Additive versus Multiplicative Clause Weighting for SAT

John Thornton and **Duc Nghia Pham** and **Stuart Bain** and **Valnir Ferreira Jr.**
Institute for Integrated and Intelligent Systems, Griffith University
PMB 50, Gold Coast Mail Centre, 9726, Australia
email: {j.thornton, d.n.pham, s.bain, v.ferreira}@griffith.edu.au

Abstract

This paper examines the relative performance of additive and multiplicative clause weighting schemes for propositional satisfiability testing. Starting with one of the most recently developed multiplicative algorithms (SAPS), an experimental study was constructed to isolate the effects of multiplicative in comparison to additive weighting, while controlling other key features of the two approaches, namely the use of random versus flat moves, deterministic versus probabilistic weight smoothing and multiple versus single inclusion of literals in the local search neighborhood.

As a result of this investigation we developed a pure additive weighting scheme (PAWS) which can outperform multiplicative weighting on a range of difficult problems, while requiring considerably less effort in terms of parameter tuning. We conclude that additive weighting shows better scaling properties because it makes less distinction between costs and so considers a larger domain of possible moves.

Introduction and Background

Clause weighting algorithms for satisfiability testing have formed an important research area since their first introduction in the early 1990s. Since then various improvements have been proposed, resulting in the two best known algorithms of today: the discrete Lagrangian method (DLM) (Wu & Wah 2000) and scaling and probabilistic smoothing (SAPS) (Hutter, Tompkins, & Hoos 2002). While these methods differ in important aspects, both use the same underlying trap avoiding strategy: increasing weights on unsatisfied clauses in local minima and then periodically adjusting weights to maintain reasonable weight differentials during the search.

The earliest clause weighting algorithms, such as Breakout (Morris 1993), repeatedly increased weights on unsatisfied clauses and so allowed unrestricted weight growth during the search. Flips were then chosen on the basis of minimizing the combined weight of the unsatisfied clauses. In 1997, Frank proposed a new weight decay algorithm that updated weights on unsatisfied clauses using a combination of a multiplicative decay rate and an additive weight increase. While Frank's work laid the ground for future advances, his decay scheme produced relatively small improvements over earlier weighting approaches. At this point, clause weighting algorithms proved competitive on many smaller problems but were unable to match the performance of faster and simpler heuristics, such as Novelty, on larger problem instances (McAllester, Selman, & Kautz 1997). As a key reason for developing incomplete local search techniques is to solve problems beyond the reach of complete SAT solvers, the poor scalability of clause weighting was a major disadvantage.

It was not until the development of DLM that a significant performance gain was achieved. In its simplest form, DLM follows Breakout's weight increment scheme, but additionally decrements clause weights after a fixed number of increases. DLM also alters the point at which weight is increased by allowing *flat* moves that leave the weighted cost of the solution unchanged. These flat moves are in turn controlled by a tabu list and by a parameter which limits the total number of consecutive flat moves (Wu & Wah 2000). In empirical tests DLM proved successful at solving a range of random and structured SAT problems, and in particular was able to outperform the best non-weighting algorithms on many larger and more difficult problem instances.

In another line of research, Schuurmans and Southey (2000) developed a fully multiplicative weighting algorithm: smoothed descent and flood (SDF). SDF introduced a new method for breaking ties between equal cost flips by additionally considering the number of true literals in satisfied clauses. In situations where no improving moves are available, SDF multiplicatively increases weights on unsatisfied clauses and then normalizes (or *smooths*) clause weights so that the greatest cost difference between any two flips remains constant. SDF's reported flip performance was promising in comparison to DLM, but these results did not look at the more difficult problems for which DLM was especially suited. In addition, SDF's time performance did not compare well, due to the overhead of adjusting weights on all clauses at each local minimum.

In subsequent work, SDF evolved into the exponentiated subgradient algorithm (ESG) (Schuurmans, Southey, & Holte 2001), which in turn formed the basis of the scaling and probabilistic smoothing (SAPS) algorithm (Hutter, Tompkins, & Hoos 2002). ESG and SAPS dispensed with SDF's augmented cost function, and SAPS further improved on the run-time performance of ESG by only smoothing weights periodically, and only increasing weights on *vio-*

lated clauses in a local minimum[1].

