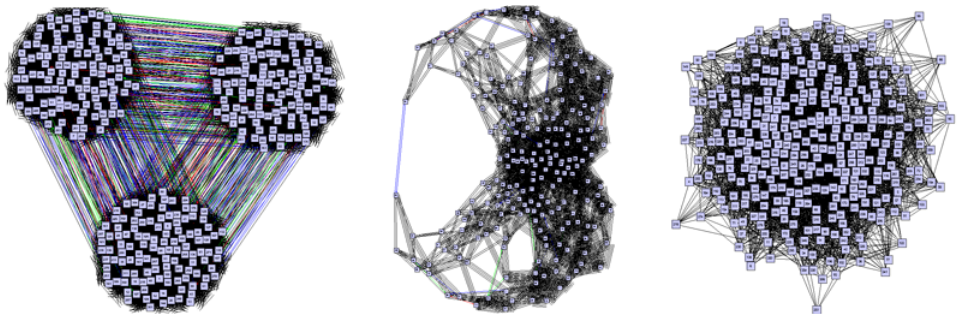

Figure 1.1: Three different Boolean formulas in constraint graph form (the nodes represent variables, the edges represent binary relationships between the variables). From left to right: a hand-crafted graph colouring problem (363 variables, 9559 clauses), a small practical scheduling problem (226 variables, 1078 clauses), a random 3-SAT problem (360 variables, 1530 clauses). Due to their structure, different classes of problems may require different solving techniques. These graphs were generated using the DPvis visualization tool (http://www.carstensinz.de).

son, many state-of-the-art solvers opt out of implementing a value ordering heuristic altogether, and always try a fixed polarity first. The work presented in this thesis attempts to further our understanding of value ordering heuristics.

SAT is arguably the most important special case of the much larger class of Constraint Satisfaction Problems (CSP). Interestingly, CSPs are solved by backtracking search similar to DPLL, and many concepts such as variable and value ordering also play a tremendous role in CSP solvers. Since the variables in CSPs can take on more than two values, value ordering heuristics are more prominent in CSP research than in SAT research. Unfortunately, there appears to be a growing rift between CSP and SAT research, and discoveries in the two fields do not always benefit each other.

For a long time, the performance of SAT solvers was evaluated solely on random formulas, as those are easy to generate and reason about. This led to the discovery of many interesting properties of random formulas such as the phase-transition phenomenon [CA93], as well as a number of promising variable and value ordering heuristics. It soon became evident, however, that SAT solvers which excelled at random problems did not always do well on real-world problems and vice-versa. Unlike random formulas, real problems such as scheduling exhibit a great deal of structure which requires a different approach. For this reason, it has become customary to evaluate SAT solvers on three separate classes of problems: randomly-generated, hand-crafted (such as puzzles and graph colouring problems), and application (real-world problems such as chip verification, cryptography and instruction scheduling). Figure 1.1 shows three Boolean formulas, each belonging to a different class, in graph form. The stark difference between the structure of each formula is apparent.

The evolution of SAT solvers is by no means over. *SAT-Race*[1], an annual SAT competition first held in 2002, has created an active and competitive environment for SAT research. Each year, participants submit new and modified solvers to SAT-Race,

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

where their performance is evaluated on a set of standard benchmarks and ranked against each other. Researchers also contribute benchmarks and technical papers which help further understanding of SAT algorithms. While SAT-Race primarily focuses on the classical SAT problem, it also provides dedicated competitions for many other flavours of Boolean Satisfiability such as Maximum Satisfiability, Minimal Unsatisfiable Subsets, and Quantified Boolean Formulas.

1.2 Contributions of the Thesis

In this thesis, we propose three novel value ordering heuristics for DPLL-based SAT solvers and demonstrate their effectiveness on several classes of SAT problems. The three heuristics determine preferred polarities by first converting the Boolean formula into a continuous optimization problem, and then applying local search techniques to approximate the location of the solution. The benefit of this approach is that there exist infinitely many ways of defining continous optimization problems that in some way correspond to the SAT instance being solved, which enables us to try and evaluate many different possible parametrizations. This approach also allows us to tap into the existing tools of multivariate calculus and a number of well-known iterative optimization techniques. Furthermore, unlike existing heuristics, our approach computes prefered polarities for *all* variables at once, which means the heuristic does not need to be evaluated at every decision node.

In order to define the optimization problems, we translate the exponentially-sized space of all assignments $\{True, False\}^n$ to the continous domain $[0, 1]^n$ corresponding to the unit hypercube. This transformation also offers a convenient geometric interpretation of the inner workings of each heuristic. The three methods presented in this thesis differ in their formulation of the optimization problem and the local search method used.

The first heuristic models the SAT problem as a system of non-linear equations crafted in a way such that the satisfying assignments of the Boolean formula (if they exist) correspond to the roots of the system. We then apply Newton's root-finding algorithm to this system to approximate the location of the solution, and use this estimate as the preferred polarity choice. We show that the resulting Jacobian matrix is sparse, which facilitates an efficient implementation.

The second heuristic represents the problem as a single objective function rather than a system of equations. This objective function, which we call a *conflict potential*, measures the number of clauses conflicting with each possible assignment. If the formula is satisfiable, there exists at least one corner of the unit hypercube where the conflict potential is zero. We show that the gradient and the second-order Hessian matrix of the conflict potential can be computed in closed form, which allows us to use the quadratically-converging Newton's Method for scalar functions. We present two possible ways of defining and implementing the conflict potential.

The third heuristic builds on the idea of conflict potentials by adding a stochastic element to the search. Instead of using a fixed parametrization, this heuristic samples a fixed number of related potentials from a distribution of objective functions, and

then lets them vote on the best gradient descent direction. The advantage of this approach is that randomly-sampled conflict potentials have different distributions of local extrema, which prevents local search from getting trapped in unfavourable regions of the search space.

We implement these heuristics in the widely-recognized MiniSAT framework [ES04], and evaluate their performance on a set of standard benchmarks. We compare the results against the fixed polarity policy, three well-known heuristics: MOMS [Pre96], Jeroslow-Wang [JW90], and phase-saving [PD07a], as well as six state-of-the-art SAT solvers. We demonstrate that our heuristics provide a dramatic boost in the accuracy of MiniSAT on random instances (up to 85% from 35% of instances solved), a modest gain in accuracy on application instances (up to 48% from 44%), and a small decrease in accuracy on crafted instances (down to as low as 39% from 43%). Unlike many other solvers, MiniSAT equipped with our heuristics performed reliably across all three benchmark categories. Our heuristics allowed MiniSAT to beat all six state-of-the-art solvers on crafted and application instances, and three solvers in the random category.

1.3 Organization of the Thesis

The thesis is organized as follows. In Chapter 2, we formally state the Boolean Satisfiability Problem, introduce DPLL and CDCL solvers, and define the problem of value ordering in backtracking search. We also present three well-known iterative optimization schemes used in this thesis: gradient descent, Newton's Root-Finding Method for systems of equations, and Newton's Method for scalar functions.

In Chapter 3, we review literature related to the problem of value ordering in SAT, and discuss known heuristics in DPLL, CDCL and lookahead solvers. We also review existing local search techniques in incomplete SAT solvers, and disucss discrepancy-based search.

In Chapter 4, we present our three value ordering heuristics based on non-linear optimization methods.

In Chapter 5, we empirically evaluate our heuristics on random, crafted and industrial benchmarks used in recent SAT-Race competitions. We discuss the choice of parameters, analyze the strengths and weaknesses of each approach, and compare the results to five other value ordering heuristics as well as six state-of-the-art solvers.

In Chapter 6, we conclude our work by highlighting its contributions and significant results. We also outline future work and potential enhancements to our approach.

Chapter 2

Background

In this chapter, we provide the necessary background in Boolean Satisfiability and non-linear optimization, and formally define the problem of value ordering as it pertains to DPLL and CDCL solvers. We also outline the features of our base solver MiniSAT, and discuss viable policies a value ordering heuristic may pursue.

2.1 Boolean Satisfiability

2.1.1 Boolean Formulas

Boolean logic is an intuitive and robust mechanism for reasoning about facts which are either $True$ or $False$. Expressions which symbolically represent relationships between Boolean variables are known as *Boolean formulas*. More precisely, a Boolean formula is a mapping of the form $F : \{True, False\}^n \rightarrow \{True, False\}$ which contains Boolean variables $v_1, v_2, ..., v_n$, brackets, and the operators $\wedge$ (logical and), $\vee$ (logical or), and $\neg$ (logical negation). Each variable can take on one of two values: $True$ or $False$. We also refer to the value of the variable as the *polarity* of the variable. The polarity is positive if the variable has the value $True$, otherwise the polarity is negative. Throughout the thesis, we use the common "bar-notation" to denote negation: $\bar{v} = \neg v$. We also use the notation $F[v_i = True]$ and $F[v_i = False]$ to denote a reduced formula; that is, the variable v_i is assigned a truth value in F.

If there exists an assignment to the variables which makes the formula evaluate to $True$, the formula is said to be *satisfiable*; otherwise, it is *unsatisfiable*. A satisfying assignment is known as a *model* of the formula. A formula which evaluates to $True$ under all assignments is known as a *tautology*. A variable that has the same polarity in all models of a satisfiable formula is known as a *backbone* variable.

Example 2.1 (Boolean Formulas)**.** *The following expressions are examples of Boolean formulas. F_1 is a satisfiable formula of three variables. F_1 has a single model: $(v_1, v_2, v_3) = (False, True, True)$. F_2 is an unsatisfiable formula. F_3 is the negation of F_2, and is therefore a tautology. $F_1[v_1 = False]$ is the reduced form of F_1 with the variable v_1 set to $False$.*

$$F_1 = \bar{v}_1 \wedge (v_1 \vee (v_2 \wedge v_3))$$
$$F_2 = (v_1 \vee v_2) \wedge (\bar{v}_1) \wedge (\bar{v}_2)$$
$$F_3 = (\bar{v}_1 \wedge \bar{v}_2) \vee (v_1) \vee (v_2)$$
$$F_1[v_1 = False] = (v_2 \wedge v_3)$$

2.1.2 Conjunctive Normal Form

Boolean formulas in *Conjunctive Normal Form* (CNF) are particularly useful for representing SAT problems. A CNF formula is a conjunction (logical and) of some *clauses*, and each clause is a disjunction (logical or) of some *literals*. A literal is a variable or its negation.

$$F = \bigwedge_{i=1}^{m} C_i \quad \text{where} \quad C_i = \bigvee_{j} l_{ij} \tag{2.1}$$

If a literal is the negation of a variable, the literal is said to be negative; otherwise, it is positive. By convention, we require that no clause contains a literal and its negation at the same time. We also recognize a special case in which a clause contains no literals; this is known as an *empty clause*. Empty clauses always evaluate to $False$. A clause containing a single literal is called a *unit clause*. A literal which has the same polarity in all clauses is called a *pure literal* (or, *monotone literal* in some literature). Throughout the thesis, we use n to denote the number of variables in a formula, and m to denote the total number of clauses.

Example 2.2 (CNF Formulas)**.** *The following expressions are examples of Boolean formulas in Conjunctive Normal Form. F_1 and F_2 each contain three clauses; F_3 contains a single clause. The clause $(\bar{v}_2)$ of F_2 is a unit clause and $\bar{v}_3$ in F_1 is a pure negative literal.*

$$F_1 = (v_1 \vee v_2) \wedge (v_1 \vee \bar{v}_2) \wedge (\bar{v}_1 \vee \bar{v}_2 \vee \bar{v}_3)$$
$$F_2 = (v_1 \vee v_2) \wedge (\bar{v}_1 \vee \bar{v}_2) \wedge (\bar{v}_2)$$
$$F_3 = (\bar{v}_2 \vee \bar{v}_3)$$

A CNF formula is satisfied if and only if all of its clauses are satisfied, which makes CNF formulas a natural way of encoding lists of necessary conditions which must hold in a given problem. Tseitin [Tse68] showed that any Boolean formula can be transformed into a corresponding CNF formula in a way which preserves the satisfiability property, while incurring at most a linear blow-up in the number of literals. This makes CNF formulas a standard choice for automated theorem proving, circuit analysis, and electronic design automation algorithms. When used as input to computer programs, CNF formulas are most commonly encoded in the `DIMACS` file format.

Example 2.3 (DIMACS File Format). *CNF formula* $F = (\bar{v}_1) \wedge (v_1 \vee v_2) \wedge (\bar{v}_1 \vee \bar{v}_2)$ *with* $n = 2$ *variables and* $m = 3$ *clauses encoded in the DIMACS file format.*

```
p cnf 2 3
-1 0
1 2 0
-1 -2 0
```

2.1.3 Problems in Satisfiability

We now define one of the most fundamental problems in Boolean logic and the main topic of this thesis.

Definition 2.1 (Boolean Satisfiability). *Given a Boolean formula* $F(v_1, v_2, ..., v_n)$*, the Boolean Satisfiability Problem (SAT) is to find an assignment to variables* $v_1, v_2, ..., v_n$ *such that* F *evaluates to* $True$*, or to determine that no such assignment exists.*

Boolean Satisfiability was the first problem shown to be NP-Complete [Coo71]. The complexity of the problem is exponential in the number of variables and, unless P=NP, a polynomial time algorithm capable of determining satisfiability of an arbitrary Boolean formula does not exist. It should be noted, however, that there exist CNF formulas which can be solved in polynomial time, most notably *2-SAT* (CNF formulas with two literals in each clause), and *Horn-SAT* (CNF formulas with at most one positive literal in each clause). These special forms are known as *tractable subclasses*. Much research has gone into identifying such special forms as well as finding *backdoor sets*—assignments to variables which reduce Boolean formulas to tractable subclasses.

Boolean Satisfiability itself is one of the most important subclasses of a much larger family of problems known as Constraint Satisfaction Problems (CSPs).

Definition 2.2 (Constraint Satisfaction Problem). *A CSP is a triple* (V, D, C) *where* V *is a set of* n *variables,* D *is a set of variable domains, and* C *is a set of* m *constraints. A solution to a CSP is an assignment to* $v_1, v_2, ..., v_n$ *where* $v_i \in D_i$ *which satisfies all the constraints.*

A classic example of a CSP is the game of *Sudoku*, which can be represented as 81 variables with domains $D_i = \{1, 2, .., 9\}$ and constraints on variables in the same rows, columns, and 3×3 sub-squares. It is readily seen that SAT is a CSP where each $D_i = \{True, False\}$. Both SAT and CSPs are powerful mechanisms for representing and solving a wide range of problems in artificial intelligence, engineering and mathematics. Since any CSP problem can be converted to an equivalent SAT instance through methods such as the *direct encoding* [DK89] and the *support encoding* [Gen02b], advances in SAT and CSPs can mutually benefit each other.

Although we focus on the classic SAT problem in this thesis, it should be noted that many other types of Satisfiability problems exist. A few notable examples are *Maximum Satisfiability* (MaxSAT) which seeks assignments satisfying as many clauses as possible, *Minimal Unsatisfiable Core* (MUC) which is the problem of extracting the smallest unsatisfiable subset of clauses, and *Quantified Boolean Formulas* (QBF) which deal with Boolean formulas involving universal and existential quantifiers.

2.2 SAT Solvers

Due to the ubiquity of SAT problems in practical applications, efficient SAT solvers are in high demand. A brute-force algorithm which iterates through all possible variable assignments, and checks whether each assignment satisfies the formula has runtime complexity $O(2^n|F|)$ where $|F|$ denotes the length of the formula. Clearly, such a technique is impractical for real-world SAT problems such as FPGA routing which involve thousands of variables and millions of clauses. Modern SAT solvers employ a wide range of techniques designed to avoid traversing the entire search space. Today's algorithms are capable of solving Boolean formulas with thousands of variables in mere minutes.

SAT solvers can belong to one of two varieties: complete and incomplete. Incomplete solvers are usually stochastic in nature, and cannot guarantee finding a solution or disproving a formula. They are, however, often used in practice because of their high efficiency on large instances. Complete solvers, if given enough time, will find a solution if one exists or disprove the formula. Although we focus on complete solvers in this thesis, we make use of several concepts used in incomplete solvers.

The input to a solver is a CNF-formula, and the output is a model if the formula is satisfiable, or `UNSATISFIABLE` if a model does not exist. Some SAT solvers are specifically geared toward producing a complete proof of unsolvability known as a *certificate*.

2.2.1 Resolution

The history of modern SAT solvers begins with the Davis-Putnam procedure (DPP) [DP60] which was primarily based on the rule of *resolution*.

Definition 2.3 (Resolution)**.** *In the context of SAT solvers, the rule of resolution is defined as follows. Given a CNF formula with clauses* $(v \vee A) \wedge (\bar{v} \vee B)$ *where* A *and* B *are (possibly empty) disjunctions of literals, we can* resolve *on variable* v *to produce the* resolvant *clause* $(A \vee B)$*.*

It is easy to see why this rule holds. If $v = True$, the satisfiability of the formula is determined completely by B. If $v = False$, the satisfiability of the formula is determined completely by A. In either case, $(A \vee B)$ alone determines the satisfiability of the formula, and the value of v is irrelevant. Therefore, resolution provides a method for systematically eliminating variables from a CNF formula. The key step of the DPP algorithm does just that: for each variable v, the solver forms the resolvants, and then deletes the original clauses containing v and $\bar{v}$. A satisfiable formula can be completely reduced by DPP, while an unsatisfiable formula will eventually resolve into at least one empty clause.

The downside of resolution is that forming all resolvants for each pair v and $\bar{v}$ significantly increases the length of the formula during the variable elimination process. In other words, the formula blows up before it shrinks. The complexity of this blowup has been studied in depth. A lower bound was given by Tseitin [Tse68], and was later improved by Galil [Gal77], and Haken [Hak85]. All research points to

the unfortunate fact that in the worst case, the blowup is exponential in the number of variables, which makes resolution completely impractical for solving large SAT instances. Earliest implementations of DPP could only handle short formulas with a handful of variables, especially since RAM was very expensive in those days.

While modern SAT solvers no longer use resolution in its pure form as the primary solving technique, it still plays a prominent role in many helper algorithms such as formula preprocessing and simplification, clause learning and local search. Some recent work has been done to integrate resolution back into the main SAT solving process [ZHG11].

2.2.2 DPLL

Since resolution has an exponential memory requirement, it is prohibitively expensive for solving large practical formulas. This limitation was lifted by Loveland and Logemann [DLL62] who modified DPP into what is now known as the Davis-Putnam-Logemann-Loveland (DPLL) algorithm. Unlike its predecessor, DPLL uses a linear amount of memory, and is only limited by CPU time. DPLL forms the basis for the vast majority of modern complete SAT solvers; it is also the algorithm we focus on in this thesis.

The idea behind DPLL is based on resolution, but rather than forming the resolvants explicitly, the algorithm employs the so-called *splitting rule*. At each step, DPLL chooses an unassigned variable and assigns it to either $True$ or $False$. This step is called *branching* on a *decision literal*. If all clauses are satisfied by the assignment, the algorithm has found a model and terminates. If on the other hand, the assignment violates at least one clause, the algorithm is said to have encountered a *conflict* and *backtracks* one level to try the opposite polarity of the decision literal. If DPLL exhausts all branches, the formula is determined to be unsatisfiable. DPLL therefore treats the search space as a binary tree which is systematically explored in a depth-first manner. Because complete resolvants are not generated, the formula always gets shorter when a decision variable is assigned; this guarantees a linear bound on the amount of memory required. A simple illustration of the DPLL search process is shown in Figure 2.1.

Unlike the brute force algorithm, DPLL does not wait until all variables are assigned before the clauses are checked for consistency; instead, each branch decision is immediately checked against all clauses, which allows the algorithm to detect conflicts early on. The checking step is efficiently implemented using *watch lists* (mappings of variables to the clauses they appear in). The pseudocode for DPLL is given in Listing (2.1). `UnitPropagate(F, c)` assigns the literal of unit clause c in a way that satisfies c. In doing so, other clauses may become unit, and the procedure is repeated until no more unit clauses remain. `AssignPureLiteral(F, l)` simplifies the formula by assigning a pure literal l to a satisfying value and deleting all satisfied clauses from the formula. `ChooseNextLiteral(F)` is responsible for choosing the next branch variable and its polarity.

It is known that the choice of the decision variable and its polarity has a tremendous impact on the efficiency of DPLL. Well-made decisions guide the algorithm

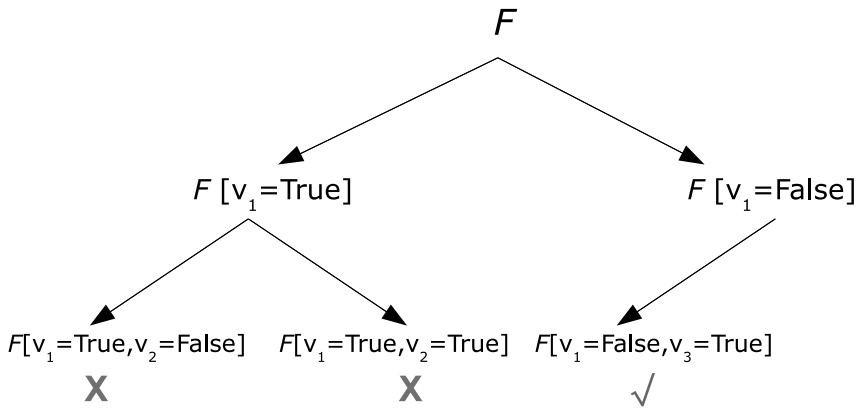

Figure 2.1: A simple example illustrating the backtracking DPLL algorithm. The search begins with a CNF formula F which depends on Boolean variables v_1, v_2, v_3. DPLL selects a variable (this decision is known as variable ordering) and a polarity (this decision is known as value ordering), and performs the splitting rule by assigning the chosen variable to the chosen polarity. In this example, DPLL chose v_1 and $True$. This produces a shorter formula $F[v_1 = True]$ and the process is recursively repeated. The next decision ($v_2 = False$) leads to a conflict, and DPLL backtracks one level to try $v_2 = True$. Eventually, DPLL encounters the reduced formula $F[v_1 = False, v_3 = True]$ in which all clauses are satisfied. At this point, DPLL has found a satisfying assignment and the search ends.

towards promising areas of the search space early on which can lead to exponential savings in the amount of search effort. Perhaps unsurprisingly, choosing the optimal decision variable is NP-hard, which was shown by Liberatore [Lib00]. Similarly, the task of choosing the optimal polarity is also in NP (otherwise, any satisfiable formula could be solved in polynomial time by sequentially assigning each of the n variables). For these reasons, sophisticated *variable ordering* and *value ordering* heuristics have become central to the design of modern SAT solvers. While variable ordering heuristics have attracted a considerable amount of interest, value ordering heuristics have drawn less attention. In this thesis we focus on the value ordering process.

2.2.3 CDCL

Much research has gone into improving the performance of the basic DPLL algorithm. One of the most successful variants of DPLL is the class of so-called Conflict-Driven Clause-Learning (CDCL) algorithms. In CDCL, some or all of the encountered conflicts are analyzed and recorded in the form of new clauses. While these new clauses are "redundant" in the sense that they are implied by original formula, their structure may aid the solver in pruning the search space more efficiently. Furthermore, conflict analysis allows CDCL solvers to backtrack multiple levels at once, which also significantly speeds up the search process. This technique, known as *non-chronological backtracking* or *backjumping*, was popularized by early efficient SAT solvers *ZChaff* [MMZ+01], and *berkmin* [NG02]. CDCL solvers also use resolution to simplify learned clauses, and use information collected from conflicts to drive various aspects of the

Algorithm 2.1: DPLL(F)

input : A CNF formula F
output: $True$ if F is satisfiable; $False$ otherwise.

if all clauses of F are satisfied **then**
  return $True$
else if F contains an empty clause **then**
  return $False$
foreach unit clause c in F **do**
  F $\leftarrow$ `UnitPropagate(`F, c`)`
foreach pure literal l in F **do**
  F $\leftarrow$ `AssignPureLiteral(`F, l`)`
$l \leftarrow$ `ChooseNextLiteral(`F`)`
return `DPLL(`$F \wedge l$`)` or `DPLL(`$F \wedge \bar{l}$`)`

search process such as variable ordering and restarts.

We implemented and tested our heuristics in the *MiniSAT 2.2* framework. MiniSAT[1] [ES04] is a well-documented and robust open-source DPLL-CDCL solver developed by Niklas Eén and Niklas Sörensson. MiniSAT placed first in the industrial category in SAT-Race 2005, and is widely-recognized as a solid framework for SAT research. Many state-of-the-art SAT solvers such as *GlueMiniSAT* and *CryptoMiniSAT* were derived from the MiniSAT core. We now briefly outline several of MiniSAT's features which are common to many modern CDCL solvers:

- **Clause Learning and Subsumption**: MiniSAT analyzes and stores all conflicts in the form of new clauses. Whenever possible, multiple learned clauses are combined and replaced with shorter clauses through *clause subsumption* based on resolution. For example, two learned clauses $(a \vee b \vee c)$ and $(a \vee b \vee \bar{c})$ can be safely replaced with $(a \vee b)$ by resolving on c. Learned clauses aid the search process, but also slow down branching because more conflicts need to be checked at every decision node. To curb the number of recorded clauses, MiniSAT places a limit on the maximum number of learned clauses. This limit is raised at a user-defined rate during the search. MiniSAT also regularly purges a fixed fraction of all learned clauses to prevent memory overflow.
- **Restarts**: It has been empirically shown that aggressively restarting the search process improves solver performance on hard problems [Hua07]. Restarts are designed to prevent the solver from getting trapped in unfavourable regions of the search space. After restarting, a different sequence of decisions is made. MiniSAT restarts the search after encountering a certain number of conflicts. The conflict limit is progressively raised in a special form of the geometric progression known as the *Luby Sequence* [LSZ93], which has been shown to be effective on a wide range of problems.

[1] http://minisat.se

0: 1
1: 1 1 2
2: 1 1 2 1 1 2 4
3: 1 1 2 1 1 2 4 1 1 2 1 1 2 4 8
...

The Luby sequence ($\times 100$) defines restart limits in MiniSAT.

- **Activity-based Variable Ordering**: MiniSAT implements the popular Variable State Independent, Decaying Sum (VSIDS) heuristic for choosing the next variable to branch on. This heuristic, popularized by the solver *Chaff* [MMZ+01], gives preference to variables which occur in more conflicts during search. Whenever a conflict is encountered, variables belonging to the conflicting clauses are "bumped up" in the priority heap and then gradually decay with time. The variable with the highest activity score is then selected. The intuition is that branching on such variables places more restrictions on the search space.
- **Phase Saving**: Whenever the algorithm backtracks from a conflict, the polarities of all variables are cached, and then re-used in future branch decisions. The intuition is that polarities which have worked in the past may also work in the future. This mechanism was first proposed by Ginsberg [Gin93] in the context of CSP solvers, and was later adapted to SAT solvers by Pipatsrisawat and Darwiche [PD07a].
- **Formula Pre-Processor**: As of version 2.2, MiniSAT integrates a CNF formula pre-processor called *SatELite*. This pre-processor attempts to simplify the input formula through subsumption, self-subsuming resolution and variable elimination before passing it to the main search algorithm. SatELite has demonstrated a greater than 50% reduction in the number of literals on many industrial problems [EB05]. Most modern SAT solvers pre-process their inputs in a similar manner.

