A Study of Divide and Distribute Fixed Weights and its Variants*

Cayden R. Codel and Marijn J. H. Heule

Carnegie Mellon University, Pittsburgh, Pennsylvania, United States
{ccodel,mheule}@cs.cmu.edu

Abstract

Divide and Distribute Fixed Weights (DDFW) is a stochastic local search Boolean satisfiability (SAT) algorithm 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 Yet despite its success, DDFW has received little research interest, and its initial results are out of date with respect to more modern SAT benchmarks.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 algorithmic variants based off of ideas from similar SAT algorithms such as ProbSAT and SAPS. We then test these variants against a set of modern hard benchmarks. We present three main findings. The first is a confirmation that a greedy variable selection process is optimal for DDFW. The second is that a linear weight transfer rule is more effective than a fixed additive one. Testing reveals a linear transfer rule configuration that performs 40% better than the original DDFW algorithm. The third is that it is sometimes more effective for unsatisfied clauses to borrow clause weight from their entire neighborhood rather than a singular clause in local minima.

1 Introduction

Stochastic local search (SLS) is the SAT-solving paradigm of choice for random SAT instances and for several classes of structured problem instances, such as the n-queens problem and some graph coloring problems [18, 19]. When it comes to a majority of structured problem instances, conflict-driven clause learning (CDCL) (cf. [3], Chapter 4) is usually the more effective solving method. Yet every rule has its exceptions: CNF encodings of matrix multiplication problems [10] that appear hard for CDCL solvers are solved by YalSAT [4], an SLS solver developed in 2014, in minutes on a single CPU. Another exception is the SLS algorithm DDFW [15]. DDFW is the only algorithm in the SLS-framework UBCSAT [22] able to solve the Pythagorean triples $n = 7284$ instance [11] on a single CPU with a timeout of one million flips, and it does so in under a minute. Such a quick solve time is remarkable when compared to the thousands of CPU hours of CDCL search that were unsuccessful in solving the $n = 7824$ instance. The success of DDFW on this challenging instance motivates our examination of its techniques in this paper.

At a high level, DDFW associates a weight to each clause and then attempts to find a satisfying assignment by flipping variables that minimize the amount of weight held by the unsatisfied clauses. When DDFW reaches a state where flipping any variable would increase the amount of unsatisfied weight, it doesn't flip a variable and instead increases the weight of the unsatisfied clauses and decreases the weight of the satisfied clauses. Eventually, DDFW returns to a state where it may flip variables to reduce the unsatisfied weight.

The method of assigning a weight to each clause and then minimizing the amount of unsatisfied weight is not a new SLS solving technique: DDFW is derived from the ideas present in the SLS algorithms SAPS ("Scaling and Probabilistic Smoothing") [12] and PAWS ("Pure

*Based on the research thesis available at https://contrib.andrew.cmu.edu/~ccodel/ddfw_thesis.pdf

Additive Weighting Scheme") [21]. Both SAPS and PAWS flip variables that reduce the amount of unsatisfied weight at each step, and both algorithms reweight clauses in local minima. The novelty of DDFW is to reweight clauses along neighborhood relationships. Transferring weight directly from satisfied clauses to unsatisfied clauses ensures that the total amount of clause weight remains constant, thus ridding DDFW of the need for a weight renormalization step—a step present in both SAPS and PAWS.

In this paper, we propose three DDFW variants. The first variant changes how DDFW selects which variable to flip at each step. The second variant changes how DDFW computes how much weight to transfer in local minima. The third variant builds on the second and changes how DDFW moves weight within clause neighborhoods. We then test these variants against a set of modern hard SAT benchmarks to see how the performance of the variants compares to the base algorithm. We present three main findings:

- DDFW flips those variables that reduce the amount of weight held by the unsatisfied clauses the most at each step. We propose two new distributions from which to select variables to flip: a proportional distribution and a uniform distribution. **Testing showed that the original distribution is the optimal of the three.**
- In local minima, DDFW distributes weight from satisfied clauses to unsatisfied clauses in fixed constant amounts. We propose a linear transfer rule that transfers more weight from a clause the more weight it has. Experiments show that certain configurations of the linear rule **perform 40% better** and have **a 3x solve rate** when compared to the original DDFW algorithm.
- In local minima, DDFW distributes weight from one satisfied clause to one unsatisfied clause at a time. We propose a new method of transferring weight from entire neighborhoods. Our experimental results show that distributing the same amount of weight across a larger number of clauses gives a **45% improvement** on the most challenging problem instances, but does not perform as well on easier instances.

The organization of the paper is as follows: we cover preliminaries in Section 2. Prior work on similar SLS solvers to DDFW is presented in Section 3. The base DDFW algorithm and its variants are presented in Section 4. Our experimental setup and results are discussed in Section 5. We make concluding remarks and propose future work in Section 6.

