A Study of Divide and Distribute Fixed Weights and its Variants

Cayden R. Codel
ccodel@andrew.cmu.edu

Advised by Marijn J. H. Heule
mheule@cs.cmu.edu

Carnegie Mellon University, Pittsburgh, PA

Abstract. Divide and Distribute Fixed Weights (DDFW) is a stochastic local search Boolean satisfiability (SAT) solver that has achieved a high level of performance on select problem instances, including the Pythagorean triples instance for $n = 7824$. Yet despite its success, DDFW has received little research interest, and its initial results are out of date with respect to more modern SAT benchmarks. To address both those research needs, we examine DDFW in depth and propose modifications to the algorithm based off of ideas from similar SAT solvers such as ProbSAT and SAPS. We then take these modifications and test DDFW against a set of modern hard benchmarks. We present three main findings. The first is a confirmation that a greedy variable selection process in focused search is optimal for DDFW. The second is that a linear weight transfer rule is more effective than a fixed additive one. The third is that it is more effective for unsatisfied clauses to borrow clause weight from its entire neighborhood rather than a singular clause in local minima, as it does in the original algorithm. The second and third strategies produce modifications of the DDFW algorithm that perform 30-50% better than the original, with 3x higher solve rates.

1 Introduction

As the first proven NP-hard problem [11], the Boolean satisfiability problem (SAT) has inspired much research. Not only is the SAT problem of theoretical interest but, because any NP-hard problem may be reduced to SAT, it also has a vast number of applications. For example, SAT solvers—algorithms that solve SAT problem instances—have found use in industry and academia as black-box algorithms, assisting in termination analysis of term-rewriting systems [14], planning in AI [28], bounded model checking of ANSI-C programs [10], managing software packages [42], and inferring haplotype in bioinformatics [30]. If a problem can be converted into SAT, then a SAT solver may be a powerful tool.

However, the success of SAT is plagued by its inherent limitations. SAT is an NP-hard problem, so unless P = NP, an efficient algorithm for all SAT instances is unobtainable. Nevertheless, decades of research have produced hundreds of SAT solvers, each sporting their own flavors of solving techniques and strategies. Over the years, SAT solvers have come to be classified into one of several SAT-solving paradigms. We discuss the two most prevalent here.

The first SAT-solving paradigm is conflict-driven clause learning (CDCL). The core of CDCL solvers is the DPLL backtracking algorithm proposed by Davis, Putnam, Logemann, and Loveland [12]. CDCL solvers use unit propogation and resolution to explore the search space until either a satisfying assignment to the Boolean variables of the SAT instance is found or a conflict is reached. By adding conflict clauses to the overall formula, CDCL solvers are guaranteed to find a satisfying assignment if one exists or conclude that there is no satisfying assignment. In addition to merely stating that a problem instance is unsatisfiable, many CDCL solvers have been developed which can output a proof of unsatisfiability that can be formally checked by proof checkers. These proofs are instrumental

in verifying groundbreaking results in combinatorial problems [8,18,21]. The first CDCL solver was arguably GRASP [31]. Other CDCL solvers of note are CaDiCaL [6] and Lingeling [4]. For an overview of CDCL, cf. chapter 4 of the Handbook of Satisfiability [3].

The second SAT-solving paradigm is stochastic local search (SLS). Generally, SLS solvers start with some initial truth assignment to the Boolean variables and incrementally flip the truth value of a single variable at a time until a satisfying assignment is discovered. SLS solvers are incomplete, meaning that they are not guaranteed to produce a satisfying assignment. Yet, SLS solvers can sometimes quickly find satisfying assignments to problem instances that take state-of-the-art CDCL solvers CPU years to find. SLS solvers perform best on random SAT instances, which are notoriously hard for CDCL solvers, but success can also be found on structured problem instances. For example, encodings of matrix multiplication problems into SAT appear hard for CDCL solvers and many SLS solvers [20], yet YalSAT [5], an SLS solver developed in 2014, solves these challenge instances in minutes on a single CPU.

The SLS algorithm that is the focus of this thesis is the Divide and Distribute Fixed Weights (DDFW) algorithm, first presented in 2005 [27]. At a high level, DDFW associates a weight to each clause and then attempts to find a satisfying assignment by minimizing the amount of weight held by the unsatisfied clauses. The novelty of DDFW is that when a local minimum is reached, weight is transferred from satisfied clauses to unsatisfied clauses along clause neighborhood relationships, rather than across all clauses. Like YalSAT, DDFW has found success on a particular structured problem instance. Of the 33 solvers present in the SLS framework UBCSAT [41], DDFW is the only SLS solver able to complete the Pythagorean triples instance for $n = 7824$ [21] on a single CPU with a timeout of one million flips, and it does so in under a minute.[1] Such a quick solve time is remarkable when compared to the many thousands of CPU hours that were required to initially solve the $n = 7824$ instance. It is unfortunate, then, that only a handful of publications exist examining DDFW. Due to the lack of research literature on DDFW and the age of the research literature that does exist, two research needs arise.

The first is the need to examine the solving techniques of DDFW more thoroughly. DDFW assumes that it is optimal to flip variables which decrease the total amount of weight held by the unsatisfied clauses. However, ProbSAT [2] and its implementation YalSAT have shown that it is not always optimal to flip the best-appearing variable at each step, instead preferring a more spread-out probability distribution from which to select variables to flip. DDFW also assumes that reweighting unsatisfied clauses along clause neighborhood relationships is an effective technique for escaping local minima. As this idea is unique to DDFW, this reweighting strategy has not been explored outside of its original publications and so requires investigation. Research into these two assumptions and additional

[1] Four solvers were able to find an assignment that made the instance evaluate to a single digit number of unsatisfied clauses. These algorithms were g2wsat, novelty++, novelty+p, and PAWS.

solving techniques for DDFW may reveal a heuristic that is more effective on a larger number of SAT instances, increasing the success of DDFW.

The second research need is the need to update the literature on how effective DDFW is against more recent hard SAT instances. Benchmarks which were considered hard a decade ago can now be solved in seconds by state-of-the-art SAT solvers. Because DDFW was first proposed in 2005, its initial results are outdated. When combined with results from the first research direction, experimental testing may reveal a heuristic for DDFW that is effective on modern benchmarks.

In this thesis, I contribute to both of these research needs. I propose modifications to DDFW in how it selects which variables to flip and in how it reweights clauses in local minima. The modifications I propose are inspired by similar SAT solvers which have also enjoyed success, such as YalSAT and SAPS [24]. I then test these heuristics against a suite of hard SAT benchmark instances culled from research publications and applications of the last few years. My contributions are not exhaustive on either front—the space of variable selection and clause reweighting heuristics is rich, and additional hard benchmarks are published annually at the SAT Competition[2]—and so this thesis also contributes an implementation of DDFW that allows for easy testing of additional heuristics against updated benchmarks.

I present three main findings from my experiments. They are as follows:

- DDFW attempts to find a satisfying assignment by flipping variables that reduce the amount of weight held by unsatisfied clauses the most at each step. I propose two additional variable selection probability distributions and test them against a new test set of hard SAT instances. **Testing showed that the original selection distribution is optimal.**
- In local minima, DDFW distributes weight from satisfied clauses to unsatisfied clauses according to fixed amounts. I propose that weight be transferred according to a linear transfer rule instead. Experiments show that a combination of multiplicative and additive constants produce a DDFW algorithm that performs **30% better than the original, with 3x the solve rate.**
- In local minima, DDFW distributes weight from a single satisfied clause to a single unsatisfied clause at a time. I propose that weight be transferred from entire satisfied neighborhoods instead. Experiments show that distributing roughly the same amount of weight from entire neighborhoods achieves a **50% improvement over the original DDFW algorithm**.

The organization of the thesis is as follows: In Section 2, we cover preliminaries of general notation and definitions used in the remainder of the thesis. Section 3 reviews prior work into similar SLS solvers. Section 4 discusses the DDFW algorithm in-depth, as well as the new strategies investigated by this thesis. The section ends with some notes on the implementation of DDFW used in experiments. Section 5 presents the experimental results of testing DDFW and the new strategies against an updated set of hard SAT benchmark instances. We make concluding remarks and propose future work in Section 6.

[2]Visit http://www.satcompetition.org/ for more information.

2 Preliminaries

A Boolean satisfiability (SAT) problem instance is a propositional formula $\mathcal{F}$ consisting of a set of Boolean variables $\{x_1, x_2, ..., x_n\}$ and their negations joined by logical connectives. The goal of a SAT solver is to find an assignment of true and false values to the Boolean variables such that the overall formula evaluates to true. Such an assignment is called a *satisfying assignment*. While any well-formed Boolean formula constitutes a SAT problem instance, it is standard to express formulas in conjunctive normal form (CNF). It is well known that any Boolean formula can be converted into CNF, so SAT solvers expecting problem instances in CNF form does not preclude solving. We define CNF here:

Definition 1 (Conjuctive normal form (CNF)). *A Boolean formula $\mathcal{F}$ is in **conjunctive normal form** if it is an indexed set of clauses $\mathcal{C} = \{C_1, C_2, ..., C_m\}$ joined by logical ANDs:*

$$\mathcal{F} = \bigwedge_{j=1}^{m} C_j$$

where each clause C_j is an indexed set of Boolean literals $C_j = \{v_1, v_2, ..., v_{|C_j|}\}$ joined by logical ORs:

$$C_j = \bigvee_{k=1}^{|C_j|} v_k$$

with each $v_k \in \{x_i, \neg x_i\}$ for some $i \in [1, n]$.

In other words, $\mathcal{F}$ is a conjunction of clauses, each of which is a disjunction of literals. We take the conventions that each clause is unique up to ordering of its literals and that each clause contains unique literals, such that for every distinct pair $v_k, v_\ell \in C_j$, $v_k \neq v_\ell$ and $\neg v_k \neq v_\ell$.

Let us denote assignments of truth values to the n Boolean variables as α such that $\alpha(x_i) \in \{\top, \bot\}$ for all $i \in [1, n]$, with $\top$ representing true and $\bot$ representing false. Let $U(\mathcal{F}, \alpha)$ give the set of clauses in $\mathcal{F}$ that have no literal which evaluates to true under α. We call U the set of unsatisfied clauses in $\mathcal{F}$. Let $u(\mathcal{F}, \alpha) := |U(\mathcal{F}, \alpha)|$. Define $S(\mathcal{F}, \alpha)$ and $s(\mathcal{F}, \alpha)$ in a similar way for the set and number of satisfied clauses under α. We drop $\mathcal{F}$ if the formula α acts on is clear by context. In an abuse of notation, if $C \subseteq \mathcal{C}$ is a subset of the clauses of $\mathcal{F}$, let $U(C, \alpha)$ (respectively, $S(C, \alpha)$) give the set of clauses in C which are left unsatisfied (satisfied) under α, and let u and s be defined similarly.

Formulas in CNF have the nice property that flipping $\alpha(x_i)$ from $\bot$ to $\top$ makes any clause containing x_i evaluate to true. If a literal x_i occurs in many clauses, then it is usually advantageous for an SLS solver to set $\alpha(x_i) = \top$ (respectively, $\neg x_i$ and $\alpha(\neg x_i) = \bot$). It is thus useful to relate clauses which share literals. We call these clauses neighbors. Groups of clause neighbors are called a neighborhood. We formally define clause neighborhoods here:

Definition 2 (Neighboring clause). *Let $\mathcal{F}$ be a Boolean formula in CNF on n variables $\{x_1, x_2, ..., x_n\}$. Fix a clause $C_j \in \mathcal{F}$. Then $C_k \in \mathcal{F}$, $C_k \neq C_j$ is a*

same-sign neighbor *of C_j if there is a literal $v \in \{x_i, \neg x_i\}$ for some i such that $v \in C_j \cap C_k$. We say that C_j and C_k are neighbors on v. Let $N(C_j)$ be the set of all such same-sign neighboring clauses $C_k \neq C_j$.*

Same-sign neighboring clauses are clauses that share at least one literal of the same sign. For brevity, we write that two clauses are neighbors to mean that they are same-sign neighbors, unless otherwise indicated.