The feature of greatest interest to the current study is that, ignoring the issue of additive versus multiplicative clause weighting, the weight update scheme of SAPS is almost identical in structure to the weight update scheme of DLM: both increase weight when a local minimum is identified (although using different identification criteria), and both periodically adjust weights according to a parameter value that varies for different problems[2]. SAPS differs from DLM only in using the parameter to probabilistically determine when weight is reduced, whereas DLM deterministically reduces weight after a fixed number of increases.

The aim of this study is to investigate whether an additive or multiplicative weight update scheme is better for satisfiability testing. Given that SAPS and DLM both have some claim to be considered as the state-of-the-art in local search for SAT and that both have separately hit upon a similar underlying weighting structure, it now becomes possible to compare additive and multiplicative clause weighting without their relative performance being disguised by differing implementation details. To perform this comparison, we started with the authors' original version of SAPS and changed it in small steps until it became an effective additive clause weighting algorithm. By examining and empirically testing the effect of each step, we set out to isolate exactly those features that are crucial for the success of each approach. This resulted in the development of a new pure additive weighting scheme (PAWS). As the published results for SAPS have only looked at relatively small problems, we also decided to evaluate SAPS and PAWS on an extended test set that includes a selection of the more difficult problems for which DLM was developed. In the remainder of the paper we describe in more detail the development of PAWS from SAPS and DLM, and then present the results and conclusions of our empirical study.

Clause Weighting Algorithms for SAT

DLM has been described as "ad hoc" (Schuurmans, Southey, & Holte 2001) and criticized for requiring a large number of parameters to obtain optimum performance. However, DLM has evolved through several versions, the last of which was developed specifically to solve the larger towers of Hanoi and parity learning problems from the DIMACS benchmarks (Wu & Wah 2000). As already discussed, the basic structure of DLM is similar to SAPS, except for the heuristic used to control the taking of flat moves. In addition, although the latest version of DLM has 27 parameters, in practice only three of these require adjustment in the SAT domain.

[1]ESG's approach is to scale and smooth the weight on all clauses in every local minima.

[2]Additionally, a third clause weighting algorithm, GLSSAT (Mills & Tsang 1999), uses a similar weight update scheme, additively increasing weights on the least weighted unsatisfied clauses and multiplicatively reducing weights whenever the weight on any one clause exceeds a predefined threshold. Although GLSSAT performed well in comparison with Walksat, it could not match DLM on larger problems and so is not considered further in this study.

Of particular interest is that DLM uses a single parameter to control the weighting process (corresponding to Max_{inc} in Figure 2), which determines when weights are to be reduced. In contrast, SAPS requires two further parameters (α and ρ) to determine the amount that weights are multiplicatively scaled or smoothed (in DLM clause weight increases and decreases are implemented by adding or subtracting one). The other two DLM parameters (θ_1 and θ_2) are used to control the flat move heuristic: Using the terms from Figure 1, if $best < 0$, DLM will randomly select and flip any $x_i \in L$. Otherwise, if $best = 0$, and the number of immediately preceding consecutive flat moves is $< \theta_1$ and $L_t \neq \emptyset$, then DLM will randomly select and flip any $x_i \in L_t$, where L_t contains all flat move literals that have not been flipped in the last θ_2 moves. Otherwise clause weights are additively updated, as per Figure 2.

Although SAPS implements a fairly "pure" weighting algorithm, there are a few implementation details that distinguish it from DLM (see Figure 1). The first is the wp parameter which probabilistically controls whether a random flip is taken when no improving cost move is available. This acts as an alternative to DLM's flat move heuristic. The second is that the set of local neighborhood of moves for SAPS contains a single copy of each literal that can *make* a false clause (i.e. turn it from false to true). In DLM, the neighborhood consists of all literals in all false clauses. This means that if a literal appears in more than one false clause, it will appear more than once in the local neighborhood, thereby increasing the probability that it will be selected. Finally, as noted earlier, SAPS uses *probabilistic* smoothing when adjusting clause weights, i.e. if P_{smooth} is set to 5% then there is a 1 in 20 chance that weight will be adjusted after an increase. In contrast, DLM's third parameter fixes the exact number of increases before weight is decreased, and so represents a *deterministic* weight reduction scheme.