2.2.4 Value Ordering

We now discuss the central topic of our work: value ordering heuristics in DPLL-CDCL solvers.

Definition 2.4 (Value Ordering in SAT)**.** *Value ordering is the problem of deciding which polarity of the decision variable should be tried first. The optimal choice is one which results in the fewest number of nodes explored in the DPLL search tree.*

Although we seek a heuristic which minimizes the size of the search tree, we must recognize that if the computational cost of the decision outweighs the benefits of the reduced search tree, a value ordering heuristic is useless in practice. Balancing the cost and the reward of heuristics is notoriously difficult, which has led many SAT solvers to not implement a value ordering heuristic at all (for example, MiniSAT always branches on $False$). Heuristics can be *static* (that is, computed once before the search begins), or *dynamic* (computed at all, or some decision nodes during the search process). Ideally, we seek a heuristic that is as accurate and as efficient

as possible, which is a challenging task, because finding an optimal heuristic is as difficult as solving the original SAT problem itself.

In theory, value ordering can be significantly more powerful than variable ordering: an ideal value ordering heuristic always leads to the model of a satisfiable SAT problem without any backtracking at all; this result does not hold for variable ordering heuristics.

When enumerating all solutions in DPLL search, the choice of polarity does not matter because all subtrees must be explored independently. By the same argument, the choice of polarity is irrelevant for unsatisfiable formulas. In CDCL search, however, the choice of polarity always matters, because different decisions lead to different learned clauses which directly affect future search space exploration.

There are several objectives a value ordering heuristic may pursue:

- **Promise**: The heuristic may choose polarities which appear to lead to a solution faster. This involves estimating the number of solutions in each subtree at the decision variable. If the formula is unsatisfiable, this method may be detrimental to the overall performance.

- **Fail-First**: Alternatively, the heuristic may pursue polarities which appear to lead to a conflict as soon as possible. This allows DPLL to quickly eliminate parts of the search space and enables CDCL to learn shorter conflict clauses which may be beneficial in future.

- **Formula Reduction**: The heuristic may pursue polarities which simplify the formula as much as possible. In doing so, the heuristic attempts to reach *either* a conflict or a solution in the fewest number of decisions. This is usually accomplished by a greedy choice (for example, the polarity which falsifies more clauses).

- **Better Learned Clauses**: The heuristic may pursue polarities which end up generating "better" learned clauses. Good learned clauses help decrease the total number of nodes explored in the future, however defining what "good" means is difficult, and is up to the algorithm (for example, shorter clauses or clauses containing more active variables).

- **Other Objectives**: The heuristic may choose polarities which accomplish some other objective that leads to a reduction in search effort. These may include polarities which allow backjumping farther, identify backbone variables, or discover backdoor sets.

2.3 Nonlinear Optimization

The heuristics presented in this thesis rely heavily on iterative methods for optimization of non-linear functions. In practice, most non-linear equations cannot be solved analytically and one must fall back on approximate methods and iterative local search

schemes. We review two first-order optimization schemes: gradient descent and Newton's Method for systems of equations and a second-order formulation of Newton's Method for scalar equations.

2.3.1 Gradient Descent

A common task in computer science and other fields is minimizing some cost function $f(\boldsymbol{x})$, where $\boldsymbol{x} = (x_1, x_2, ..., x_n)^T$ is a vector of n continuous variables[2]. Alternatively, we may seek to maximize some reward function. In either case, our goal is to *optimize* some given *objective function*. Without loss of generality, we focus on minimization techniques, as any maximization problem can be turned into a minimization problem by multiplying the objective function by -1. In a minimization problem, we seek an optimal parameter $\boldsymbol{x}^*$ such that,

$$\boldsymbol{x}^* = \arg\min_{\boldsymbol{x}} f(\boldsymbol{x})$$

Finding the global minimum of the objective function is usually difficult, and one has to rely on *local search* methods to obtain an approximate answer. Gradient descent (also known as *steepest descent*) is one of the simplest and best-known methods for local minimization on continuous and differentiable functions. This method is based on a simple observation that at any given point $\boldsymbol{x}$, a differentiable function increases the fastest in the direction of its gradient $\nabla f(\boldsymbol{x}) = (\frac{\partial f}{\partial x_1}, \frac{\partial f}{\partial x_2}, ..., \frac{\partial f}{\partial x_n})^T$. Steepest descent begins with an initial guess $\boldsymbol{x}_0$, and iteratively refines it by going *against* the gradient in order to locally minimize the objective function as much as possible at each step. Typically, the gradient is multiplied by a small step size $\lambda \leq 1$ to avoid "overshooting" a potential solution. The algorithm repeats until it exceeds the maximum allowed number of iterations, or satisfies some user-defined stopping criteria (for example, the gradient reaching a near-zero magnitude). In our heuristics, we do not define any stopping criteria other than the maximum iteration limit. The primary motivation behind this decision is the large computational cost associated with computing the gradient of the objective function. As explained in Chapter 4, the objective functions used in our approach often contain hundreds of thousands of terms depending on the Boolean formula being solved. In our preliminary tests, we have observed that the cost of computing a stopping condition typically outweighs its benefits. For the same reason, we chose not to implement any algorithm for dynamically adjusting the step size (such as line search).

The pseudocode for the algorithm is given in Listing 2.2.

Like all local search methods, gradient descent does not guarantee finding the optimal assignment $\boldsymbol{x}^*$. If the objective function is highly non-linear, or if the initial guess $\boldsymbol{x}_0$ is chosen poorly, the search may easily get trapped in a *local minimum*. This method may also "overshoot" a solution if the step size λ is too big.

The main advantages of gradient descent are that it has few parameters and is extremely efficient when the partial derivatives $\frac{\partial f}{\partial x_i}$ are known in closed form. In

[2]In this section, we treat all vectors as column vectors to make equations more readable.

Algorithm 2.2: Gradient Descent($f(\boldsymbol{x}), \boldsymbol{x}_0$)

input : Objective function $f(\boldsymbol{x})$, an initial guess $\boldsymbol{x}_0$
output : $\boldsymbol{x}$ which locally minimizes $f(\boldsymbol{x})$
parameters: Step size λ and the maximum number of iterations max_iterations

$\boldsymbol{x} \leftarrow \boldsymbol{x}_0$
for *iteration* $\leftarrow 1$ **to** max_iterations **do**
  $\boldsymbol{x} \leftarrow \boldsymbol{x} - \lambda \nabla f(\boldsymbol{x})$

return $\boldsymbol{x}$

practice, this is not always the case, and one has to approximate the partial derivatives using techniques such as *forward differencing*. Fortunately, as we show in Chapter 4, the partial derivatives used in our heuristics can be computed in closed form, which allows us to use gradient descent in its original formulation.

2.3.2 Newton's Method For Systems of Equations

Newton's Method is an extension of the gradient descent method for finding roots of a system of coupled equations $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), ..., f_m(\boldsymbol{x}))^T$. Specifically, it aims to find an assignment to variables $\boldsymbol{x} = (x_1, x_2, ..., x_n)^T$ such that $f_i(\boldsymbol{x}) = 0$ for all $i = 1..m$ by iteratively refining an initial guess $\boldsymbol{x}_0$. As with gradient descent, we require each f_i to be continuous and differentiable on the domain of local search.

To obtain the iterative scheme, we first perform a first-order Taylor series expansion of $\boldsymbol{f}$ around $\boldsymbol{x}_i$:

$$\boldsymbol{f}(\boldsymbol{x}_{i+1}) \approx \boldsymbol{f}(\boldsymbol{x}_i) + J(\boldsymbol{f})(\boldsymbol{x}_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)$$

where $J(\boldsymbol{f})(\boldsymbol{x}_i)$ is the $m \times n$ Jacobian matrix of first-order partial derivatives of each objective function in the system of equations:

$$J(\boldsymbol{f})(\boldsymbol{x}_i) = \begin{pmatrix} \frac{\partial f_1}{\partial x_1}(\boldsymbol{x}_i) & \frac{\partial f_1}{\partial x_2}(\boldsymbol{x}_i) & \cdots & \frac{\partial f_1}{\partial x_n}(\boldsymbol{x}_i) \\ \frac{\partial f_2}{\partial x_1}(\boldsymbol{x}_i) & \frac{\partial f_2}{\partial x_2}(\boldsymbol{x}_i) & \cdots & \frac{\partial f_2}{\partial x_n}(\boldsymbol{x}_i) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial f_m}{\partial x_1}(\boldsymbol{x}_i) & \frac{\partial f_m}{\partial x_2}(\boldsymbol{x}_i) & \cdots & \frac{\partial f_m}{\partial x_n}(\boldsymbol{x}_i) \end{pmatrix} \tag{2.2}$$

To find the roots of the system, we set $\boldsymbol{f}(\boldsymbol{x}_{i+1}) = \boldsymbol{0}$, where $\boldsymbol{0}$ is a column vector of zeros, and solve the system for $\boldsymbol{x}_{i+1}$ to obtain the update step:

$$\boldsymbol{0} = \boldsymbol{f}(\boldsymbol{x}_i) + J(\boldsymbol{f})(\boldsymbol{x}_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)$$

$$\mathbf{0} = [J(\boldsymbol{f})(\boldsymbol{x}_i)]^{-1}\boldsymbol{f}(\boldsymbol{x}_i) + (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)$$

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - [J(\boldsymbol{f})(\boldsymbol{x}_i)]^{-1}$$

In practice, inverting the Jacobian is computationally expensive and error-prone. It is more efficient to rewrite the above equations in terms of an optimization step $\Delta\boldsymbol{x} \equiv \boldsymbol{x}_{i+1} - \boldsymbol{x}_i$ as follows:

$$\mathbf{0} = \boldsymbol{f}(\boldsymbol{x}_i) + J(\boldsymbol{f})(\boldsymbol{x}_i)\Delta\boldsymbol{x}$$
$$J(\boldsymbol{f})(\boldsymbol{x}_i)\Delta\boldsymbol{x} = -\boldsymbol{f}(\boldsymbol{x}_i)$$

The resulting linear system of equations can be solved for $\Delta\boldsymbol{x}$ using a robust algorithm such as Gaussian elimination, LU-decomposition or QR-factorization. Once $\Delta\boldsymbol{x}$ is found, it is used to update the current estimate $\boldsymbol{x}_i$. As with gradient descent, $\Delta\boldsymbol{x}$ is usually multiplied by a small step size $\lambda \leq 1$ to avoid overshooting a solution. The algorithm repeats for a fixed number of iterations, gradually refining the initial guess. Additional stopping criteria can be added to the algorithm if one wishes to avoid executing unnecessary iterations. For example, one could stop the search if a root of the system has been found ($\boldsymbol{f}(\boldsymbol{x}_i) = \mathbf{0}$), or if the current estimate $\boldsymbol{x}_i$ is sufficiently close to a root of the system (the magnitude of each component of $\boldsymbol{f}(\boldsymbol{x}_i)$ is within some small user-defined threshold ϵ). However, in the context of our heuristics, we observed that this algorithm virtually never finds the roots on its own (unless used on small toy instances). This observation, together with the high cost of computing $\boldsymbol{f}(\boldsymbol{x}_i)$ at each iteration, led us to drop any stopping criteria (other than the limit on the number of iterations) from our implementation. The pseudocode for the algorithm is given in Listing 2.3.

Algorithm 2.3: Newton's Method For Systems of Equations($\boldsymbol{f}(\boldsymbol{x}), \boldsymbol{x}_0$)

input : A system of equations $\boldsymbol{f}(\boldsymbol{x})$ and an initial guess $\boldsymbol{x}_0$
output : $\boldsymbol{x}$ which locally approximates $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{0}$
parameters: Step size λ and the maximum number of iterations max_iterations

$\boldsymbol{x} \leftarrow \boldsymbol{x}_0$
for *iteration* $\leftarrow 1$ **to** max_iterations **do**
  Solve $J(\boldsymbol{f})(\boldsymbol{x})\Delta\boldsymbol{x} = -\boldsymbol{f}(\boldsymbol{x})$ for $\Delta\boldsymbol{x}$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \lambda\Delta\boldsymbol{x}$

return $\boldsymbol{x}$

Newton's Method suffers from the same limitations as basic gradient descent: it is sensitive to the initial guess $\boldsymbol{x}_0$ and the "smoothness" of the objective functions f_i. If the objective functions are highly non-linear, the Jacobian J may easily become near-singular and computationally unstable, at which point the approximation to $\boldsymbol{x}_i$ quickly diverges.

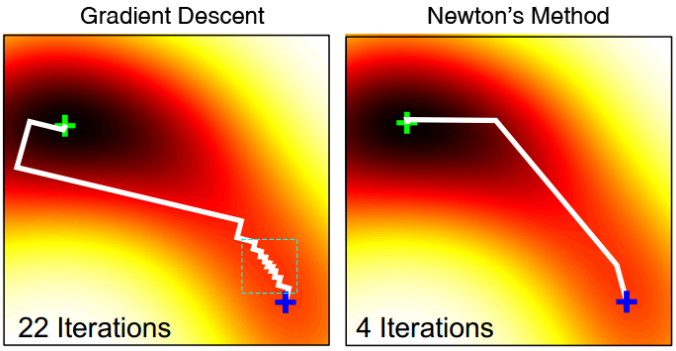

Figure 2.2: Comparison of gradient descent and Newton's Method when used to minimize the same objective function. The starting point of the search is in the lower right corner. The darker areas of the graph correspond to lower values of the objective function. Newton's Method converges in only 4 iterations, while gradient descent converges in 22 iterations due to excessive "zig-zag" steps caused by local linearization of the objective function. Diagram by Simon J.D. Prince [PRI12].

2.4 Newton's Method for Scalar Functions

Newton's Method can also be used to optimize scalar functions rather than systems of equations. In this formulation (sometimes referred to as the Newton-Raphson Method), this local search technique is similar to gradient descent, but converges much faster in practice. Whereas gradient descent uses a first-order Taylor expansion of the objective function, Newton's Method uses a second-order expansion, and locally approximates the objective function as a quadratic polynomial.

One major weakness of gradient descent is that it often ends up approaching a local minimum in a "zig-zag" pattern (see Figure 2.2), which requires many small steps to converge. Newton's Method, on the other hand, uses the curvature information of the objective function (obtained from the second-order partial derivatives) to take a more direct route to its target. The convergence rate is provably quadratic; if the objective function is a quadratic polynomial itself, the method converges in a single step (assuming step size $\lambda = 1$). The tradeoff between the two methods is that Newton's Method is more computationally expensive per step, as it requires solving a linear system of equations at each iteration.

We once again consider an objective function $f(\boldsymbol{x})$, where $\boldsymbol{x}$ is the $n \times 1$ column vector $(x_1, x_2, ..., x_n)^T$. To obtain the iterative scheme, we consider the second-order Taylor expansion of the objective function $f(\boldsymbol{x})$ around an initial guess $\boldsymbol{x}_i$,

$$f(\boldsymbol{x}_{i+1}) \approx f(\boldsymbol{x}_i) + \nabla f(\boldsymbol{x}_i)^T(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) + \frac{1}{2}(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)^T H(f)(\boldsymbol{x}_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i),$$

where $\nabla f(\boldsymbol{x}_i)$ is the gradient ($n \times 1$ column vector of first-order partial derivatives of f) evaluated at the point $\boldsymbol{x}_i$, and $H(f)(\boldsymbol{x}_i)$ is the Hessian matrix of second order partial derivatives of f evaluated at the point $\boldsymbol{x}_i$:

$$H(f)(\boldsymbol{x}_i) = \begin{pmatrix} \frac{\partial^2 f}{\partial x_1^2}(\boldsymbol{x}_i) & \frac{\partial^2 f}{\partial x_1 \partial x_2}(\boldsymbol{x}_i) & \cdots & \frac{\partial^2 f}{\partial x_1 \partial x_n}(\boldsymbol{x}_i) \\ \frac{\partial^2 f}{\partial x_2 \partial x_1}(\boldsymbol{x}_i) & \frac{\partial^2 f}{\partial x_2^2}(\boldsymbol{x}_i) & \cdots & \frac{\partial^2 f}{\partial x_2 \partial x_n}(\boldsymbol{x}_i) \\ \vdots & \vdots & \ddots & \vdots \\ \frac{\partial^2 f}{\partial x_n \partial x_1}(\boldsymbol{x}_i) & \frac{\partial^2 f}{\partial x_n \partial x_2}(\boldsymbol{x}_i) & \cdots & \frac{\partial^2 f}{\partial x_n^2}(\boldsymbol{x}_i) \end{pmatrix} \tag{2.3}$$

At any local minimum of a quadratic polynomial, the gradient vanishes. To find the local minimum of the quadratic approximation of $f(\boldsymbol{x})$, we take the gradient with resepect to $\boldsymbol{x}_{i+1}$, set it equal to $\mathbf{0}$, and solve for $\boldsymbol{x}_{i+1}$:

$$\nabla f(\boldsymbol{x}_{i+1}) \approx \nabla f(\boldsymbol{x}_i) + \frac{1}{2} H(f)(\boldsymbol{x}_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i) + \frac{1}{2} H(f)(\boldsymbol{x}_i)^T (\boldsymbol{x}_{i+1} - \boldsymbol{x}_i),$$

which can be simplified further since the Hessian matrix is symmetrical for any twice-differentiable function (as is the case here):

$$\nabla f(\boldsymbol{x}_{i+1}) \approx \nabla f(\boldsymbol{x}_i) + H(f)(\boldsymbol{x}_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)$$

$$\mathbf{0} = \nabla f(\boldsymbol{x}_i) + H(f)(\boldsymbol{x}_i)(\boldsymbol{x}_{i+1} - \boldsymbol{x}_i)$$

$$\boldsymbol{x}_{i+1} = \boldsymbol{x}_i - [H(f)(\boldsymbol{x}_i)]^{-1} \nabla f(\boldsymbol{x}_i)$$

Inverting the Hessian matrix is a costly process in practice. For this reason, it is more convenient to rewrite the above equation in terms of the incremental step $\Delta\boldsymbol{x} \equiv \boldsymbol{x}_{i+1} - \boldsymbol{x}_i$. This incremental step is known as the *Newton direction*, and is equal to

$$\Delta\boldsymbol{x} = [H(f)(\boldsymbol{x}_i)]^{-1} \nabla f(\boldsymbol{x}_i)$$

The Newton direction can be obtained (or approximated) by solving the following system of equations using any robust algorithm for linear systems.

$$H(f)(\boldsymbol{x}_i)\Delta\boldsymbol{x} = \nabla f(\boldsymbol{x}_i)$$

As with the other two optimization methods, the Newton direction is multiplied by a small step size $\lambda \leq 1$. The pseudocode for the algorithm is given in Listing 2.4.

In practice, the Hessian matrix is rarely available in closed form, and needs to be approximated by so-called *Quasi-Newton Methods* such as the Davidon-Fletcher-Powell (DFP) and Broyden-Fletcher-Goldfarb-Shanno (BFGS) algorithms. Fortunately, as we will show in Chapter 4, the Hessian matrix used in our heuristics can be easily computed in closed form which allows us to use Newton's Method directly.

Algorithm 2.4: Newton's Method For Scalar Functions($f(\boldsymbol{x}), \boldsymbol{x}_0$)

input : Objective function $f(\boldsymbol{x})$ and an initial guess $\boldsymbol{x}_0$
output : $\boldsymbol{x}$ which locally minimizes $f(\boldsymbol{x})$
parameters: Step size λ and the maximum number of iterations max_iterations

$\boldsymbol{x} \leftarrow \boldsymbol{x}_0$
for *iteration* $\leftarrow 1$ **to** max_iterations **do**
    Solve $H(f)(\boldsymbol{x})\Delta\boldsymbol{x} = \nabla f(\boldsymbol{x})$ for $\Delta\boldsymbol{x}$
    $\boldsymbol{x} \leftarrow \boldsymbol{x} + \lambda\Delta\boldsymbol{x}$

return $\boldsymbol{x}$

2.5 Summary

In this chapter, we have reviewed Boolean formulas, backtracking SAT solvers, and formally defined the problem of value ordering in DPLL-CDCL search. We have also outlined three methods for non-linear optimization which will serve as the basis for our value ordering heuristics. In the next chapter, we review literature related to value ordering in DPLL, CDCL and lookahead solvers, and discuss local search and discrepancy based search methods.

Chapter 3

Related Work

In this chapter, we review literature relevant to the problem of value-ordering in Boolean Satisfiability. We discuss known static and dynamic heuristics used in DPLL and CDCL search, explain the intuition behind them and comment on their effectiveness. We also briefly touch on local search methods in incomplete SAT solvers, as well as discrepancy based search.

3.1 Simple Ordering Policies

As mentioned in the previous chapter, SAT value-ordering heuristics are less widely used than variable-ordering heuristics. The amount of research in this area has been on a decline since the late 1990's and it seems that no consensus has been reached on what a "good" generic value-ordering heuristic should be. Earlier work focused on identifying polarities which simplified the formula the most, while more recent heuristics are based on lookahead methods, problem relaxation and learning.

Many solvers opt out of implementing a value-ordering heuristic altogether. For example, our base framework MiniSAT and many other state-of-the-art solvers resort to fixed branching schemes. Earlier versions of MiniSAT allowed the user to specify one of three static branching behaviours: $\{False, True, Random\}$, however newer versions have eliminated this option and always branch on $False$ first. According to MiniSAT's developers [MVVW06], this change is due to the fact that branching on $False$ is empirically better on most standard benchmarks. This is likely a by-product of the instance encoders (algorithms which translate problems into a SAT formulation).

A slightly more advanced scheme is used in RSAT [PD07b]. This solver always tries the last cached polarity of a variable first (initially all variables are set to $False$). The intuition is that if a given polarity was successful before, it may be successful in another part of the search space. This concept has proved to be effective on a range of problems and was adopted by many SAT and CSP solvers (for example, the *phase-saving* feature in MiniSAT).

3.2 Literal Ordering

In SAT, value-ordering heuristics rarely appear on their own; rather, they are bundled into *literal-ordering* heuristics which pick a decision variable *and* a corresponding value at the same time. Literal-ordering heuristics typically begin by computing two scores $h(v)$ and $h(\bar{v})$ for each variable v; then, the two scores are used to rank the polarity, and their combination $H(h(v), h(\bar{v}))$ is used to rank the variable.

The best known literal selection heuristics are *Jeroslow-Wang*, *MOMS* and *BOHM*. All three are based on the notion that literals which appear in shorter clauses are preferable because branching on them is more likely (at least in the short term) to lead to unit propagations, formula simplification and ultimately, a conflict or a solution. These heuristics have a low computational cost which allows them to be used dynamically.

The Jeroslow-Wang heuristic [JW90] is a popular scheme that favours literals in shorter clauses. For a literal l, the score $h(l)$ is defined as,

$$h(l) = \sum_{i:\, l \in C_i} 2^{-|C_i|},$$

where $|C_i|$ denotes the number of literals in clause C_i and the sum is over all clauses C_i which contain the literal l. If $h(l) > h(\bar{l})$, the heuristic branches on l, otherwise it branches on $\bar{l}$. The intuition is as follows: since there are 2^n possible variable assignments in the search space, a clause of length $|C_i|$ eliminates exactly $2^{n-|C_i|}$ assignments. For example, in a three-variable formula $F(a, b, c)$ the clause (a) eliminates 4 out of 8 possible assignments, while the clause $(a \vee b)$ only eliminates 2. The heuristic then greedily chooses the literal with the highest sum of potential reductions over all clauses. Jeroslow-Wang is used in the `PicoSAT` solver where it has demonstrated performance gains over other value ordering heuristics [Bie08]. The idea of scaling clauses by powers of 2 was first introduced by Johnson [Joh73] in the context of MAX-SAT.

Several variants of Jeroslow-Wang have been studied. Hooker and Vinay [HV95] propose the *Two-Sided Jeroslow-Wang* heuristic, which first selects the variable with largest $h(l)+h(\bar{l})$ and then branches on l if $h(l) > h(\bar{l})$ and on $\bar{l}$ otherwise. Van Gelder and Tsuji [VGTUoC95] use the same approach in their *DSJ* algorithm, but select the variable which maximizes the *product* $h(l) \cdot h(\bar{l})$. These modifications are designed to give additional preference to variables that occur frequently, and are therefore more likely to reduce the formula in the short term.

Pretolani's [Pre96] MOMS heuristic is an acronym for "Maximum Occurrences on clauses of Minimum Size". As the name suggests, MOMS selects a literal which occurs most frequently in unsatisfied clauses that have the shortest length in the formula. Pretolani examines MOMS on a range of random problems and concludes that the heuristic performs best when the formula contains a high number of binary (two-literal) clauses and when the polarity of the decision literal is chosen in a way that *falsifies* as many clauses as possible (fail-first policy). This observation was reported for both satisfiable and unsatisfiable instances.

The BOHM heuristic [BB92] can be thought of as a more informed version of MOMS. BOHM also prefers literals that occur in short clauses, but takes into account clauses of all sizes rather than just the clauses of minimum length. The algorithm selects variable v having the maximal vector $(H_1(v), H_2(v), ..., H_s(v))$ under the lexicographic order, where s is the length of the longest clause, and H_i is defined as,

$$H_i(v) = \alpha\ max(h_i(v), h_i(\bar{v})) + \beta\ min(h_i(v), h_i(\bar{v})),$$

where $h_i(v)$ is the number of clauses of length i which contain v. The authors use weights $\alpha = 1$ and $\beta = 2$. Since $H_i(v) = H_i(\bar{v})$, the polarity of the decision variable is then selected as follows: v is chosen over $\bar{v}$ if,

$$\sum_i h_i(v) > \sum_i h_i(\bar{v}).$$

Many variations of MOMS and BOHM exist, and both have shown to be successful on random and structured problems (for example, see the study conducted by Freeman [Fre95]).

Two simple heuristics based on literal counting are proposed by Marques-Silva [Ms99]. Let C_p and C_n be the number of unsatisfied clauses containing v and $\bar{v}$ respectively. *Dynamic Largest Combined Sum* (DLCS) first selects a variable for which $C_p + C_n$ is maximum, and then branches on v if $C_p \geq C_n$, and on $\bar{v}$ otherwise. *Dynamic Largest Individual Sum* (DLIS) is identical to DLCS, except that the variable is selected based on the largest value of C_p and C_n, rather than their sum. Both heuristics prefer literals which occur more frequently, in hope of reducing the formula quickly. DLCS and DLIS were successfully used in the `GRASP` solver. The author also presents a study of several well-known literal-ordering heuristics and notes that heuristics like MOMS can be overly-greedy and are often outperformed by random polarity selection.