As is discussed in Section 4, DDFW speculates that the property of clause neighborhoods becoming satisfied together is key to finding a satisfying assignment, and so it attempts to satisfy neighborhoods together. To further encapsulate this idea, DDFW lets an unsatisfied clause C_j borrow weight from satisfied clauses in its neighborhood $S(N(C_j), \alpha)$. The concept of assigning weight to and moving weight between clauses is not unique to DDFW, as we shall see in Section 3. However, that DDFW moves weight within clause neighborhoods *is* unique. Therefore, we define the notation needed to work with clause weights:

Definition 3 (Clause weighting function). *Let $\mathcal{F}$ be a Boolean formula in CNF. Then let the* ***weight*** *of clause $C_j \in \mathcal{C}$ be given by $W(C_j)$, where $W : \mathcal{C} \to \mathbb{R}^+$ assigns a positive real value to each clause. Denote the sum of weights of the unsatisfied clauses under an assignment α as $W_U(\mathcal{F}, \alpha)$ and the sum of weights of the satisfied clauses under α as $W_S(\mathcal{F}, \alpha)$.*

Definition 4 (Unsatisfied weight reducing variable). *Let $\mathcal{F}$ be a Boolean formula in CNF, α be an assignment, and W be a clause weighting function. Then x_i is an* ***unsatisfied weight reducing variable*** *if*

$$\sum_{C \in U(\mathcal{C}, \alpha')} W(C) < \sum_{C \in U(\mathcal{C}, \alpha)} W(C)$$

where $\alpha'(x_i) = \neg\alpha(x_i)$ and $\alpha'(x_j) = \alpha(x_j)$ for all other $j \neq i$. Let $R(\mathcal{F}, \alpha)$ be the set of unsatisfied weight reducing variables in $\mathcal{F}$ under α.

If the goal of an SLS solver is to reduce $W_U(\mathcal{F}, \alpha)$, then it is advantageous to flip the truth values of unsatisfied weight reducing variables in $R(\mathcal{F}, \alpha)$. If $|R(\mathcal{F}, \alpha)| = 0$, then a local minimum has been reached.

Section 4 shows how DDFW uses the concepts of clause neighborhoods and unsatisfied weight reducing variables to solve CNF formulas. But before we discuss DDFW, we first discuss similar SLS solvers and potential motivations for modifications to DDFW.

3 Prior work in SLS

Stochastic local search (SLS) algorithms attempt to solve Boolean satisfiability (SAT) problem instances by making incremental changes to an assignment α until a satisfying assignment is found. These incremental changes almost always take the form of changes in sign ("flips") of a single Boolean variable at a time. Many SLS solvers choose to flip Boolean variables that reduce the number of unsatisfied clauses $u(\mathcal{F}, \alpha)$, but there are many exceptions to this rule. Yet no matter how SLS algorithms prioritize the variables they flip, the eventual goal is to end with a satisfying assignment.

There are a few drawbacks to the SLS method. The first is that SLS solvers are not guaranteed to find a satisfying assignment to a satisfiable problem instance in a finite amount of time. As a result, SLS algorithms are classified as incomplete. Many solvers use randomness to achieve approximate completeness [22], but ultimately, randomness does not guarantee completeness. The second drawback is that SLS solvers cannot prove that a problem instance is unsatisfiable. The reason for this comes from the first drawback: after any finite number of flips, SLS solvers cannot differentiate between a satisfiable instance and an unsatisfiable one. As a result, SLS solvers in practice are given problem instances which are known or expected to be satisfiable.

Yet despite these drawbacks, SLS solvers still enjoy success over the more systematic CDCL solvers on many classes of problem instances. SLS is the method of choice for randomly constructed SAT instances and on encodings of some graph coloring problems and the N-queens problem [36,37]. Work has also been done to make SLS solvers more effective on general structured problem instances [9,15,23]. Some SLS solvers perform surprisingly well on singular classes of structured problem instances, such as YalSAT [5] on matrix multiplication problems [20] and DDFW [27] on the Pythagorean triples $n = 7824$ instance [21].

In this section, we discuss several SLS solvers and how the ideas of their implementations may relate to DDFW or inspire potential modifications. The first solver, Walksat, is presented in Section 3.1. Walksat is a simple and influential SLS solver that captures the essence of many SLS solvers. While not a direct predecessor to DDFW, it is still instructive to examine. The second solver, ProbSAT, is covered in Section 3.2. ProbSAT generalizes the method of choosing which variable to flip when there are many potential candidates. The methods of variable selection used in ProbSAT and DDFW are different, but the success of ProbSAT inspires potential modifications for DDFW when it comes to variable selection. The third and final solver is SAPS, discussed in Section 3.3. SAPS is a direct predecessor to DDFW, as it also reweights clauses in local minima. The reweighting method in SAPS is more global, though: weights are normalized across all clauses, as opposed to the method in DDFW of moving weight within clause neighborhoods. Although DDFW improves on the ideas in SAPS, SAPS suggests that reweighting from larger groups of clauses may be more effective than singular clauses, as is the method in DDFW.

3.1 A look at Walksat

Each SLS algorithm answers the question of how it selects which Boolean variables to flip differently. Naively, solvers should prioritize making flips that cause the greatest decrease in $u(\mathcal{F}, \alpha)$. If there are no variables that decrease u when flipped, then a local minimum has been reached. The space of heuristics to escape local minima is vast, and many strategies have been proposed.

One strategy to escape local minima is to make flips that keep $u(\mathcal{F}, \alpha)$ constant until a flip which decreases u becomes available. Flips that keep u at the same value are called "sideways moves." If u is plotted on a y-axis and the number of flips on the x-axis, then this strategy would cause u to decrease until it reaches a "plateau." u stays on this plateau until a u-decreasing variable is found, at which point u drops again, potentially into another plateau. This algorithmic idea of sideways moves is implemented in the SLS algorithm known as GSAT [37]—where the "G" stands for greedy—to some experimental success.

However, some plateaus cannot be escaped simply by making sideways moves. One way to "dislodge" α from these kinds of local minima is to intentionally flip a variable which increases u. The hope is that slight perturbations in u causes faster and more effective escapes from local minima than only taking sideways moves. Introducing the occasional "random walk" flip gives WALKSAT [35]. In experimental testing, WALKSAT performed an order of magnitude better than GSAT in both the time and the number of flips needed to find a satisfying assignment, and WALKSAT was able to solve problem instances twice as large as those solved by GSAT. Due to its simplicity and effectiveness, WALKSAT has become an influential SLS algorithm, off of which many other solvers are based. Pseudocode for WALKSAT can be found in Algorithm 1.

Algorithm 1: WALKSAT(p)

```
Input: n Boolean variables x_1, ..., x_n and a CNF formula F;
for MAX-TRIES times do
    α ← randomly generated truth assignment;
    for MAX-FLIPS times do
        if α satisfies F then
            return α;
        else
            if rand(0, 1) ≤ p then
                C_j ← random unsatisfied clause in U(α);
                Flip a random literal in C_j;
            else
                Flip a literal which decreases u(α) the most;
            end
        end
    end
end
return "No satisfying assignment";
```

The parameter p determines the odds of a random walk flip occurring. With probability p, a random unsatisfied clause C_j is chosen, and a random literal in C_j is flipped, satisfying the clause. Of course, the random variable flip may cause any number of satisfied clauses with the opposite-signed literal to become unsatisfied. u may thus increase with this kind of flip. With probability $1-p$, the GSAT search procedure is conducted, called "focused search." u will decrease or engage in sideways moves with this kind of flip. In experimental testing, p was found to be optimal between 0.5 and 0.6 for the set of random SAT instances selected in the study, but p is naturally sensitive to the exact problem instance being considered.[3]

WALKSAT embodies two main elements of an SLS solver succinctly: many flips are made to directly improve some metric (e.g. u), while others are made to perturb α in order to escape from local minima. While the implementations of other SLS solvers differ wildly when compared to WALKSAT, it is often the case that other solvers share these two elements with WALKSAT.

3.2 A look at ProbSAT

When considering WALKSAT in Algorithm 1, one may ask whether one literal should be preferred over another when performing a random walk. Perhaps the solver should prioritize literals that, when flipped, decrease u the most among those in the selected clause. Or perhaps the solver should instead prioritize literals that cause the least number of clauses to become unsatisfied, regardless of how many clauses become satisfied. By changing the underlying probability distribution the literals of the clause are chosen from, the SLS solver can be configured to, on average, make more effective random walk flips.

The generalization of the probability distribution in the random walk portion of WALKSAT is found in PROBSAT [2]. Instead of a flat probability distribution, PROBSAT picks literals proportional to each literal's score under an abstract function $f(v, \alpha)$. Pseudocode of PROBSAT is presented in Algorithm 2. An implementation of PROBSAT with additional optimizations is YALSAT [5].

In WALKSAT, we saw that focused search flips variables which decrease u the most. PROBSAT goes one step further and examines whether it is more effective to flip variables that satisfy the most number of clauses ("make") or variables that cause the least number of satisfied clauses to become unsatisfied ("break"). An interesting result from the original publication was that the effect of make could be effectively ignored. In other words, it was better to prioritize variables which minimized break than maximized make. It was also found that a probability distribution of the form $f(v, \alpha) = (c_b)^{-\text{break}(v,\alpha)}$ was better than a polynomial one. The parameter c_b was found to have an optimal value of around 2.5. Such an exponential distribution may prove useful when considering how to sample variables to flip in other SLS solvers, such as DDFW.

[3] While too much of a tangent to explore here, it is worthwhile to note the literature on the hardness of random satisfiable SAT instances and how hard random instances may be generated (cf. [13,16,17,29,32,38]). In short, their generation is nontrivial.

Algorithm 2: ProbSAT(f)

```
Input: n Boolean variables x_1, ..., x_n and a CNF formula F;
for MAX-TRIES times do
    α ← randomly generated truth assignment;
    for MAX-FLIPS times do
        if α satisfies F then
            return α;
        else
            C_j ← random unsatisfied clause in U(α);
            for v ∈ C_j do
                compute f(v, α);
            end
            v ← random variable according to probability f(v,α) / Σ_{v∈C_j} f(v,α);
        end
    end
end
return "No satisfying assignment";
```

3.3 A look at SAPS

Both of the previously-discussed algorithms attempt to minimize $u(\mathcal{F}, \alpha)$ in their search for a satisfying assigment. Yet u is not, *a priori*, the best metric to minimize. Notions of minimizing the sum of unsatisfied clause sizes, the sum of unsatisfied clause neighborhood sizes, or the number of distinct variables in unsatisfied clauses are all metrics that, when equal to 0, mean that a satisfying assignment has been found. Clearly, a rich space exists in which to explore possible minimization metrics for SLS algorithms.

As alluded to already in Definition 3, assigning a weight to each clause in $\mathcal{F}$ via a weighting function W gives a minimization metric $W_U(\mathcal{F}, \alpha)$, the sum of weights held by the unsatisfied clauses under an assignment α. When $W_U(\mathcal{F}, \alpha) = 0$, then we have found a satisfying assignment (as W gives strictly positive weights to each clause). We note that if $W(C_j) = 1$ for every clause, then there is no difference between minimizing W_U and minimizing u. In this sense, W_U is a generalization of u.