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. An assignment of true and false values to the Boolean variables that makes the formula evaluate to true is called a *satisfying assignment.* It is well known that any Boolean formula can be efficiently converted to conjunctive normal form (CNF), where the formula $\mathcal{F} = \{C_1, C_2, ..., C_m\}$ is expressed as an indexed set of disjunctions joined by conjunctions:

$$\mathcal{F} = \bigwedge_{j=1}^{m} \bigvee_{k=1}^{|C_j|} v_k$$

where $v_k \in \{x_i, \overline{x}_i\}$ for some $i \in [1, n]$.

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 $u(\mathcal{F}, \alpha) := |U(\mathcal{F}, \alpha)|$.

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. If we fix a clause $C_j \in \mathcal{F}$, then any clause $C_k \in \mathcal{F}$, $C_k \neq C_j$ is a *neighbor* of C_j if $|C_j \cap C_k| > 0$. If $v \in C_j \cap C_k$, then we say that C_j and C_k are neighbors on v. Let $N(C_j)$ give the set of all neighboring clauses $C_k \neq C_j$.

Some SLS algorithms associate a weight to each clause. Let us define a weighting function $W : \mathcal{F} \to \mathbb{R}^+$ that assigns a positive real value to each clause. We write $W_U(\mathcal{F}, \alpha)$ to mean the sum of weights of the unsatisfied clauses under an assignment α. Because $W(C_j) > 0$ for all $C_j \in \mathcal{F}$, then $W_U(\mathcal{F}, \alpha) = 0$ indicates that α is a satisfying assignment. Therefore, flipping the sign of variables that minimize W_U is a greedy method of solving CNF formulas. Variables that, when flipped, reduce W_U are called *unsatisfied weight reducing variables*. Let $R(\mathcal{F}, \alpha)$ be the set of unsatisfied weight reducing variables in $\mathcal{F}$ under α. When $|R(\mathcal{F}, \alpha)| = 0$, then a local minimum has been reached.

3 Prior work in SLS

The ideas in DDFW and the variants proposed in Section 4 are inspired by several SLS algorithms. We discuss those SLS algorithms here. Specifically, we discuss ProbSAT, an SLS algorithm that samples variables to flip from a larger distribution than DDFW does; and the family of solvers preceding DDFW, including SAPS and PAWS.

3.1 A look at ProbSAT

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 such strategy is to intentionally perturb α in a way that increases u by flipping a varaible at random (a "random walk"). Walksat [17] is a simple but successful and influential SLS algorithm that implements this idea.

Another successful strategy is implemented in ProbSAT [2]. Instead of flipping variables which reduce u globally, ProbSAT selects an unsatisfied clause at random and then flips a variable in that clause proportional to its "score" under a scoring function f. The paper that introduced ProbSAT [2] explored a polynomial and an exponential scoring function dependent on how many clauses become satisfied ("make") and how many clauses become unsatisfied ("break") when flipping a variable. Interestingly, it was discovered that the role of "make" could be ignored, and that an exponential function of only "break" scores was most effective.

The core idea of ProbSAT—generalizing the probability distribution that variables to flip are chosen from—can be extended to DDFW. DDFW selects variables to flip greedily, but different probability distributions may lead to a more effective algorithm.

3.2 A look at algorithms similar to DDFW

Many SLS 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. By assigning a weight to each clause, $W_U(\mathcal{F}, \alpha)$

becomes a generalization of $u(\mathcal{F}, \alpha)$, and SLS algorithms can minimize W_U rather than u. A greedy W_U-minimizing algorithm is to flip an unsatisfied weight reducing variable at each step. However, when $|R(\mathcal{F}, \alpha)| = 0$, then the SLS solver has arrived at a local minimum. The question arises of how to escape it.

SAPS ("Scaling and Probabilistic Smoothing") [12] is a W_U-minimizing SLS algorithm that 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

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

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 z is a normalization factor between 0 and 1. $\overline{W(\mathcal{F})}$ denotes the average clause weight before smoothing. In the publication introducing SAPS [12], the optimal values for these parameters were $a = 1.3$, $z = 0.8$, and $p_{\text{smooth}} = 0.05$. SAPS also had a random walk probability of $p_{\text{walk}} = 0.01$. Pseudocode of SAPS is presented in Algorithm 1.