Overall, there is little difference between DLM and SAPS in terms of parameter tuning. While SAPS has four parameters (α, ρ, wp and P_{smooth}) and a basic version of DLM has three, in practice at least one of the SAPS parameters can be treated as a constant and the others adjusted to suit (in this study wp is set at 1%). For both algorithms the process of parameter tuning is time consuming, as optimal performance is highly dependent on the correct settings. This compares poorly with simpler non-weighting algorithms, such as adaptive Walksat (Hoos 2002), which only requires the automatic tuning of a single noise parameter. To address this, a version of SAPS called Reactive SAPS (RSAPS) was developed (Hutter, Tompkins, & Hoos 2002) that automatically adjusts the P_{smooth} parameter during the search. However we found this algorithm did not perform as well as a properly tuned SAPS on our problem set, so we did not consider it further.

Hence, the main design criticism that can be levelled at DLM is that it relies on a somewhat complex flat move heuristic, whereas SAPS can search purely on the basis of weight guidance (while taking the occasional random move). From this it could be argued that multiplicative weighting is superior to additive weighting because it makes finer distinctions between moves and so avoids the need to

```
procedure SAPS
begin
    generate random starting point
    for each clause c_i do: set clause weight w_i ← 1
    while solution not found and not timed out do
        best ← ∞
        for each literal x_i appearing in at least one false clause do
            Δw ← change in false clause Σw caused by flipping x_i
            if Δw < best then L ← x_i and best ← Δw
            else if Δw = best then L ← L ∪ x_i
        end for
        if best < −0.1 then randomly flip x_i ∈ L
        else if probability ≤ wp then randomly flip any literal
        else
            for each false clause f_i do: w_i ← w_i × α
            if probability ≤ P_smooth then
                μ_w ← mean of current clause weights
                for each clause c_j do: w_j ← w_j × ρ + (1 − ρ) × μ_w
            end if
        end if
    end while
end
```

Figure 1: Scaling and probabilistic smoothing (SAPS)

```
procedure PAWS
begin
    generate random starting point
    for each clause c_i do: set clause weight w_i ← 1
    while solution not found and not timed out do
        best ← ∞
        for each literal x_ij in each false clause f_i do
            Δw ← change in false clause Σw caused by flipping x_ij
            if Δw < best then L ← x_ij and best ← Δw
            else if Δw = best then L ← L ∪ x_ij
        end for
        if best < 0 or (best = 0 and probability ≤ P_flat) then
            randomly flip x_ij ∈ L
        else
            for each false clause f_i do: w_i ← w_i + 1
            if # times clause weights increased % Max_inc = 0 then
                for each clause c_j|w_j > 1 do: w_j ← w_j − 1
            end if
        end if
    end while
end
```

Figure 2: The pure additive weighting scheme (PAWS)

search plateau areas. However, this assumes that the overall performance of SAPS is at least as good as DLM's and that the effectiveness of additive weighting depends on plateau searching, both issues we shall address later in the paper.

The Pure Additive Weighting Scheme (PAWS)

SAPS has demonstrated that effective local search guidance can be given by a reasonably simple manipulation of clause weights. It has also outperformed DLM on a range of SATLIB benchmark problems, both in terms of time and median number of flips (Schuurmans, Southey, & Holte 2001; Hutter, Tompkins, & Hoos 2002). From this work several questions arise: firstly how does SAPS perform on the larger DIMACS benchmark problems for which DLM was developed? Secondly, the SAPS code is based on a very efficient implementation of Walksat[3], so to what extent is the superior time performance of SAPS based on the details of this implementation? And finally, does the success of SAPS depend on multiplicative weighting? i.e. can we obtain the same kind of guidance using additive weighting, avoiding the complication of multiplicative update parameters and without resorting to the further complication of a plateau searching strategy?