A very different approach to literal selection is presented by Bruni and Sassano [BS01]. Their *Clause Hardness Adaptive Evaluation* heuristic chooses literals in a way that satisfies "hard" clauses first. The "hardness" of a clause C_i is defined as,

$$H(C_i) = \frac{s_i + p \times f_i}{|C_i|},$$

where s_i the number of times the clause has been encountered during search, f_i is the number of times C_i failed, and p is an empirically-determined clause penalty. At each step, the clause with highest "hardness" estimate H is chosen, and its literals are instantiated in a way to satisfy it. Intuitively, short clauses which are visited often and fail frequently are hard to satisfy, and thus place more restrictions on the literals contained in them. This heuristic was specifically designed towards unsatisfiable instances, and was successful at extracting unsatisfiable cores from the DIMACS benchmark set and a data collecting problem.

Hsu et al. [HMBM08] achieved some success in identifying the polarity of backbone variables (variables which have the same polarity in all solutions) by probabilistically

inferring the *bias* (tendency of a variable to be either positive or negative) from a pool of known solutions.

Heule et al. [HvM08] also successfully exploit observed statistical bias on random k-SAT problems. The authors examine search trees produced by several well-known value-ordering heuristics and measure the distribution of solutions in individual subtrees. The observed bias is then extrapolated to new instances. The authors propose a novel backjumping mechanism called *distribution jumping* (back-jumping to a subtree more likely to contain a solution based on the observed bias), and implemented it in the `march_ks` solver. The algorithm performed well on random instances, but showed no improvement on real-world problems.

3.3 Lookahead Value Ordering

We now examine value-ordering techniques based on lookahead. Lookahead solvers are DPLL-based algorithms which make their branching decisions by examining the immediate effect of setting individual variables to $True$ or $False$. Specifically, for each variable v, a lookahead solver performs two unit propagations to obtain reduced formulas $F[v = False]$ and $F[v = True]$. The resulting formulas are then compared to the original formula F by some *difference measure* $\texttt{Diff}(F, F')$ which aids in deciding which of the literals should be branched on. The two estimates $L \equiv \texttt{Diff}(F, F[v = False])$ and $R \equiv \texttt{Diff}(F, F[v = True])$ are usually combined using some mixing function `MixDiff`(L,R) to rank the variable. In most solvers, `MixDiff` is simply the product of L and R.

Lookahead is a powerful technique because it lets the algorithm see one step ahead of the search process. Unfortunately, examining all variables at each step is usually too expensive. For this reason, most lookahead solvers operate on small subsets of promising variables. The *variable pre-selection* step is also heuristic in nature. Unlike CDCL algorithms, lookahead solvers do not record conflict clauses. For this reason, lookahead heuristics typically select promising branches, rather than fail-first branches. The main objective of lookahead heuristics is to ensure the search tree is small and balanced.

The first lookahead solver with these features was `POSIT` [Fre95]. The branching heuristics of this solver are heavily inspired by MOMS which was one of the few known heuristics at the time. The solver branches on v if v occurs in more clauses of minimum length than $\bar{v}$, and on $\bar{v}$ otherwise. The variable-ordering heuristic involves several intricate tie-breakers, but for the most part also follows the MOMS paradigm.

Another highly successful lookahead solver based on clauses of minimum length is `SatZ` by Li and Anbulagan [LA97]. In this solver, $\texttt{Diff}(F, F')$ is defined as the number of clauses of minimum size in F' but not in F, and the mixing function is,

$$\texttt{MixDiff} = 1024 \times LR + (L + R).$$

The sum $L+R$ is used for tie-breaking, while the factor 1024 is used to give preference to the product LR, and is otherwise arbitrary. The variable with the highest `MixDiff`

is chosen in hopes of reducing the formula quickly. The value-ordering heuristic always branches on $True$.

Dequen and Dubois [DD04] describe `kcnfs`, a lookahead solver specifically targeted for solving satisfiable random k-SAT problems. While the variable selection is based on a rather involved *Backbone Search Heuristic* (BSH), the variable-ordering heuristic simply branches on v if v occurs more often than $\bar{v}$ and on $\bar{v}$ otherwise. This approach is reminiscent of literal-ordering heuristics DLCS and DLIS discussed earlier.

Mpekas et al. [MVVW06] describe a hybrid SAT solver which combines lookahead techniques and CDCL search in the MiniSAT framework. The branching heuristic ranks the literals by their "reduction quality" (the literal's potential to simplify the formula) defined as,

$$RQ(l) = lq(l)lq(\bar{l}) + lq(l) + lq(\bar{l}),$$

where $lq(l)$ is the "literal quality" of l. The *Propagations Based Heuristic* (PBH) defines the literal quality as,

$$lq(l) = |IUP(l)|,$$

where $IUP(l)$ is the set of all literals assigned during unit propagation on literal l. The more literals are forced to assume a value as a result of l, the stronger the literal quality. The *Ternary Clauses Heuristic* (TCH) defines the literal quality as,

$$lq(l) = \sum_{x_i \in IUP(l)} 1 + w(x_i),$$

where $w(x_i) = \#occ3(\bar{x}_i)$ is the number of 3-literal clauses containing $\bar{x}_i$. By including the weight $w(x_i)$, TCH takes into account the number of binary clauses that are created after branching on l. Intuitively, literals which fix many literals *and* create a lot of short clauses have more potential to reduce the formula. TCH can be seen as looking one step ahead of BPH. Whereas most lookahead heuristics are promise-based, this hybrid solver performed best with a fail-first approach which combined PBH with branching on literals with the highest reduction quality first.

A relaxation lookahead heuristic is used by Kullmann [Kul02] in `OKsolver`. At the decision variable, the heuristic prefers the polarity whose resulting formula minimizes

$$\sum_{k \geq 2} -|Q_k| ln(1 - 2^{-k}),$$

where Q_k denotes the set of all clauses of length k. Kullman shows that this sum is proportional to the probability that a random Boolean formula of the same size is falsified by a random assignment. In other words, the heuristic approximates the formula by a random formula having the same structure, and selects a value which is more likely to satisfy it.

The solver `march_eq`, developed by Heule et al. [HDVZVM05], is one of the most efficient lookahead solvers inspired by `SatZ`. This solver chooses the variable having the maximal product LR, and branches on v if $L < R$ and on $\bar{v}$ otherwise. The

difference measure `Diff` is defined as,

$$\texttt{Diff}(F, F[l]) = |B(l)| + \sum_{Q_i \in C(l)} eq(|Q_i|),$$

where $eq(k) = 5.5 \times 0.85^k$ and $C(l)$ is the set of all equivalence clauses[1] reduced during unit propagation, $eq(k)$ is an empirically-determined clause weighting function, and $B(l)$ is the set of all newly-created binary clauses after unit propagation on l. Intuitively, `march_eq` prefers variables which create a lot of binary clauses, and affect a large number of equivalent literals. The choice of polarity, on the other hand, minimizes short-term reductions in hopes of finding a solution (promise policy). `march_eq` performs well on random and crafted instances.

3.4 Local Search in SAT

Although we focus on complete solvers in this thesis, the heuristics presented in the next chapter borrow several concepts from incomplete SAT solvers. Specifically, our heuristics are based on local search, and use the number of unsatisfied clauses as the primary objective function; both of these properties commonly appear in incomplete SAT solvers. For this reason, we provide a brief overview of existing SAT solvers based on local search.

As stated previously, incomplete solvers do not guarantee finding a solution, nor can they disprove a formula. This class of solvers typically begins with a random guess, which is gradually refined until a satisfying assignment is found, or the algorithm gives up. The development of incomplete SAT solvers began in the early 1990's, when very large instances of the famous N-Queens problem were successfully solved using local search techniques [MJPL90]. Local search quickly garnered interest from researchers, as it was able to solve instances which were beyond the reach of complete solvers available at the time. Local search also attracted interest because it could be readily adapted to tackle the closely related Maximum Satisfiability problem (MAX-SAT).

Selman el al. [SLM92] developed the highly influential incomplete SAT solver `GSAT` and showed that it outperformed many backtracking algorithms on random and crafted graph colouring instances. `GSAT` is based on a simple greedy variable "flip" (polarity switch) procedure. The algorithm begins with a random variable assignment. If the assignment satisfies the formula, the algorithm has found a model and terminates. Otherwise, the algorithm selects a variable which, once flipped, causes the largest number of unsatisfied clauses to become satisfied. This process is repeated until all clauses are satisfied, or the maximum number of iterations is exceeded.

`GSAT` effectively minimizes the following objective function: each of the 2^n assignments is labeled with a number between 0 and m corresponding to the number of

[1]Equivalence clauses are of the form $(l_i \leftrightarrow l_j \leftrightarrow ... \leftrightarrow l_k)$ and represent equivalence relationships between literals. These clauses are derived in a pre-processing step.

clauses the assignment is in conflict with. A satisfying assignment (if one exists) has no conflicts, and thus has the value 0. The algorithm aims to find such an assignment, by always making a greedy choice among the n neighbours of the current estimate (each neighbour corresponding to a single variable flip). In this formulation, a local minimum is an assignment whose all n neighbours conflict with more clauses than the assignment itself. It has been observed that local search rarely gets trapped in local minima when n is large; instead, the algorithm spends a lot of time making "sideways moves" (variable flips which neither improve nor worsen the current estimate). These areas of the search space have become known as "plateaus", and the research effort has shifted towards escaping plateaus quickly [SKC94].

Inspired by simulated annealing [KGJV83], the `WalkSAT` algorithm [SKC93] added a *random walk strategy* to the basic `GSAT` framework. The new algorithm used a user-defined *noise parameter* $0 \leq p \leq 1$ to add randomized moves to the search process. In addition, all flips were restricted to unsatisfied clauses only. These seemingly minor tweaks have enabled `WalkSAT` to escape plateaus extremely quickly, and made `WalkSAT` the basis for some of the most powerful incomplete solvers used today.

The pseudocode for `WalkSAT` is presented in listing 3.1[2]. The algorithm's local update step functions as follows: let the *penalty* of a variable be defined as the number of currently satisfied clauses which would become falsified if the variable is flipped. If a penalty-free flip can be made, `WalkSAT` performs it. Otherwise, the algorithm chooses an unsatisfied clause C; then, with probability p, the algorithm flips a random variable in C, and with probability $1 - p$ flips a variable in C which incurs the smallest penalty. When $p = 1$, the algorithm executes a pure random walk if a penalty-free step is not available. It has been shown that a single choice of p can work well across multiple instances from the same category of problems. For example, Kroc et al. [KSS10] conduct an empirical study on the noise parameter.

Modern variants of the `WalkSAT` architecture employ a variety of tweaks and modifications designed to speed up local search. The diversity of these algorithms is vast; we briefly highlight the `Novelty` family of algorithms which has been very succesful in recent SAT-Races. `Novelty` [MSK97] introduced a tie-breaking procedure to `WalkSAT` by choosing the most recently flipped variable in case of a penalty tie. `Novelty+` [Hoo99] added a second probabilistic parameter to the algorithm which independently governs the random walk. `AdaptNovelty+` [Hoo02] automatically tunes the noise parameter p: whenever the algorithm finds itself stagnant, the noise is gradually increased to help escape plateaus; eventually, the noise is lowered again. `G`^{2}`WSAT` [LH05] ties `Novelty` with the concept of Tabu search [G$^+$89] which temporarily blacklists variables which do not strictly decrease the objective function.

Several alternative approaches to the `WalkSAT` architecture have been studied, most prominently Dynamic Local Search (DLS). In DLS, clauses are assigned non-negative weights, and the objective function is usually defined to be the weighted sum of the unsatisfied clauses. As the search progresses, the weights are manipulated in such a way that the plateaus and local minima are "flooded" to reduce the chance

[2] `Uniform`$(0, 1)$ returns a number randomly sampled from the uniform distribution between 0 to 1.

of getting trapped in them again. This concept gave rise to a vast diversity of algorithms such as the Breakout Method [Mor93] and Scaling and Probabilistic Smoothing (SAPS) [HTH02]. We forego the detailed discussion of these methods as they are less relevant to our heuristics. More information and practical implementations of many of these algorithms can be found in the multi-faceted `UBCSAT` solver[3].

Algorithm 3.1: WalkSAT(F)

input : A CNF formula F
output : An assignment which satisfies F, or `NOTFOUND`
parameters: Iteration limits max_tries, max_flips, noise parameter p

```
for i ← 1 to max_tries do
    v ← random Boolean assignment
    for j ← 1 to max_flips do
        if F(v) = True then return v
        C ← randomly-chosen unsatisfied clause of F
        if some variable v_d ∈ C can be flipped without a penalty then
            Flip v_d
        else if Uniform(0, 1) < p then
            // Random walk.
            Flip a randomly-chosen variable ∈ C.
        else
            // Best local move.
            Flip a variable ∈ C with the lowest penalty.
return NOTFOUND
```

3.5 Discrepancy Based Search

The last topic we touch on is *Discrepancy Based Search*. This family of techniques is concerned not with how branching heuristics should work, but rather with how branching heuristics should be used in backtracking search. Heuristics are generally less informed at the top of the search tree and are more likely to make mistakes there. Unfortunately, a few bad decisions near the root of the search tree can quickly trap the solver in a large unfavourable region of the search space. While techniques such as restarts and clause learning may mitigate the effects of early mistakes, discrepancy based search is specifically designed to address this issue.

A *discrepancy* is a decision made against the heuristic. Depth-first search (DFS), which serves as the basis for most complete solvers, always trusts the branching heuristic to make the right decision, and goes against the heuristic only after backtracking. This makes it extremely costly for DFS to undo mistakes made near the root of the

[3]http://www.satlib.org/ubcsat/algorithms/index.html

tree. To tackle this issue, an alternative to DFS called Discrepancy Based Search was developed. This technique systematically explores the search tree, but in an order different from DFS. Discrepancy search acknowledges that heuristics are imperfect, and purposely goes against the heuristic at some nodes.

Limited Discrepancy Search (LDS) proposed by Harvey and Ginsberg [HG95] laid the foundation for this class of algorithms. LDS explores the search tree by progressively raising the limit on the number of allowed discrepancies. On the first iteration, LDS visits the path which completely adheres to the heuristic. On the second iteration, LDS visits all paths which contain at most one discrepancy ("wrong turn"). The process is repeated until all paths have been traversed. The last path explored is the one that goes completely against the heuristic. The authors demonstrate, both theoretically and empirically, that LDS outperforms DFS when informed branching heuristics are used. A conceptually identical traversal method was suggested earlier by Basin and Walsh [BW92], albeit in a less practical formulation.

One drawback of LDS is that it ends up re-visiting many leaf nodes in the search tree (at iteration k, LDS visits all paths with *at most* k discrepancies). This redundancy was removed by *Improved Limited Discrepancy Search* (ILDS) proposed by Korf [Kor96]. At iteration k, ILDS visits all paths with *exactly* k discrepancies, which insures that no leaf node is visited twice.

While LDS and ILDS allow for discrepancies, neither algorithm specifically addresses the fact that heuristics are more likely to make mistakes at the top of the search tree. This motivated the development of *Interleaved Depth-first Search* (IDFS) by Meseguer [Mes97] and *Depth-bounded Discrepancy Search* (DDS) by Walsh [Wal97]. These search algorithms bias discrepancies higher up in the search tree. IDFS effectively runs multiple depth-first searches in parallel, with each search originating from a different node near the root of the tree. DDS places a limit on the depth below which discrepancies are not allowed, and progressively lowers this limit with each iteration. DDS has been shown to significantly outperform DFS and ILDS on random 3-SAT problems.

We have briefly described the most influential discrepancy search methods. It should be noted that many variants of these techniques have been studied and integrated into practical solvers. For example, Furcy and Koenig [FK05] describe *Limited Discrepancy Beam Search* which combines LDS with beam search techniques and backtracking, and evaluate it on a range of CSP problems. Mijnders et al. [MDWH$^+$10] propose *Advanced Limited Discrepancy Search* (ALDS) which combines the strengths of LDS and DDS, and successfully use it in a lookahead solver on satisfiable 3-SAT instances.

3.6 Summary

In this chapter, we have reviewed existing value ordering heuristics in DPLL, CDCL, and lookahead SAT solvers. Most of these heuristics are based on greedy decisions which appear to lead to either shortening or falsifying the formula in the short term; other heuristics select polarities by attempting to estimate the probability of finding

a solution through relaxation and counting methods. We have also touched on the subjects of local search in SAT, and discrepancy-based search.

In the next chapter, we propose three novel value ordering heuristics. Unlike the algorithms outlined in this chapter, our heuristics are based on local search and non-linear optimization, and are independent of the variable ordering heuristic.

Chapter 4

Our Approach to Value Ordering

In this chapter, we present our three value-ordering heuristics based on non-linear optimization techniques.

4.1 Real-Valued Formulation of CNF

The heuristics presented in this chapter are based on the same basic principle. First, we transform the CNF formula into an unconstrained optimization problem defined on a real-valued domain. We then apply an iterative local search method to this problem to approximate the location of a solution. Finally, we use this approximation to determine which polarity the DPLL decision variable should take on first. The heuristics differ in their representation of the problem, and the optimization method used.

In order to use iterative optimization methods such as gradient descent or Newton's Method, the problem must be ported to a real-valued domain. A natural choice is to map each Boolean variable $v_i \in \{True, False\}$ to a real-valued variable $x_i \in [0, 1]$. The optimization domain is therefore the interior and the boundary of the unit hypercube D,

$$D = \{(x_1, x_2, ..., x_n) : x_i \in \mathbb{R}, 0 \leq x_i \leq 1\},$$

whose corners correspond to the set of all possible assignments to n Boolean variables. For example, the assignment $\boldsymbol{v} = (True, False, False)$ of a CNF formula in three variables corresponds to the point $\boldsymbol{x} = (1, 0, 0)$; the corresponding domain D is shown in Figure 4.1. While we can always convert v_i to x_i, converting x_i to v_i cannot be done directly if x_i is not exactly 0 or 1. In situations like this, the simplest course of action is to round x_i to the nearest Boolean value. As with Boolean variables, we can "negate" any x_i by simply subtracting it from 1. For example, if v_3 maps to x_3, $\bar{v}_3$ corresponds to $1 - x_3$.

Now that we have replaced Boolean variables with real-valued ones, we need a way to convert the Boolean formula into a real-valued optimization problem defined as a function of $\boldsymbol{x}$. There are infinitely many ways of accomplishing this transformation, and the heuristics presented in this chapter demonstrate a few possible choices. The

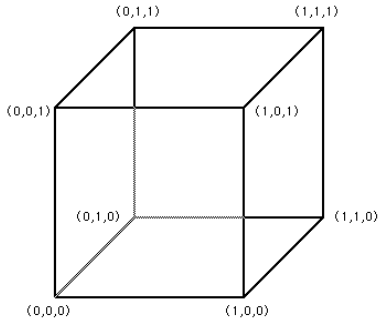

Figure 4.1: The interior of the unit hypercube forms the domain on which we define our optimization problems. Here, the domain for a formula with $n = 3$ variables is a regular unit cube. The 2^n corners of the unit hypercube correspond to the set of all possible Boolean assignments of the n variables.

only restriction for defining the optimization problem is that it should be based on functions that are continuous and differentiable on D, otherwise local search methods will not work. Next, we examine each heuristic in detail.

4.2 Method 1: System of Equations

The first method is a direct transformation of a Boolean SAT problem to the real-valued domain. We first replace each clause C with a real-valued function f whose roots correspond to the satisfying assignments of C, and then use Newton's Method to approximate the solution of the resulting system of equations. Noting that a clause $C = (l_1 \vee l_2 \vee ... \vee l_k)$ is satisfied whenever any of its literals evaluate to $True$, we can define a matching real-valued function f as,

$$f(\boldsymbol{x}) = \prod_{l \in C} z(l), \qquad \text{where } z(l) = \begin{cases} 1 - x_u & \text{if } l = v_u \\ x_u & \text{if } l = \bar{v}_u \end{cases} \tag{4.1}$$

Since f is a product of several linear functions, any assignment causing at least one of its factors to become 0 is also a root of f. Notice that any such assignment has at least one $x_i \in \{0, 1\}$ and thus corresponds to a satisfying Boolean assignment of C. If f evaluates to 0 for a subset of the variables, the values of the remaining variables are irrelevant and can be set to either 0 or 1.

Example 4.1. *The real-valued counterpart of the clause $C = (v_1 \vee \bar{v}_2 \vee \bar{v}_3)$ is $f = (1 - x_1)x_2x_3$. C is satisfied whenever $v_1 = True$ or at least one of v_2 and v_3 is $False$. C is falsified only when $(v_1, v_2, v_3) = (False, True, True)$. Similarly, its real-valued counterpart f evaluates to 0 whenever $x_1 = 1$ or one of x_2 and x_3 is 0. f evaluates to 1 only when $(x_1, x_2, x_3) = (0, 1, 1)$*[1].

[1]On the unit hypercube, this is the only choice of x_1, x_2, x_3 for which f evaluates to 1. Outside the search domain, however, infinitely many such assignments exist, e.g. $(x_1, x_2, x_3) = (0, 2, 0.5)$.

A CNF formula F is satisfied if and only if each clause is satisfied. That is, we seek an assignment $\boldsymbol{v}$ of Boolean variables such that $C_1(\boldsymbol{v}) = True, C_2(\boldsymbol{v}) = True, ..., C_m(\boldsymbol{v}) = True$. We now consider a system of corresponding real-valued clauses $\boldsymbol{f}(\boldsymbol{x}) = (f_1(\boldsymbol{x}), f_2(\boldsymbol{x}), ..., f_m(\boldsymbol{x}))$ as defined above. Such a system is "satisfied" if and only if each real-valued clause evaluates to 0. That is, we seek a vector $\boldsymbol{x}$ such that $f_1(\boldsymbol{x}) = 0, f_2(\boldsymbol{x}) = 0, ..., f_m(\boldsymbol{x}) = 0$, or $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$ for short.

If such assignment $\boldsymbol{x}$ exists, the original formula F is satisfiable, and its model $\boldsymbol{v}$ can be reconstructed from $\boldsymbol{x}$ by setting $v_i = True$ if $x_i = 1$, $v_i = False$ if $x_i = 0$. If x_i is neither 0 or 1, v_i can be set to an arbitrary Boolean value. This situation occurs when both polarities of variable v_i appear in the model. If $\boldsymbol{x}$ does not exist, F is unsatisfiable.

In order to find the satisfying assignment, we need to solve the non-linear system $\boldsymbol{f}(\boldsymbol{x}) = \boldsymbol{0}$ with m equations and n variables. Not surprisingly, this is also an NP-Complete problem [JG79]. While we cannot in general solve such a system in closed form, we can use iterative optimization techniques to approximate the location of the solution and then use this information to guide DPLL towards promising areas of the search space.

The heuristic functions as follows: we first make an initial uninformed guess $\boldsymbol{x}_0$ for each of the unassigned variables. Since we do not want to bias any variable towards either 0 or 1, we can choose $\boldsymbol{x}_0 = (\frac{1}{2}, \frac{1}{2}, ..., \frac{1}{2})$ as the starting point. Geometrically, this corresponds to starting the search at the center of the n-dimensional unit hypercube D. We then apply a fixed number of iterations of Newton's Method to refine our estimate. Finally, we round the obtained estimate for $\boldsymbol{x}$ to the nearest corner of the unit hypercube, which corresponds to a Boolean assignment. We thus obtain a vector of preferred polarities for *all* of the unassigned variables at once.

Due to the large size and non-linearity of the problem, it is unlikely that the local search method alone will solve the system even when a solution exists[2]. Furthermore, rounding $\boldsymbol{x}$ to the nearest Boolean assignment is not guaranteed to preserve or improve the current estimate computed by Newton's Method. Quite often, the rounded value of $\boldsymbol{x}$ is much worse than the estimate obtained through local search. In practice, however, this approach can often aid DPLL in making better polarity choices.

In order to use Newton's Method on the resulting system of equations, we must first compute its Jacobian matrix $J\boldsymbol{f}(\boldsymbol{x})$ whose elements are $J_{ij} = \frac{\partial f_i}{\partial x_j}$. In other words, we need a way of differentiating a real-valued clause f with respect to any variable x_j for $j = 1..n$. If the variable v_j does not appear in the corresponding CNF clause C, the partial derivative $\frac{\partial f_i}{\partial x_j}$ simply vanishes. Otherwise, v_j appears in some literal[3] $p \in C$. Since f is defined as a product, we differentiate the factor of f in which x_j appears, while keeping all other factors constant. The derivative $\frac{\partial z(l)}{\partial x_j}$ follows directly from the definition of $z(l)$.

[2]For medium and large SAT problems, Newton's Method usually settles in a local minimum in under ten iterations.

[3]Notice that there can be only one such literal, because by definition, a disjunctive clause in a CNF formula contains each variable at most once.

$$\frac{\partial f}{\partial x_j} = \frac{\partial z(p)}{\partial x_j} \prod_{\substack{l \in C \\ l \neq p}} z(l), \qquad \text{where } \frac{\partial z(l)}{\partial x_j} = \begin{cases} -1 & \text{if } l = v_j \\ 1 & \text{if } l = \bar{v}_j \end{cases} \tag{4.2}$$

Example 4.2 (Differentiating Real-Valued Clauses)**.** *Consider the real-valued clause $f = (1 - x_1)x_2x_3$ from the previous example. Assume that the system of equations in which it appears has four variables: x_1, x_2, x_3, x_4. The corresponding partial derivatives are: $\frac{\partial f}{\partial x_1} = -x_2x_3$, $\frac{\partial f}{\partial x_2} = (1 - x_1)x_3$, $\frac{\partial f}{\partial x_3} = (1 - x_1)x_2$ and $\frac{\partial f}{\partial x_4} = 0$.*

So far, we have presented this heuristic as following the promise paradigm (that is, its primary objective is to approximate the location of the satisfying assignment), but we can also easily modify this approach to function as a fail-first heuristic. In the promise mode, we seek to satisfy as many clauses as possible, which corresponds to optimizing the system $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{0}$. In the fail-first mode, we seek to *falsify* as many clauses as possible, which corresponds to optimizing the system $\boldsymbol{f}(\boldsymbol{x}) = \mathbf{1}$. To accomplish this, we simply apply the optimization method to a modified system of equations $\boldsymbol{g}(\boldsymbol{x}) = \mathbf{0}$ where each $g_i \equiv f_i - 1$ for all i. Note that the Jacobian of $\boldsymbol{g}(\boldsymbol{x})$ is the same as the Jacobian of $\boldsymbol{f}(\boldsymbol{x})$, so no additional computation is needed to use it with Newton's Method. The fail-first version of the heuristic guides DPLL towards regions of the search space where conflicts are more likely to occur. This may lead to shorter learned clauses, and ultimately, a smaller search tree.