Algorithm 1: SAPS($p_{\text{smooth}}, p_{\text{walk}}, a, z$)

```
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 return α
6  |  |  if |R(F, α)| > 0 then Flip a literal in R that reduces W_U the most
7  |  |  else
8  |  |  |  if rand(0,1) ≤ p_walk then Flip a random literal in F
9  |  |  |  else
10 |  |  |  |  for C_j ∈ U(F, α) do
11 |  |  |  |  |  W(C_j) ← a × W(C_j)
12 |  |  |  |  if rand(0,1) ≤ p_smooth then
13 |  |  |  |  |  for C_j ∈ F do
14 |  |  |  |  |  |  W(C_j) ← z × W(C_j) + (1 − z) × avg(W(F))
15 return "No satisfying assignment"
```

Algorithm 1 can be modified in two main ways. One way is to modify how variables from $R(\mathcal{F}, \alpha)$ are selected to be flipped, as in line 6. CCAnr [6] contains ideas of a possible modification. CCAnr incorporates the concepts of age and configuration checking in its variable selection, the latter meaning preventing variable flips if the neighborhood has not changed since the previous flip, preventing cycling. The heuristics of configuration changing appears to make SLS algorithms more effective on structured problem instances. However, modifications in this vein were not investigated in this paper.

Another way to modify Algorithm 1 is to change how clauses are reweighted in local minima. SAPS uses a multiplicative updating rule with smoothing. PAWS ("Pure Additive Weighting Scheme") [21] is a successor of SAPS that uses an additive rule instead of a multiplicative one. PAWS appeared to perform better than SAPS on several benchmarks, but the reason for this was not clear. The authors remarked that PAWS was more efficient, as it used integers for weights rather than floating-point values, and so it was able to perform more flips per second than SAPS. Yet the authors also pointed out that the floating-point weights of SAPS were

more expressive than the integer weights of PAWS, and so SAPS performed better on easier problem instances. Regardless of the exact reason for the discrepancy in performance, neither PAWS nor SAPS dominated the other in terms of performance or solve rates.

We note that SAPS uses a purely mutliplicative reweighting rule and that PAWS uses a purely additive one. The mixed successes of both algorithms suggests that more complex reweighting functions are promising candidates for algorithmic variation.

4 DDFW and its variants

Divide and Distribute Fixed Weights (DDFW) [15] is an SLS algorithm that seeks to minimize $W_U(\mathcal{F}, \alpha)$. It does so by assigning a weight to each clause. It 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 flipping unsatisified weight reducing variables begins anew.

Despite DDFW being the only SLS algorithm, to our knowledge, that exploits clause neighborhood relationships in local minima, remarkably little literature has been published on DDFW or its techniques [13, 14]. 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, 8] and in Microsoft's open-source Z3 Theorem Prover [7].

In this section, we present the original DDFW algorithm. We then extend prior work by proposing three algorithmic variants, which we test experimentally in Section 5.

4.1 DDFW

We present the DDFW algorithm as it appeared in its original publication [15]. 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. 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 $C_j \in U(\mathcal{F}, \alpha)$ and moves weight from one of its satisfied neighbors in $N(C_j)$ 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 no satisfied neighbors of weight at least w_{init}, 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 2.

Not mentioned in the original publication but appearing in the code supplementing it was the random walk in line 13. 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. The implementation used for experimentation included this random walk.

Two features of DDFW distinguish it from similar SLS algorithms. 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.