To answer all three of these questions we developed a pure additive weighting scheme (PAWS), which we embedded directly into the SAPS source code[4] (so the same efficiencies were obtained), and tested PAWS on both the SATLIB benchmarks used for SAPS and a selection of the DIMACS benchmarks used for DLM. We term PAWS as a pure weighting scheme because it does away with DLM's plateau searching heuristic and only relies on weight guidance to determine the search trajectory. However, PAWS retains DLM's preference for taking flat moves when no improving moves are available, by selecting random moves from the domain of available flat moves. In addition, PAWS retains DLM's deterministic weight reduction scheme and the multiple inclusion of literals that appear in more than one false clause.

Figure 2 shows the complete PAWS procedure which is now controlled by two parameters: P_{flat} which decides whether a randomly selected flat move will be taken (corresponding to wp in SAPS), and Max_{inc} which determines at which point weight will be decreased (corresponding to P_{smooth} in SAPS). As with wp in SAPS, we found that P_{flat} can be treated as a constant, and for all subsequent experiments it was set at 15%. Hence PAWS only requires the tuning of a single parameter, Max_{inc}, which we found to have roughly the same settings and sensitivity as the equivalent parameter in DLM. On all our test problems the optimum value of Max_{inc} was relatively easy to find, generally showing a similar concave shaped relationship with local search cost as that observed for Walksat's noise parameter in (Hoos 2002). The requirement to only tune a single parameter with a fairly stable relationship to cost gives PAWS a considerable practical advantage over DLM and SAPS, which typically need considerably more effort to set up for a particular class of problem.

While PAWS comes close to being an additive version of SAPS, as discussed earlier, it differs in three aspects:

- Random Flat (RF): PAWS probabilistically takes a random flat move when no improving move is available (rather than allowing cost increasing moves).
- Deterministic Reduction (DR): PAWS deterministically reduces weights after Max_{inc} number of increases (rather than reducing weights with probability P_{smooth}).

[3] http://www.cs.washington.edu/homes/kautz/walksat/walksat-dist.tar.Z.uu

[4] http://www.int.gu.edu.au/~johnt/paws.tar

- Multiple Inclusion (MI): PAWS allows optimal cost flips that appear in n false clauses to also appear n times in its move list L (rather than exactly once).

To test the effects of these differences, three additional versions of PAWS were developed, each with one of these features replaced by the alternative heuristic used in SAPS. Similarly, three further versions of SAPS were developed each using the alternative PAWS heuristic.

Empirical Study

Problem Set

We firstly set out to reproduce the problem set reported in the original study on SAPS (Hutter, Tompkins, & Hoos 2002). This involved selecting the median and hardest problems from several SATLIB problem classes. As we were unable to verify the exact problems with the authors, we ran our own tests with SAPS, using the published parameter settings, to identify the median and hardest instances for the flat100, flat200, uf100 and uf250 problem sets. Secondly, to test performance on larger problem instances, we included the SATLIB bw-large.d blocks world problem, the two most difficult DIMACS graph coloring problems (g125.17 and g250.29) and the median and hardest DIMACS 16-bit parity learning problems (par16). We then generated three sets of random 3-SAT problems from the accepted hard region, each containing 20 instances, the first with 400 variables, the second with 800 variables and the last with 1600 variables. To these we added the f400, f800 and f1600 DIMACS problems and repeated the earlier process to identify the median and hardest problem from each set. Finally, we generated a range of random binary CSPs, again from the accepted hard region, and transformed them into SAT problems using the multivalued encoding described in (Prestwich 2003). These problems were divided into 4 sets of five problems each, according to the number of variables (v), the domain size (d), and the constraint density (c) in the original CSP, giving the 30v10d40c, 30v10d80c, 50v15d40c and 50v15d80c problem sets from each of which the hardest problem was selected.

Complete versus Local Search

One of the key motivations for the development of local search techniques for SAT was to solve problems beyond the reach of existing complete solvers. Complete solvers, even if slower on particular instances, have the advantage of unambiguously reporting if an instance is unsatisfiable. Hence, local search for SAT is most applicable to problems that are too difficult for complete search to solve in a reasonable time frame. This means the scalability of local search is important, and that evaluations on problems that can easily be solved by a complete solver are not conclusive. To clarify this issue we additionally tested our problem set using the well-known complete solver, Satz (Li & Anbulagan 1997).