If W_U is our minimization metric, then it is likely advantageous to flip variables that reduce W_U. Definition 4 supplies us with a GSAT-like algorithm for minimizing W_U: flip unsatisfied weight reducing variables whenever possible. Of course, the question arises of what to do in a local minimum.

SAPS ("Scaling and Probabilistic Smoothing") [24], a W_U-minimizing SLS algorithm, answers that question with a method of clause reweighting. When a local minimum is reached, the weights of all unsatisfied clauses clauses are mulitplied by a scaling factor a. Then, with probability p_{smooth}, all clause weights are "smoothed" to the average weight value via

$$W(C_j) \leftarrow n \times W'(C_j) + (1 - n) \times \overline{W(\mathcal{C})}$$

where $W'(C_j)$ is the updated weight value equal to $W(C_j)$ if C_j is satisfied and $a \times W(C_j)$ if C_j is unsatisfied, and where n is a normalization factor between 0 and 1. $\overline{W(\mathcal{C})}$ denotes the average clause weight before smoothing. In the publication introducing SAPS, optimal values for these parameters were $a = 1.3$, $n = 0.8$, $p_{\text{smooth}} = 0.05$, and a random walk probability of $p_{\text{walK}} = 0.01$ (see below).

The rest of SAPS is familiar to what we've seen before—start with a random initial assignment, pick the best unsatisfied weight reducing variable to flip, and sometimes take a random walk—and is presented in Algorithm 3. For more details, see the cited publication.

Algorithm 3: SAPS($p_{\text{smooth}}, p_{\text{walk}}, a, n$)

```
1  Input: n Boolean variables x_1, ..., x_n and a CNF formula F;
2  for MAX-TRIES times do
3      α ← randomly generated truth assignment;
4      for MAX-FLIPS times do
5          if α satisfies F then
6              return α;
7          else
8              if |R(F, α)| > 0 then
9                  Flip a literal in R that reduces W_U the most;
10             else
11                 if rand(0, 1) ≤ p_walk then
12                     Flip a random literal in F;
13                 else
14                     for C_j ∈ U(C) do
15                         W(C_j) ← a × W(C_j);
16                     end
17                     if rand(0, 1) ≤ p_smooth then
18                         for C_j ∈ C do
19                             W(C_j) ← n × W(C_j) + (1 − n) × W(C)‾;
20                         end
21                     end
22                 end
23             end
24         end
25     end
26 end
27 return "No satisfying assignment";
```

Algorithm 3 lays out a base procedure for how to flip variables that reduce W_U and how to reweight clauses in a local minimum. The procedure can be extended with more complex ways of selecting variables and reweighting clauses. One such exmaple of an "extension" to how SAPS selects which variables to flip can be found in CCANR [9]. CCANR incorporates the concepts of age

and configuration changing, the latter meaning preventing variable flips if the neighborhood has not changed since the previous flip, preventing cycling. The heuristic of configuration changing appears to make SLS algorithms more effective on structured problem instances.

The idea of clause reweighting present in SAPS is not new; rather, the solvers using clause reweighting strategies form a kind of family tree. SAPS inherited many of its ideas from an SLS algorithm called ESG ("Exponential Sub-Gradient method"),[4] and it passes its ideas to two solvers. The most direct descendent of SAPS is an algorithm called PAWS ("Pure Additive Weighting Scheme") [40]. The only significant difference between the two algorithms is that PAWS reweights clauses additively instead of multiplicatively. The study that introduced PAWS found that the additive reweighting method was less CPU intensive. As a result, PAWS was able to perform more flips than SAPS in the same amount of time, possibly contributing to the success of PAWS over SAPS on a majority of the test set. Yet, SAPS outperformed PAWS on several problem classes, particularly smaller problem instances. The authors speculated that the difference in weight value types—SAPS with floating-point weight values and PAWS with integral ones—contributed to this performance discrepancy by allowing SAPS to have more expressive clause weights. However, the authors remarked that the exact reason for this was unknown and left it for future work.

PAWS, in turn, passes on its algorithmic ideas to DDFW. DDFW reweights clauses additively, like PAWS, but in a novel way, building on its predecessors. In its original form, DDFW only distributes weights in an additive way, staying "purely PAWS." But SAPS succeeded over PAWS for *some* reason. Therefore, when we turn to modifying DDFW in the next section, we keep SAPS—and the other solvers discussed above—in mind.

[4]The interested reader may also refer to SDF [34], a "cousin" of ESG.

4 DDFW and its modifications

Divide and Distribute Fixed Weights (DDFW) [27] is an SLS algorithm that seeks to minimize $W_U(\mathcal{F}, \alpha)$. It does so by assigning a weight to each clause and then flips variables which reduce the amount of weight held by unsatisfied clauses. When a local minimum is reached, weight is moved from satisfied clauses to unsatisfied clauses via clause neighborhood relationships. Eventually, enough weight will be transferred to the unsatisfied clauses that at least one variable may be flipped to minimize W_U, and the focused search begins anew.

Despite DDFW being the only SLS solver, to my or the original authors' knowledge, that exploits clause neighborhood relationships in local minima, remarkably little literature has been published on DDFW. Aside from its introduction, only a couple of other papers explore modifications to the base algorithm [25,26].[5] DDFW is also not prevalent in the SAT solving community, but it has seen some use as a black-box SAT solver in other publications [1,19] and in Microsoft's open-source Z3 Theorem Prover.[6]

To ameliorate the lack of study into DDFW, we embark on an examination of several parts of the algorithm. We first cover the original implementation of DDFW in Section 4.1. Modifications to the probability distribution for how unsatisfiable weight reducing variables are chosen are discussed in Section 4.2, and modifications to the weight transfer rule are discussed in Section 4.3. A strategy for abandoning the current assignment in favor of reverting back to a more optimal assignment is introduced in Section 4.4. All of these modifications see experimental testing in Section 5. We finish in Section 4.5 with notes on the implementation of DDFW used for the experiments.

4.1 DDFW

We present the DDFW algorithm as it appeared in its original publication [27]. After reading in a Boolean formula $\mathcal{F}$ in CNF, DDFW assigns a fixed starting weight to every clause, i.e. $W(C_j) = w_{\text{init}}$ for every C_j. The original publication posited that a value of $w_{\text{init}} = 8$ was best. Then an initial random assignment α is generated and focused search begins. At each step, a variable from $R(\mathcal{F}, \alpha)$ which reduces W_U the most is flipped. If $|R(\mathcal{F}, \alpha)| = 0$, then with probability 0.15, a sideways move is taken, if one exists. Otherwise, DDFW determines it has reached a local minimum, and so it moves to its reweighting phase.

To reweight, DDFW takes each unsatisfied clause $C_j \in U(\mathcal{C}, \alpha)$ and moves weight from one of its satisfied neighbors in $S(N(C_j), \alpha)$ to C_j. DDFW makes sure to not take too much weight away from any one clause at a time: for a value of $w_{\text{init}} = 8$, the most amount of weight moved between clauses is 2. If there are

[5] Of note is the publication made in 2021 by one of the original DDFW authors [25] during the undertaking of this thesis! Even though the publication is new, the benchmarks used for testing in the paper match the ones from the 2005 publication, and so there is still need to test DDFW against an updated set of benchmarks.

[6] https://github.com/Z3Prover/z3

no satisfied neighbors of a great enough weight, then a random satisfied clause of sufficient weight is used instead for the weight transfer.

The above algorithmic description is presented in detail in Algorithm 4.

Algorithm 4: DDFW

```
1  Input: n Boolean variables x_1, ..., x_n and a CNF formula F;
2  Initialize each clause's weight to w_init;
3  for MAX-TRIES times do
4      α ← randomly generated truth assignment;
5      for MAX-FLIPS times do
6          if α satisfies F then
7              return α;
8          else
9              if |R(F, α)| > 0 then
10                 Flip a literal in R which decreases W_U the most;
11             else if rand(0, 1) ≤ 0.15 and a sideways move exists then
12                 Flip a literal which does not increase W_U;
13             else
14                 for C_j ∈ U(F, α) do
15                     C_k ← arg max_{C_k} {W(C_k) : C_k ∈ S(N(C_j), α)};
16                     if W(C_k) < w_init or rand(0, 1) ≤ 0.01 then
17                         C_k ← random satisfied clause with W(C_k) ≥ w_init;
18                     end
19                     if W(C_k) > w_init then
20                         Transfer a weight of two from C_k to C_j;
21                     else
22                         Transfer a weight of one from C_k to C_j;
23                     end
24                 end
25             end
26         end
27     end
28 end
29 return "No satisfying assignment";
```

Not mentioned in the original publication but appearing in the code supplementing it was the random walk in line 16. With small probability $p = 0.01$, the maximum-weight neighbor C_k is discarded, and a random satisfied clause with weight at least w_{init} is chosen instead. This change was included in the implementation used for experimentation in Section 5.

Two features of DDFW distinguish it from similar SLS solvers. The first is in how DDFW applies its reweighting rule. Similar solvers escape local minima using a two-step process: first, the weights of the unsatisfied clauses are increased; and second, all weights are normalized so as to keep the total weight from growing too large. DDFW combines these two steps into one by moving weight directly

from satisfied clauses to unsatisfied clauses. In this way, DDFW obeys a kind of weight conservation law, and so the total weight remains constant.

The second distinguishing feature of DDFW is in which satisfied clauses share weight. DDFW exploits the properties of same-sign neighborhoods as discussed underneath Definition 2: namely, that flipping a shared literal helps all clauses that are neighbors on that literal by increasing the number of literals in those clauses that evaluate to true. Thus, borrowing weight from satisfied neighbors to satisfy an unsatisfied clause will never harm any of the neighbors from which that weight was borrowed. DDFW explicitly creates these "alliances" between same-sign neighbors in how it transfers weight in a local minimum.

A couple notes on particular implementation details of DDFW are in order. Firstly, DDFW chooses to increase the weight of all unsatisfied clauses in a local minimum, as opposed to decreasing the weight of all satisfied clauses, due to speedup achieved in practice. In general, SLS solvers start with an initial assignment under which tens of thousands of clauses may remain unsatisfied. Within thousands of flips, though, the number of unsatisfied clauses drops quickly. Applying a reweighting rule to the remaining unsatisfied clauses means applying the rule to a small number of clauses a majority of the time.

Secondly, the clause weights are all integers ($w_{\text{init}} = 8$, the reweighting rule transfers a weight of 1 or 2). The clause weights need not be integers, but actual implementations of DDFW prefer them for several reasons: integer operations are faster than floating-point ones on most CPUs, integers don't suffer from floating-point rounding error, and small integers allow for various optimizations in the variable selection and clause reweighting structures maintained by the algorithm. Yet the choice to use integers unnecessarily restricts the expressivity of the variable selection and reweighting strategies, as we shall see.[7]

DDFW in its original form enjoys success on selected random SAT instances and the Pythagorean triples instance. But the value of $w_{\text{init}} = 8$ and the weight transfer values of 1 and 2 are fixed constants. While these values may work best on some problem instances, they are not guaranteed to work on all. A deeper understanding of why these values work may lead to an improvement on the algorithm. To that end, we generalize two aspects of the DDFW algorithm.

The simpler generalization is abstracting away the probability distribution from which literals in $R(\mathcal{F}, \alpha)$ are drawn during focused search. DDFW flips only those literals which cause the greatest decrease in W_U, but as seen in ProbSAT in Algorithm 2, it is sometimes advantageous to flip less-than-optimal variables. We may instead draw literals from R proportional to their score under some f.