Algorithm 2: 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 return α
7  |   |   if |R(F, α)| > 0 then Flip a literal in R which decreases W_U the most
8  |   |   else if rand(0,1) ≤ 0.15 and a sideways move exists then
9  |   |   |   Flip a literal which does not increase W_U
10 |   |   else
11 |   |   |   for C_j ∈ U(F, α) do
12 |   |   |   |   C_k ← arg max_{C_k} {W(C_k) : C_k ∈ N(C_j), C_k satisfied}
13 |   |   |   |   if W(C_k) < w_init or rand(0,1) ≤ 0.01 then
14 |   |   |   |   |   C_k ← random satisfied clause with W(C_k) ≥ w_init
15 |   |   |   |   if W(C_k) > w_init then Transfer a weight of two from C_k to C_j
16 |   |   |   |   else Transfer a weight of one from C_k to C_j
17 return "No satisfying assignment"
```

The second distinguishing feature of DDFW is in which satisfied clauses share weight. DDFW exploits the properties of neighborhoods: 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 are 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 neighbors in how it transfers weight in a local minimum. DDFW chooses to increase the weight of all unsatisfied clauses in local minima, as opposed to decreasing the weight of all satisfied clauses, due to speedup achieved in practice, as the value of $u(\mathcal{F}, \alpha)$ is often low after thousands of flips.

4.2 A variation of the variable selection probability distribution

Line 7 in Algorithm 2 selects a random literal in $R(\mathcal{F}, \alpha)$ that causes the greatest decrease in W_U when flipped. However, we need not greedily choose which variable to flip. We can instead apply a probability distribution determined by a function f as in ProbSAT [2]. In this way, we may explore which method of unsatisfied weight reducing variable selection is optimal.

We propose two additional probability distributions here. The first is a 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. A directly linear function f is most straightforward. Future work could consider an exponential distribution, as in ProbSAT. Line 7 then becomes

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

where $\Delta W(x_i)$ is the reduction of unsatisfied weight $W_U(\mathcal{F}, \alpha)$ if the value of $\alpha(x_i)$ is flipped.

4.3 A variation of the weight transfer rule

Lines 15 and 16 in Algorithm 2 transfer fixed weights of 2 and 1 from satisfied neighboring clauses to unsatisfied clauses in local minima. For a value of $w_{\text{init}} = 8$, the percentage of a clause's weight moved in a single transfer is capped at 25% of w_{init}. 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 15 and 16 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 $0 < a < 1$, 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. The implementation of DDFW we used for experimentation 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 $W(C_k) \leq w_{\text{init}}$.

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 variant proposed 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 t could be introduced to control the number of maximum-weight neighbors to take weight from, any fixed t runs into the same problem that $t = 1$ 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.

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 proportional to that clause's weight. We call this kind of rule application the "proportional transfer method."

5 Experimental results

To test DDFW and the modifications proposed above, we 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 addition, our implementation is modular and allows for easy slotting in of additional variable selection heuristics and reweighting rules.

We take DDFW and the variants proposed in Section 4 and test them against a set of modern hard benchmarks. All experiments were conducted on the StarExec community servers [20]. 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. We thank the StarExec community for providing the computational resources. In all experiments except for the UBCSAT baseline, the initial clause weight $w_{\text{init}} = 100$ to allow for easy conversion to percentages of w_{init} transferred in local minima.

5.1 Benchmark instances and UBCSAT baseline

All benchmarks used can be found at https://github.com/marijnheule/benchmarks, 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. All instances in the test set are hard for state-of-the-art CDCL and SLS solvers. The benchmarks are: two encodings of the Pythagorean triples instance for $n = 7824$ [11], ten encodings of matrix multiplication problems [10], two encodings of the almost-squares-in-almost-square (asias) problem [5], three encodings of Steiner triple problems [16], and the ten 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.