The pseudocode for the heuristic is presented in Listing 4.1. The helper function `EvalClause`$(C_i, \boldsymbol{x})$ evaluates the real-valued counterpart f_i of clause C_i at point $\boldsymbol{x}$. Similarly, the helper function `DiffClause`$(C_i, j, \boldsymbol{x})$ differentiates f_i with respect to the variable x_j, and evaluates the resulting partial derivative at point $\boldsymbol{x}$ as defined in Equation (4.2).

At first glance, it may appear that using Newton's Method would be inefficient in this situation, both in terms of storage and runtime. The matrix J which defines the linear system to be solved is quite large (even for a medium SAT problem with 200 variables and 5000 clauses, J has one million elements), and solving it by conventional methods would be problematic since most practical linear system solvers have a runtime complexity of $O(n^3)$ [Atk09]. Fortunately, the matrix J is extremely sparse. To see this, consider a 3-SAT problem where each clause depends on exactly three variables. This implies that each row in J has at most three non-zero partial derivatives. In general, the number of non-zero entries in J is at most $|F|$ (the total number of literals of F), as opposed to mn. If the formula consists of relatively short clauses, $|F|$ is much smaller than mn. In practice, this nearly always holds, as real-world SAT problems are usually dominated by very short clauses.

Sparse matrices can be stored efficiently and solved much more quickly than dense matrices. For our implementation, we used the `CSparse` library by Timothy A. Davis[4] [Dav06]. This open-source C library offers competitive performance on standard benchmarks, and offers a number of well-known direct algorithms for solving sparse matrices (QR, LU and Cholesky decompositions). Since the number of clauses is

[4]http://www.cise.ufl.edu/research/sparse/CSparse

Algorithm 4.1: HeuristicA(F, mode)

input : A CNF formula F with n variables and m clauses $C_1, C_2, .., C_m$ and the mode of operation (promise or fail-first)
output : A vector of preferred polarities for all variables
parameters: The maximum number of iterations max_iterations, step size λ

```
// Begin search at the center of the unit hypercube
```
for $i \leftarrow 1$ **to** n **do**
  $x_i \leftarrow \frac{1}{2}$
$J \leftarrow m \times n$ sparse matrix
for *iteration* $\leftarrow 1$ **to** max_iterations **do**
  ```
  // Apply Newton's Method to refine the estimate
  ```
  for $i \leftarrow 1$ **to** m **do**
    $f_i \leftarrow$ `EvalClause`$(C_i, \boldsymbol{x})$
    if mode $= fail\text{-}first$ **then**
      $f_i \leftarrow f_i - 1$
    for $j \leftarrow 1$ **to** n **do**
      $J_{ij} \leftarrow$ `DiffClause`$(C_i, j, \boldsymbol{x})$
  Solve $J\Delta\boldsymbol{x} = -\boldsymbol{f}$ for $\Delta\boldsymbol{x}$
  $\boldsymbol{x} \leftarrow \boldsymbol{x} + \lambda\Delta\boldsymbol{x}$
```
// Round x to the nearest Boolean assignment
```
for $i \leftarrow 1$ **to** n **do**
  if $x_i > \frac{1}{2}$ **then** polarity$[i] \leftarrow True$
  else polarity$[i] \leftarrow False$
return polarity

always equal to, or greater than the number of variables, the Jacobian matrix is almost always overdetermined (has more equations than unknowns). For this reason, we used QR decomposition as our primary solver algorithm, as this method does not require the matrix to be square, invertible, or positive definite. In essence, the answer obtained through QR decomposition is closely related to the linear least squares problem; the answer fits all the equations of the Jacobian matrix as close as possible.

4.3 Method 2: Conflict Potentials

Unlike the first heuristic, the second method aims to find favourable polarities by optimizing a single objective function rather than a system of equations. Perhaps the most obvious way to convert a CNF formula to a single real-valued objective function would be to rewrite it as a "product of sums":

$$A(\boldsymbol{x}) = \prod_{i=1}^{m} \sum_{l \in C_i} z(l), \qquad \text{where } z(l) = \begin{cases} x_u & \text{if } l = v_u \\ 1 - x_u & \text{if } l = \bar{v}_u \end{cases} \tag{4.3}$$

In this formulation, A evaluates to 0 for any conflicting assignment, and to a value greater than 0 for any satisfying assignment[5]. Unfortunately, $A(\boldsymbol{x})$ is a degree-m multivariate polynomial that is poorly suited for local search techniques. Although it is possible to compute the partial derivatives of this function, great care is required to avoid numerical instability. Rather than trying to optimize $A(\boldsymbol{x})$, we focus on a different objective function $R(\boldsymbol{x})$ defined as a "sum of products",

$$R(\boldsymbol{x}) = \sum_{i=1}^{m} \prod_{l \in C_i} z(l), \qquad \text{where } z(l) = \begin{cases} 1 - x_u & \text{if } l = v_u \\ x_u & \text{if } l = \bar{v}_u \end{cases} \tag{4.4}$$

or simply,

$$R(\boldsymbol{x}) = \sum_{i=1}^{m} f_i(\boldsymbol{x}), \tag{4.5}$$

where $f_i(\boldsymbol{x})$ is the real-valued counterpart of clause C_i as defined in Equation (4.1).

In this formulation, $R(\boldsymbol{x})$ evaluates to zero only when each of its real-valued clauses evaluates to zero. Since this situation corresponds to each clause C_i being satisfied, any such point $\boldsymbol{x}$ maps to a model of F. In general, for any Boolean assignment, all satisfied real-valued clauses evaluate to 0, and all conflicting ones evaluate to 1. Because $R(\boldsymbol{x})$ sums over all f_i, it is therefore a measure of the number of clauses a particular Boolean assignment conflicts with. On the interior of the domain D, the value of this function can be thought of as an *interpolation* between the values of $R(\boldsymbol{x})$ at the corners of the unit hypercube. Intuitively, minimizing $R(\boldsymbol{x})$ leads to assignments closer to the solution, and maximizing it leads to conflict-rich areas of the search space.

[5]To be precise, $A \geq 1$ for any satisfying assignment, because the value of each "clause" is at least 1 for all such assignments (corresponding to at least one satisfied literal).

Functions of this form have appeared in the literature [War99]. A discrete version of $R(\boldsymbol{x})$ is, in fact, the standard objective function used in most incomplete SAT solvers[6] such as `GSAT` [SLM92] and `WalkSAT` [SKC93]. For convenience, we shall refer to such functions as *conflict potentials*.

Example 4.3 (Conflict Potentials)**.** *The CNF formula* $F = \bar{v}_1 \wedge (v_1 \vee \bar{v}_2) \wedge (\bar{v}_1 \vee v_2)$ *has a corresponding conflict potential* $R(\boldsymbol{x}) = x_1 + (1 - x_1)x_2 + x_1(1 - x_2)$*. The values of this function at the corners of* D *are* $R(0, 0) = 0$*,* $R(0, 1) = R(1, 1) = 1$ *and* $R(1, 0) = 2$*. These values indicate that the Boolean assignment* $(False, False)$ *satisfies* F*,* $(False, True)$ *and* $(True, True)$ *conflict with one clause of* F *each, and* $(True, False)$ *conflicts with two clauses of* F*.*

Apart from having an intuitive geometric interpretation, $R(\boldsymbol{x})$ is also much easier to differentiate than objective function $A(\boldsymbol{x})$, which makes it a good candidate for iterative optimization methods. Since $R(\boldsymbol{x})$ is defined as a sum, any partial derivative of $R(\boldsymbol{x})$ is simply the sum of corresponding partial derivatives of $f_i(\boldsymbol{x})$. This fact leads to an efficient and numerically-stable method for computing both the gradient $\nabla R(\boldsymbol{x})$,

$$\nabla R(\boldsymbol{x}) = \sum_{i=1}^{m} \nabla f_i(\boldsymbol{x})$$

and the Hessian matrix $H(R)(\boldsymbol{x})$ of the conflict potential,

$$H(R)(\boldsymbol{x}) = \sum_{i=1}^{m} H(f_i)(\boldsymbol{x}).$$

We already have a method for finding first order partial derivatives of f_i as defined in equation (4.2). Finding second-order partial derivatives of f_i is also straightforward. Suppose a real-valued counterpart f of clause C is differentiated with respect to variable x_j followed by x_k. There are three possible cases. If C does not depend on both v_j and v_k, the derivative vanishes. If $j = k$ (that is, we differentiate with respect to the same variable twice) and v_j occurs in literal $p \in C$, we differentiate the corresponding factor of f_i twice, while keeping all other factors constant:

$$\frac{\partial^2 f}{\partial x_j^2} = \frac{\partial^2 z(p)}{\partial x_j^2} \prod_{\substack{l \in C \\ l \neq p}} z(l) \tag{4.6}$$

Lastly, if $j \neq k$ and v_j occurs in literal $p \in C$ and v_k occurs in literal $q \in C$, we differentiate the corresponding factors of f_i once, while keeping all other factors constant:

$$\frac{\partial^2 f}{\partial x_j \partial x_k} = \frac{\partial z(p)}{\partial x_j} \frac{\partial z(q)}{\partial x_k} \prod_{\substack{l \in C \\ l \neq p, l \neq q}} z(l) \tag{4.7}$$

[6]The discrete version of $R(\boldsymbol{x})$ used in incomplete solvers is restricted to Boolean assignments only, which corresponds to the corners of our search domain D.

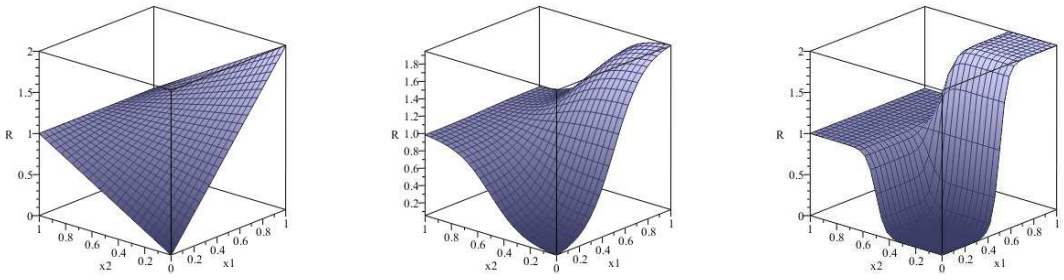

Figure 4.2: Three conflict potentials for the Boolean function $F = \bar{v}_1 \wedge (v_1 \vee \bar{v}_2) \wedge (\bar{v}_1 \vee v_2)$ using different parametrizations $z(l)$. From left to right: linear parametrization, sigmoid parametrization with $K = 8$, sigmoid parametrization with $K = 30$. The magnitude of the potentials at the corners of the domain is the number of clauses in conflict with the corresponding Boolean assignment. For example, the point $(0, 0)$ does not conflict with any of the clauses, which means $(False, False)$ is the model of F. The point $(1, 0)$ corresponding to $(True, False)$, on the other hand, conflicts with two clauses of F (clauses 1 and 3) and is the least optimal assignment.

The first-order partial derivative $\frac{\partial z(l)}{\partial x_j}$ is given in Equation (4.2). To find the second-order partial derivative $\frac{\partial^2 z(l)}{\partial x_j^2}$, we simply differentiate $\frac{\partial z(l)}{\partial x_j}$ with respect to x_j one more time. Notice that in this case, $\frac{\partial^2 z(l)}{\partial x_j^2}$ always evaluates to 0. This, however, only holds for the linear choice of $z(l)$ as defined in Equation (4.1).

Example 4.4 (Differentiating Conflict Potentials)**.** *The gradient and the Hessian matrix of conflict potential* $R(\boldsymbol{x}) = x_1 + (1 - x_1)x_2 + x_1(1 - x_2)$ *from the previous example can be easily computed by differentiating each term individually, and then summing the results together:*

$$\nabla R(\boldsymbol{x}) = \left(\frac{\partial R}{\partial x_1}, \frac{\partial R}{\partial x_2} \right) = (2 - 2x_2, 1 - 2x_1)$$

$$H(R)(\boldsymbol{x}) = \begin{pmatrix} \frac{\partial^2 R}{\partial x_1^2} & \frac{\partial^2 R}{\partial x_1 \partial x_2} \\ \\ \frac{\partial^2 R}{\partial x_2 \partial x_1} & \frac{\partial^2 R}{\partial x_2^2} \end{pmatrix} = \begin{pmatrix} 0 & -2 \\ \\ -2 & 0 \end{pmatrix}$$

So far, we have modeled real-valued clauses f_i as products of linear functions as defined in equation (4.1). While this representation is valid, some difficulties arise when it is used to define conflict potentials. While each f_i satisfies $0 \leq f_i \leq 1$ on D, its value becomes meaningless outside the unit hypercube. Depending on where f_i is evaluated, the value could be negative or greater than 1, which has no meaningful interpretation when mapped back to the Boolean search space. For this reason, we resort to "clipping" $\boldsymbol{x}$ at each optimization step to confine it to the unit hypercube. Unfortunately, doing so often interferes with the integrity of local search, and some-

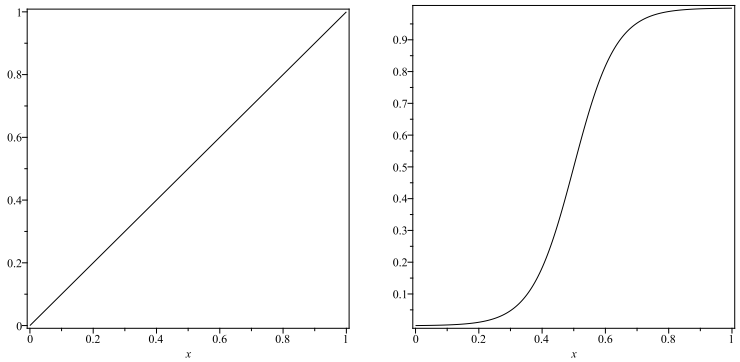

Figure 4.3: The real-valued clauses f_i can be represented as products of many different kinds of 0-1 functions. While the linear function (left) is the most natural choice, the sigmoid function (right) has the advantage of being restricted to the range from 0 to 1 for all values of x which can improve local search.

times causes the Hessian matrix to become numerically unstable. In order to avoid this problem, it is desirable to represent f_i as a product of continuous, differentiable functions $z(l)$ that satisfy $0 \leq z(l) \leq 1$ *everywhere*. This would eliminate the need for clipping $\boldsymbol{x}$ because each f_i would have a meaningful value on the entire space $\mathbb{R}^n$.

One parametrization that satisfies these properties is the sigmoid function. The sigmoid is a logistic function commonly used to approximate the discrete step function and is widely used in neural networks, image processing and other areas of computer science. The sigmoid function is defined as follows[7],

$$S(x) = \frac{1}{1+\rho}, \qquad \rho = e^{-K(x-\frac{1}{2})},$$

where K is the "stiffness factor". The larger the value of K, the closer the sigmoid is to a discrete step function. For smaller values of K, the sigmoid approaches a linear function asymptotically squeezed between 0 and 1. The first and second-order partial derivatives of this function can be shown to be,

$$\frac{\partial S(x)}{\partial x} = K\rho S^2(x)$$

$$\frac{\partial^2 S(x)}{\partial x^2} = K\rho(\rho-1)S^3(x)$$

By substituting these expressions for $z(l)$, $\frac{\partial z(l)}{\partial x_j}$ and $\frac{\partial^2 z(l)}{\partial x_j^2}$ in equations (4.1) and (4.2), a sigmoid parametrization of SAT is obtained. While the sigmoid is a convenient parametrization for real-valued clauses, there are some drawbacks associated with it:

- computing the sigmoid and its derivatives is costly.

[7]The standard sigmoid has a phase transition at $x = 0$. We shift the sigmoid to the right by $\Delta x = \frac{1}{2}$ to place the phase transition at the center of our domain D.

- the convergence speed of Newton's Method is sensitive to the value of the "stiffness" parameter K and choosing a robust value of K that works across multiple problems is difficult, and
- unlike the linear parametrization, the second-order derivative of the sigmoid is not zero which makes for a denser Hessian matrix.

The pseudocode for the heuristic is presented in Listing 4.2. Since we can easily compute the Hessian matrix in closed form, we can use Newton's Method for scalar equations in place of a simple gradient descent to speed up the search. Unlike the $m \times n$ Jacobian matrix used in our first heuristic, the size of the Hessian matrix is only $n \times n$. Since the number of clauses is always greater than the number of variables[8], the Hessian matrix is much smaller than the Jacobian, and can be solved more quickly. Similar to the first heuristic, this method can be made to function in fail-first mode; to achieve this, we simply negate the objective function. For convenience, we also provide a sample C++ implementation (see Appendix A) of this heuristic (the static version) which uses gradient descent instead of Newton's Method.

Helper functions $\texttt{DiffR}(F, i, \boldsymbol{x})$ and $\texttt{DoubleDiffR}(F, i, j, \boldsymbol{x})$ compute the first and second order partial derivatives of the conflict potential of F with respect to variables x_i and x_j, and evaluate the result at point $\boldsymbol{x}$.

We now show that the well-known Jeroslow-Wang (JW) value ordering heuristic can be seen as a special case of the conflict potential method. Specifically, we show that JW is equivalent to a single gradient descent optimization step of a linear conflict potential starting from the center of the unit hypercube. As described in Chapter 3, the JW heuristic sets the decision variable v_u to $True$ if an only if,

$$\sum_{i\colon v_u \in C_i} 2^{-|C_i|} > \sum_{i\colon \bar{v}_u \in C_i} 2^{-|C_i|}$$

with the intuition that a larger sum corresponds to a greater fraction of potential assignments that can be eliminated if v_u is assigned accordingly. Consider now a conflict potential $R(\boldsymbol{x})$ as defined in Equation (4.4). A single optimization step amounts to computing the gradient $\nabla R(\boldsymbol{x})$ at the point $(\frac{1}{2}, \frac{1}{2}, ..., \frac{1}{2})$ and making a small step against its direction. Since we only consider one step, the preferred polarity of each Boolean variable v_i is completely determined by the sign of its corresponding gradient component (i.e. the partial derivative of $R(\boldsymbol{x})$ with respect to x_i). We now explicitly compute the derivative $\frac{\partial R(\boldsymbol{x})}{\partial x_u}$ for the decision variable v_u. The expression follows directly from Equation (4.6):

$$\frac{\partial R(\boldsymbol{x})}{\partial x_u} = \sum_{i=1}^{m} \frac{\partial f_i(\boldsymbol{x})}{\partial x_u}$$

We can split the sum into two parts: terms which contain x_u (corresponding to clauses containing $\bar{v}_u$) and terms which contain $1 - x_u$ (corresponding to clauses containing v_u). Terms which do not depend on x_u contribute nothing to the partial derivative and can be omitted.

[8] In practice, m can be orders of magnitude greater than n.

Algorithm 4.2: HeuristicB(F, mode)

input : A CNF formula F with n variables and m clauses, mode of operation (promise or fail-first)
output : A vector of preferred polarities for all variables
parameters: The maximum number of iterations max_iterations, step size λ

```
// Begin search at the center of the unit hypercube
for i ← 1 to n do
    x_i ← 1/2
H ← n × n sparse matrix
for iteration ← 1 to max_iterations do
    // Apply Newton's Method to refine the estimate
    for i ← 1 to n do
        G_i ← DiffR(F, i, x)
        for j ← 1 to n do
            H_ij ← DoubleDiffR(F, i, j, x)
    Solve HΔx = G for Δx
    if mode = promise then
        x ← x − λΔx
    else
        x ← x + λΔx
    // Restrict x to the unit hypercube
    for i ← 1 to n do
        x_i ← max(0, min(1, x_i))
// Round x to the nearest Boolean assignment
for i ← 1 to n do
    if x_i > 1/2 then polarity[i] ← True
    else polarity[i] ← False
return polarity
```

$$\frac{\partial R(\boldsymbol{x})}{\partial x_u} = \sum_{i\colon v_u \in C_i} \frac{\partial f_i(\boldsymbol{x})}{\partial x_u} + \sum_{i\colon \bar{v}_u \in C_i} \frac{\partial f_i(\boldsymbol{x})}{\partial x_u}$$

A real-valued clause f_i which depends on x_u is a product of $|C_i|$ factors, at most one of which is a function of x_u. After differentiating f_i with respect to x_u, we are left with $|C_i| - 1$ factors and a sign as specified in Equation (4.2). At the center of the unit hypercube, all variables have coordinates $x_i = \frac{1}{2}$. Therefore, all remaining factors evaluate to $\frac{1}{2}$ (this also holds for positive literals, since $1 - \frac{1}{2} = \frac{1}{2}$). The magnitude of each differentiated term is therefore $(\frac{1}{2})^{|C_i|-1}$.

$$\begin{aligned}
\left.\frac{\partial R(\boldsymbol{x})}{\partial x_u}\right|_{\boldsymbol{x}=(\frac{1}{2},\frac{1}{2},\ldots,\frac{1}{2})} &= \sum_{i\colon v_u \in C_i} (-1) \prod_{\substack{l \in C_i \\ l \neq v_u}} \left(\frac{1}{2}\right) + \sum_{i\colon \bar{v}_u \in C_i} (+1) \prod_{\substack{l \in C_i \\ l \neq \bar{v}_u}} \left(\frac{1}{2}\right) \\
&= -\sum_{i\colon v_u \in C_i} \left(\frac{1}{2}\right)^{|C_i|-1} + \sum_{i\colon \bar{v}_u \in C_i} \left(\frac{1}{2}\right)^{|C_i|-1}
\end{aligned}$$

For the decision variable v_u to be set to $True$, the partial derivative has to be negative (recall that we go *against* the gradient in gradient descent). Setting up the inequality, rearranging and dividing both sides by 2, we obtain the final condition, which is equivalent to the Jeroslow-Wang formulation.

$$-\sum_{i\colon v_u \in C_i} 2^{-|C_i|+1} + \sum_{i\colon \bar{v}_u \in C_i} 2^{-|C_i|+1} < 0$$

$$\sum_{i\colon v_u \in C_i} 2^{-|C_i|} > \sum_{i\colon \bar{v}_u \in C_i} 2^{-|C_i|}$$

The result is interesting in that the Jeroslow-Wang formula was derived from a completely discrete point of view (by counting the number of rows in the truth table that can be eliminated), while our conflict potential formulation is completely continuous in nature.

4.4 Method 3: Stochastic Optimization

The third method builds on the idea of conflict potentials by adding a stochastic element to the search. In doing so, we attempt to escape local extrema and obtain a better polarity estimate.

There are many ways of adding randomized behaviour to optimization algorithms such as ours. Perhaps the simplest modification would be to use a random starting guess $\boldsymbol{x}_0$ instead of the fixed "center of the hypercube" assignment $(\frac{1}{2}, \frac{1}{2}, ..., \frac{1}{2})$. We have had some success with this technique, however due to the high dimensionality of the problem, the quality of the approximation is very sensitive to the choice of the starting position, which negatively impacts the repeatability of results.

We propose a different technique, which has shown to be more robust and effective. The starting guess is $(\frac{1}{2}, \frac{1}{2}, ..., \frac{1}{2})$ as before, however the decision on how to refine the starting guess is made by examining not one, but *multiple* conflict potentials. In most applications, local search methods have only one objective function to work with and have to rely on stochastic methods such as simulated annealing and genetic algorithms to escape local extrema. In this situation, however, we have the advantage of being able to generate as many objective functions for the same CNF formula as we need.

As stated previously, there exists an infinite number of ways of representing real-valued clauses of a CNF formula, and each parametrization gives rise to a unique conflict potential. For example, both $R_1 = x_1 x_2$ and $R_2 = {x_1}^2 x_2$ are valid conflict potentials for the Boolean function $F = (\bar{v}_1 \wedge \bar{v}_2)$. R_1 and R_2 "conflict" (evaluate to 1) only for $(x_1, x_2) = (1, 1)$ corresponding to the only falsifying Boolean assignment $(v_1, v_2) = (True, True)$. This result stems from the simple observation that the curves x and x^2 have the same endpoints at $x = 0$ and $x = 1$.

In general, for a given function F, we can *generate* as many different conflict potentials as we wish by simply choosing different parametrizations for the real-valued clauses f_i. All such potentials will have the same values at the corners of D, but the shape of the function (and the distribution of local extrema) will be different for each choice of parametrization. Inevitably, some potentials will be better suited for local optimization methods than others, and there is no simple way of determining which parametrization is best ahead of time. We can however, sample a few such potentials and let them vote on where to go next.

The heuristic repeatedly samples from a family of conflict potentials $R_{\boldsymbol{\theta}}$, where $\boldsymbol{\theta} = (\theta_1, \theta_2, ..., \theta_n)$ is a vector of exponents uniquely identifying the potential. Each $R_{\boldsymbol{\theta}}$ is a sum of real-valued clauses represented as products of exponential curves defined as follows,

$$z(l) = \begin{cases} x_u^{\theta_u} & \text{if } l = v_u \\ (1 - x_u)^{\theta_u} & \text{if } l = \bar{v}_u \end{cases} \tag{4.8}$$

Notice that the basic conflict potential defined in equation (4.1) is also a member of $R_{\boldsymbol{\theta}}$. Specifically, it is the potential with $\boldsymbol{\theta} = (1, 1, ..., 1)$.