The Wilcoxon Rank-Sum Test

Local search run-times can vary significantly on the same problem instance, as determined by the initial starting point and any subsequent randomized decisions. For this reason empirical studies require the same problem to be solved multiple times, and at least for the mean, median and standard deviation to be reported. However, it is still unclear exactly how much confidence we can have in the reported differences between algorithms. Standard deviation is informative for normally distributed data, but local search run-times are generally not normally distributed, often having the median to the left of the mean and a number of unpredictably distributed outliers. Hence standard comparisons that assume normality, such as a two-sample t-test, are not reliable, and the level of statistical confidence in differences between algorithms is rarely investigated.

However, nonparametric measures, such as the Wilcoxon rank-sum test, do not rely on normality, and only assume that the distributions to be compared have a similar shape. To use the Wilcoxon test requires that the run-times (or number of flips) from two sets of observations, A and B, are sorted in ascending order. Then each observation is ranked (from $1 \ldots N$) and the sum of the ranks for distribution A is calculated. This value (w_A) can now be used to test the hypothesis that distribution A lies to the left of distribution B, i.e. $H_1 : A < B$, using the normal approximation to the Wilcoxon distribution (Gibbons & Chakraborti 1992)[5]:

$$z = (w_A - n_A(N+1)/2 - 0.5)/\sqrt{n_A n_B (N+1)/12}$$

where n_A and n_B are the number of observations in distributions A and B respectively and $N = n_A + n_B$. Using the standard $Z \sim \text{Normal}(0, 1)$ tables, z will give the probability P that the null hypothesis, $H_0 : A \geq B$, is true.

Results

Table 1 shows the results for the original SAPS problem set from (Hutter, Tompkins, & Hoos 2002), except the f400 problems which came from our own problem distribution (as the SAPS uf400 problems were unavailable). Table 2 shows the results for the larger DIMACS benchmarks and our random 3-SAT and binary CSP problems[6]. In both tables, the Wilcoxon values give the probability that the null hypothesis $A \geq B$ is true, where A is the distribution of flips or run-times that has the *smaller* rank-sum value. We record P-values against distribution A and take $P < 0.05$ to indicate that A is significantly less than B, marking such results with '*'. Also, due to space limitations, we only report the base version performance of SAPS and PAWS, and discuss the overall performance of each variant in the next section.

Table 1 shows SAPS and PAWS to be fairly evenly matched on the original SAPS problem set. SAPS is significantly better on the flat-med and uf250-hard problems, and slightly better on the flat-hard, f400-hard and bw_large.b problems, whereas PAWS is significantly better on the bw_large.c, logistics.c, uf100 and f400-med problems and slightly better on bw_large.a and ais10. However, the results also show that most of these problems are not difficult for

[5] assuming $n_A > 12$, $n_B > 12$ and that no rank values are tied

[6] All results are for 100 runs with a 20 million flip cut-off except 50v15d40c which had a 50 million cut-off.