DDFW is implemented in the SLS framework UBCSAT [22]. We used UBCSAT's implementation of DDFW as a performance baseline. The results are summarized in Table 1. The baseline shows that the matrix, 3-SAT, and Steiner-729 problem instances are the most challenging, while both encodings of the Pythagorean triples problem and the first two Steiner triples instances are the easiest, as they are the only four with positive solve percentages.

5.2 Variable selection probability distribution variation 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 unequivocally holds in Figure 1.

It is possible that DDFW favors one of the two newly proposed distributions under a different weight transfer rule. Yet it is our experience that the original variable selection distribution is the optimal of the three even under the linear transfer rule variants. Of course, these findings do not rule out the possibility that there exists a more optimal variable selection heuristic for

Table 1: Baseline results for DDFW on the test set using the UBCSAT implementation. The "$\overline{u}$" 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" 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.

Instance	$\overline{u}$	Min u	% solve
matrix	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	4.16	0	1
Steiner-405	4.98	0	4
Steiner-729	28.26	4	0
3-SAT	35.97	25.3	0
Overall	36.57	24.07	0.85

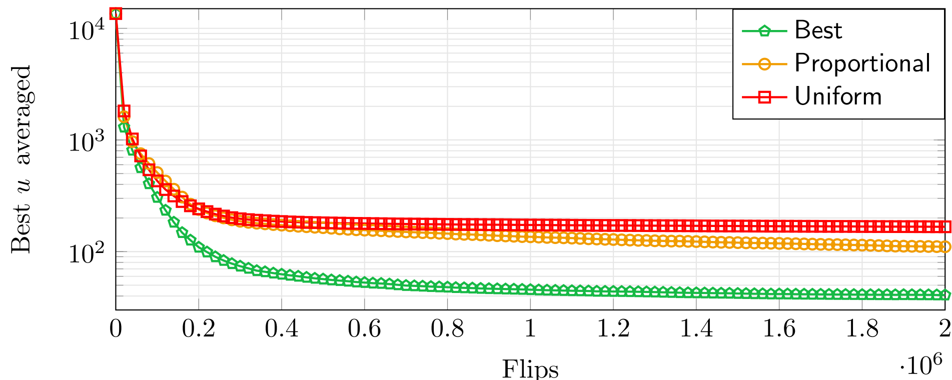

Figure 1: A log plot of the lowest number of unsatisfied clauses found over a number of flips. The u values are averaged over all problem instances, with 20 iterations per problem instance. "Best" refers to flipping variables which reduce W_U the most. "Proportional" refers to selecting variables proportional to their W_U reduction. "Uniform" refers to a uniform probability distribution across all unsatisfied weight reducing variables.

DDFW. For example, the configuration change method of CCAnr [6] mentioned in Section 3 is one such variable selection heuristic that may help DDFW. Due to the results in Figure 1, we conduct the remaining experiments using the original distribution of always flipping the best unsatisfied weight reducing variable whenever $|R(\mathcal{F}, \alpha)| > 0$.

5.3 Linear rule results

In Section 4.3, we proposed a linear weight transfer rule to be used in local minima. We examine the performance of DDFW using that linear transfer rule here.

Our implementation of DDFW follows the distinction made by the original DDFW algorithm

Table 2: Results from three parameter searches across $(c_>, c_\leq)$, $(a_>, a_\leq)$, and $(a_>, c_>)/(a_\leq, c_\leq)$ configurations. Each row presents the optimal configuration for that problem instance or set of instances discovered in each parameter search. The $\overline{u}$ columns report the average lowest $u(\mathcal{F}, \alpha)$ value found before timeout at five million flips. The % column reports the percentage of the 100 iterations that solved the instance by the configuration.

Instance	$(c_>, c_\leq)$	$\overline{u}$	%	$(a_>, a_\leq)$	$\overline{u}$	%	(a, c)	$\overline{u}$	%
matrix	(5, 25)	43.91	0	(0.1, 0.05)	45.87	0	(0.1, 5)	45.29	0
asias-20	(20, 45)	2.56	0	(0.1, 0.35)	2.37	0	(0, 45)	2.48	0
asias-34	(40, 30)	11.79	0	(0.1, 0.5)	10.54	0	(0.05, 10)	11.14	0
bce7824	(50, 45)	0.94	30	(0.3, 0.3)	1.03	31	(0.2, 45)	0.7	33
plain7824	(50, 50)	1.12	27	(0.5, 0.5)	1.11	28	(0.25, 50)	0.99	22
Steiner-243	(20, 5)	0.74	26	(0.05, 0.05)	0.32	70	(0.15, 5)	0.22	78
Steiner-405	(35, 5)	0.96	4	(0.25, 0.25)	1.45	16	(0.1, 50)	1.42	37
Steiner-729	(50, 45)	1.0	0	(0.05, 0.4)	8.71	16	(0.2, 45)	11.39	0
3-SAT	(10, 5)	21.29	0	(0.05, 0.05)	11.21	0	(0.1, 5)	10.16	0
Overall	(10, 25)	29.16	1.7	(0.05, 0.05)	23.82	2.96	(0.1, 5)	22.05	2.67

and applies a different linear rule depending on whether $W(C_j) > w_{\text{init}}$ or $W(C_j) \leq w_{\text{init}}$. Let us call the pair of additive-multiplicative parameters used in the former case $(a_>, c_>)$ and the pair used in the latter case $(a_\leq, c_\leq)$. To investigate the effects of these additive-multiplicative pairs, we perform three parameter searches. The first fixes $a_> = a_\leq = 1$ and varies the c values from 5 to 50 in steps of 5. In this way, the parameter search investigates the effect of only the additive constants, and so it can be viewed as a search on the PAWS heuristic. The second parameter search fixes $c_> = c_\leq = 0$ and varies the a values from 0.05 to 0.5 in steps of 0.05. This parameter search investigates the effects of only the multiplicative constants and can be viewed as a search on the SAPS heuristic. The final parameter search collapses the two linear rule pairs into one by setting $a_> = a_\leq$ and $c_> = c_\leq$. The a values are ranged from 0.05 to 0.5 in steps of 0.05, and the c values are ranged from 5 to 50 in steps of 5. All configurations were tested against the test set for 100 iterations with a timeout of five million flips. The averaged lowest $u(\mathcal{F}, \alpha)$ values and solve percentages for all three parameter searches are presented in Table 2. 3D plots of the parameter searches are displayed in Figure 2.