Example 4.5 (Family of Conflict Potentials)**.** *The family of conflict potentials associated with the CNF formula* $F = \bar{v}_1 \wedge (v_1 \vee \bar{v}_2) \wedge (\bar{v}_1 \vee v_2)$ *is* $R_{\boldsymbol{\theta}}(\boldsymbol{x}) = x_1^{\theta_1} + (1 - x_1)^{\theta_1} x_2^{\theta_2} + x_1^{\theta_1}(1 - x_2)^{\theta_2}$*. For example, the conflict potential* $R_{(2,1)}(\boldsymbol{x}) = x_1^2 + (1 - x_1)^2 x_2^1 + x_1^2(1 - x_2)^1$ *is a member of this family for* $\boldsymbol{\theta} = (2, 1)$*.*

The pseudocode for the heuristic is presented in Listing 4.3. `Uniform`(a, b) returns a random real number sampled from the uniform distribution in $[a, b]$. The algorithm repeatedly samples multiple vectors of exponents $\boldsymbol{\theta}$, constructs corresponding conflict potentials $R_{\boldsymbol{\theta}}$, and uses their gradients evaluated at the current estimate $\boldsymbol{x}$ to vote on a common gradient direction. The algorithm then advances $\boldsymbol{x}$ by a small step in that direction. Since the vote vector is not a real gradient, we cannot rely on its magnitude for gradient descent. Instead, we opt for making fixed-size steps Δx in the direction of the vote vector. The geometric interpretation of this method is that the

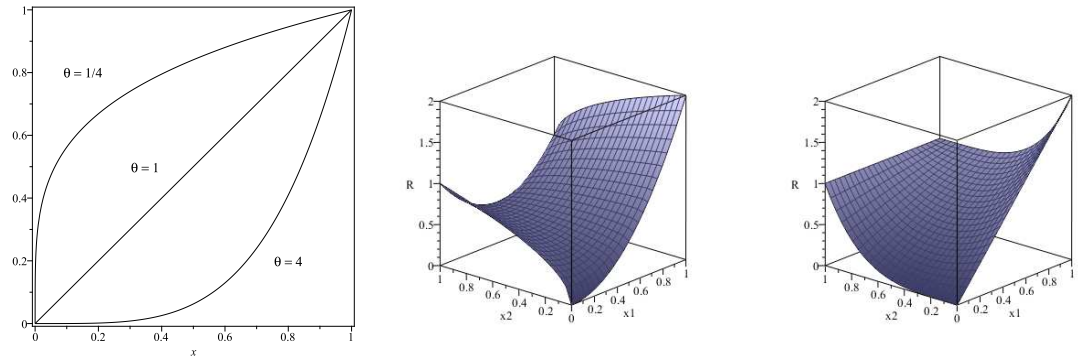

Figure 4.4: The stochastic heuristic represents real-valued clauses as products of exponential curves with randomly-sampled exponents θ_i. This parametrization gives rise to a family of conflict potentials $R_{\boldsymbol{\theta}}$. On the right, two members of this family, $R_{(2,0.5)}$ and $R_{(1,4)}$, are plotted for the CNF formula $F = \bar{v}_1 \wedge (v_1 \vee \bar{v}_2) \wedge (\bar{v}_1 \vee v_2)$. All potentials in $R_{\boldsymbol{\theta}}$ have the same values at the corners of the unit hypercube, but different distributions of local extrema; this useful property is exploited by the stochastic gradient descent.

member functions of the $R_{\boldsymbol{\theta}}$ family represent different interpolations between fixed values at the corners of the unit hypercube. We do not know apriori which of these interpolations is best for performing gradient descent on; we can however, poll several interpolations and decide on a common gradient direction.

For this heuristic, we use a simple gradient descent in favour of more advanced techniques. Our experiments showed that Newton's Method tends to become increasingly unstable when faced with the severe non-linearity of randomly-sampled conflict potentials. Furthermore, since many samples are required at each step, Newton's Method becomes prohibitively expensive.

The exponents θ_i are sampled in a way that generates a mixture of convex functions (such as $x^{0.5}$) and concave functions (such as x^2). To achieve this, we first sample a value σ from a uniform distribution between 1 and an upper bound σ_{max}, and then with probability $\frac{1}{2}$, set θ_i to either σ or $1/\sigma$.

The gradient of each f_i is found using equation (4.2) with the following rule for literal p which stems directly from the definition of the exponential curves used to represent the real-valued clauses,

$$\frac{\partial z(l)}{\partial x_j} = \begin{cases} \theta_j x_j^{\theta_j - 1} & \text{if } l = v_j \\ -\theta_j (1 - x_j)^{\theta_j - 1} & \text{if } l = \bar{v}_j \end{cases}$$

Similarly to the other two heuristics, this method can be made to function in fail-first mode by simply reversing the direction of the gradient. Given enough time and a high number of samples, this method has demonstrated remarkably good results, however it is computationally expensive and is better suited for use as a static heuristic.

Algorithm 4.3: HeuristicC(F, mode)

input : CNF formula F with n variables and m clauses, and the mode of operation (promise or fail-first)
output : A list of preferred polarities for all variables
parameters: The number of iterations max_iterations, the number of samples max_samples, maximum exponent σ_{max}, step size λ

for $i \leftarrow 1$ **to** n **do**
  $\boldsymbol{x}_i \leftarrow \frac{1}{2}$
for *iteration* $\leftarrow 1$ **to** max_iterations **do**
  `// Create a mixture of convex and concave potentials`
  for $i \leftarrow 1$ **to** n **do**
    $\text{votes}_i \leftarrow 0$
    Sample $\sigma \sim$ `Uniform`$(1, \sigma_{max})$
    if `Uniform`$(0, 1) > 0.5$ **then** $\theta_i \leftarrow \sigma$ **else** $\theta_i \leftarrow \frac{1}{\sigma}$
  `// Vote on a common direction`
  for *sample* $\leftarrow 1$ **to** max_samples **do**
    gradient $\leftarrow \nabla R_{\boldsymbol{\theta}}(\boldsymbol{x})$
    for $i \leftarrow 1$ **to** n **do**
      if $\text{gradient}_i > 0$ **then**
        $\text{votes}_i \leftarrow \text{votes}_i + 1$
      else if $\text{gradient}_i < 0$ **then**
        $\text{votes}_i \leftarrow \text{votes}_i - 1$
  `// Update` $\boldsymbol{x}$ `and restrict it to the unit hypercube`
  for $i \leftarrow 1$ **to** n **do**
    if mode $=$ *promise* **then**
      $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i - sign(\text{votes}_i)\lambda$
    else
      $\boldsymbol{x}_i \leftarrow \boldsymbol{x}_i + sign(\text{votes}_i)\lambda$
    $\boldsymbol{x}_i \leftarrow \max(0, \min(1, \boldsymbol{x}_i))$
for $i \leftarrow 1$ **to** n **do**
  if $\boldsymbol{x}_i > \frac{1}{2}$ **then** $\text{polarity}[i] \leftarrow True$
  else $\text{polarity}[i] \leftarrow False$
return polarity

4.5 Using the Heuristics

We now describe how to use our heuristics in a SAT solver. Since all three heuristics are computationally expensive and share the unusual property that they compute preferred polarities for all unassigned variables at once, special care needs to be taken when using them in practice.

In order to use our polarity heuristics, the SAT solver needs to maintain a global vector of n Boolean variables which store the preferred polarity for each variable. Minisat comes equipped with such a vector (the `polarity[]` vector of the `Solver` class) which is normally used for the phase-saving feature. Other solvers may need to be extended with such a vector.

In order to use the heuristics statically, one simply executes the desired heuristic once right after the formula has been pre-processed, but before the search begins. At this point, all variables are unassigned and all clauses are still unsatisfied; this fact can be exploited to implement a particularly efficient version of the heuristic by avoiding a number of special cases and checks. The output of each heuristic is a vector of preferred polarities which is then stored in the global preference vector. At each branch decision, the solver consults this vector for the polarity to try first. When the heuristic is used statically, we can afford to invest extra time into obtaining a good initial estimate; this corresponds to using a large number of iterations at a fine resolution (small step size λ).

Using the heuristic dynamically is more challenging. Since all three heuristics are computationally expensive and compute the preferred polarities for all unassigned variables at once, it would be extremely wasteful to compute them at every branch decision. Moreover, the computational complexity of the heuristics which rely on solving sparse matrices varies greatly from instance to instance, and cannot be gauged easily.

Intuitively, the decisions made early on (higher up in the search tree) carry a heavier weight than the decisions made later on. A "good" decision near the root can provide exponential savings in the number of nodes explored. For this reason, the approach we propose in this thesis uses the depth of the branch decision as a criterion for when to use the heuristic. Specifically, the user supplies a parameter $0 < \tau < 1$ which denotes the fraction of the height of the search tree where the heuristic should be computed. Since the exact height of the search tree is not known apriori, we can simply use n as the upper bound. If the branching decision occurs at depth between 0 and $\frac{n}{\tau}$, the heuristic is computed, and the result is stored in the global polarity preference vector. All decisions made at depths below this threshold simply use the polarities computed above, as in the static approach.

Another dynamic method would be to apply the heuristics after a certain number of uninstantiated variables remain, and/or a certain number of unsatisfied clauses remain. For example, one could choose to apply the heuristic whenever $n < 100$ and $m < 1000$. In doing so, the heuristic examines only small (and bounded) sub-trees within the main search tree. This approach may not be very accurate, but it is a practical way to place an upper bound on the computational cost of each heuristic computation. This is of particular importance to matrix-based heuristics as they tend

#	Problem Representation	Optimization Method	Parametrization
1	System of equations	NM for systems of equations	Linear
2	Conflict potential	NM for scalar functions	Linear / Sigmoid
3	Family of conflict potentials	Stochastic gradient descent	Exponential

Table 4.1: The three heuristics presented in this thesis differ in their choice of problem representation, the optimization method used, and the parametrization of the real-valued clauses.

to consume a lot of resources on large SAT instances.

4.6 Summary

We have presented three value ordering heuristics based on local optimization techniques. All three methods convert the SAT problem into a real-valued optimization problem defined on the unit hypercube, and then use different optimization techniques to approximate the optimal assignment (promise mode), or the least optimal assignment (fail-first mode). The heuristics then round this estimate to the nearest Boolean assignment and use it as a polarity guide for DPLL. The differences between the three heuristics are highlighted in Table 4.1.

Next, we evaluate these heuristics on different classes of SAT problems, discuss parameter selection, and analyze the strengths and weaknesses of each approach.

Chapter 5

Experimental Evaluation

In this chapter, we empirically evaluate our heuristics. First, we describe our testing environment and discuss the benchmarks used in our experiments. Then, we calibrate the parameters of our algorithms and compare the performance of our heuristics on a set of randomly-generated instances. In addition to our heuristics, we also examine three standard value-ordering heuristics: MOMS, Jeroslow-Wang, and phase-saving. To gain more insight into the convergence rates of our approach, we conduct additional experiments with varying amounts of pre-processing and analyze its effects on the accuracy of the results. We then proceed to apply our methods to large-scale crafted and real-world instances. Finally, we aggregate our results and compare the performance of our heuristics against six other state-of-the-art SAT solvers, including a lookahead solver and two incomplete solvers. We conclude with a simulation of heuristic portfolios (combinations of heuristics) and a discussion on how multi-core CPUs can benefit our approach.

5.1 Experimental Setup

We implemented static versions of our heuristics in the MiniSAT 2.2 framework[1] [ES04] with the CSparse library [Dav06] for sparse matrix operations. MiniSAT was compiled using `gcc` with the *-O3* optimization switch on a 64-bit Linux kernel 2.6. All tests were run on a cluster of 64-bit 2 × Dual Core AMD Opteron machines clocked at 2.6GHz with 10GB of RAM.

For each SAT instance, we recorded the solver output (SAT, UNSAT or INDET), the model (if any), as well as basic statistics (the elapsed CPU time, the total number of decisions made, the total number of restarts, and the total number of conflicts). We enforced a CPU timeout of 1200 seconds (20 minutes) per instance[2], which is a standard policy in SAT competitions. The timeout applies to both the static heuristic and the main search process (that is, more pre-processing means less time for the actual search). No memory limit was enforced. For stochastic heuristics, the results are averaged over 30 runs with different seeds. The result of the solver is verified

[1] Latest version as of 2012.

[2] Throughout this section, all measures of time refer to CPU time, rather than wall clock time.

at a later stage to ensure correctness. The verification step is not counted as search time. For satisfiable instances, the verification step consists of simply plugging the model into the formula; for unsatisfiable instances, we rely on the results of state-of-the-art solvers, as well as benchmark descriptions (many benchmarks are specifically designed to be unsatisfiable). None of our tests produced erroneous results, which is not surprising since MiniSAT is known to be a reliable framework. Considering that the only component of the search we alter is the value ordering heuristic, there is no risk of weakening the integrity of MiniSAT.

Clearly, all components of a CDCL solver interact with each other in non-trivial ways during search; for example, value ordering decisions affect subsequent variable ordering, restarts, learned clauses, and vice versa. It would be desirable to test the merits of our value ordering heuristic in isolation (that is, with other CDCL enhancements and heuristics disabled), as some researchers advocate [Hoo95]. This, however, is problematic because modern crafted and real-world instances are usually almost intractable with CDCL enhancements turned off. The goal of our experiments was to see how our value ordering heuristics can benefit MiniSAT as a whole.

With the exception of the value ordering heuristic, we used all the default settings of MiniSAT 2.2. In order to improve the repeatability of our results, we did not use the random variable selection heuristic (MiniSAT can be configured to occasionally pick a random decision variable to introduce a small amount of noise into the search process.) For experiments where the phase-saving heuristic is enabled (the default behaviour of MiniSAT), we used the standard "full phase-saving" mode (*--phase-saving=2*).

5.2 Algorithms

The methods described in the previous chapter give rise to numerous choices for implementing the heuristics in practice. For example, the conflict potential approach can be implemented in promise or fail-first mode, with a linear or sigmoid parameterization, and optimized using gradient descent or Newton's method. We selected the most representative and telling combinations of techniques and parameterizations. Our preliminary tests suggested that unlike promise-based heuristics, all fail-first gradient-based methods (linear potential, sigmoid potential, stochastic vote) produced very similar results in terms of accuracy. For this reason, we focused our attention on the linear potential for computing fail-first polarities, as it is the cheapest heuristic among the three. In our naming convention, we use a subscript plus sign to denote promise-based heuristics, and a subscript minus sign to denote fail-first heuristics. We implemented and tested the Jacobian-based heuristics NS_+ and NS_-, the Hessian-based promise heuristic HS_+, a stochastic vote promise heuristic VS_+, and three gradient-based methods: linear potential heuristics GS_+ and GS_-, and a sigmoid potential promise heuristic SS_+. The detailed summary of heuristics used in our evaluation is presented in Table 5.1.

In addition to our selection of heuristics, we also tested a few simple branching policies: always branching on $False$, always branching on $True$, and branching on

Heuristic	Type	Formulation	Optimization Method
NS_+	promise	System of equations	Newton's method
NS_-	fail-first	System of equations	Newton's method
GS_+	promise	Linear conflict potential	Gradient descent
GS_-	fail-first	Linear conflict potential	Gradient ascent
HS_+	promise	Linear conflict potential	Newton's method
SS_+	promise	Sigmoid conflict potential	Gradient descent
VS_+	promise	Family of conflict potentials	Stochastic vote descent

Table 5.1: The heuristics examined in our evaluation.

$False$ with phase-saving enabled (the default behaviour of MiniSAT). In our results, we refer to these heuristics as $False$, $True$ and PS respectively. The reason we test these three heuristics separately is twofold: first, we wish to investigate whether enabling phase-saving has any substantial merit over simply branching on $False$ in practice. Secondly, we wish to investigate whether branching on $False$ has in fact a consistent advantage over always branching on $True$. There exist benchmarks (such as the `battleship_sat` benchmark used in our evaluation) where branching on $True$ solves almost twice as many instances as branching on $False$. However, if we were to simply negate each variable in that benchmark, we would obtain SAT instances of equivalent difficulty, but with the opposite preferred branch polarity. For this reason, we evaluate both $True$ and $False$ heuristics separately.

We have also implemented two well-known greedy dynamic value ordering heuristics, MOMS and Jeroslow-Wang, as described in Chapter 2. Since both heuristics only examine clauses which contain the decision variable, they can be implemented quite efficiently in the MiniSAT framework by tapping into the existing literal watch list mechanism. Our implementation does not use learned clauses for making decisions (our preliminary tests on random instances suggested that using learned clauses drastically slowed down the decision process while offering little in return).

Lastly, we examine the bigger picture of where MiniSAT equipped with our heuristics stands in the world of modern SAT algorithms. To do this, we investigate how our heuristics fare against a selection of state-of-the-art solvers. Our test suite consists of well-known solvers selected from the winners of the most recent (as of this writing) SAT Competition 2011[3]. These solvers have won gold, silver, and bronze medals in at least one benchmark category. Although the primary focus of our thesis is on complete DPLL-CDCL solvers, we wanted to see how MiniSAT equipped with our heuristic performs relative to other solver architectures. We selected four complete solvers, one of which is a lookahead solver, and two incomplete stochastic solvers. We used `glucose` (won gold in SAT+UNSAT application, silver in UNSAT application, bronze in UNSAT crafted), `clasp` (gold in UNSAT crafted), `lingeling` (bronze in SAT+UNSAT crafted), `march_rw` (gold in UNSAT random), as well as two incomplete solvers `sparrow2011` (gold in SAT random) and `adaptg2wsat` (bronze in SAT random at SAT Competition 2009). The detailed list of solvers used in our

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

Solver	Version / Build	Type	Reference
MiniSAT	2.2	Complete	[ES04]
glucose	2.0	Complete	[AS09]
clasp	2.0-R4092	Complete	[GKNS07]
lingeling	release 587	Complete	[Bie10]
march_rw	SAT11 Competition build	Complete (lookahead)	[HvZD11]
sparrow2011	SAT11 Competition build	Incomplete (SLS)	[SS11]
adaptg2wsat	SAT11 Competition build	Incomplete (SLS)	[LWZ07]

Table 5.2: SAT solvers used in our evaluation. With the exception of our reference solver (MiniSAT), all have scored high at SAT Competition 2011. Although our heuristics are geared toward complete DPLL-CDCL algorithms, we have also included two incomplete solvers, and one look-ahead solver for comparison.

evaluation is given in Table 5.2. The solvers were compiled from source in our testing environment. The source code and detailed descriptions of the solvers are available from the SAT Competition website. We used the same rules (1200 second timeout, no memory limit, default settings) for each of our reference solvers.

5.3 Benchmark Selection

We now turn our attention to the SAT benchmarks used in our experiments. Our goal was to select a wide range of modern benchmarks that are sufficiently difficult to solve. The number of variables (or clauses) in a formula is not necessarily a good indicator of the difficulty of an instance. In our test set, there are instances with hundreds of thousands of variables that get solved within seconds, and instances with as few as 180 variables that stump even the most advanced solvers. For example, the once-challenging SATLIB and DIMACS benchmarks[4] which have been used for evaluating SAT solvers for many years are much too easy for modern solvers. Our preliminary tests have shown that MiniSAT can successfully solve the largest uniform random 3-SAT SATLIB benchmarks (uf250-1065 and uuf250-1065) in under 10 seconds per instance just by using its default branching rules. On these instances, the time it takes to compute our static heuristics often outweighs the time actually spent searching. Similarly, the dynamic heuristics MOMS and Jeroslow-Wang did not offer any measurable performance boost. For this reason, we focus our attention on benchmarks that can be considered to have "medium" to "hard" difficulty by modern (complete) solver standards. We were interested in benchmarks where MiniSAT on its default settings solves less than half of all instances within our 20 minute limit. We avoided benchmarks which are too hard (MiniSAT solves less than 5% of the instances) or too easy (MiniSAT solves more than 50% of the instances within 1 minute).

Nonetheless, the fact that the cost of our static heuristics can sometimes outweigh the actual search time on easy instances can be seen as the first observable weakness

[4]Available at http://www.satlib.org

of our approach. Since there is no simple and reliable way of estimating how difficult a SAT instance is ahead of time, we cannot tune the amount of work (for example, the number of iterations) that goes into computing the static heuristic. One simple way of addressing this shortcoming would be to start the search normally, and only engage our static heuristic if the instance still has not been solved after some fixed amount of time. This would allow easy instances to be solved without incurring the time penalty of our heuristic.

With the above-mentioned criteria in mind, we selected all our benchmarks from the submissions to the most recent SAT Competitions. Instances from all these benchmarks have been used in SAT-Race 2009 and SAT-Race 2011 and are available for download from the SAT Competition website. We also used a few instances from SAT-Race 2007 for parameter calibration and preliminary testing. We used the benchmarks as they are provided, without removing or modifying any instances. As is usually done in SAT solver evaluation, we have constructed three separate test sets: randomly-generated (uniform 3-SAT formulas), crafted (puzzles and games), and application (cryptography, circuits, bioinformatics, and other real-world sets).

The random set (`r3sat`) consists of all satisfiable medium-sized 3-SAT instances from SAT-Race 2009. These 110 uniform formulas have each 360 to 560 variables and a clause-to-variable ratio of 4.25. This ratio ensures that the instances are hard to solve. This "phase-transition phenomenon" is one of the most striking empirical results obtained in the history of SAT research. The phase transition region for 3-SAT formulas lies between clause-to-variable ratios of at least 3.42 [KKL06] and at most 4.506 [DBM00]. This benchmark proved to be sufficiently challenging for the complete solvers used in our evaluation, with MiniSAT on default settings solving only about 35% of the instances within the 20 minute limit. We also experimented with unsatisfiable random 3-SAT formulas, although finding a good benchmark proved to be difficult. We settled on the random unsatisfiable 3-SAT instances from SAT-Race 2007, because the more recent benchmarks turned out to be too hard for MiniSAT. We refer to this set as `r3unsat`.

The crafted and application test sets were constructed from a wide selection of SAT benchmarks varying in type (puzzles, proofs, circuit verification, cryptography), the number of variables (ranging from a few dozen to over half a million), satisfiability (satisfiable only, unsatisfiable only, or mixed), and difficulty. We also tried to keep the number of satisfiable and unsatisfiable instances balanced as much as possible. We briefly describe the different benchmarks used in our experiments below; for more information, see the supplied references.

The crafted test suite consists of 224 instances in 10 benchmarks. These instances are primarily puzzles, games, and graph-based challenges.

- `edgematching` [Heu09]: a puzzle involving the placement of differently-shaped pieces on a grid in a way such that the edges of the connected pieces match. The problem is known to be NP-complete. Since there is no straightforward encoding of the problem into SAT, the benchmark experiments with a number of different encodings of the same problem.
- `battleship_sat` and `battleship_unsat` [SH11]: a simple game which involves

sinking moving battleships (see reference for the exact rules of the game). The instances encode the decision problem whether a ship can be sunk in a given number of shots.

- `mosoi289_sat` and `mosoi289_unsat` [Mos11]: determining whether a 17×17 grid is 4-colourable. The puzzle is based on a challenge by Lance Fortnow[5].
- `sgen_sat` and `sgen_unsat`: SAT instances generated using the `sgen1` tool by Ivor Spence[Spe09]. This tool employs a number of techniques for generating very small but extremely hard instances.
- `automata_sync` [ST11]: discovery of synchronizing words in a finite state automaton (FSA). A synchronizing word (also known as a reset sequence) is an input which sends every state of the FSA to the same state. Although determining whether a synchronizing word exists can be done in polynomial time, finding a synchronizing word of a fixed length is an NP-complete problem.
- `modcircuits` [KKY09]: an optimization problem for finding efficient circuits for the MOD function.
- `vanderwaerden` [AKS11]: discovery of van der Waerden numbers [Ros04] which have applications in combinatorics and number theory. The instances are encoded as a hypergraph colouring problem.

The application test suite consists of 142 instances in 6 benchmarks. These benchmarks are real-world problems in the fields of bioinformatics, software verification, and cryptography.

- `aprove09` [Fuh09]: proving termination properties of programs (specifically, Term Rewrite Systems [BN]). The benchmark was generated using the `AProVE` software verification tool.
- `bioinfo` [CFT10]: discovery of parameters in discrete genetic networks (specifically, Gene Regulatory Networks). The problem is first formulated as a CSP, and then translated into SAT.
- `bioinstances` [BJ09]: computation of two different distance measures between evolutionary trees. Both measures are known to be NP-hard to compute. Accurate distance measures allow reconstruction of species lineages.
- `desgen` [SJ09b], `md5` [SJ09a], and `aes_32` [GK10]: key-finding problems for three well-known encryption standards: Data Encryption Standard (DES), MD5, and Advanced Encryption Standard (AES). These encryption methods are considered to be quite secure; correspondingly, the instances should be hard to solve.
- `slp_aes` [FSK10]: modeling an encryption algorithm as an XOR circuit with a minimal number of gates; based on the work of Boyar and Peralta [BP10].

The complete list of crafted and application benchmarks is given in Table 5.3.

[5] http://blog.computationalcomplexity.org/2009/11/17x17-challenge-worth-28900-this-is-not.html

[6] This benchmark is erroneously labeled as unsatisfiable. We found that at least one instance (289-unsat-6x30.cnf) is in fact satisfiable.

Crafted benchmarks (224 instances)

Benchmark	# of instances	Type	n_{max}	m_{max}	Reference
automata_sync	12	SAT+UNSAT	63882	123981	[ST11]
battleship_sat	28	SAT	3655	78604	[SH11]
battleship_unsat	17	UNSAT	900	13950	[SH11]
edgematching	30	SAT	24356	1884008	[Heu09]
modcircuits	19	SAT+UNSAT	1698	903653	[KKY09]
mosoi289_sat	15	SAT	720	28980	[Mos11]
mosoi289_unsat	15	SAT+UNSAT[6]	3360	661080	[Mos11]
sgen_sat	13	SAT	300	720	[Spe09]
sgen_unsat	13	UNSAT	153	316	[Spe09]
vanderwaerden	62	SAT+UNSAT	705	260122	[AKS11]

Application benchmarks (142 instances)

Benchmark	# of instances	Type	n_{max}	m_{max}	Reference
aes_32	10	SAT	1116	4312	[GK10]
aprove09	18	SAT	94663	318353	[Fuh09]
bioinfo	20	SAT+UNSAT	670867	3513333	[CFT10]
bioinstances	30	SAT+UNSAT	4692	582514	[BJ09]
desgen	19	SAT	32465	97748	[SJ09b]
md5	10	UNSAT	70139	228634	[SJ09a]
slp_aes	35	SAT+UNSAT	130182	440311	[FSK10]

Table 5.3: Crafted and application benchmarks used in the evaluation of our heuristics. n_{max} denotes the largest number of variables and m_{max} denotes the largest number of clauses in each benchmark. All benchmarks were taken from SAT Competition 2009 and SAT Competition 2011.

5.4 Parameter Selection

The heuristics developed in the previous chapter have several parameters, each of which can have a significant impact on the net efficiency and accuracy of the solver. Rather than trying to fine-tune each parameter to individual benchmarks and test sets, we attempt to find reliable parameters which work across many types of instances and do not significantly slow down the solver. All of our heuristics share two key properties: the total number of optimization steps, and the step size. In addition, the sigmoid-based method SS_+ requires a "stiffness" parameter K which determines how steep the underlying sigmoid curves are. The stochastic method VS_+ requires the total number of objective functions to sample and the maximum perturbation exponent θ_{max} which defines the family of sampled conflict potentials.

First, we focus on the number of iterations, as it is directly proportional to the net computational cost of each heuristic. Intuitively, more iterations at a smaller step size should yield better results. We must, however, set a reasonable limit on these parameters, since we are bound by the 20 minute execution limit. The computational cost per iteration for the three methods are shown in Figure 5.1. We evaluated the Jacobian-based approach NS_+ (averaged over 100 iterations), the gradient-based approach GS_+ (averaged over 10000 iterations), and the Hessian-based approach HS_+ (averaged over 100 iterations) on the random satisfiable set `r3sat`. Note that the corresponding fail-first versions (NS_-,GS_-, and HS_-) have the exact same complexity and speed since the only difference between them is the direction of optimization. The complexity of the sigmoid-based approach SS_+ is identical to that of GS_+, although the speed is understandably slightly lower due to the overhead of computing the sigmoid function and its derivatives. Finally, the computational cost of the stochastic vote method VS_+ is simply a multiple of the cost of GS_+ since all it does is sample a fixed number of objective functions from a family of potentials, whereas GS_+ always uses the same fixed potential $R_{\boldsymbol{\theta}=(1,1,\ldots,1)}$.