The other generalization we may make is abstracting away the weight transfer rule applied in a local minimum. Currently, DDFW transfers a fixed constant

[7] One could point out that using large enough integers would allow DDFW to "scale-up" operations so the effect of floating-point numbers could be approximated, but large integers invite their own host of problems, including overflow and the temptation to use floating-point operations in more complex weight transfer rules. Because I did not wish to navigate these issues, I decided to bite the bullet and use floating-point numbers, despite their drawbacks.

amount of weight from a single satisfied neighboring clause to an unsatisfied one. We may ask how the performance of DDFW is affected if weight is moved according to more complex functions, or if weight is taken from larger numbers of clauses in the neighborhoods.

In the sections directly below, we explore potential other implementations for both of these generalized aspects.

4.2 Modifications to the variable selection probability distribution

Line 10 in Algorithm 4 selects a random literal in $R(\mathcal{F}, \alpha)$ that causes the greatest decrease in W_U when flipped. However, we are not limited to only those variables in R. We can instead apply a probability distribution determined by a function f as in ProbSAT. In this way, we may explore which method of unsatisfied weight reducing variable selection is optimal for DDFW.

We propose two additional probability distributions here. The first is the uniform distribution: flip a random literal in R. The second is a weighted distribution proportional to the amount flipping the literal would decrease W_U by. Under this distribution, literals which decrease W_U more tend to be flipped more often, but less-preferred variables sometimes get flipped as well. A directly linear function f is most straightforward, and was investigated in Section 5. Future work could consider an exponential distribution, as in ProbSAT. The replacement to line 10 for this probability distribution would look like

$$v \leftarrow \text{random variable according to probability } \frac{\Delta W(x_i)}{\sum_i \Delta W(x_i)},\ x_i \in R(\mathcal{F}, \alpha)$$

where $\Delta W(x_i)$ is the reduction of unsatisfied weight if x_i is flipped in α.

4.3 Modifications to the weight transfer rule

Lines 20 and 22 in Algorithm 4 transfer fixed weights of 2 and 1 from a single satisfied neighboring clause to an unsatisfied clause when DDFW reaches a local minimum. When $w_{\text{init}} = 8$, the percentage of a clause's weight moved in a single transfer is capped at 25% of the initial weight value. But there may be situations where local minima can only be escaped after multiple weight transfers, such as when a supermajority of the clause weight is held by the satisfied clauses. In these situations, it is more advantageous to transfer larger amounts of weight at a time to "short-circuit" the need for multiple transfers. A more complex weight transfer rule, dependent on the amount of weight in the neighborhood, could solve this problem and may lead to a more effective reweighting strategy.

There are two immediate ways of moving more weight in a single transfer: take more from the maximum-weight clause, or take weight from a larger number of clauses. For the first, we consider a linear transfer rule dependent on a multiplicative parameter a and an additive parameter c. A linear rule is chosen to capture the reasoning in the above paragraph: when there is more weight

available in the maximum-weight clause, more weight should be transferred. Lines 20 and 22 then become:

$$\text{Transfer a weight of } (a \times W(C_k)) + c \text{ from } C_k \text{ to } C_j$$

Note that this rule does not differentiate between $W(C_k)$ above or at most w_{init}. On the one hand, the linear rule dispenses with this problem: for $a < 1$, when the amount of weight $W(C_k)$ draws closer to or falls below w_{init}, less weight is transferred, which approximates the effect of the original transfer rule. But on the other hand, there may be utility in switching between transfer rules in high-weight and low-weight scenarios. My implementation of DDFW, as mentioned in Section 4.5, thus allows for *two* sets of linear parameters to be specified: one pair to be applied when $W(C_k) > w_{\text{init}}$, and one pair to be applied when $W(C_k) \leq w_{\text{init}}$. Setting both pairs of constants equal to each other makes the linear rule blind to whetehr $W(C_k)$ is greater than or at most w_{init}.

Of course, there are plenty of functions more expressive than linear ones that may be used for the weight transfer rule. Polynomials of higher order, logistic functions, and exponential functions are all candidate transfer rules as well. But the parameter space posed by a and c in combination with the modification to transfer weight from larger groups of clauses in the neighborhood (see below) is large enough without introducing new classes of functions, and so we leave exploration into these alternative transfer rules for future work.

The second way of moving more weight in a single transfer is to take weight from a larger number of clauses. While it is certainly the case that a parameter n could be introduced to control the number of maximum-weight neighbors to take weight from, any fixed n runs into the same problem that $n = 1$ does in DDFW: a static parameter does not allow for reaction to the current state of the algorithm. For example, in situations where an unsatisfied clause has an above-average sized satisfied neighborhood, then satisfying the clause would potentially help an above-average number of clauses. Thus, more weight should be transferred to that clause. To that end, we will consider applying the transfer rule to all satisfied clauses in the neighborhood.

Algorithm 5: Transfer rule for all neighboring clauses

```
1 Input: an unsatisfied clause C_j;
2 for C_k ∈ S(N(C_j), α) do
3 |   Transfer weight from C_k to C_j according to the transfer rule;
4 end
```

We have some choice in how we apply the transfer rule to these clauses. Ultimately, we want to transfer weight from every applicable clause to the unsatisfied clause. We could do so by applying the transfer rule to each individual clause. We call this kind of rule application the "individual transfer method." But we could also compute how much weight we'd like to take from the neighborhood

as a whole using an aggregate weight statistic and then transfer that amount of weight spread across the neighboring clauses in some fashion. We propose two such methods here.

The first method applies the transfer rule to the average of the weights in the neighborhood. However much weight needs to be transferred is thus split across the neighbors evenly. We call this the "average transfer method."

The second method is like the first, except the weight is taken from neighboring clauses proportional to their weight. For example, if three neighbors had weight 2, 4, and 10, and if the amount of weight to be transferred is 4, then 0.5 weight is taken from the first neighbor, 1 from the second, and 2.5 from the third. We call this the "proportional transfer method."

The randomness of line 16 will be present for these two transfer methods as well. With a small probability p, a random satisfied clause is selected instead of the current neighboring clause. Such randomness is important to prevent particular neighborhoods from hoarding all the weight.

4.4 A restarting strategy

At times, DDFW may reach a local minimum in a state such that no matter how many times the weight transfer rule is applied, DDFW remains in that local minimum. One way this situation could occur is DDFW flips a variable, runs out of unsatisfied weight reducing variables, transfers weight, and then flips the same variable back, only to end up in the exact same state. Such a situation is unlikely, especially given the elements of randomness present in the original DDFW algorithm. Even so, DDFW is not immune to this and other pitfalls that befall W_U-minimizing algorithms, and no doubt there exist CNF formulas that are adversarial for DDFW.

It may then be optimal to restore DDFW to a previous or more neutral state when a local minimum has been occupied for a significant number of flips. Work on this front has already been done to an extent: an extension to DDFW called DDFW+ was published only a year after the original publication [26]. The core idea of DDFW+ was to increment a counter every time a variable flip did not improve the best number of unsatisfied clauses found so far. When the counter reached a particular value (originally, the number of literals present in the CNF formula), then the weights of all clauses were reset to fixed values (2 for satisfied clauses, 3 for unsatisfied clauses—note that for DDFW+, $w_{\text{init}} = 2$). Experiments showed that DDFW+ had a 30-75% speedup when compared to DDFW on some selected test instances from the 2005 SAT competition and on the original DDFW test instance set. These results were not seen over all test instances, however, and the experiments were run on what are now considered outdated problem instances. Whether the exact method of reweighting in DDFW+ is effective on modern benchmarks remains to be seen.

For this thesis, we propose a simpler restarting strategy: after a fixed number of flips where the lowest number of unsatisfied clauses found has not improved, set all weights back to their initial values and restore the assignment α to the one causing the formula to have the fewest number of unsatisfied clauses, as

discovered on a previous flip. Restoring the clause weights back to w_{init} may break DDFW out of the local minimum by causing many new variables to be flipped. We present initial results in Section 5.

4.5 An implementation of DDFW

In order to test DDFW and the modifications proposed above, I implemented DDFW in `C`. The repository can be found at https://github.com/ccodel/ddfw. The implementation allows for configuration of the various parameters of DDFW at the command line. In particular, all of the following may be specified:

- Timeouts for a maximum number of flips and CPU seconds
- The initial weight to assign to each clause
- The multiplicative and additive constants used in the linear reweighting rule
- The clause groups and transfer methods
- The probability distribution to select unsatisfied weight reducing variables
- A number of times to rerun the algorithm with a fresh α
- A number of flips to wait in local minima before restarting

In addition, my implementation is modular and allows for easy slotting in of additional variable selection heuristics, reweighting rules, and restarting strategies. Also included in the repository is a set of Python scripts that parse DDFW output and create .csv and .tsv files for data analysis.

My implementation uses a number of data structures to hold computed values or to reduce the amount of computation needed at each step. While this makes my implementation competitive with other SLS solvers on a wide array of problem instances, the use of these data structures does lead to some issues. Most notably, problem instances which have a small number of variables (≤ 100) and thousands of clauses—meaning many large neighborhoods—cannot make it past the data structure initialization phase in under fifteen minutes. I suspect that modifying my implementation to handle problem instances such as these will lead to an incredibly inefficient algorithm, as maximum weight neighbors must be calculated over large neighborhoods at each local minimum. Any implementation of DDFW is sure to run into this dilemma. Inherently, then, my implementation of DDFW is not useful on these classes of problem instances.

I chose to implement DDFW from scratch because the only existing implementation of DDFW was implemented in the SLS framework UBCSAT [41]. While UBCSAT is easy to use, it is not easily extensible, and so many of the modifications I proposed above could not be implemented. My resulting implementation of DDFW is competitive in speed to UBCSAT's, even though mine uses floating point operations. In addition, I claim that my implementation is more modular and extensible than UBCSAT's, and should serve as a starting point for future investigation into DDFW and similar W_U-minimizing algorithms.

5 Experimental results

We take DDFW and the modifications proposed in Section 4 and test them against a set of modern hard benchmarks. All experiments were conducted on the StarExec community servers [39]. The specs for the compute nodes can be found at https://starexec.org/starexec/public/about.jsp. The compute nodes that ran the experiments were Intel Xeon E5 cores with 2.4 GHz, and all experiments ran with 8 GB of memory. Each configuration and set of parameters was run for 100 iterations with a five million flip timeout, unless otherwise noted.

Generally, DDFW was configured to initialize every clause's weight to 100 ($w_{\text{init}} = 100$). An initial weight of 100 was chosen to allow for easy conversion between additive constants and a percentage of the initial weight transferred. To compare to the original DDFW settings, under a value of $w_{\text{init}} = 100$, the amount of weight transferred from a satisfied neighboring clause C_k in a local minimum is either 25 or 12.5, depending on whether $W(C_k) > w_{\text{init}}$ or $W(C_k) \leq w_{\text{init}}$.

In this section, we present the set of modern SAT problem instances used to test DDFW, and we present and discuss the experimental results. The test set is introduced in Section 5.1. To get a baseline against which to compare modifications of DDFW, we present the performance of UBCSAT's implementation of DDFW against the test set in Section 5.2. We then turn to testing the modifications to DDFW proposed in the previous section. The performance of DDFW for the different weight-reducing variable selection probability distributions is discussed in Section 5.3. An initial look at the linear weight transfer rule is given in Section 5.4. We then examine the effects of the linear weight transfer rule under different weight transfer methods in Section 5.5.

5.1 Benchmark instances

DDFW, like many algorithms, requires parameter tuning. A common way to tune algorithmic parameters is to run the algorithm against a set of challenging problem instances. The hope is that if the algorithm is tuned to perform well on a set of challenging instances, then the algorithm will be tuned to perform well more generally. However, it is important that the algorithmic parameters not be overfitted to the test set. To prevent overfitting, we used a variety of problem classes in our experiments.