We make several observations about Table 2. The first is that there is evidence to suggest that it is not always optimal to transfer more weight from satisfied neighboring clauses with weight $W(C_j) > w_{\text{init}}$. Looking at the column for the $(c_>, c_\leq)$ parameter search, we see that the optimal configurations for the matrix problems, asias-20, the 3-SAT problems, and the overall best average configuration all had $c_> < c_\leq$. A reason for this trend could be that clauses with weight greater than w_{init} received that weight by being unsatisfied in a local minimum and so represent harder-to-satisfy clauses. Thus, DDFW should not be too hasty in transferring weight off of those clauses to maintain those clauses' status as harder to satisfy. However, the evidence for this trend is not too strong: there are fewer $(a_>, a_\leq)$ configurations that follow this trend, and when $a_> = a_\leq$, as in many optimal instance configurations, the effective result is that more weight is transferred when $a_>$ is applied compared to when $a_\leq$ is. We speculate that weight transfer rules that factor in "difficulty in satisfying" may present a promising research direction.

The second observation we make is that all three parameter searches support the assumptions made in the original DDFW publication regarding how much weight should be transferred in

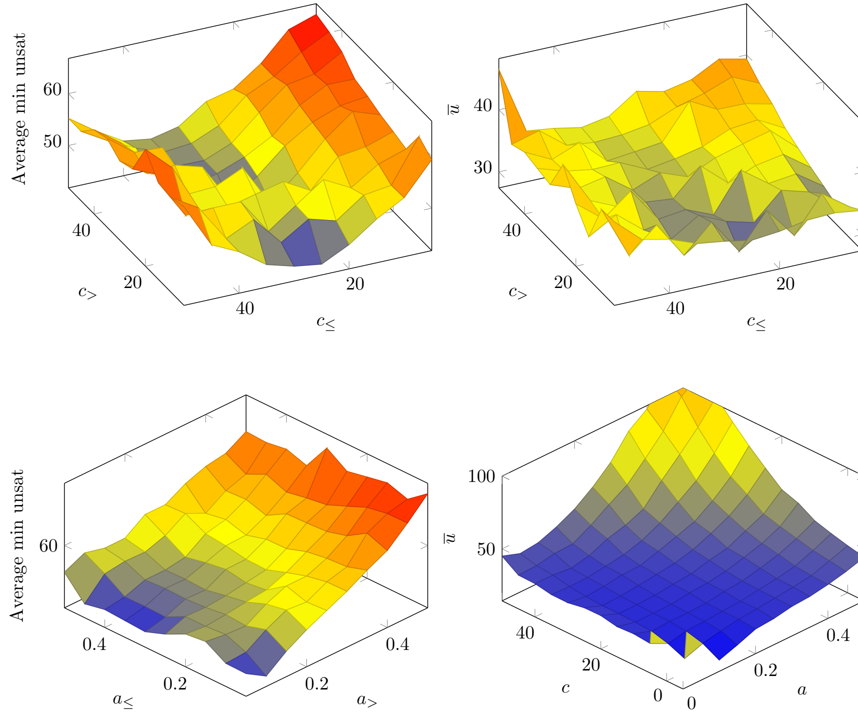

Figure 2: 3D plots of the three parameter searches performed on the linear rule. The $\overline{u}$ axis measures the average lowest number of unsatisfied clauses found before a timeout of five million flips. Reading from the top-left, the plots are of the constant parameter search on the matrix instances alone, the constant parameter search over all test instances, the multiplicative parameter search on all test instances, and the full linear parameter search on all test instances. Other than the matrix-only plot, the general shapes of the plots are shared across instances.

local minima. DDFW as in Algorithm 2 transfers 12.5% and 25% of w_{init} from satisfied clauses to unsatisfied clauses. Most of the configurations in Table 2 transfer 15-35% of w_{init}, which is close to the original transfer percentages.

The third observation is that all three parameter searches show an overall configuration that performs much better than the UBCSAT baseline. The best of the three overall configurations, $(a, c) = (0.1, 5)$, achieves a $\overline{u}$ value of 22.05 and a solve rate of 2.67%, which represents a 40% improvement and a 3x solve rate when compared to the UBCSAT baseline. These results strongly indicate that the linear rule is more successful than the fixed rule of Algorithm 2.

The final observation we make is that it is usually optimal to keep the multiplicative constant a smaller than the additive constant c. This trend holds when comparing the additive-only parameter search to the multiplicative-only parameter search and when looking at the (a, c) pairs in the third parameter search. The tendency for a to be close to 0 indicates that while taking more weight from clauses with greater weight values helps DDFW, the effect is beneficial only in small amounts.

We now turn to examining the 3D plots in Figure 2. The top-left plot in Figure 2 shows that for the additive-only parameter search on only the matrix instances, the value of $c_\leq$ was more influential on the performance than the value of $c_>$, but this trend did not hold over

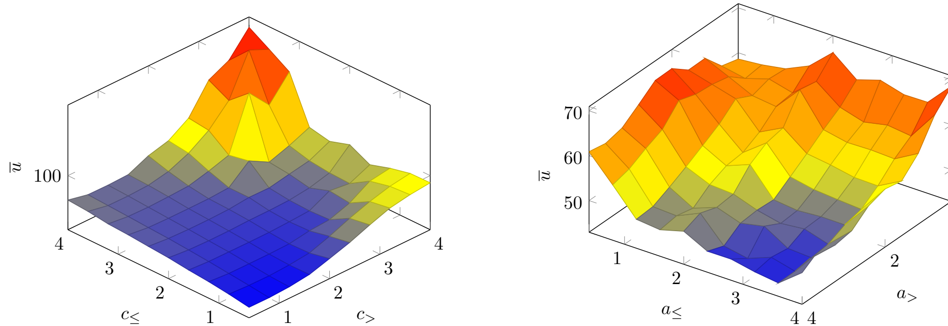

Figure 3: 3D plots of the average lowest number of unsatisfied clauses found at a timeout of five million flips over the entire test set. The left plot shows the $(c_>, c_\leq)$ parameter search on all test instances when the weight is distributed over the neighborhood proportional to each clause's weight, and the right plot shows the parameter search when the linear rule is applied to each individual clause. Note the change in axis between the two plots.

the entire test set, as shown in the top-right plot. The bottom-left plot shows that in the multiplicative-only parameter search, the value of $a_>$ was more influential on the $\overline{u}$ value than $a_\leq$, which is further evidence that DDFW should not take away weight from clauses with weight above w_{init} too quickly. Finally, the bottom-right plot echoes what is in Table 2: that optimal (a, c) pairs have relatively small values but that the effect of the optimal pairs is to transfer 15-35% of w_{init} from each clause in a local minimum.