Several observations can be immediately made. The gradient-based GS_+ is extremely cheap with CPU time requirements of approximately 0.16ms per iteration for the largest instances in the set (560 variables). The complexity is linear in the number of literals, which in this set is directly proportional to the number of variables (since the formulas have three literals per clause, and a fixed clause-to-variable ratio, the total number of literals is $|F| = 3 \times 4.25n$). On the other hand, the two matrix-based heuristics are orders of magnitude slower to compute: HS_+ clocks in at 0.21s for the largest instances, while NS_+ takes 1.15s per iteration. Both methods rely on the CSparse library for solving sparse matrices, the complexity of which is cubic in the worst case. HS_+ takes less time to compute since it only has to solve the $n \times n$ Hessian matrix, whereas NS_+ solves the $m \times n$ Jacobian matrix. In this set, $m = 4.25n$, which corresponds to a 560×560 Hessian and a 2380×560 Jacobian for the largest instances in set `r3sat`. As is evident from the graph, these methods do not scale well for large instances such as those in the application test set. The largest instances that we were able to reliably test with NS_+ and NS_- (given our implementation and time limit) are about 3,000 variables and 60,000 clauses in size. Three crafted benchmarks (`automata_sync`, `edgematching`, `mosoi289_unsat`)

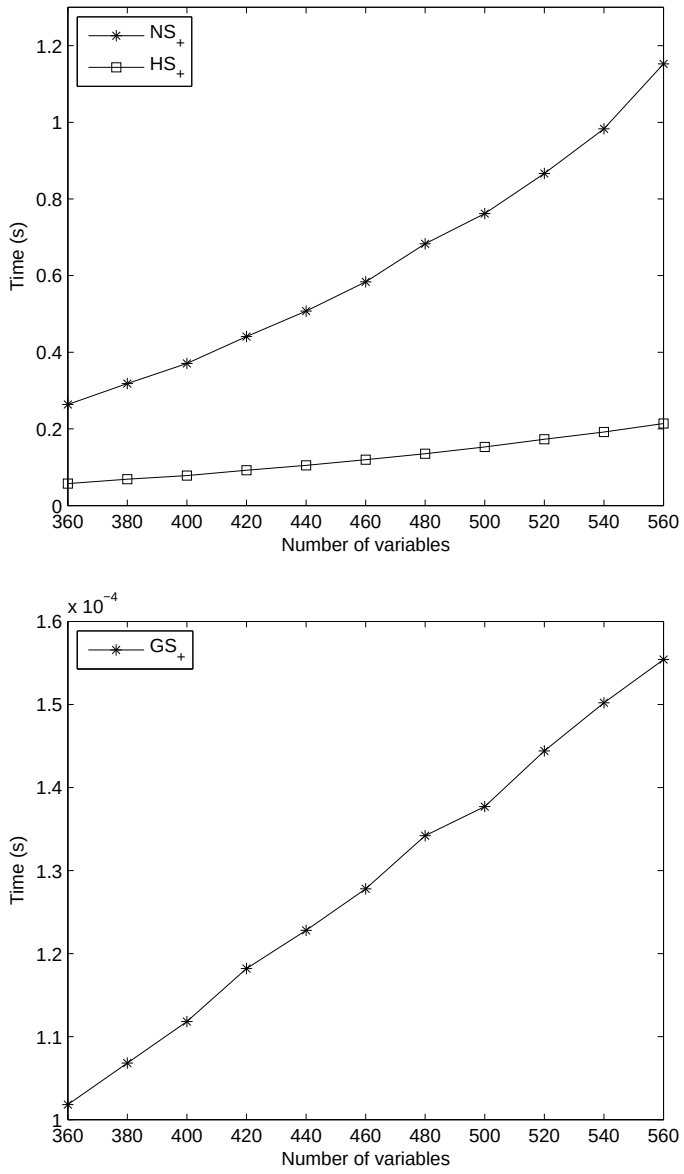

Figure 5.1: The cost of computing a single iteration of the three main heuristics: NS_+, HS_+, and GS_+ as a function of the number of variables. The results were obtained on the random satisfiable test set `r3sat`. The cost is measured in CPU time. The gradient-based GS_+ is extremely cheap and scales linearly with the number of literals. NS_+ and HS_+ do not scale well, as they both rely on an external sparse matrix solver which has cubic complexity. HS_+ is more efficient than NS_+ as it only requires solving the $n \times n$ Hessian matrix which is much smaller than the $m \times n$ Jacobian.

were too large for the matrix-based approaches (NS_+, NS_-, and HS_+). Similarly, all but one application benchmark (`aes_32`) were too large for the matrix-based methods. The `bioinstances` benchmark, however, was still within practical limits for the Hessian-based HS_+ heuristic.

In the case of matrix-based methods, the computational cost is completely dominated by solving, rather than filling the matrix. Computing the partial derivatives and setting up the matrix amount to less than 2% of the net cost of computing the heuristic on the random test set. The simplest way to speed up these heuristics would be to replace CSparse with a more efficient library. Such specialized libraries exist and are usually fine-tuned to the hardware of the machine. Some libraries even utilize the graphics processing unit (GPU) and multiple CPU cores for carrying out matrix operations at high speed. Nonetheless, the complexity of these algorithms is not as attractive as the gradient-based method when it comes to very large crafted and real-world application benchmarks.

Since the iterations can get expensive for larger instances, we must ensure that the solver does not waste the 20 minutes on the heuristic, and has enough time to actually perform the search. We chose to reserve at most 10% of the allocated time budget for computing the static heuristic. In other words, our version of MiniSAT has at most 2 minutes of pre-processing, and at least 18 minutes of search time per instance. For smaller instances, this time limit is rarely an issue because the heuristics are sufficiently fast to carry out the required number of iterations within the first 2 minutes. For large instances, the iterations are costly (especially in the case of NS_+), and the pre-processing time limit is often reached. This means many large instances (primarily in the application test set) often do not have a chance to carry out the target number of optimization steps.

To select a target number of iterations and the step size, we conducted a few preliminary tests on a random 5-SAT calibration test set consisting of an equal number of large satisfiable and unsatisfiable instances. The reason we chose 5-SAT is because it more closely matches the structure of real-world instances: most SAT instances are dominated by very short (2 and 3 literal) clauses, while clauses of length 10 and above occur rarely even for large application benchmarks. By trial and error, we found a reasonably high number of iterations that can be safely executed within the 2 minute pre-processing limit, and then tried several step sizes at that number of iterations. The step size with the highest accuracy rate was selected as the step size to be used on all other benchmarks. These results corresponded to approximately 2,000 iterations for GS_+, GS_-, and SS_+ with step size $\lambda = 0.001$, and 1,000 iterations for VS_+ with step size $\lambda = 0.01$. Since the matrix-based methods NS_+, NS_-, and HS_- are a lot more costly to compute, we set the target iteration count at 100; in practice, this number of iterations is rarely achieved in medium to large instances. The step size for these methods is also much larger ($\lambda = 0.3$).

We now turn our attention to the sigmoid-based method SS_+. The larger the stiffness parameter K, the closer a sigmoid function is to a regular step function. A true step function is not suited for iterative optimization since its derivative is discontinuous (the derivative is zero on either side of the step function and infinite at the transition). Similarly, a product of several sigmoid functions with a high

stiffness parameter suffers from the same problem: the derivatives outside the unit hypercube quickly approach zero and can effectively stall the optimization process due to rounding errors. We found that this regularly occurs for values of K above 40. This threshold is even lower for instances with longer clauses (since each literal in a clause results in an additional sigmoid factor, which weakens the derivative). On the other hand, choosing a value of K that is too small stretches the sigmoid potential out and requires more optimization steps to converge. With this in mind, we tried several K values on 30 random 5-SAT formulas (15 satisfiable and 15 unsatisfiable) used in SAT-Race 2009. The experimental data from our calibration set suggested that random 5-SAT formulas can be reliably optimized using values of K between 3 and 20. For simplicity, we used $K = 10$ for all our experiments. We do not, however, exclude the possibility that much better choices of K exist.

Finally, we calibrate the stochastic vote method VS_+. The perturbation exponents for this heuristic are sampled in the range from $\frac{1}{\theta_{max}}$ to θ_{max}. When θ_{max} is small (that is, close to 1), all sampled potentials are very close to the potential $R_{\boldsymbol{\theta}=(1,1,\ldots,1)}$ which is the default linear conflict potential used in gradient-based methods GS_+ and GS_-. In this case, virtually all votes will point in the same direction (same as the direction of the gradient), and the accuracy of the method will be the same as that of the GS heuristics, but at a multiple of the cost (for example, ten samples means ten times the computational cost). On the other hand, if the value of θ_{max} is too large, each sampled potential will be extremely non-linear ("wavy"); this is not a bad thing per se, since all conflict potentials sampled in this way are *correct* in the sense that they all can be used to find the least/most conflicting assignments of the same Boolean formula. The caveat is that, when sampling from a wider gamut of conflict potentials, we need to take more samples to have a meaningful (noise-free) gradient vote. Ideally, we would like to sample an infinite number of samples from the widest range of conflict potentials possible ($\theta_{max} \to \infty$). In practice, however, we are very much bound by the number of samples we can take, since the computational cost of the heuristic is directly proportional to the number of sampled potentials. Therefore, we first decide on the number of samples, and then tune θ_{max} to it. We decided that we can afford a ten-fold slowdown relative to GS_+, which corresponds to sampling 10 potentials. As with the matrix-based methods, this heuristic rarely manages to carry out the required number of iterations on very large instances. We examined several values of θ_{max} (1.5, 2, 5, 10, 20) on the random 5-SAT set and obtained an 11% accuracy boost at $\theta_{max} = 5$ relative to the next best results for 2 and 10. We therefore used $\theta_{max} = 5$ as the maximum perturbation exponent for all subsequent experiments.

5.5 Results and Discussion

We present the detailed timing results for each heuristic using time vs. accuracy graphs which have become standard in SAT evaluations. The vertical axis corresponds to the percentage of instances solved, and the horizontal axis corresponds to the time cut-off in the range from 0 to 1200 seconds. Intuitively, curves that grow faster and higher

are more desirable (meaning, the algorithm solves more instances in less time). The rightmost point on each curve indicates the final percentage of instances the algorithm managed to solve.

We first turn our attention to the random set `r3sat`. The results for this set are presented in Figure 5.2. The following observations can be made. First, there is very little difference between always branching on $False$ and always branching on $True$. Always branching on $True$ is slightly (about 3%) more accurate on medium instances, but both heuristics end up solving approximately the same number of instances (34% for $False$ and 36% for $True$) in the 1200 second timeframe. This is expected, since randomly-generated problems are unlikely to have a bias towards either polarity. The curve for the default MiniSAT branching rule (branching on $False$ with phase-saving enabled) actually underperforms both $True$ and $False$ for easy instances, and is eventually sandwiched between the two curves at a total of 35% instances solved. In this test set, enabling phase-saving offers little to no advantage. One could argue that the reason for this is that caching previous decisions is not helpful in random instances due to the lack of a repeatable structure.

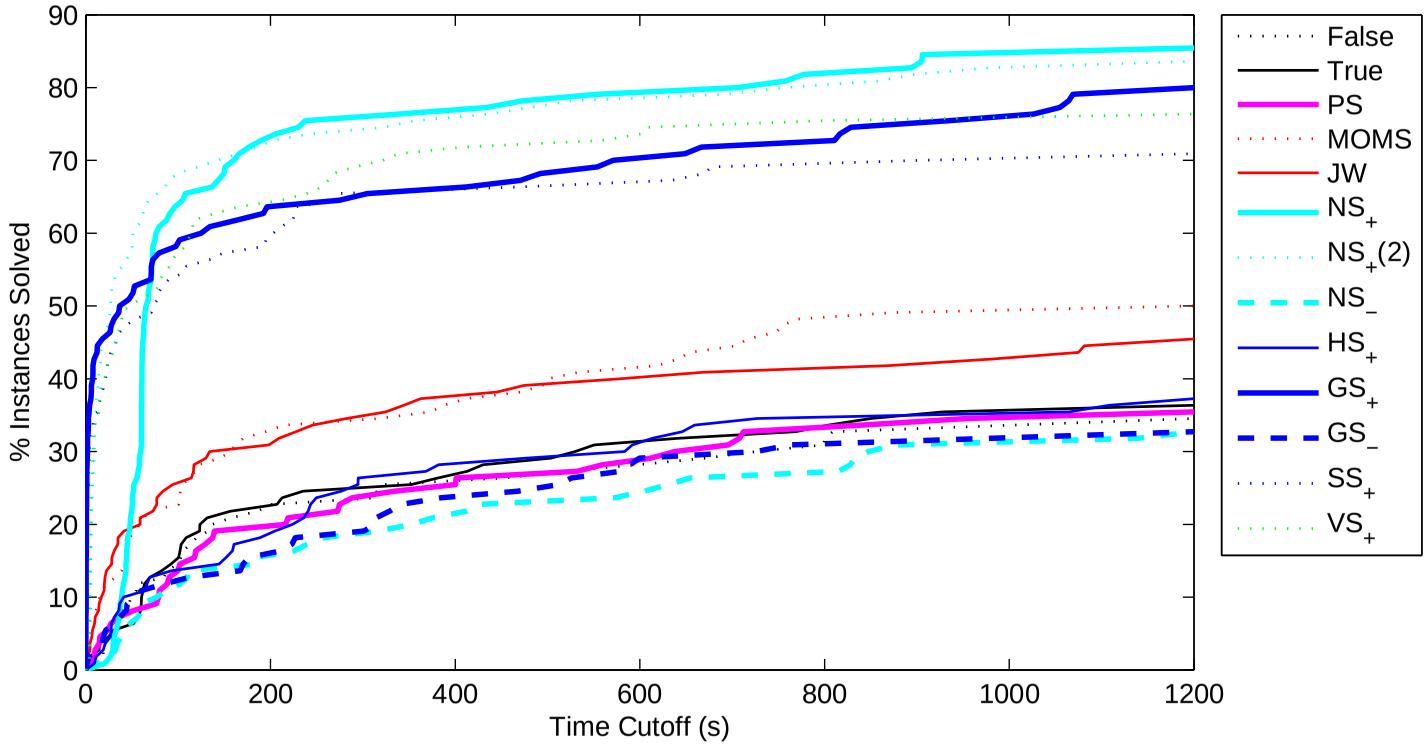

Figure 5.2: Time vs. accuracy for the application satisfiable random set `r3sat`. Jacobian-based NS_+ based on a system-of-equations and the gradient-based GS_+ easily outperform the baseline MiniSAT branching rules. Always branching on *True* and always branching on *False* have nearly identical results, and enabling phase-saving has virtually no benefit.

The two greedy dynamic heuristics JW and MOMS fared much better, solving 45% and 50% of instances respectively. Since all the clauses in this benchmark are at most three literals in length, and the distribution of variables across them is uniform, these greedy methods work quite well for detecting imbalances between the two polarities of the decision variable. Both methods perform similarly for easy to medium instances, but MOMS quickly outpaces JW for more difficult instances. Since both heuristics have the same complexity, it is not immediately apparent why MOMS outperforms JW. We note, however, that the curve for JW is much smoother, likely due to the fact that this heuristic considers a weighted sum of the clauses rather than simply going for the polarity which appears in the maximum number of shortest clauses.

The worst performing heuristics are GS_- and NS_- (both solving only 32% of the instances) which lag behind all other algorithms on virtually every instance. These are fail-first heuristics, and it is evident that their predictions are the complete opposite of what we seek in this benchmark. The matrix-based NS_- is likely more accurate at computing the fail-first preference vector than its gradient-based peer, and hence performs even worse on medium instances. We can also conclude, that a fail-first approach does not generate sufficiently strong learned clauses to help MiniSAT find the model quicker on random formulas. In contrast, the two best performing heuristics turned out to be their promise-based duals GS_+ and NS_+, solving 80% and 85% of instances respectively. This makes sense, as the gradient-based methods are conceptually similar to incomplete stochastic solvers which excel at random satisfiable formulas. The difference between our method and incomplete solvers is that the optimization is deterministic and takes place inside the unit hypercube, rather than on its corners. The system-of-equations approach, while computationally expensive, is even more accurate since it is the most natural translation of the CNF formula into a real-valued optimization problem. Unlike the gradient-based methods, NS_+ tries to "satisfy" all real-valued clauses simultaneously, rather than minimizing their sum. This method is nearly as accurate even when the number of iterations is reduced: the curve NS_+2 is the same algorithm as NS_+ but with only 10 instead of 100 iterations, which gives it a superiour performance on virtually all instances, including the easy ones. Notice that the curves for many of our heuristics rise almost vertically right after the completion of the optimization, which indicates that the heuristic is nearly optimal and the solver almost never backtracks. This is especially true for easy instances. GS_+, SS_+, and VS_+ solve more instances in 5 seconds than the baseline MiniSAT (branching on $False$ with phase-saving) solves in 20 minutes.

The most disappointing results were for the Hessian-based HS_+ heuristic which came in at only 37% of instances solved. The curve for this heuristic is omitted from the graph for clarity, as it essentially overlaps the phase-saving curve. We expected this algorithm to perform as well as the gradient-based approach, but in fewer iterations (after all, both algorithms optimize the same objective function, with the difference that HS_+ uses a second-order local search method which typically converges faster). The algorithm worked well on isolated instances during preliminary testing, but offered only a negligible 1% boost over the baseline MiniSAT branching rules. Upon closer examination of the optimization process, it became apparent that Newton's method, which is the driving force behind HS_+, became trapped in "plateaus"

within the unit hypercube, and unlike its gradient-based peer, was unable to drop down to lower levels sufficiently fast. Decreasing the step size ($\lambda = 0.01$, $\lambda = 0.001$, and $\lambda = 0.00001$) did not remedy the problem across the benchmark. Nonetheless (as will be demonstrated shortly), the method performs reasonably well on crafted instances. We can only conclude that a second-order approximation is not the best optimization technique for conflict potentials stemming from randomly-generated 3-SAT instances. The local quadratic approximation ends up tossing the optimization steps around aimlessly.

It is interesting to note that reducing the number of iterations of NS_+ tenfold had virtually no impact on its accuracy. Inspired by this observation, we decided to investigate the relationship between the number of iterations a static heuristic performs versus its accuracy. Specifically, we measure the accuracy as the percentage of instances solved within the 1200 second time limit. The intuition is that more iterations should yield better results (that is, more instances solved). This is indeed the trend we observed in our preliminary tests, and our next experiment quantifies these results. We focus our attention on algorithms NS_+ and GS_+ which performed best on the random satisfiable set. The two heuristics represent the system-of-equations approach and the gradient-based approach respectively. To investigate how the accuracy of these two methods depends on the number of iterations, we conducted an additional series of full solver runs on `r3sat` for each heuristic, gradually raising the number of optimization steps while keeping the time limits on pre-precessing and search constant. The ranges we examined were from 1 to 250 iterations for NS_+, and from 1 to 20,000 for GS_+ (recall that gradient-based methods are much cheaper to compute). For each run, we recorded the net number of instances solved.

The accuracy results for NS_+ and GS_+ are presented in Figure 5.3. First, notice that the horizontal axis is logarithmic. Several interesting observations can be made. The matrix-based NS_+ yields 65% accuracy after just one iteration, and quickly converges to about 83% accuracy in only 100 iterations. The gradient-based GS_+, on the other hand, results in a 54% accuracy after one iteration, and steadily increases to about 79% for 20,000 iterations. Since the data points for GS_+ appear to lie on a roughly straight line, the convergence rate of this heuristic is approximately logarithmic in this region. Note that the leftmost point on each curve corresponds to a single iteration; the "zero iterations" point corresponds to reverting to MiniSAT's default branching rules which yield about 35% accuracy on this test set. This means that even a single iteration of NS_+ and GS_+ increases the accuracy from 35% to 65% and 54% respectively. Considering that the gradient-based methods are orders of magnitude cheaper to compute than the matrix-based methods and scale much better for larger problems, they are the preferred choice for the crafted and application benchmarks. For example, to achieve an accuracy of 75%, NS_+ executes about 5 iterations at an average cost of 0.6s per iteration for a total pre-processing time of 3s. To achieve the same accuracy, GS_+ executes 5,000 iterations at an average cost of 0.14ms per iteration for a total cost of only 0.7s. We should also point out that the two accuracy curves are not monotone, which we attribute to "snapping" (rounding) the real-valued estimate to the nearest Boolean assignment. While the continuous objective function may be well-optimized at a certain point, its value may not be as

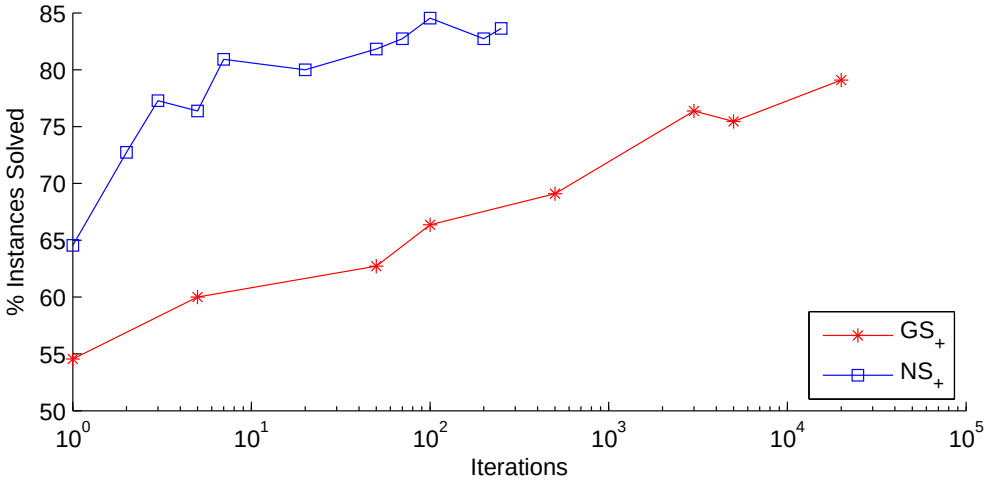

Figure 5.3: The accuracy of the two most successful heuristics: NS_+ (system of equations solved with Newton's method) and GS_+ (linear conflict potential solved by gradient descent) as a function of the number of optimization iterations. The results were obtained on the random test set `r3sat`. NS_+ reaches a plateau of about 83% in only 100 iterations, while GS_+ steadily improves from about 54% to 79% in 20,000 iterations. Notice that zero iterations (not shown on the graph) corresponds to reverting to MiniSAT's default branching behaviour, which has accuracy of about 35% on this test set.

good at the nearest Boolean assignment.

The time vs. accuracy graph for the crafted test set is presented in Figure 5.4, and the detailed breakdown of the results on individual benchmarks is summarized in Table 5.4. Surprisingly, the default MiniSAT branching rules beat all other heuristics on virtually all benchmarks with a total of 97 out of 224 instances solved. The stochastic vote method and always branching on $True$ trail behind at 92 and 91 instances respectively. While slightly more accurate in the long run, the stochastic vote heuristic incurs a heavy pre-processing penalty which degrades its performance on easy and medium instances. With this observation factored in, the gradient-based GS_+ is a more reliable candidate as it performs better across all instances for a total of 90 instances solved. The performance of GS_+ closely trails the performance of always branching on $True$. MOMS and Jeroslow-Wang underperform all of our gradient-based promise methods at 82 and 84 solved instances respectively. Interestingly, always branching on $False$ has the worst accuracy of all aforementioned heuristics. At 81 instances solved, always branching on $False$ underperforms always branching on $True$ by 10 instances (mostly due to the benchmarks `battleship_sat` and `vanderwaerden`). This difference in accuracy can be attributed to the fact that, unlike random instances, crafted instances are likely to have a bias toward either polarity. In this case, always branching on $False$ (the default polarity of MiniSAT) is actually significantly worse than always branching on $True$.

The time vs. accuracy graph for the application test set is presented in Figure 5.5 and the detailed breakdown of the results on individual benchmarks is summarized in Table 5.4. Similarly to the crafted set, the net accuracy of each heuristic is within

a narrow range (from 43% to 47%). However, two of our gradient-based methods (linear and sigmoid) edge out the baseline MiniSAT by 2 and 5 instances respectively. The sigmoid method SS_+ comes in first at 68 out of 142 instances solved, with GS_+ trailing behind at 65 instances solved. For SS_+, the bulk of the accuracy boost comes from the `slp_aes` cryptography benchmark where it solved 4 times as many instances as baseline MiniSAT did. As with the crafted set, the accuracy of SS_+ comes at the cost of a hefty pre-processing penalty, which weakens its performance on easy and medium instances. GS_+, on the other hand, closely trails the performance of the default branching rules, and finally edges out on the hard instances. The remaining algorithms come in at 61 instances solved and generally underperform the three leading heuristics across all instances with the fail-first GS_- substantially lagging behind despite its short pre-processing time. Interestingly, always branching on $True$ and always branching on $False$ are virtually indistinguishable in terms of accuracy. This is somewhat surprising, as we expected real-world instances to have a clearly defined polarity bias. Since real-world instances are of particular importance to SAT research, we wanted to investigate the three winning heuristics further. We conducted additional experiments with the time limit raised to 1 hour. Unfortunately, we did not observe any significant changes in accuracy with the increased time limit. Both SS_+ and GS_+ solved an additional 7 instances each, while the baseline MiniSAT solved another 8 instances bringing the totals to 75, 72, and 71 solved instances respectively.