All benchmarks used in this thesis can be found on GitHub[8], with the exception of the random 3-SAT instances, which were taken from the 2018 SAT Competition.[9] The benchmarks are all satisfiable CNF formulas in DIMCAS format.[10] All instances in the test set are hard for state-of-the-art CDCL and SLS solvers, although there were several SLS solvers from the 2018 SAT Competition that performed well on the random 3-SAT instances. The benchmarks are comprised of

[8] https://github.com/marijnheule/benchmarks

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

[10] A description of the DIMACS format can be found at https://www.cs.utexas.edu/users/moore/acl2/manuals/current/manual/index-seo.php/SATLINK___DIMACS.

- Two encodings of the Pythagorean triples instance for $n = 7824$ [21]
- Ten selected encodings of matrix multiplication problems [20]
- Two encodings of the almost-squares-in-almost-square (asias) problem [7]
- Three encodings of Steiner triple problems [33]
- The 3-SAT random instances from the 2018 SAT Competition

Notably, all except for the random 3-SAT instances are structured instances, which SLS solvers are generally not as performant on.

Because these are all challenging problem instances, DDFW does not locate satisfying assignments for many of these instances under a majority of configurations. Thus, the metric for our experiments becomes that of MaxSAT: minimizing the number of unsatisfied clauses that remain at timeout. A common metric used in the experimental results below is the lowest $u(\mathcal{F}, \alpha)$ achieved on any flip before timeout. The reason for this metric is simple: we should expect that those configurations that find assignments with the least number of unsatisfied clauses before timeout are the ones that, if left to run for a greater number of flips, will discover a satisfying assignment first.

5.2 UBCSAT baseline

Before making any modifications to DDFW, we first need a baseline of its performance on the test set. The implementation in UBCSAT serves as that baseline. The UBCSAT implementation of DDFW was run against each test instance for 100 iterations, each with a timeout of five million flips. The results are summarized in Table 1.

Table 1. Baseline results for DDFW on the test set using the UBCSAT implementation. The "Avg $u(\mathcal{F}, \alpha)$" column reports the average value of the lowest number of unsatisfied clauses found for any flip before timeout at five million flips. The "Min $u(\mathcal{F}, \alpha)$" column reports the lowest number of unsatisfied clauses found for any flip over all 100 iterations. The table reports an average over the matrix and 3-SAT problem instances due to their similarities in difficulty and solve times. An overall average for the test suite is reported on the final row.

Instance	Avg $u(\mathcal{F}, \alpha)$	Min $u(\mathcal{F}, \alpha)$	% solve
Matrix problems	57.02	38.1	0
asias-20	2.97	2	0
asias-34	14.02	10	0
bce7824	1.37	0	9
plain7824	1.63	0	9
Steiner-243-45	4.16	0	1
Steiner-405-70	4.98	0	4
Steiner-729-112	28.26	4	0
3-SAT problems	35.97	25.3	0
Overall	36.57	24.07	0.85

We see that the matrix, 3-SAT, and Steiner-729 problem instances are the most challenging of the test set, while both encodings of the Pythagorean triples problem (bce7824, plain7824) and the first two Steiner triples instances are the easiest, as they are the only four with positive solve percentages. Overall, DDFW gets to about 60 remaining unsatisfied clauses for the matrix instances, 35 for the 3-SAT instances, and close to single digits for the remaining instances. Configurations of DDFW which achieve lower u values and higher solve percentages are thus desirable for this test set.

5.3 Variable selection probability distribution results

In Section 4.2, we explored two new probability distributions for how to select unsatisfied weight reducing variables to flip. We examine the effects of flipping variables according to those distributions here.

Naively, we should expect that the more the distribution favors flipping unsatisfied weight reducing variables that reduce $W_U(\mathcal{F}, \alpha)$ the most, the better DDFW will perform. And indeed, we see that this assumption unequivically holds in Figure 1. For the original DDFW configuration in Algorithm 4, it is clear that taking the best variable at each flip is advantageous.

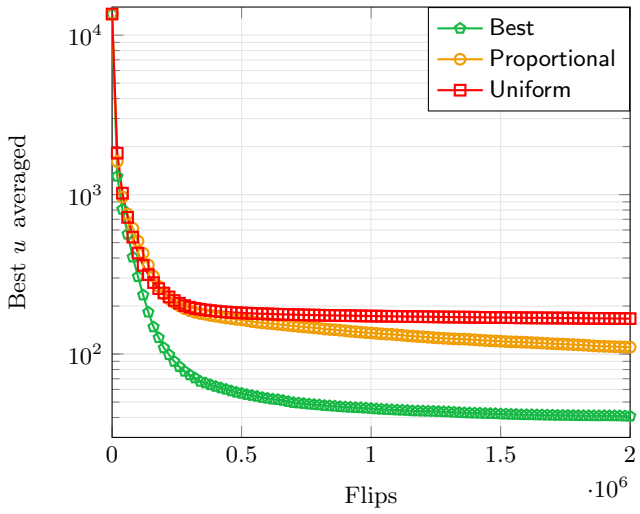

Fig. 1. A log plot of the lowest number of unsatisfied clauses found over a number of flips. The u values are averaged over all problem instances, with 20 iterations per problem instance. The original DDFW settings of a weight transfer of 2/1 were used, as in Algorithm 4. "Best" refers to flipping variables which reduce W_U the most. "Proportional" refers to selecting variables proportional to their W_U reduction. "Uniform" refers to a uniform probability distribution across all unsatisfied weight reducing variables.

It is possible that the results in Figure 1 favor one of the two newly proposed distributions under a different weight transfer rule. Yet it is my experience in prior testing that there is almost always at least a factor of 2 difference between the number of unsatisfied clauses at timeout for the original DDFW distribution and for the two new distributions. If there is a particular configuration or range of configurations under, say, the linear transfer rule that makes the proportional distribution optimal, then I have not discovered it in my testing, and all indicators point to such a configuration being brittle.

The results in Figure 1 are negative in the sense that both of the proposed variable selection probability distributions do not improve DDFW beyond what the original probability distribution does. However, such negative results are important when scoping out which methods are most effective in the minimization of W_U.

Of course, these findings do not rule out the possibility that there exists a more optimal variable selection heuristic for DDFW. For example, the configuration change method of CCANR [9] mentioned in Section 3 is one such variable selection heuristic that seeks to cut down on the amount of time an SLS algorithm spends in a local minimum by choosing variables which prevents cycling. Adopting that method for DDFW may show some promise, but it was not pursued in the course of this thesis.

For the remainder of this thesis, we conduct the experiments using the original DDFW distribution of always flipping the best variable during focused search.

5.4 Linear rule results

We now examine the effect of the linear weight transfer rule on the performance of DDFW. As mentioned in Section 4.5, my implementation of DDFW allows for two pairs of additive and multiplicative constants to be specified. One additive-multiplicative pair is used to calculate how much weight should be tranferred when the weight of the maximum-weight neighboring clause is at most w_{init}. Let us call this pair of constants $(a_{\leq}, c_{\leq})$. The other pair of constants is used when the weight of the maximum-weight neighboring clause is strictly greater than w_{init}. Let us call that pair of constants $(a_{>}, c_{>})$.

Before investigating the full parameter space posed by the linear rule, let us fix $a_{\leq} = a_{>} = 1$ and examine the effect of varying only the additive constants. The original DDFW algorithm has $c_{>} > c_{\leq}$ in order to transfer more weight from neighboring clauses that have more weight to give. Over a large number of flips, the effect of $c_{>} > c_{\leq}$ serves to keep any one clause from acrruing too much weight. If this reasoning is beneficial, then as we search for the best configuration of $(c_{>}, c_{\leq})$, we should expect the relationship $c_{>} > c_{\leq}$ to hold.

This is surprisingly not always the case. The results of a parameter search across c values of 5 to 50 in steps of 5 for each additive constant are summarized in Table 2. 3D plots of the average number of unsatisfied clauses remaining at timeout are shown in Figure 2.

Table 2 shows that it is not always optimal to have $c_{>} > c_{\leq}$. In particular, DDFW performed best on the matrix problem instances and the first of the asias

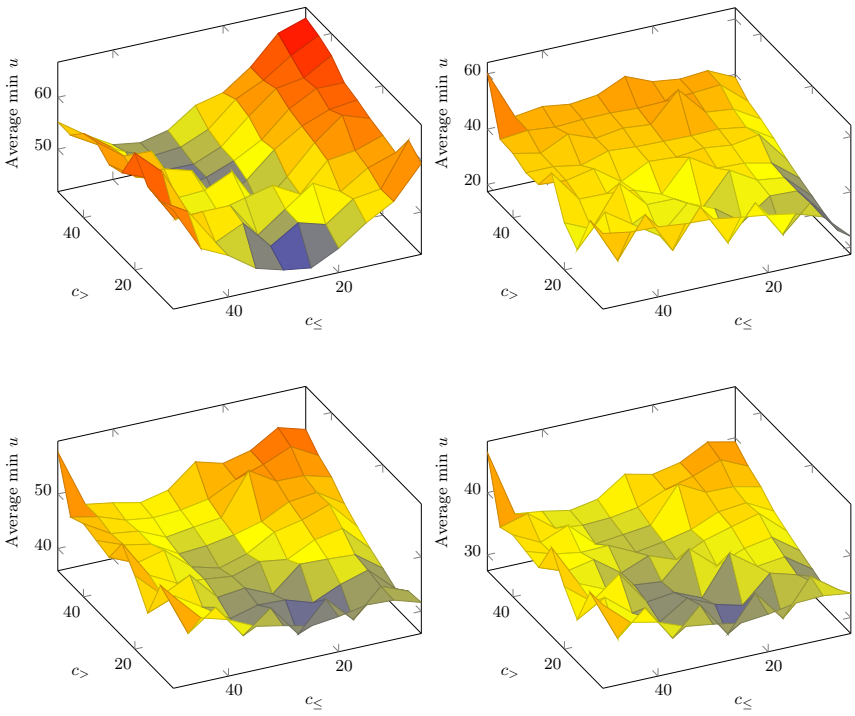

Fig. 2. 3D plots of the average lowest number of unsatisfied clauses found at a timeout of five million flips for values of $(c_>, c_\leq)$ from 5 to 50 in steps of 5. Each pair of c values was run for 100 iterations on each test instance, and the average u values at timeout were taken. Reading from the top-left, the plots show the parameter search on the matrix instances, the 3-SAT instances, all remaining instances, and an overall average across all instances. Note that all minimums occur with $c_> \leq c_\leq$, which is contrary to the original DDFW algorithm.

Table 2. Configurations of additive constants $c_>$ and $c_\leq$ from a parameter search of values of 5 to 50 in steps of 5 which give the least number of unsatisfied clauses before timeout at five million flips, on average. The best configuration of $(c_>, c_\leq)$ pairs are reported for each instance or group of instances, and the best configuration over all instances is reported at the bottom.

Instance	$c_>$	$c_\leq$	Avg $u(\mathcal{F}, \alpha)$	% solve
Matrix problems	5	25	43.91	0
asias-20	20	45	2.56	0
asias-34	40	30	11.79	0
bce7824	50	45	0.94	30
plain7824	50	50	1.12	27
Steiner-243-45	20	5	0.74	26
Steiner-405-70	35	5	0.96	4
Steiner-729-112	50	45	1.0	0
3-SAT problems	10	5	21.29	0
Overall	10	25	29.16	1.7

instances with a $c_>$ value at least 20% of w_{init} less than $c_\leq$. Also, the overall best configuration, on average, was $(c_>, c_\leq) = (10, 25)$. Together, these configurations indicate that it is *not always desirable* to take a proportionally greater amount of weight from neighbors with weight above w_{init} than from those with weight at most w_{init}. One reason for this trend may lie in how clauses end up with more than w_{init} weight in the first place: by being unsatisfied in a local minimum. Unsatisfied clauses in local minima represent the most difficult clauses to satisfy. Transferring weight to these clauses in local minima makes DDFW prioritize satisfying them. Once satisfied, it appears that DDFW should not be so hasty in bringing their weight back to w_{init}, as then the clauses lose their status as difficult-to-satisfy. Yet the trend of $c_> > c_\leq$ holds for many other instances in the test set, including all Steiner triple instances and the 3-SAT random instances, so the prior reasoning is not always the rule.