5.4 Linear rule transfer method results

We next test the variant of DDFW that applies the weight transfer rule to every satisfied clause in the neighborhood. We perform a parameter search on only the additive constants $(c_>, c_\leq)$ across the two transfer methods (individual and proportional). Because the rule is being applied to many more clauses than one, we vary $(c_>, c_\leq)$ across smaller values that, when mutliplied by the average clause neighborhood size, approximates the amount of weight transferred in the typical linear rule. The values range from 0.4 to 4 in steps of 0.4. Table 3 summarizes the best configurations found in the parameter search. 3D plots of these parameter searches are shown in Figure 3.

The results in Table 3 are initially disappointing. The $\overline{u}$ values for the overall best average configurations do not even best those in the UBCSAT baseline in Table 1. However, the most interesting feature of Table 3 is that both the proportional and individual transfer methods achieve $\overline{u}$ values of about 30 on the matrix instances, which are the most challenging instances of the test set. When compared against the three linear rule parameter searches in Table 2, we see that the new transfer methods achieve a 33% improvement over the typical linear rule and about a 45% improvement over the UBCSAT baseline.

The lack of consistency in the results in Table 3 may be due to the disparate clause neighborhood sizes between the instances of the test set. For example, the average clause neighborhood size of the matrix instances was about 15 clauses, whereas the average neighborhood size of the 3-SAT instances was about 4. Future work could focus on how to tune how much weight should be transferred within clause neighborhoods of different sizes.

Table 3: Results from two additive-constant only parameter searches on the two proposed neighborhood transfer variants. The "$\overline{u}$" column reports the average value of the lowest number of unsatisfied clauses found for any flip before timeout at five million flips. The table reports an average over the matrix and 3-SAT problem instances due to their similarities in difficulty and solve times. The "% solve" column is omitted because no optimal configuration solved any instance on any of the 100 iterations.

Transfer method	Proportional		Individual	
Instance	$(c_>, c_\leq)$	$\overline{u}$	$(c_>, c_\leq)$	$\overline{u}$
matrix	(0.4, 0.8)	**27.00**	(4.0, 3.6)	**33.91**
asias-20	(0.8, 0.4)	2.79	(3.6, 3.2)	3.48
asias-34	(1.6, 3.6)	16.06	(2.4, 3.2)	24.19
bce7824	(2.0, 2.8)	11.39	(3.6, 2.8)	7.84
plain7824	(0.8, 0.8)	11.24	(4.0, 2.4)	8.97
Steiner-243	(3.2, 0.8)	1.2	(3.6, 2.4)	1.37
Steiner-405	(4.0, 0.8)	1.68	(1.6, 1.6)	1.8
Steiner-729	(3.2, 0.4)	2.13	(3.2, 0.4)	1.92
3-SAT	(0.8, 0.4)	94.02	(1.6, 0.4)	76.26
Overall	(0.8, 0.4)	46.91	(3.6, 2.8)	45.29

6 Conclusions and future work

The results presented above are heartening to the study of DDFW and similar W_U-minimizing SLS algorithms. We showed that through simple modification of the weight transfer method used to escape local minima, large improvements in the lowest number of unsatisfied clauses found before timeout and in solve rates can result.

However, this study was not exhaustive in its investigation of the three proposed variants to DDFW. As a result, there are many directions for future work. Already remarked above is the incorporation of configuration change methods from CCAnr into the variable selection distribution used by DDFW to choose which variables to flip. Also mentioned above is the opportunity for more complex transfer rules, such as higher-order polynomials or rules that take into account additional factors of clause neighborhood size, to be investigated in future work.

Future work could also focus on the effects of distributing clause weight across local structures in local minima. In this study, we focused on borrowing weight from entire satisfied neighborhoods instead of a single satisfied clause. Yet additional heuristics may be more effective, such as different underlying weight transfer rules or distributing weight across different local structures than clause neighborhoods. The space of heuristics to explore is vast, and as already shown in this study, slight modifications to the DDFW algorithm can result in significant improvements.

References

[1] Tanbir Ahmed, Oliver Kullmann, and Hunter Snevily. On the van der waerden numbers w(2;3,t). *Discrete Applied Mathematics*, 174, 09 2014.