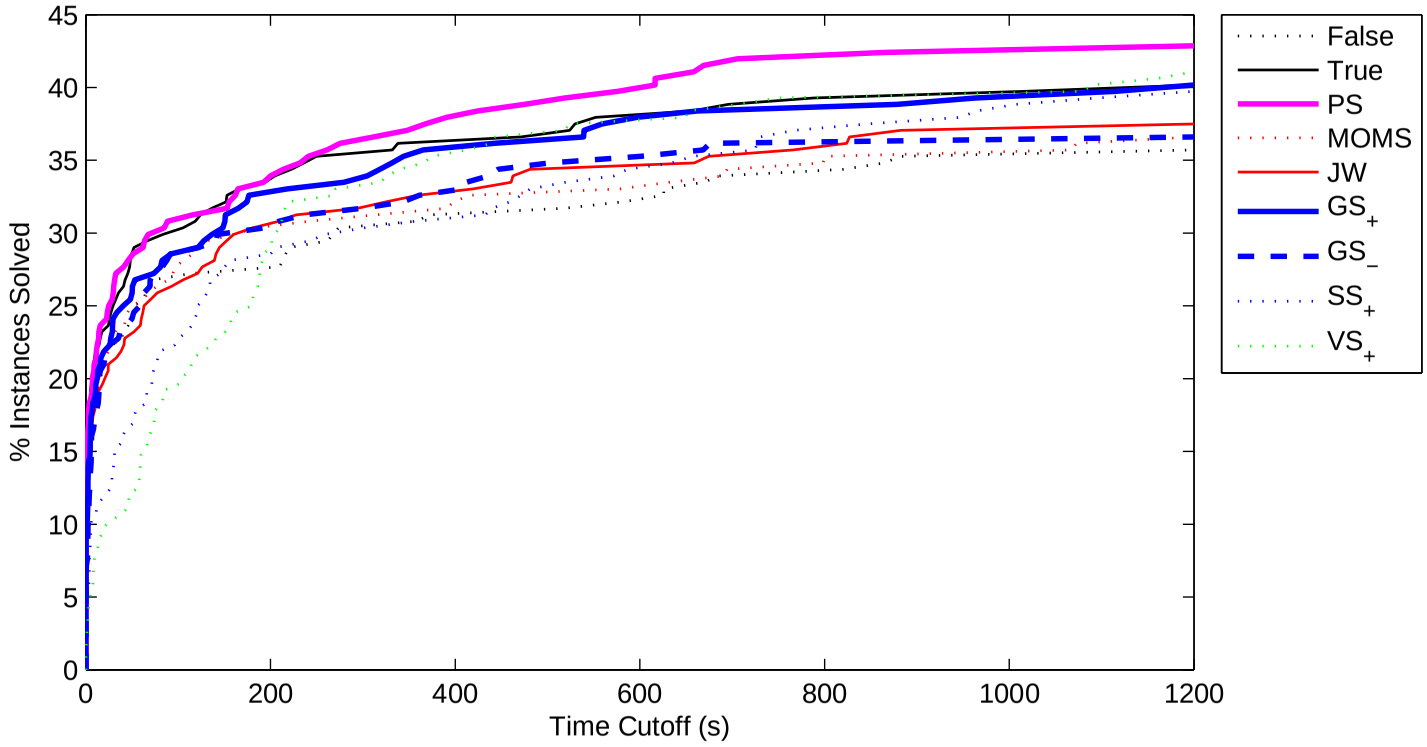

Figure 5.4: Time vs. accuracy for the crafted test set. This is the only set where the default behaviour of MiniSAT (always branching on *False* with phase-saving enabled) significantly outperforms all other heuristics. The stochastic vote algorithm VS_+ comes in second at the cost of more pre-processing. The gradient method GS_+ is slightly worse than always branching on *True*, which in this set outperforms always branching on *False*. The curves for the matrix-based methods are not shown because the benchmarks contained too many instances that were too large to reliably apply the heuristics.

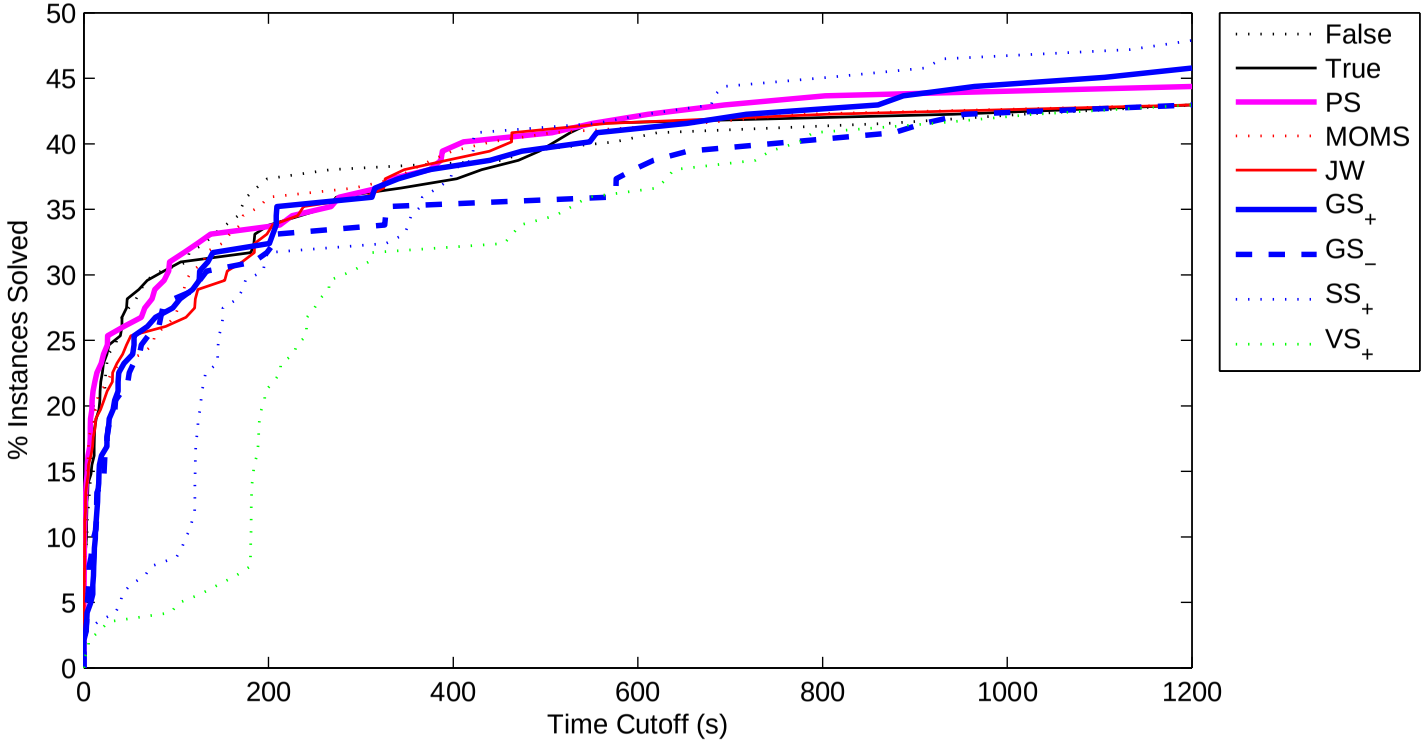

Figure 5.5: Time vs. accuracy for the application test set. The sigmoid-based heuristic SS_+ solves the most instances at the cost of more pre-processing (and thus, worse performance on easy instances). The gradient-based GS_+ comes in second, but has better overall performance. The curves for the matrix-based methods are not shown because the benchmarks contained too many instances that were too large to reliably apply the heuristics.

There are several possible explanations why our heuristics do not drastically improve the accuracy of MiniSAT on the crafted and application benchmarks. The first factor is the size of the instances. More variables and more clauses means increased computational cost of the heuristic and fewer iterations executed (our matrix-based algorithms have timed out after completing as little as 3 optimization steps on some of the larger instances). It is also possible that larger instances require far more optimization iterations than we expect. Secondly, with the increased number of variables, there is an increased room for error; even if the heuristic gets 80% of the polarities right, an instance with over half a million variables (such as those in the `bioinfo` benchmark) can easily trap the solver in unfavourable regions of the search space if it simply branches on a few incorrectly predicted variables near the root of the search tree. We believe discrepancy based search (described in Chapter 3) may alleviate this issue. Thirdly, the composition of the instance matters. Instances from the random set `r3sat` have many nice properties (all clauses have the same length, the number of clauses is a fixed multiple of the number of variables, the distribution of variables across clauses is uniform) which do not hold for real-world instances. Lastly, the constraint structure of the instances likely plays a crucial role. The local information gathered from random instances through iterative optimization is more likely to lead to a solution quickly due to a uniform distribution of solutions across the search space, whereas crafted and application instances can have intricate structures and complex distributions of solutions which may be more difficult to identify using non-linear optimization.

Now that we have identified the winning heuristics in each category, we can compare their performance to state-of-the-art solvers and other solver architectures. The results for the different solvers on the random, crafted, and application benchmarks are presented in Figures 5.6, 5.7, and 5.8. The entries are sorted in increasing accuracy order and the baseline MiniSAT results are highlighted.

We first turn our attention to the random set. Unsurprisingly, the two incomplete solvers `sparrow2011` and `adaptg2wsat` solve all 110 instances, since random satisfiable formulas is the category stochastic local search solvers typically excel at. The complete lookahead solver `march_rw` also performs exceptionally well, solving 104 instances. Complete solvers `lingeling` and `glucose` come in last at 17 and 21 solved instances respectively. Whereas the baseline MiniSAT implementation solves only 39 instances, our systems-of-equations approach NS_+ and the gradient-based GS_+ offer a significant accuracy boost with 87 and 92 instances solved respectively. All of our promise heuristics performed better than `clasp` (57) and MiniSAT equipped with MOMS (55) and JW (50). This makes our most successful combination (MiniSAT with NS_+) the second best complete solver and fourth best solver overall in the random category.

Whereas the results for the random set display a wide gamut of accuracy scores (ranging from 15% to 100%), the range of scores is much narrower in the crafted set (ranging from 29% to 43%) with the lookahead solver `march_rw` performing worst (65 out of 224 solved) and the baseline MiniSAT performing best (97 out of 224 solved). Even though `march_rw` was the top complete solver in the random category, it significantly lags behind on the crafted set, which once again illustrates that the structure

Crafted benchmarks

Benchmark	Total	False	True	PS	MOMS	JW	GS_+	GS_-	SS_+	VS_+	NS_+	NS_-	HS_+
automata_sync	12	8	8	8	7	8	8	8	8	8	-	-	-
battleship_sat	28	9	17	18	8	8	17	8	16	19	16	16	16
battleship_unsat	17	8	8	8	8	8	8	8	8	8	8	8	8
edgematching	30	18	16	18	19	18	15	17	19	19	-	-	-
modcircuits	19	6	4	5	4	5	4	6	4	3	5	4	6
mosoi289_sat	15	15	15	15	15	15	15	15	15	15	15	15	15
mosoi289_unsat	15	0	0	1	0	0	0	0	0	1	-	-	-
sgen_sat	13	1	4	4	1	2	4	1	3	2	4	1	3
sgen_unsat	13	3	3	3	3	3	3	3	3	3	3	3	3
vanderwaerden	62	13	16	17	17	17	16	16	13	14	13	16	16
ALL	224	81	91	**97**	82	84	90	82	89	92	64	63	67

Application benchmarks

Benchmark	Total	False	True	PS	MOMS	JW	GS_+	GS_-	SS_+	VS_+	NS_+	NS_-	HS_+
aes_32	10	5	5	6	5	4	5	4	5	4	4	4	4
aprove09	18	16	16	17	16	18	17	17	17	18	-	-	-
bioinfo	20	20	20	20	20	20	20	20	20	20	-	-	-
bioinstances	30	8	8	8	7	7	8	9	8	7	-	-	7
desgen	19	5	5	5	5	5	6	4	5	7	-	-	-
md5	10	5	5	5	5	5	5	5	5	4	-	-	-
slp_aes	35	2	2	2	3	2	4	2	8	1	-	-	-
ALL	142	61	61	63	61	61	65	61	**68**	61	4	4	11

Table 5.4: The number of instances solved in 1200 seconds by each MiniSAT heuristic in each benchmark. Dashes indicate that a particular benchmark contained too many instances which were too large to reliably apply the heuristic (this mainly applies to matrix-based methods).

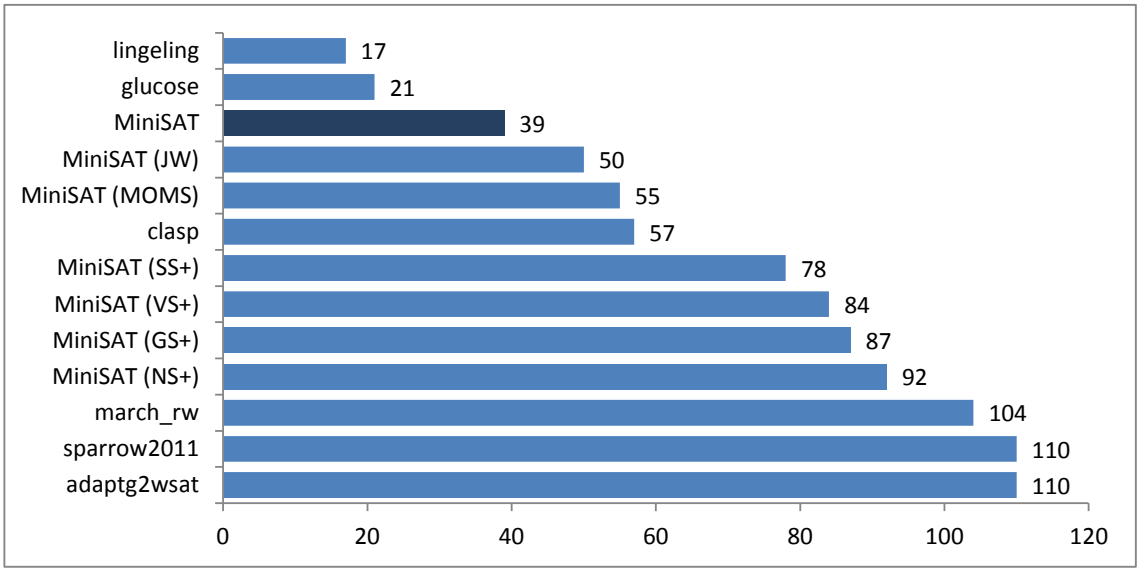

Figure 5.6: The total number of instances solved by different solvers and heuristics in the satisfiable random benchmark (110 instances). The two incomplete solvers easily solve all instances. Our gradient and systems-of-equations heuristics raise the baseline MiniSAT accuracy from 39 to 87 and 92 instances solved respectively.

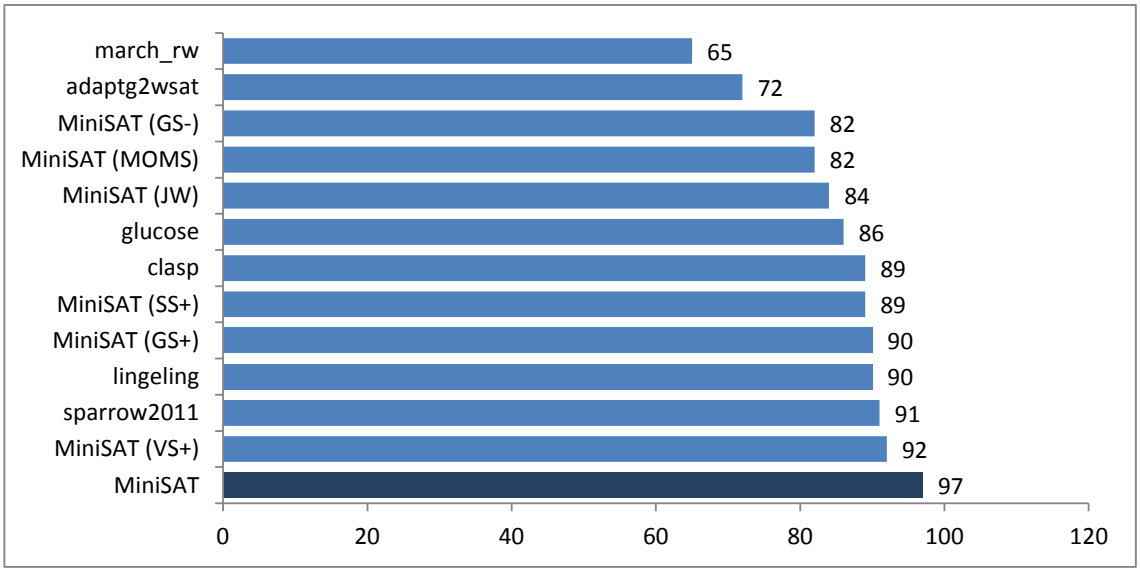

Figure 5.7: The total number of instances solved by different solvers and heuristics in the crafted benchmark (224 instances). Surprisingly, the basic MiniSAT finishes first with 97 instances solved, followed by the stochastic vote method (92), and sparrow2011 (91). The lookahead solver march_rw has the worst accuracy at only 65 instances solved. Our heuristics end up hindering the baseline MiniSAT branching rules, but not by much.

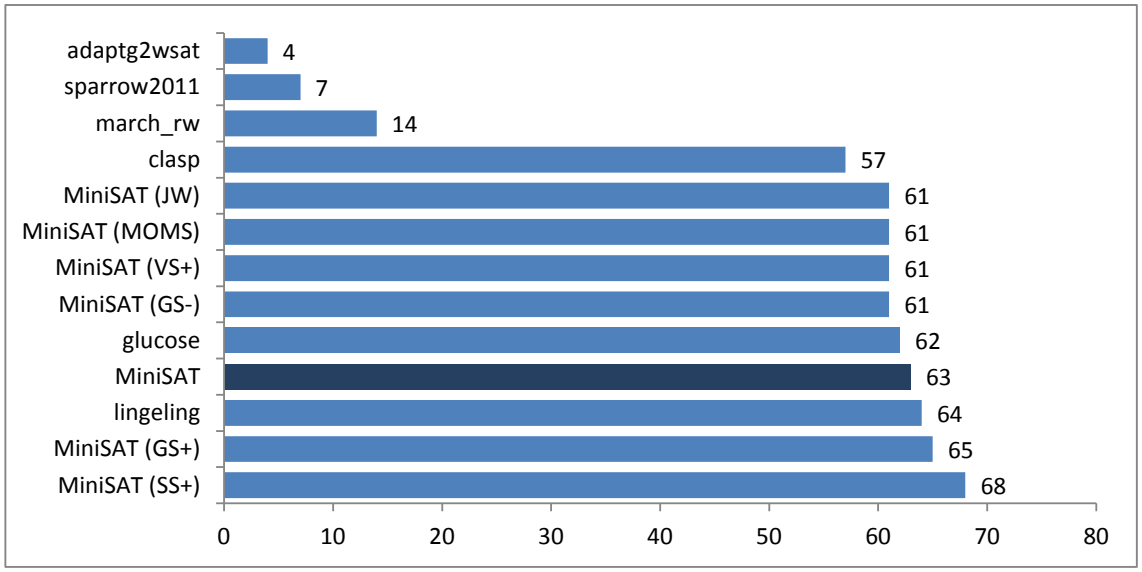

Figure 5.8: The total number of instances solved by different solvers and heuristics in the application benchmark (142 instances). The two incomplete solvers and the lookahead solver perform poorly, solving less than 9% of all instances. Heuristics GS_+ and SS_+ modestly boost the accuracy of MiniSAT by 2 and 5 solved instances respectively. The sigmoid potential heuristic finishes first with 68 instances solved versus 64 for lingeling and 63 for MiniSAT.

of the problem can be crucial to an algorithm's performance. As discussed above, the baseline MiniSAT edges out all of our heuristics; what's even more surprising is that it beats all other solvers. Our stochastic vote method VS_+, while less accurate than baseline MiniSAT comes in second (92 instances solved), surpassing all other solvers by a small margin.

In the application category, the algorithms fall into one of two accuracy ranges: 3% to 10% (`adaptg2wsat`, `sparrow2011`, `march_rw`) and 40% to 48% (the rest). The incomplete and lookahead solvers perform worst, which is the complete opposite of the results observed on the random test set where `adaptg2wsat`, `sparrow2011`, and `march_rw` were the top three solvers. On the other hand, the solver `lingeling` (which was the worst performing algorithm on the random set) comes in third beating the baseline MiniSAT by one instance. Our algorithms SS_+ (68 out of 142 solved) and GS_+ (65 out of 142 solved) dominate not only the baseline MiniSAT, but all other solvers as well. It is also interesting to note that the fail-first heuristic GS_- is not that much worse than its promise-based counterpart (61 vs. 65 instances solved) and just as good as MOMS, JW, and the stochastic vote method.

It should be noted that we cannot expect the incomplete solvers to perform exceptionally well on both the crafted and the application sets, as those contain unsatisfiable instances which stochastic local search cannot solve (unless the solver uses a formula pre-processor capable of determining unsatisfiability before search begins). The fact that none of the methods was able to solve even 50% of the crafted and application sets is a testament to the difficulty of the chosen benchmarks. MiniSAT

also reasserts itself as a robust and competitive framework for SAT research across multiple types of instances. It is also evident that none of our heuristics (including the fail-first ones) significantly degrade the accuracy of the baseline MiniSAT even on both satisfiable and unsatisfiable instances. Furthermore, our promise-based heuristics consistently beat MOMS and JW in all three categories.

We close off our experimental results with a brief discussion of a potential way to improve the accuracy of our approach. The idea for this method stems from so-called *portfolio-based* SAT solvers. Rather than trying to devise a universal algorithm which works well across all SAT instances, portfolio-based solvers incorporate a multitude of different algorithms, each tailored for a particular type of SAT instances. For example, the relatively new solver `ppfolio`[7] did exceptionally well at SAT-Competition 2011 and has won several medals in multiple categories. At its core, `ppfolio` simply runs five different state-of-the-art SAT solvers in parallel and terminates the search as soon as one of the solvers finds a solution. Because different solvers excel at different types of instances, the overall boost in accuracy can often outweigh the cost of running several processes in parallel. Furthermore, modern CPUs usually spread running tasks across multiple cores, which improves the efficiency even further. Inspired by the success of this approach, we examined the possibility of running several of our heuristics in parallel. We found that our heuristics indeed excel at different instances, and that by taking the best result among the lot, we can boost the net accuracy.

We now demonstrate the potential benefits of this approach by simulating different portfolios from our existing results. Consider a portfolio with H different heuristics. On a single-core CPU architecture, the heuristics in the portfolio would have to be interleaved during run-time, which leads us to the following approximation for the best time:

$$T^1_{best} = H \times min\{T_1, T_2, .., T_H\}$$

,

where $T_1, T_2, .., T_H$ are the times for each heuristic to find the solution. We multiply by H to adjust for the interleaving (assuming all heuristics have equal priority). If the result cost exceeds 1200 seconds, the instance is marked as unsolved. For an architecture with H available cores, we simply take the best result among all the cores (no interleaving):

$$T^H_{best} = min\{T_1, T_2, .., T_H\}.$$

This, of course, is also an approximation, as it does not take into account the effects of sharing resources (such as cache, disk, and RAM) between the individual cores.

To get an upper bound on the accuracy boost, we first simulate running *all* of the available heuristics (including $False$, $True$, PS, MOMS, and JW) in parallel on an unlimited number of cores. When compared against the top heuristic in each test set, the number of instances solved rises from 92 to 102 in the random set, from 97 to 105 in the crafted set, and from 68 to 75 in the application set. This corresponds to accuracy rates of 92.7%, 46.9%, and 52.8% respectively (approximately a 3% to 6% gain). Naturally, we would like to find the smallest portfolio that still achieves

[7] http://www.cril.univ-artois.fr/~roussel/ppfolio

Random benchmarks

Portfolio	*Single-core*	*Multi-core*
{NS_+} (reference)	92	92
{NS_+ GS_+}	92	96
{NS_+ VS_+}	95	99
{NS_+ VS_+ GS_+}	95	100
{NS_+ VS_+ GS_+ SS_+ MOMS}	92	102

Crafted benchmarks

Portfolio	*Single-core*	*Multi-core*
{PS} (reference)	97	97
{PS GS_-}	95	98
{PS VS_+}	93	99
{PS GS_- VS_+}	92	101
{PS GS_- VS_+ SS_+}	88	102
{PS GS_- VS_+ SS_+ GS_+ MOMS}	87	105

Application benchmarks

Portfolio	*Single-core*	*Multi-core*
{SS_+} (reference)	68	68
{SS_+ PS}	63	69
{SS_+ GS_+}	63	71
{SS_+ GS_+ GS_-}	60	74
{SS_+ GS_+ GS_- VS_+}	53	75

Table 5.5: The simulated number of instances solved in 1200 seconds by using various heuristic portfolios. In each portfolio, the heuristics run simultaneously, and the best result is reported. The results are simulated for interleaved execution (single-core CPU) and parallel execution (multi-core CPU). For interleaved execution, portfolios almost always degrade the performance. For parallel execution, however, the number of solved instances can be increased from 92 to 102 in the random set, from 97 to 105 in the crafted set, and from 68 to 75 in the application set.

the same accuracy. Table 5.5 summarizes the simulated accuracy scores for different heuristic portfolios. The smallest portfolios that still achieve the top accuracy scores have sizes 5 (random), 6 (crafted) and 4 (application). It is not unreasonable to assume that a computer dedicated to solving SAT instances would have 4 to 6 available cores. Unfortunately, it appears that for single-core CPUs, portfolios of our heuristics degrade the performance in most cases. Nonetheless, the simulated results for the multi-core architecture make the portfolio method the top solver in the crafted and application categories, and the top complete solver in the random category.

5.6 Summary

In this chapter, we empirically evaluated our heuristics on a wide assortment of SAT instances. We obtained the best results on satisfiable random formulas, where our

systems-of-equations approach solved 85% of instances compared to 35% solved by MiniSAT's default branching rules. We found that the gradient method, although less accurate than the system-of-equations method, scales better for large instances, and that variations of the gradient method (sigmoid potential and stochastic vote) can be tuned to perform well on a wide range of benchmarks. While our approach showed no performance boost on crafted instances, we observed a modest gain in accuracy on real-world application instances (47% of instances solved compared to 44% solved by MiniSAT's default branching rules). Nonetheless, MiniSAT equipped with our heuristics is very competitive when compared to state-of-the-art SAT solvers.

In the next chapter, we conclude the thesis with a summary of our results, and a describe future work and potential improvements to our approach.

Chapter 6

Conclusion and Future Work

In this chapter, we summarize the contributions of the thesis, briefly reiterate our approach, and highlight important results. We also state open questions and list some possible directions for future work.

6.1 Conclusion

The seemingly simple question of whether a Boolean formula can ever evaluate to $True$ serves as a gateway between efficient solvers and complex real-world applications ranging from circuit design to bioinformatics. In this thesis, we explored the challenging task of designing fast and accurate value ordering heuristics for solving the Boolean Satisfiability (SAT) problem using backtracking search. The backtracking DPLL algorithm has become the most common framework for modern complete SAT solvers. When branching on a decision variable, choosing the right polarity to explore first can lead to dramatic savings in runtime and resources. Due to the inherent difficulty of deciding which polarity is better, not much research has been done on value ordering in SAT, and only few reliable heuristics are known. In this thesis, we presented three separate methods for constructing value ordering heuristics by tapping into the power of non-linear optimization methods.

Our methods first translate the Boolean formula into a corresponding continuous optimization problem defined on the unit hypercube, whose corners correspond to the 2^n Boolean assignments. Then an iterative non-linear optimization technique is applied to approximate the Boolean assignment with the most desired score; the relative location of the estimate is then used to decide the polarity of the decision variable. The three methods differ in their formulation of the optimization problem and the local search technique used.