Table 2 also shows that optimal configurations vary across problem instances, which is not surprisng. Having widely differing optimal parameter values is part and parcel part of parameter tuning on singular test instances. In fact, further modifications to DDFW could exploit this fact. For example, modifications to DDFW that attempt to dynamically set the (a, c) values during runtime could be developed to take advantage of this differing set of optimal configuration values between instances. However, such a method was not investigated in this thesis.

We now turn to analyzing the 3D plots in Figure 2. The first observation we make is that the shape for the parameter search on the matrix instances was markedly different than the ones for all other problem instances. Of course, the matrix problems could be considered the most challenging of the test set, and so the fact that small changes in c value had a larger impact on the u value is

not surprising. Yet the matrix instances show a distinctive bowl shape about $c_{\leq} = 25$ that the other plots do not emulate, and the effect of the $c_{\leq}$ is largest in the matrix plot. The reason for this is likely due to the particular encoding for the matrix problems.

Our next observation is that all plots generally show that there was an optimal pair for $(c_{>}, c_{\leq})$, and that perturbing that optimal configuration along either axis resulted in a bowl shape. The mostly smooth descent to the optimal $(c_{>}, c_{\leq})$ is evidence that extending DDFW to include a method of dynamically finding optimal $(c_{>}, c_{\leq})$ or more general (a, c) pairs at runtime is a potentially fruitful research direction and may lead to a more effective algorithm.

The third observation we will make is the tendency for u to be lowest about a $c_{>}$ value of 10 or 15. The minimums of each plot, aside from the matrix plot, occur at $c_{>} \in \{10, 15\}$. As mentioned above, that $c_{>}$ has such a low value relative to w_{init} indicates that less weight should be taken from maximum-weight neighbors that earned that weight by being unsatisfied in a local minimum.

The final observation we will make regarding these plots is a series of valleys following a $c_{>} + 5 = c_{\leq}$ line. The shape is persistent in all four plots, and in two cases, this line of valleys leads to the minimum value. I speculate that the valleys occur because the $c_{>} + 5 = c_{\leq}$ line is the first place in the parameter search where $c_{>} < c_{\leq}$, and so the effect of moving less weight from of maximum-weight neighbors is first exhibited along this line. The line of valleys could thus be considered additional evidence that it is optimal for $c_{>} < c_{\leq}$.

More qualitatively, keeping in mind the results of Table 2, the parameter searches confirm the original DDFW publication to an extent. DDFW as it appears in Algorithm 4 moves 12.5% and 25% of the initial weight value between neighboring clauses in local minima. The parameter searches show that the optimal percentages for these weight transfers hover in the 10-30% range as well, although the exact values of $(c_{>}, c_{\leq})$ differ.

We now test the symmetric version of what we did above for the multiplicative parameters: we fix $c_{>} = c_{\leq} = 0$ and vary $a_{>}$ and $a_{\leq}$. We perform a parameter search of a values from 0.05 to 0.5 in steps of 0.05. The parameter search is analogous to the one for the additive parameters, as about 5 to 50% of w_{init} is transferred during each application of the linear rule. However, because the values are multiplicative constants rather than additive, the exact amount of weight transferred for each application of the linear rule will differ depending on the weight of the neighboring clause. The findings of the multiplicative parameter search are presented in Table 3 and Figure 3.

We first compare the results in Table 3 to those in Table 2. An initial look shows that the best configurations for both parameter searches for each instance or group of instances give almost identical u values. Notably, the average u value for the matrix problems is 43.91 for the additive parameters versus 45.87 for the multiplicative parameters, and the values for the asias, Pythagorean, and Steiner instances are all roughly the same, excepting Steiner-729.

There are two major differences between the tables. The first is the multiplicative configurations had generally higher solve rates on the four instances

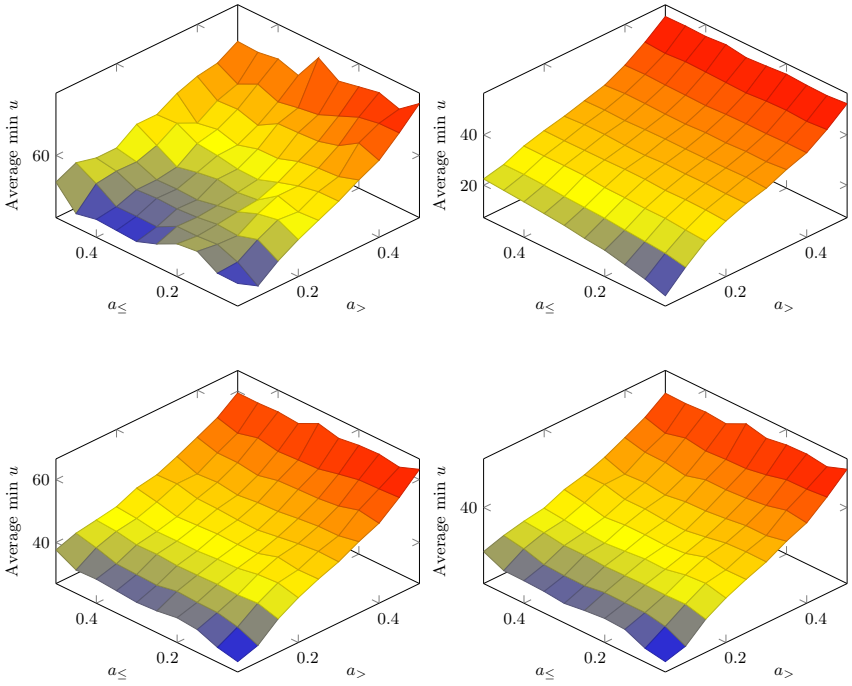

Fig. 3. 3D plots of the average lowest number of unsatisfied clauses found at a timeout of five million flips for parameter searches of $(a_{>}, a_{\leq})$ from 0.05 to 0.5 in steps of 0.05. Each pair of a values was run for 100 iterations on each test instance, and the average u values at timeout were taken. Reading from the top-left, the plots show the parameter search on the matrix instances, the 3-SAT instances, all remaining instances, and an overall average across all instances. In all four plots, the planar slope is almost solely determined by the $a_{>}$ value.

Table 3. Configurations of multiplicative constants $a_>$ and $a_\leq$ from a parameter search of values of 0.05 to 0.5 in steps of 0.05 which give the least number of unsatisfied clauses before timeout at five million flips, on average. The best configuration of $(a_>, a_\leq)$ pairs are reported for each instance or group of instances, and the best configuration over all instances is reported at the bottom.

Instance	$a_>$	$a_\leq$	Avg $u(\mathcal{F}, \alpha)$	% solve
Matrix problems	0.1	0.05	45.87	0
asias-20	0.1	0.35	2.37	0
asias-34	0.1	0.5	10.54	0
bce7824	0.3	0.3	1.03	31
plain7824	0.5	0.5	1.11	28
Steiner-243-45	0.05	0.05	0.32	70
Steiner-405-70	0.25	0.25	1.45	16
Steiner-729-112	0.05	0.4	8.71	0
3-SAT problems	0.05	0.05	11.21	0
Overall	0.05	0.05	23.82	2.96

with positive solve percentages. The Pythagorean solve percentages are almost identical, but the solve percentages for the Steiner triples are three to four times higher for the multiplicative configurations than the additive configurations.

The second major difference between the tables is that the multiplicative configurations performed better on the 3-SAT random instances. Compared to the best additive configuration, the multiplicative configuration of $(a_>, a_\leq) = (0.05, 0.05)$ found assignments that had about 50% fewer unsatisfied clauses at a timeout of five million flips. The improvement is strong evidence that multiplicative configurations should be preferred for random SAT instances, as there does not appear to be a similar improvement for structured instances.

Looking at the configuration value-pairs themselves, we see further evidence that DDFW should not take away weight from maximum-weight neighbors too quickly. Almost all $a_>$ values were at most 0.1, meaning that only 10% of the clause's weight was being transferred away in local minima. There are some exceptions to this rule: the best configuration for the plain7824 instance was $(a_>, a_\leq) = (0.5, 0.5)$, which is a much greater percentage of weight than 10%; and the best matrix configuration was $(a_>, a_\leq) = (0.1, 0.05)$, meaning that more weight is transferred from satisfied neighbors with weight above w_{init} than below it. But overall, the trend seems to hold.

We now consider Figure 3. Unlike for the additive parameter search, all four plots show the same trend: that the value of $a_>$ almost exclusively determines the performance of DDFW. Note that for any fixed $a_>$, ranging across the $a_\leq$ values gives a nearly flat line. The only plot that deviates from the general shape of the other three is the plot for the matrix instances. For that plot, there is a greater range of $(a_>, a_\leq)$ values that gives u values close to the minimum. In particular, the near-minimum configurations about $(a_>, a_\leq) = (0.05, 0.3)$ echo

the optimal configuration for the matrix problems in the additive parameter search from Table 2.

The trend in all four plots is to have a minimum near $(a_>, a_\leq) = (0.05, 0.05)$. Thus, an additional multiplicative parameter search was conducted on $(a_>, a_\leq)$ from 0.005 to 0.05 in steps of 0.005. A 3D plot of the results is shown in Figure 4.

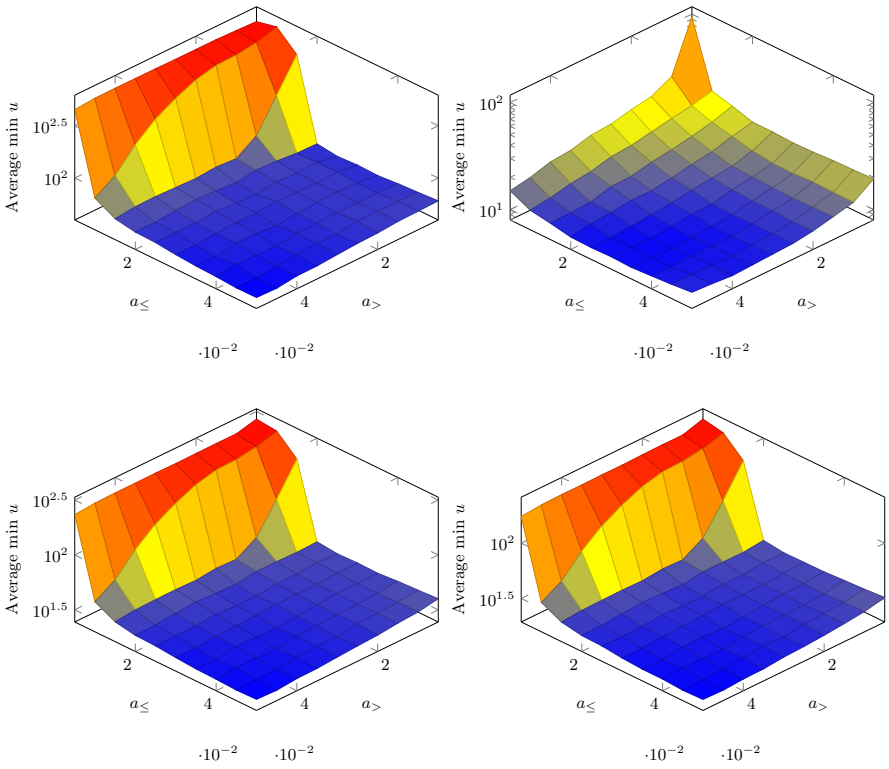

Fig. 4. 3D log plots of the average lowest number of unsatisfied clauses found at a timeout of five million flips for values of $(a_>, a_\leq)$ from 0.005 to 0.05 in steps of 0.005. Each pair of a values was run for 100 iterations on each test instance, and the average u values at timeout were taken. Reading from the top-left, the plots show the parameter search on the matrix instances, the 3-SAT instances, all remaining instances, and an overall average across all instances. Note that the axes are reversed from earlier figures, and that the u-axis is a log plot to better show the change in u value at the minimum.