[2] Adrian Balint and Uwe Schöning. Choosing probability distributions for stochastic local search and the role of make versus break. In Alessandro Cimatti and Roberto Sebastiani, editors, *Theory and Applications of Satisfiability Testing – SAT 2012*, pages 16–29, Berlin, Heidelberg, 2012. Springer.

[3] A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, NLD, 2009.

[4] Armin Biere. Yet another Local Search Solver and Lingeling and friends entering the SAT Competition 2014. In A. Balint, A. Belov, M. J. H. Heule, and M. Järvisalo, editors, *Proceedings of SAT Competition 2014*, volume 2014, pages 39–40. University of Helsinki, 2014.

[5] F. Braam, M. Moes, E. Suilen, D. V. D. Berg, and S. Bhulai. Almost squares in almost squares: solving the final instance. In *DATA ANALYTICS 2016*, 2016.

[6] Shaowei Cai, Chuan Luo, and Kaile Su. Ccanr: A configuration checking based local search solver for non-random satisfiability. In Marijn Heule and Sean Weaver, editors, *Theory and Applications of Satisfiability Testing – SAT 2015*, pages 1–8, Cham, 2015. Springer.

[7] Leonardo de Moura and Nikolaj Bjørner. Z3: An efficient smt solver. In C. R. Ramakrishnan and Jakob Rehof, editors, *Tools and Algorithms for the Construction and Analysis of Systems*, pages 337–340, Berlin, Heidelberg, 2008. Springer.

[8] Marijn J. H. Heule. Solving edge-matching problems with satisfiability solvers. 2009.

[9] Marijn J. H. Heule, Matti Järvisalo, and Martin Suda. Sat competition 2018. *Journal on Satisfiability, Boolean Modeling and Computation*, 11:133–154, 2019.

[10] Marijn J. H. Heule, Manuel Kauers, and Martina Seidl. Local search for fast matrix multiplication. In Mikoláš Janota and Inês Lynce, editors, *Theory and Applications of Satisfiability Testing – SAT 2019*, pages 155–163, Cham, 2019. Springer.

[11] Marijn J. H. Heule, Oliver Kullmann, and Victor W. Marek. Solving and verifying the boolean pythagorean triples problem via cube-and-conquer. In Nadia Creignou and Daniel Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 228–245, Cham, 2016. Springer.

[12] Frank Hutter, Dave A. D. Tompkins, and Holger H. Hoos. Scaling and probabilistic smoothing: Efficient dynamic local search for sat. In *Proceedings of the 8th International Conference on Principles and Practice of Constraint Programming*, CP '02, page 233–248, Berlin, Heidelberg, 2002. Springer-Verlag.

[13] Abdelraouf Ishtaiwi and Qasem Abu Al-Haija. Dynamic initial weight assignment for maxsat. *Algorithms*, 14(4), 2021.

[14] Abdelraouf Ishtaiwi, John Thornton, Anbulagan, Abdul Sattar, and Duc Nghia Pham. Adaptive clause weight redistribution. In Frédéric Benhamou, editor, *Principles and Practice of Constraint Programming - CP 2006*, pages 229–243, Berlin, Heidelberg, 2006. Springer.

[15] Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar, and Duc Nghia Pham. Neighbourhood clause weight redistribution in local search for sat. In Peter van Beek, editor, *Principles and Practice of Constraint Programming - CP 2005*, pages 772–776, Berlin, Heidelberg, 2005. Springer.

[16] Mauricio Resende, Rodrigo Toso, José Gonçalves, and Ricardo Silva. A biased random-key genetic algorithm for the steiner triple covering problem. *Optimization Letters*, 6:605–619, 04 2011.

[17] Bart Selman, Henry Kautz, and Bram Cohen. Local search strategies for satisfiability testing. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, 26, 09 1999.

[18] Bart Selman, Henry A. Kautz, and Bram Cohen. Noise strategies for improving local search. In *Proceedings of the Twelfth National Conference on Artificial Intelligence (Vol. 1)*, AAAI '94, page 337–343, USA, 1994. American Association for Artificial Intelligence.

[19] Bart Selman, Hector Levesque, and David Mitchell. A new method for solving hard satisfiability problems. In *Proceedings of the Tenth National Conference on Artificial Intelligence*, AAAI'92,

page 440–446. AAAI Press, 1992.

[20] Aaron Stump, Geoff Sutcliffe, and Cesare Tinelli. StarExec: A cross-community infrastructure for logic solving. In *International Joint Conference on Automated Reasoning*, volume 8562, pages 367–373. Springer, 2014.

[21] John Thornton, Duc Nghia Pham, Stuart Bain, and Valnir Ferreira. Additive versus multiplicative clause weighting for sat. In *Proceedings of the 19th National Conference on Artifical Intelligence*, AAAI'04, page 191–196. AAAI Press, 2004.

[22] Dave A. D. Tompkins and Holger H. Hoos. UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT and MAX-SAT. In Holger Hoos and David Mitchell, editors, *Revised Selected Papers from the Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004)*, volume 3542 of *Lecture Notes in Computer Science*, pages 306–320. Springer, 2005.