The first method models the optimization problem as a system of non-linear equations and attempts to find a root of the system by using Newton's Method. This amounts to repeatedly solving a linear system defined by the Jacobian matrix of the equations. The second method introduces the concept of conflict potentials; these are objective functions which measure the number of clause conflicts within the unit hypercube. The heuristic then uses gradient descent or Newton's Method to op-

timize the conflict potential (the latter involves solving a linear system defined by the Hessian matrix of the conflict potential). A similar idea is used in incomplete solvers; however, our approach optimizes the objective function on the continuous space within the unit hypercube, rather than on its corners. The third method builds on the concept of conflict potentials, but adds a stochastic element to the search. Instead of optimizing a single conflict potentials, the method samples a fixed number of related conflict potentials (perturbed by small random exponents) and lets them vote on a common optimization direction at each iteration. Since the distribution of local extrema is different among the sampled potentials, the heuristic has less chance of getting stuck during the optimization process. We implemented our heuristics in the renowned MiniSAT framework.

All of our heuristics share the unusual property that the polarity estimates are computed for all unassigned variables at once (as opposed to the decision variable only). While computationally expensive, this property allows the heuristic to be used both dynamically and statically.

6.2 Summary of Results

Since it is well known that SAT solvers have varying degrees of accuracy across different classes of instances, we tested our heuristics on random, crafted, and real-world application benchmarks used in recent SAT-Race competitions. The heuristic parameters were selected based on the algorithms' performance on random formulas, and were not otherwise tuned to individual test sets. We measured the accuracy as the number of instances solved within a 1200 second time limit.

We obtained by far the best results on random satisfiable formulas, boosting the baseline MiniSAT accuracy from 35% to 80% (gradient-based approach) and to 85% (matrix-based approach). The matrix-based method NS_+ converges very quickly (in under 100 iterations), but the iterations are costly. The gradient-based GS_+, on the other hand, is cheap to compute, but requires more iterations; we determined that its accuracy increases approximately logarithmically with the number of iterations. We also found that even a single optimization iteration of NS_+ boosts MiniSAT's accuracy from 35% to 65% on the random set.

Our heuristics did not have such a dramatic impact on the crafted and application sets. On the application set, we obtained a modest gain, raising the accuracy of MiniSAT from 44% to 48% with SS_+. Surprisingly, MiniSAT came in first in the crafted category, and none of our heuristics managed to improve its accuracy; on the contrary, our promise-based heuristics degraded the accuracy by as much as 4%.

The sigmoid-based method SS_+ shines in the application category, where it beats not only MiniSAT, but all state-of-the-art solvers. On other benchmarks, its performance is generally similar to that of its linear counterpart GS_+. Nonetheless, its exceptional performance on the cryptography benchmark `slp_aes` confirms that a change in the parametrization of the conflict potential can have a significant impact on the performance of the heuristic.

The Hessian-based heuristic HS_+ performed surprisingly poorly on the random

set (37% accuracy) compared to its peers, but showed average performance on the other two sets. Our expectation that the method would be as accurate as its gradient descent version was shown to be incorrect. It appears that a second-order optimization method does not offer any advantage over plain gradient descent for the types of formulas we examined.

The stochastic vote method VS_+ shows average performance, except on the crafted set where it comes in second, surpassing its peers and `sparrow2011`. It should be noted that this method has a high pre-processing cost which degrades its performance on easy instances compared to other heuristics. However, this method has many parameters, and it is conceivable that a more refined calibration would have yielded better results.

In general, the heavy computational cost of our heuristics makes them less attractive for solving easy instances, as the time spent computing the heuristic can often outweigh the actual search time. For this reason, we focused our attention on medium and hard instances only. The complexity of the gradient-based methods is linear in the length of the formula, which allows them to scale to large instances easily. The matrix-based methods, on the other hand, have cubic complexity in the dimension of the matrix to be solved ($n \times m$ for Jacobian-based methods, $n \times n$ for Hessian-based methods) and do not scale beyond medium-sized instances, even though these matrices are extremely sparse. The largest instances that we were able to use the matrix-based methods on had about 3,000 variables and 60,000 clauses.

Overall, MiniSAT performed reliably across all categories, and our promise-based heuristics either boosted its accuracy (35% to 85% for random, 44% to 48% for application) or lightly degraded it (down to as low as 39% from 43% for crafted). On crafted and application sets, MiniSAT equipped with our heuristics beat all six state-of-the-art solvers. On the random set, MiniSAT equipped with our heuristics comes in fourth behind the lookahead and incomplete solvers, but still surpassing `lingeling`, `glucose`, and `clasp`. Unlike MiniSAT, the other solvers displayed a huge variance in accuracy scores across the different test sets. For example, the incomplete solvers `sparrow2011` and `adaptg2wsat` achieved 100% accuracy on the random set, but under 5% accuracy on real-world instances. Similarly, `lingeling` came in 2nd in the application category but last in the crafted category.

We observed that the promise-based versions of our heuristics always outperformed the fail-first versions. This is contrary to our initial assumption that branching towards conflict-rich regions of the search space would help MiniSAT produce stronger learned clauses and hence speed up the search. Interestingly, our promise-based heuristics appear to be very reliable: they usually increase the accuracy on satisfiable instances and do not (significantly) decrease the accuracy on unsatisfiable instances.

We also showed that the well-known Jeroslow-Wang heuristic can be seen as a single iteration of our gradient-based method GS_+ evaluated at the center of the unit hypercube. Unlike Jeroslow-Wang, GS_+ goes beyond a single iteration, and repeatedly refines the initial guess. This may explain why our promise-based heuristics outperform both Jeroslow-Wang and the related heuristic MOMS on all test sets. Both Jeroslow-Wang and MOMS have similar accuracy scores, with MOMS outperforming Jeroslow-Wang by about 5% on hard random instances, and underperforming

by about 2% on the crafted set.

In our tests, we did not observe a significant advantage of always branching on $False$ (the default MiniSAT polarity) vs. always branching on $True$; in fact, on our crafted set, always branching on $True$ solved 10 instances more. We also observed that enabling phase-saving (the default MiniSAT behaviour) does indeed provide a small accuracy boost (a few percentage points) on most benchmarks.

Lastly, we examined a portfolio-based approach, where many successful heuristics work in parallel until one of them finds the solution. While the results of our simulation are not exact, they suggest that by bundling 4 to 6 heuristics together and running them on separate cores, the accuracy of MiniSAT can be boosted further.

6.3 Future Work

In this thesis, we modified just one component of MiniSAT, namely, the value ordering heuristic. Naturally, this is not the only aspect affecting the efficiency of search. All of MiniSAT's features such as variable ordering, clause learning, clause pruning, and restarts play an important role and affect each other in non-trivial ways. In particular, it would be interesting to modify the variable ordering heuristic to match the strategy of our value ordering heuristics more closely. The default variable ordering heuristic VSIDS is inherently a fail-first heuristic, which may or may not be the best choice when the value ordering heuristic operates in a different mode. It would be insightful to test our value ordering heuristic with other existing variable ordering heuristics, or perhaps devise a new variable ordering heuristic based on the information obtained from the non-linear optimization process. For example, in our stochastic heuristic VS_+, sampled conflict potentials vote on a common direction. The sign of the vote is used to make a polarity choice; its magnitude, however, is not taken into account. Perhaps, the variable which has received the largest number of consenting votes could be branched on next with the intuition that the polarity of that variable has a higher measure of confidence. Similarly, we can extract this "confidence score" from the the gradient-based methods GS_+ and SS_+ by measuring the magnitude of each component of the gradient.

Another potential improvement would be to integrate our value ordering heuristics into a discrepancy-based SAT solver (as described in Chapter 3). Discrepancy-based solvers systematically branch against the value ordering heuristic in a predefined pattern which causes the search tree to be explored more sporadically. This way, a bad decision made high up in the search tree has less chance of trapping the solver in unfavourable regions of the search space. We noticed that in the case of satisfiable instances, the polarity vector computed by our heuristics usually contains a very large fraction of correctly-guessed polarities; unfortunately, a few bad decisions at the top of the search tree can render this good initial guess useless. We speculate that discrepancy-based search could help bring out those correctly-guessed polarities in the long run.

It would also be interesting to apply our heuristics in the closely-related field of Maximum Satisfiability (MAX-SAT) which attempts to find a variable assignment

which satisfies as many clauses as possible. In essence, that is exactly what our non-linear optimization methods are designed to do. The conflict potential (the objective function in two of our methods) is a measure of clause conflicts within the unit hypercube. By minimizing this conflict potential, our heuristics discover Boolean assignments which violate as few clauses as possible (or equivallently, satisfy as many clauses as possible). This would allow us to transform our heuristics into MAX-SAT algorithms.

Appendix A

Sample Implementation

For convenience, we provide a sample C++ implementation of one of the simpler static heuristics GS_+ (or GS_- when run in fail-first mode) based on a linear conflict potential with gradient-based optimization.
(See next page).

```
/* Assume the following global variables have been defined and initialized.
 {int n}: the number of variables in the instance.
 {int m}: the number of clauses in the instance.
 {Clause clauses[m]}: a vector of all the clauses in the instance.
 {bool preferred_polarities[n]}: a vector to store the preferred polarities.
 The Clause class is essentially a vector of Literal objects.
 The Literal class represents a single literal and has the following methods:
    {int  Literal::var()}: returns the variable of the literal.
    {bool Literal::is_positive()}: returns true iff the literal's variable is not negated.
*/

/* Computes the gradient of the linear conflict potential at point x[n].
   The result is stored in grad[n]. Only to be used with static heuristics. */
void gradR(double x[], double grad[]) {
    for (int i=0; i<n; i++) grad[i] = 0;
    for (int i=0; i<m; i++) {               // Iterate over clauses.
        Clause c = clauses[i];
        for (int j=0; j<c.size(); j++) {    // Iterate over the literals in the clause.

            Literal l = c[j];               // Differentiate clause c with respect to literal l.
            double product = l.is_positive() ? -1 : 1;
            for (int k=0; k<c.size(); k++) {
                if (j==k) continue;         // Do not differentiate the clause c against l twice.
                Literal ll = c[k];
                product *= ll.is_positive() ? (1-x[ll.var()]) : x[ll.var()];
            }

            grad[l.var()] += product;       // Add the clause's contribution.

        }
    }
}

/*  Fills the global vector preferred_polarities[n] with
    the preferred polarities for each variable computed by performing
    gradient descent (promise_mode=true) or ascent (promise_mode=false) on the
    linear conflict potential. Static version only. */
void GS(bool promise_mode, int max_iterations, double step_size) {

    double x[n], grad[n];
    for (int i=0; i<n; i++) x[i] = 0.5;     // Start search at the center of the unit hypercube.

    if (promise_mode) step_size *= -1.0;    // In promise mode, we minimize the conflict potential.

    int iteration = 0;
    while (++iteration < max_iterations) {

        gradR(x, grad);                     // Compute the gradient at the current estimate.
        for (int i=0; i<n; i++) {
            x[i] += step_size * grad[i];    // Perform gradient descent or ascent.
            x[i] = max(0, min(1, x[i]));    // Restrict the estimate to the search domain.
        }
    }

    for (int i=0; i<nVars(); i++)
        preferred_polarities[i] = x[i] > 0.5;   // Snap to the nearest corner (Boolean assignment).
}
```

Bibliography

[AKS11] Tanbir Ahmed, Oliver Kullmann, and Hunter Snevily. On the van der Waerden numbers $w(2;3,t)$. Technical Report arXiv:1102.5433v1 [math.CO], arXiv, February 2011.

[AS09] G. Audemard and L. Simon. Glucose: a solver that predicts learnt clauses quality. *SAT Competition*, pages 7–8, 2009.

[Atk09] K.E. Atkinson. *An introduction to numerical analysis*. Wiley-India, 2009.

[BB92] M. Buro and H.K. Büning. *Report on a SAT competition*. Fachbereich Math.-Informatik, Univ. Gesamthochschule, 1992.

[Bie08] A. Biere. Picosat essentials. *Journal on Satisfiability, Boolean Modeling and Computation*, 4(75-97):45, 2008.

[Bie10] A. Biere. Lingeling, plingeling, picosat and precosat at sat race 2010. *FMV Report Series Technical Report*, 10(1), 2010.

[BJ09] M. Bonet and K. John. Efficiently calculating evolutionary tree measures using sat. *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 4–17, 2009.

[BN] F. Baader and T. Nipkow. Term rewriting and all that. 1998.

[BP10] J. Boyar and R. Peralta. A new combinational logic minimization technique with applications to cryptology. *Experimental Algorithms*, pages 178–189, 2010.

[BS01] R. Bruni and A. Sassano. Restoring satisfiability or maintaining unsatisfiability by finding small unsatisfiable subformulae. In *In LICS Workshop on Theory and Applications of Satisfiability Testing*, page 2001, 2001.

[BW92] D. Basin and T. Walsh. Difference unification. 1992.

[CA93] J.M. Crawford and L.D. Auton. Experimental results on the crossover point in satisfiability problems. In *Proceedings of the National Conference on Artificial Intelligence*, pages 21–21. Citeseer, 1993.

[CFT10] Fabien Corblin, Eric Fanchon, and Laurent Trilling. Applications of a formal approach to decipher discrete genetic networks. *BMC Bioinformatics*, 11:385, 2010.

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

[Dav06] T.A. Davis. *Direct methods for sparse linear systems*, volume 2. Society for Industrial Mathematics, 2006.

[DBM00] O. Dubois, Y. Boufkhad, and J. Mandler. Typical random 3-sat formulae and the satisfiability threshold. In *Proceedings of the eleventh annual ACM-SIAM symposium on Discrete algorithms*, pages 126–127. Society for Industrial and Applied Mathematics, 2000.

[DD04] G. Dequen and O. Dubois. Kcnfs: An efficient solver for random k-sat formulae. In *Theory and Applications of Satisfiability Testing*, pages 305–306. Springer, 2004.

[DK89] J. De Kleer. A comparison of atms and csp techniques. In *Proceedings of the Eleventh International Joint Conference on Artificial Intelligence*, volume 1, pages 290–296. Citeseer, 1989.

[DLL62] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem-proving. *Communications of the ACM*, 5(7):394–397, 1962.

[DP60] M. Davis and H. Putnam. A computing procedure for quantification theory. *Journal of the ACM (JACM)*, 7(3):201–215, 1960.

[EB05] Niklas Een and Armin Biere. Effective preprocessing in sat through variable and clause elimination. In *In proc. SAT05, volume 3569 of LNCS*, pages 61–75. Springer, 2005.

[ES04] N. Eén and N. Sörensson. An extensible sat-solver. In Enrico Giunchiglia and Armando Tacchella, editors, *Theory and Applications of Satisfiability Testing*, volume 2919 of *Lecture Notes in Computer Science*, pages 333–336. Springer Berlin / Heidelberg, 2004.

[FK05] D. Furcy and S. Koenig. Limited discrepancy beam search. In *International joint conference on artificial intelligence*, volume 19, page 125. Lawrence Erlbaum Associates ltd, 2005.

[Fre95] Jon William Freeman. *Improvements to propositional satisfiability search algorithms*. PhD thesis, Philadelphia, PA, USA, 1995. UMI Order No. GAX95-32175.

[FSK10] C. Fuhs and P. Schneider-Kamp. Synthesizing shortest linear straight-line programs over gf (2) using sat. *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 71–84, 2010.

[Fuh09] C. Fuhs. Sat instances for termination analysis with aprove. *SAT 2009 competitive events booklet: preliminary version*, page 63, 2009.

[G+89] F. Glover et al. Tabu search-part i. *ORSA Journal on computing*, 1(3):190–206, 1989.

[Gal77] Z. Galil. On the complexity of regular resolution and the davis-putnam procedure. *Theoretical Computer Science*, 4(1):23–46, 1977.

[Gen02a] I.P. Gent. Arc consistency in sat. In *ECAI*, volume 2, pages 121–125, 2002.

[Gen02b] I.P. Gent. Arc consistency in sat. In *ECAI*, volume 2, pages 121–125, 2002.

[Gin93] M.L. Ginsberg. Dynamic backtracking. *Arxiv preprint cs/9308101*, 1993.

[GK10] M. Gwynne and O. Kullmann. *Attacking AES via SAT*. PhD thesis, BSc Dissertation (Swansea), 2010.

[GKNS07] M. Gebser, B. Kaufmann, A. Neumann, and T. Schaub. clasp: A conflict-driven answer set solver. *Logic Programming and Nonmonotonic Reasoning*, pages 260–265, 2007.

[Hak85] A. Haken. The intractability of resolution. *Theoretical Computer Science*, 39:297–308, 1985.

[HDVZVM05] M. Heule, M. Dufour, J. Van Zwieten, and H. Van Maaren. March_eq: Implementing additional reasoning into an efficient look-ahead sat solver. In *Theory and Applications of Satisfiability Testing*, pages 898–898. Springer, 2005.

[Heu09] M.J.H. Heule. Solving edge-matching problems with satisfiability solvers. *SAT 2009 competitive events booklet: preliminary version*, page 69, 2009.

[HG95] W.D. Harvey and M.L. Ginsberg. Limited discrepancy search. In *International Joint Conference on Artificial Intelligence*, volume 14, pages 607–615. Citeseer, 1995.

[HMBM08] E. Hsu, C. Muise, J. Beck, and S. McIlraith. Probabilistically estimating backbones and variable bias: Experimental overview. In *Principles and Practice of Constraint Programming*, pages 613–617. Springer, 2008.

[Hoo95] J.N. Hooker. Testing heuristics: We have it all wrong. *Journal of Heuristics*, 1(1):33–42, 1995.

[Hoo99] H.H. Hoos. On the run-time behaviour of stochastic local search algorithms for sat. In *Proceedings of the national conference on artificial intelligence*, pages 661–666. JOHN WILEY & SONS LTD, 1999.

[Hoo02] H.H. Hoos. An adaptive noise mechanism for walksat. In *Proceedings of the national conference on artificial intelligence*, pages 655–660. Menlo Park, CA; Cambridge, MA; London; AAAI Press; MIT Press; 1999, 2002.

[HTH02] F. Hutter, D.A.D. Tompkins, and H.H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. *Lecture notes in computer science*, 2470:233–248, 2002.

[Hua07] Jinbo Huang. The effect of restarts on the efficiency of clause learning, 2007.

[HV95] J. N. Hooker and V. Vinay. Branching rules for satisfiability. *Journal of Automated Reasoning*, 15:359–383, 1995.

[HvM08] M.J.H. Heule and H. van Maaren. Whose side are you on? *Journal on Satisfiability, Boolean Modeling and Computation*, 4:117–148, 2008.

[HvZD11] M.J.H. Heule, J.E. van Zwieten, and M. Dufour. marchrw sat solver. http://www.st.ewi.tudelft.nl/sat/, 2011.

[JG79] D.S. Johnson and M.R. Garey. Computers and intractability: A guide to the theory of np-completeness. *Freeman&Co, San Francisco*, 1979.

[Joh73] David S. Johnson. Approximation algorithms for combinatorial problems. In *Proceedings of the fifth annual ACM symposium on Theory of computing*, STOC '73, pages 38–49, New York, NY, USA, 1973. ACM.

[JW90] Robert G. Jeroslow and Jinchang Wang. Solving propositional satisfiability problems. *Annals of Mathematics and Artificial Intelligence*, 1:167–187, 1990. 10.1007/BF01531077.

[KGJV83] S. Kirkpatrick, C.D. Gelatt Jr, and M.P. Vecchi. Optimization by simulated annealing. *science*, 220(4598):671–680, 1983.

[KKL06] A.C. Kaporis, L.M. Kirousis, and E.G. Lalas. The probabilistic analysis of a greedy satisfiability algorithm. *Random Structures & Algorithms*, 28(4):444–480, 2006.

[KKY09] A. Kojevnikov, A. Kulikov, and G. Yaroslavtsev. Finding efficient circuits using sat-solvers. *Theory and Applications of Satisfiability Testing-SAT 2009*, pages 32–44, 2009.

[Kor96] R.E. Korf. Improved limited discrepancy search. In *Proceedings of the National Conference on Artificial Intelligence*, pages 286–291, 1996.

[KSS10] L. Kroc, A. Sabharwal, and B. Selman. An empirical study of optimal noise and runtime distributions in local search. *Theory and Applications of Satisfiability Testing–SAT 2010*, pages 346–351, 2010.

[Kul02] O. Kullmann. Investigating the behaviour of a sat solver on random formulas. *Submitted to Annals of Mathematics and Artificial Intelligence*, 2002.

[LA97] Chu Min Li and Anbulagan Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of the 15th international joint conference on Artifical intelligence - Volume 1*, pages 366–371, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[Lev73] L. Levin. Universalnyıe perebornyıe zadachi (universal search problems: in russian). *Problemy Peredachi Informatsii*, 9(3):265–266, 1973.

[LH05] C. Li and W. Huang. Diversification and determinism in local search for satisfiability. In *Theory and Applications of Satisfiability Testing*, pages 158–172. Springer, 2005.

[Lib00] Paolo Liberatore. On the complexity of choosing the branching literal in dpll. *Artificial Intelligence*, 116:200–0, 2000.

[LSZ93] Michael Luby, Alistair Sinclair, and David Zuckerman. Optimal speedup of las vegas algorithms. *Information Processing Letters*, 47:173–180, 1993.

[LWZ07] C. Li, W. Wei, and H. Zhang. Combining adaptive noise and look-ahead in local search for sat. *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 121–133, 2007.

[MDWH+10] S. Mijnders, B. De Wilde, M.J.H. Heule, D. Mitchell, and E. Ternovska. Symbiosis of search and heuristics for random 3-sat. In *Proceedings of the Third International Workshop on Logic and Search (LaSh 2010)*, pages 231–240. Academic Service, 2010.

[Mes97] P. Meseguer. Interleaved depth-first search. In *International Joint Conference on Artificial Intelligence*, volume 15, pages 1382–1387. Lawrence Erlbaum Associates ltd, 1997.

[MJPL90] S. Minton, M.D. Johnston, A.B. Philips, and P. Laird. Solving large-scale constraint satisfaction and scheduling problems using a heuristic repair method. In *Proceedings of the eighth National conference on Artificial intelligence*, pages 17–24, 1990.

[MMZ+01] M.W. Moskewicz, C.F. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient sat solver. In *Proceedings of the 38th annual Design Automation Conference*, pages 530–535. ACM, 2001.

[Mor93] P. Morris. The breakout method for escaping from local minima. In *Proceedings of the national conference on artificial intelligence*, pages 40–40. JOHN WILEY & SONS LTD, 1993.

[Mos11] A. Mosoi. Grid colouring challenge. SAT 2011 benchmark description, 2011.

[Ms99] Joo Marques-silva. The impact of branching heuristics in propositional satisfiability algorithms. In *In 9th Portuguese Conference on Artificial Intelligence (EPIA*, pages 62–74, 1999.

[MSK97] D. McAllester, B. Selman, and H. Kautz. Evidence for invariants in local search. In *Proceedings of the national conference on artificial intelligence*, pages 321–326. JOHN WILEY & SONS LTD, 1997.

[MVVW06] D. Mpekas, M. Van Vlaardingen, and S. Wieringa. The first steps to a hybrid sat solver. *Delft University of Technology*, 2006.

[NG02] Y. Novikov and E. Goldberg. Berkmin: a fast and robust sat-solver. *Proceedings of Design, Automation and Test in Europe (DATE02)*, page 0142, 2002.

[PD07a] K. Pipatsrisawat and A. Darwiche. A lightweight component caching scheme for satisfiability solvers. *Theory and Applications of Satisfiability Testing–SAT 2007*, pages 294–299, 2007.

[PD07b] K. Pipatsrisawat and A. Darwiche. Rsat 2.0: Sat solver description. *Solver description, SAT competition*, 54:9–10, 2007.

[Pre96] D. Pretolani. Efficiency and stability of hypergraph sat algorithms. *Cliques, coloring, and satisfiability: second DIMACS implementation challenge, October 11-13, 1993*, 26:479, 1996.

[PRI12] S.J.D. PRINCE. Computer vision: models, learning, and inference. *Recherche*, 67:02, 2012.

[Ros04] V. Rosta. Ramsey theory applications. *the electronic journal of combinatorics*, 1000(0):DS13–Dec, 2004.

[SH11] Evgeny Skvortsov and Carl Hubinette. Moving battleship problem. SAT 2011 benchmark description, 2011.

[SJ09a] M. Sesum and P. Janicic. Generator of sat instances of unknown satisfiability. *SAT 2009 competitive events booklet: preliminary version*, page 87, 2009.

[SJ09b] M. Sesum and P. Janicic. Generator of satisfiable sat instances. *SAT 2009 competitive events booklet: preliminary version*, page 85, 2009.

[SKC93] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. *Cliques, coloring, and satisfiability: Second DIMACS implementation challenge*, 26:521–532, 1993.

[SKC94] B. Selman, H.A. Kautz, and B. Cohen. Noise strategies for improving local search. In *Proceedings of the national conference on artificial intelligence*, pages 337–337. JOHN WILEY & SONS LTD, 1994.

[SLM92] B. Selman, H. Levesque, and D. Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the tenth national conference on Artificial intelligence*, pages 440–446, 1992.

[Spe09] I. Spence. sgen1: A generator of small, difficult satisfiability benchmarks. *SAT 2009 competitive events booklet: preliminary version*, page 95, 2009.

[SS11] K.A. Sakallah and L. Simon. Proceedings of the 14th international conference on theory and application of satisfiability testing. 2011.

[ST11] E. Skvortsov and E. Tipikin. Experimental study of the shortest reset word of random automata. *Implementation and Application of Automata*, pages 290–298, 2011.

[Tse68] G S Tseitin. *Studies in constructive mathematics and mathematical logic*, 8(115-125):234–259, 1968.

[VGTUoC95] A. Van Gelder, Y. Tsuji, and Santa Cruz. Computer Research Laboratory University of California. *Satisfiability testing with more reasoning and less guessing*. Computer Research Laboratory,[University of California, Santa Cruz, 1995.

[Wal97] T. Walsh. Depth-bounded discrepancy search. In *International joint conference on artificial intelligence*, volume 15, pages 1388–1395. LAWRENCE ERLBAUM ASSOCIATES LTD, 1997.

[Wal00] T. Walsh. Sat v csp. *Principles and Practice of Constraint Programming–CP 2000*, pages 441–456, 2000.

[War99] J.P. Warners. *Nonlinear approaches to satisfiability problems*. Univ., 1999.

[ZHG11] M. Zhou, F. He, and M. Gu. An efficient resolution based algorithm for sat. In *Theoretical Aspects of Software Engineering (TASE), 2011 Fifth International Symposium on*, pages 60–67. IEEE, 2011.