Figure 4 shows that most $(a_>, a_\leq)$ pairs near (0.05, 0.05) gives u value close to the minimum for most instances. However, the only problem instances that showed improvement over those configurations in the original multiplicative parameter search were

- Steiner-243-45: optimal $(a_>, a_\leq) = (0.005, 0.035)$ with Avg $u(\mathcal{F}, \alpha) = 0.03$ and solve percentage 97, versus 0.32 and 70
- Steiner-405-70: optimal $(a_>, a_\leq) = (0.015, 0.01)$ with Avg $u(\mathcal{F}, \alpha) = 1$ but solve percentage 0, versus 1.45 and 16
- Steiner-729-112: optimal $(a_>, a_\leq) = (0.005, 0.005)$ with Avg $u(\mathcal{F}, \alpha) = 1.11$, versus 8.71
- 3-SAT problems: optimal $(a_>, a_\leq) = (0.04, 0.035)$ with Avg $u(\mathcal{F}, \alpha) = 10.0$ versus 11.21

These improvements are not suprising: those with optimal values near $(a_>, a_\leq) = (0.05, 0.05)$ were the only ones that benefitted from the additional parameter search. What is surprising is the improvement in the solve percentage for the first Steiner instance and the drop in the u value for the third Steiner instance.

We now turn to performing a parameter search on both additive and multiplicative values. We set $a_> = a_\leq$ and $c_> = c_\leq$ as we vary the multiplicative and additive values. We vary a from 0 to 0.5 in steps of 0.05 and c from -5 to 50 in steps of 5. The results are shown in Figure 5 and Table 4.

Table 4. Configurations of multiplicative and additive constants a and c for values of 0 to 0.5 in steps of 0.05 for a and values of -5 to 50 in steps of 5 for c which give the least number of unsatisfied clauses before timeout at five million flips, on average. The best configuration of (a, c) pairs are reported for each instance or group of instances, and the best configuration over all instances is reported at the bottom.

Instance	a	c	Avg $u(\mathcal{F}, \alpha)$	% solve
Matrix problems	0.1	5	45.29	0
asias-20	0	45	2.48	0
asias-34	0.05	10	11.14	0
bce7824	0.2	45	0.7	33
plain7824	0.25	50	0.99	22
Steiner-243-45	0.15	5	0.22	78
Steiner-405-70	0.1	50	1.42	37
Steiner-729-112	0.2	45	11.39	0
3-SAT problems	0.1	5	10.16	0
Overall	0.1	5	22.05	2.67

The results in Table 4 give similar u values to what we've seen before. Of note is that of the three parameter searches, the combined additive-multiplicative overall best configuration has the lowest average u value. These results indicate that the linear rule is not worse than either an additive-only or a multiplicative-only rule for weight transfer, as seen in DDFW and SAPS, and may in fact be better for DDFW. Further work could examine a more in-depth parameter search across all four four parameter values $(a_>, c_>)$ and $(a_\leq, c_\leq)$.

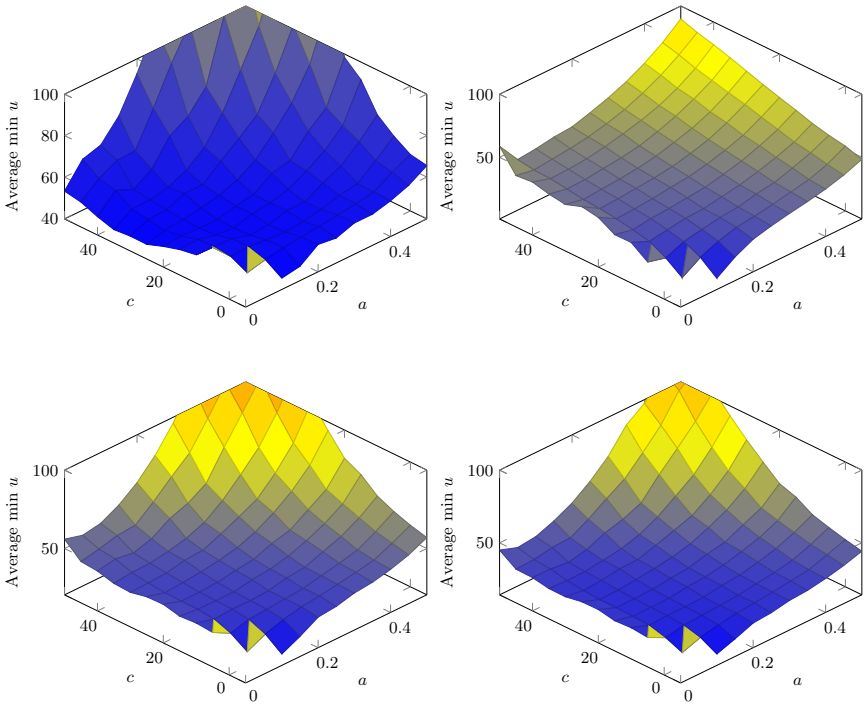

Fig. 5. 3D plots of the average lowest number of unsatisfied clauses found at a timeout of five million flips for values of (a, c) of 0 to 0.5 in steps of 0.05 for a and -5 to 50 in steps of 5 for c. Each pair of a and c values was run for 100 iterations on each test instance, and the average u values at timeout were taken. Reading from the top-left, the plots show the parameter search on the matrix instances, the 3-SAT instances, all remaining instances, and an overall average across all instances. Not pictured is a jump up of the graph to u values at least 1000 at values near $(0, 0)$.

We make an additional note about the configurations in Table 4: when combined with additive constants, the preferred value for a is close to 0, meaning that the beneficial effect of the multiplicative constant in the linear rule is small. However, this may be an artifact of varying the a and c values together. A parameter search across all four values may reveal better insight into the how the a and c values interact.

To summarize our findings in this subsection: we explored the performance of DDFW under three types of linear transfer rule. In all three cases, we found an overall configuration for DDFW that performed better than the UBCSAT baseline in Table 1. What's more, the overall best configuration for the linear rule was $(a, c) = (0.1, 5)$ and achieved an average u value that *was lower than the average minimum* u *value* in the UBCSAT baseline. Such a configuration represents an overall 30% improvement in the number of unsatisfied clauses found at timeout and a 3x improvement in the solve rate of DDFW on the test set.

5.5 Clause transfer group results

We next examine how transferring weight from larger groups of clauses in local minima affects the performance of DDFW. We start by looking at applying the additive-constants only linear rule to the entire neighborhood of an unsatisfied clause.

A parameter search like the one in Figure 2 was performed with the individual transfer method on the entire neighborhood. The results of the parameter search are shown in Figure 6.

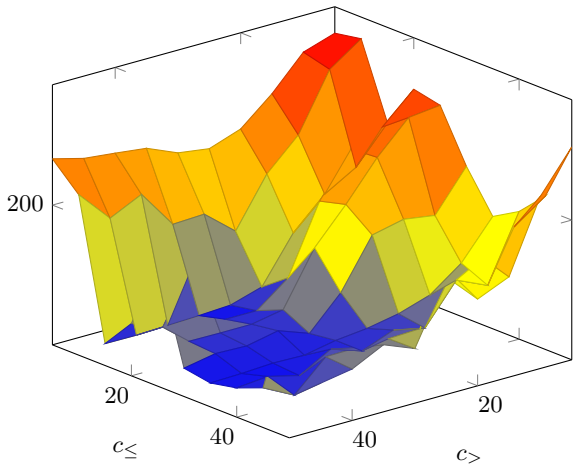

Fig. 6. A plot of the average number of unsatisfied clauses remaining after two million flips when the individual transfer method is used on the entire neighborhood. The average is taken over all test instances.

These results are initially disappointing. The minimum occurs at $(c_>, c_\leq) = (45, 15)$ and takes a u value of 47.224, which is already worse than the UBCSAT DDFW baseline in Section 5.2. However, a little bit of thought shows that this parameter search is inappropriate for the transfer method being considered. In a local minima, weight is being transferred from *each* satisfied neighbor clause. Thus, if we want to transfer about the same amount of weight as in UBCSAT DDFW, then we should take the amount of weight we want to transfer (suppose 25% of w_{init}) and divide by the neighborhood size. An analysis of the test set showed that the average neighborhood size was about 15. Therefore, we should transfer about 2 units of weight from each clause in the neighborhood. A new parameter search confirms that this is indeed the case. Refer to Figure 7.

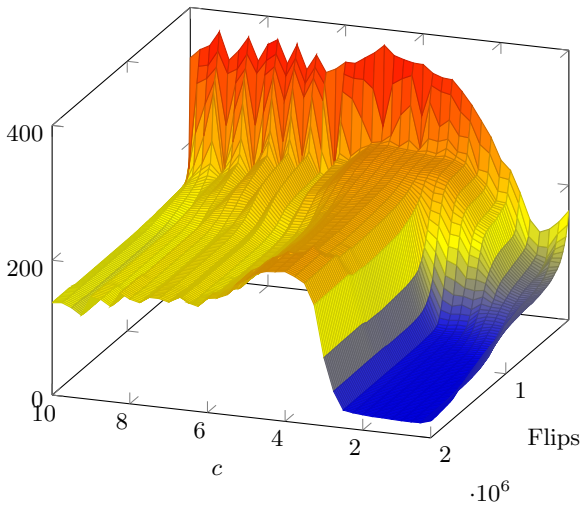

Fig. 7. A plot of the lowest number of unsatisfied clauses on average over two million flips. The individual transfer method is used here. Note the trough feature between c values of 0 and 4. The minimum occurs at $c = 0.75$ with a u value of 20.346.

The minimum of the figure occurs at $c = 0.75$ with a u value of 20.346. When compared to the UBCSAT baseline of 40.09, the individual transfer method shows an almost 50% improvement on the performance of the algorithm. It is clear that taking the same amount of weight, but taken from a larger number of clauses, leads to better performance on the number of unsatisfied clauses.

6 Conclusions and future work

In this thesis, we examined the SLS SAT solver DDFW in-depth. We began by reviewing the algorithm in its original form (Algorithm 4). We then turned to modifying key portions of the algorithm to increase the effectiveness of how DDFW chooses which variables to flip (Section 4.2) and how DDFW distributes weight between clauses to escape local minima (Section 4.3). We took these modifications and tested DDFW against a set of modern hard SAT benchmarks. Parameter searches across the constants used in the linear transfer rule for singular and neighborhood weight transfer gave generally successful results, and optimal configurations for three types of parameter search each improved the performance of DDFW when compared to a baseline performance from the UBCSAT implementation of DDFW.

In particular, we found that the overall best configuration for the linear transfer rule with a transfer of weight between the maximum-weight clause and an unsatisfied clause was $(a, c) = (0.1, 5)$. The configuration gave a 30% improvement in the average best u value found over five million flips and gave a 3x increased solve rate for those instances with positive solve rates.

We also found that DDFW worked best when weight was taken from entire neighborhoods, as opposed to a single satisfied neighbor. The best configuration found in this thesis—transferring weight from all clauses in a neighborhood by applying a linear transfer rule to each clause—achieved a 50% improvement over the original DDFW algorithm.

The results presented above are heartening to the study of DDFW and similar W_U-minimizing SLS algorithms. However, this thesis was not exhaustive in its investigation into these new methods of variable selection and clause reweighting strategies. As remarked earlier, there are two directions for future work. The first can focus on examining whether there is synergy between the "make-vs-break" variable selection probability distribution of ProbSAT and the unsatisfied weight reducing variable distribution of DDFW. The paper that introduced ProbSAT reported that the effect of "make" could be ignored, instead favoring flipping variables which do not cause satisfied clauses to become unsatisfied. Perhaps that heuristic is effective in DDFW as well: flip variables which do not cause satisfied clauses to contribute to the unsatisfied weight.

The second new research direction posed by this thesis is in which weight transfer rule is optimal in local minimua. We found that spreading weight transfer across a larger number of clauses caused better performance. However, the weight transfer rule being applied was always the linear transfer rule, regardless of the number of clauses it was applied to. Future work could focus on using more complex weight transfer rules, such as exponential or logistic functions. More complex functions would allow for more aggressive weight transfer, or perhaps weight transfer more sensitive to the distribution of weight in the clause neighborhood.

A third research direction, not immediately suggested by this thesis, but always lurking in the SAT literature, is injecting more randomness into DDFW. Randomness like in Walksat have been studied for u-minimizing algorithms, but

the literature lacks the same level of study for W_U-minimizing algorithms. Work into random flips or "random reweights" between clauses may show promise.

7 Acknowledgements

Through the course of my research, I connected with many bright minds and communities that I thank here. First and foremost, I thank my advisor, Marijn Heule. He was patient with me as he introduced me to the conventions of writing research publications, the tricks of implementing a SAT solver, and the mindset behind research and methodical scientific exploration. When I was deep in the weeds of implementing a SAT solver or examining experimental data, he always brought me back to the big picture with a simple question or observation.

Next, I thank the StarExec community for providing me with the computational resources needed to run my experiments.

Finally, I thank my friends. They've stuck with me through the course of this thesis and this pandemic, and they have done so much to keep me sane and happy this school year. I will miss them dearly as we all advance to our next stage in life. "Ekwal pai four ekwal slai."

References

1. Ahmed, T., Kullmann, O., Snevily, H.: On the van der waerden numbers w(2;3,t). Discrete Applied Mathematics **174** (09 2014). https://doi.org/10.1016/j.dam.2014.05.007
2. Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In: Cimatti, A., Sebastiani, R. (eds.) Theory and Applications of Satisfiability Testing – SAT 2012. pp. 16–29. Springer Berlin Heidelberg, Berlin, Heidelberg (2012)
3. Biere, A., Heule, M., van Maaren, H., Walsh, T.: Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications. IOS Press, NLD (2009)
4. Biere, A.: Lingeling, Plingeling, PicoSAT, and PrecoSAT at SAT race 2010. Tech. Rep. 1 (2010), http://fmv.jku.at/papers/Biere-FMV-TR-10-1.pdf
5. Biere, A.: Yet another Local Search Solver and Lingeling and friends entering the SAT Competition 2014. In: Balint, A., Belov, A., Heule, M.J.H., Järvisalo, M. (eds.) Proceedings of SAT Competition 2014. vol. 2014, pp. 39–40. University of Helsinki (2014)
6. Biere, A.: CaDiCaL, Lingeling, Plingeling, Treengeling, and YalSAT entering the SAT competition 2017. Tech. rep. (2017), https://helda.helsinki.fi/bitstream/handle/10138/224324/sc2017-proceedings.pdf
7. Braam, F., Moes, M., Suilen, E., Berg, D.V.D., Bhulai, S.: Almost squares in almost squares: solving the final instance. In: DATA ANALYTICS 2016 (2016)
8. Brakensiek, J., Heule, M., Mackey, J., Narváez, D.: The resolution of keller's conjecture. In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning. pp. 48–65. Springer International Publishing, Cham (2020)
9. Cai, S., Luo, C., Su, K.: Ccanr: A configuration checking based local search solver for non-random satisfiability. In: Heule, M., Weaver, S. (eds.) Theory and Applications of Satisfiability Testing – SAT 2015. pp. 1–8. Springer International Publishing, Cham (2015)
10. Clarke, E., Kroening, D., Lerda, F.: A tool for checking ansi-c programs. In: Jensen, K., Podelski, A. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 168–176. Springer Berlin Heidelberg, Berlin, Heidelberg (2004)

11. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the Third Annual ACM Symposium on Theory of Computing. p. 151–158. STOC '71, Association for Computing Machinery, New York, NY, USA (1971). https://doi.org/10.1145/800157.805047, https://doi.org/10.1145/800157.805047
12. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem-proving. Commun. ACM **5**, 394–397 (1962)
13. Franco, J., Paull, M.: Probabilistic analysis of the davis putnam procedure for solving the satisfiability problem. Discrete Applied Mathematics **5**(1), 77–87 (1983). https://doi.org/https://doi.org/10.1016/0166-218X(83)90017-3, https://www.sciencedirect.com/science/article/pii/0166218X83900173
14. Fuhs, C., Giesl, J., Middeldorp, A., Schneider-Kamp, P., Thiemann, R., Zankl, H.: Sat solving for termination analysis with polynomial interpretations. In: Marques-Silva, J., Sakallah, K.A. (eds.) Theory and Applications of Satisfiability Testing – SAT 2007. pp. 340–354. Springer Berlin Heidelberg, Berlin, Heidelberg (2007)
15. Gableske, O., Heule, M.J.H.: Eagleup: Solving random 3-sat using sls with unit propagation. In: Sakallah, K.A., Simon, L. (eds.) Theory and Applications of Satisfiability Testing - SAT 2011. pp. 367–368. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
16. Gent, I.P.: On the stupid algorithm for satisfiability. Tech. rep., Tech. rep. APES-02-1998, APES Research Group. Available (1998)
17. Goldberg, A.: Average case complexity of the satisfiability problem. In: Proceedings of the 4th Workshop on Automated Deduction. pp. 1–6 (1979)
18. Heule, M.: Schur number five (2018), https://www.aaai.org/ocs/index.php/AAAI/AAAI18/paper/view/16952
19. Heule, M.J.H.: Solving edge-matching problems with satisfiability solvers (2009)
20. Heule, M.J.H., Kauers, M., Seidl, M.: Local search for fast matrix multiplication. In: Janota, M., Lynce, I. (eds.) Theory and Applications of Satisfiability Testing – SAT 2019. pp. 155–163. Springer International Publishing, Cham (2019)
21. Heule, M.J.H., Kullmann, O., Marek, V.W.: Solving and verifying the boolean pythagorean triples problem via cube- and-conquer. In: Creignou, N., Le Berre, D. (eds.) Theory and Applications of Satisfiability Testing – SAT 2016. pp. 228–245. Springer International Publishing, Cham (2016)
22. Hoos, H.: On the runtime behavior of stochastic local search algorithms for SAT. In: Proceedings of AAAI'99. pp. 661–666 (1999)
23. Hossen, M.S., Polash, M.M.A.: Implementing an efficient sat solver for structured instances. In: 2019 Joint 8th International Conference on Informatics, Electronics Vision (ICIEV) and 2019 3rd International Conference on Imaging, Vision Pattern Recognition (icIVPR). pp. 238–242 (2019). https://doi.org/10.1109/ICIEV.2019.8858519
24. Hutter, F., Tompkins, D.A.D., Hoos, H.H.: Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In: Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming. p. 233–248. CP '02, Springer-Verlag, Berlin, Heidelberg (2002)
25. Ishtaiwi, A., Abu Al-Haija, Q.: Dynamic initial weight assignment for maxsat. Algorithms **14**(4) (2021). https://doi.org/10.3390/a14040115, https://www.mdpi.com/1999-4893/14/4/115
26. Ishtaiwi, A., Thornton, J., Anbulagan, Sattar, A., Pham, D.N.: Adaptive clause weight redistribution. In: Benhamou, F. (ed.) Principles and Practice of Constraint Programming - CP 2006. pp. 229–243. Springer Berlin Heidelberg, Berlin, Heidelberg (2006)

27. Ishtaiwi, A., Thornton, J., Sattar, A., Pham, D.N.: Neighbourhood clause weight redistribution in local search for sat. In: van Beek, P. (ed.) Principles and Practice of Constraint Programming - CP 2005. pp. 772–776. Springer Berlin Heidelberg, Berlin, Heidelberg (2005)
28. Kautz, H., Selman, B.: Planning as satisfiability. pp. 359–363 (01 1992)
29. Koutsoupias, E., Papadimitriou, C.H.: On the greedy algorithm for satisfiability. Information Processing Letters **43**(1), 53–55 (1992). https://doi.org/https://doi.org/10.1016/0020-0190(92)90029-U, https://www.sciencedirect.com/science/article/pii/002001909290029U
30. Lynce, I., Marques-Silva, J.a.: Efficient haplotype inference with boolean satisfiability. In: Proceedings of the 21st National Conference on Artificial Intelligence - Volume 1. p. 104–109. AAAI'06, AAAI Press (2006)
31. Marques Silva, J.P., Sakallah, K.A.: GRASP-a new search algorithm for satisfiability. In: Proceedings of International Conference on Computer Aided Design. pp. 220–227 (1996)
32. Mitchell, D., Selman, B., Levesque, H.: Hard and easy distributions of sat problems (07 1992)
33. Resende, M., Toso, R., Gonçalves, J., Silva, R.: A biased random-key genetic algorithm for the steiner triple covering problem. Optimization Letters **6**, 605–619 (04 2011). https://doi.org/10.1007/s11590-011-0285-3
34. Schuurmans, D., Southey, F.: Local search characteristics of incomplete sat procedures. Artificial Intelligence **132**(2), 121–150 (2001). https://doi.org/https://doi.org/10.1016/S0004-3702(01)00151-5, https://www.sciencedirect.com/science/article/pii/S0004370201001515
35. Selman, B., Kautz, H., Cohen, B.: Local search strategies for satisfiability testing. Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge **26** (09 1999)
36. Selman, B., Kautz, H.A., Cohen, B.: Noise strategies for improving local search. In: Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1). p. 337–343. AAAI '94, American Association for Artificial Intelligence, USA (1994)
37. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: Proceedings of the Tenth National Conference on Artificial Intelligence. p. 440–446. AAAI'92, AAAI Press (1992)
38. Selman, B., Mitchell, D.G., Levesque, H.J.: Generating hard satisfiability problems. Artificial Intelligence **81**(1), 17–29 (1996). https://doi.org/https://doi.org/10.1016/0004-3702(95)00045-3, https://www.sciencedirect.com/science/article/pii/0004370295000453, frontiers in Problem Solving: Phase Transitions and Complexity
39. Stump, A., Sutcliffe, G., Tinelli, C.: StarExec: A cross-community infrastructure for logic solving. In: International Joint Conference on Automated Reasoning. vol. 8562, pp. 367–373. Springer International Publishing (2014)
40. Thornton, J., Pham, D.N., Bain, S., Ferreira, V.: Additive versus multiplicative clause weighting for sat. In: Proceedings of the 19th National Conference on Artifical Intelligence. p. 191–196. AAAI'04, AAAI Press (2004)
41. Tompkins, D.A.D., Hoos, H.H.: UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In: Hoos, H., Mitchell, D. (eds.) Revised Selected Papers from the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004). Lecture Notes in Computer Science, vol. 3542, pp. 306–320. Springer Berlin, Heidelberg (2005)

42. Tucker, C., Shuffelton, D., Jhala, R., Lerner, S.: Opium: Optimal package install/uninstall manager. In: 29th International Conference on Software Engineering (ICSE'07). pp. 178–188 (2007). https://doi.org/10.1109/ICSE.2007.59