			Success	Time(secs)	Flips	Satz	Wilcoxon
Problem	Method	Params	%100 runs	median mean	median mean	time (secs)	t:time f:flips
bw_large.a	SAPS	P:6 α:1.30 ρ:0.80	100	0.01 0.02	2,571 3,366	0.11	
	PAWS	*Max*:34	100	0.01 0.01	2,604 3,077		$0.0749t$ $0.2955f$
bw_large.b	SAPS	P:5 α:1.30 ρ:0.80	100	0.14 0.22	24,570 38,116	0.49	$0.3124t$ $*0.0267f$
	PAWS	*Max*:50	100	0.14 0.21	29,083 45,112		
bw_large.c	SAPS	P:7 α:1.10 ρ:0.60	100	17.45 26.17	1,409,463 2,091,311	2.38	
	PAWS	*Max*:5	100	4.54 7.15	796,261 1,251,603		$*0.0000t$ $*0.0005f$
logistics.c	SAPS	P:5 α:1.30 ρ:0.90	100	0.04 0.05	7,454 9,187	0.53	
	PAWS	*Max*:∞	100	0.02 0.03	5,117 6,360		$*0.0000t$ $*0.0000f$
flat100-med	SAPS	P:5 α:1.30 ρ:0.40	100	0.01 0.01	3,940 6,669	0.20	$*0.0280t$ $*0.0060f$
	PAWS	*Max*:16	100	0.01 0.02	6,578 9,137		
flat100-hard	SAPS	P:6 α:1.30 ρ:0.80	100	0.04 0.05	23,773 28,663	0.30	$0.4064t$ $0.3147f$
	PAWS	*Max*:46	100	0.04 0.06	24,113 36,267		
flat200-med	SAPS	P:8 α:1.30 ρ:0.40	100	0.20 0.28	92,872 131,702	0.30	$*0.0396t$ $*0.0029f$
	PAWS	*Max*:9	100	0.26 0.35	137,267 188,652		
flat200-hard	SAPS	P:5 α:1.30 ρ:0.40	100	5.51 6.15	2,748,230 3,065,908	0.50	$0.3327t$ $0.1556f$
	PAWS	*Max*:74	100	5.10 7.40	2,767,914 4,026,639		
uf100-hard	SAPS	P:6 α:1.30 ρ:0.80	100	0.01 0.01	2,442 3,666	0.03	
	PAWS	*Max*:15	100	0.01 0.00	1,697 2,644		$*0.0105t$ $*0.0245f$
uf250-med	SAPS	P:5 α:1.30 ρ:0.40	100	0.01 0.02	5,024 6,236	1.42	$0.4713f$
	PAWS	*Max*:15	100	0.01 0.02	4,854 6,224		$0.4518t$
uf250-hard	SAPS	P:6 α:1.30 ρ:0.70	100	0.44 0.55	173,666 217,831	0.32	$*0.0437t$ $*0.0275f$
	PAWS	*Max*:18	100	0.54 0.78	223,331 318,772		
f400-med	SAPS	P:5 α:1.25 ρ:0.20	100	0.09 0.14	31,638 48,870	110.96	
	PAWS	*Max*:9	100	0.06 0.08	21,556 29,566		$*0.0003t$ $*0.0019f$
f400-hard	SAPS	P:5 α:1.30 ρ:0.10	100	2.28 3.34	819,408 1,195,727	371.07	$0.0970t$ $0.0632f$
	PAWS	*Max*:11	100	3.28 3.86	1,219,155 1,433,813		
ais10	SAPS	P:4 α:1.30 ρ:0.90	100	0.07 0.10	13,853 19,569	0.80	
	PAWS	*Max*:52	100	0.07 0.10	15,092 21,044		$0.0893t$ $0.2622f$
Overall	SAPS		100	0.07 2.65	22,011 490,488		
	PAWS		100	0.06 1.43	22,459 527,068		$*0.0499t$ $0.3663f$

Table 1: Results for original SAPS problem set

a complete search, with Satz having easily the best performance on bw_large.c and flat200-hard, and only being seriously challenged on the f400 problems.

In contrast, Table 2 shows PAWS to be strongly outperforming SAPS on all problems except the most difficult random binary CSP (50v15d40c). Additionally, Satz is significantly challenged on this problem set, being unable to solve the larger 3-SAT f problems, 50v CSP problems or g125/g250 problems before timing out after an hour[7]. The strong performance of PAWS in Table 2 is therefore doubly important, because it is in a domain where complete search starts to break down.

Figure 3 further illustrates the superior performance of PAWS on the Table 2 problem set by graphing the run-times for all 1300 runs of each algorithm. Here PAWS is consistently better than SAPS, in particular solving 93% versus 78% of instances within 50 seconds.

[7]All experiments were performed on a Sun supercomputer with 8 × Sun Fire V880 servers, each with 8 × UltraSPARC-III 900MHz CPU and 8GB memory per node.

			Success	Time(secs)	Flips	Satz	Wilcoxon
Problem	Method	Params	%100 runs	median mean	median mean	time (secs)	t:time f:flips
bw_large.d	SAPS	P:5 α:1.05 ρ:0.80	100	23.48 34.33	1,400,950 2,082,487	346.16	
	PAWS	*Max*:4	100	6.22 10.24	836,718 1,340,809		$*0.0000t$ $*0.0016f$
f800-med	SAPS	P:5 α:1.25 ρ:0.10	100	0.83 0.92	243,511 270,383	>3600	
	PAWS	*Max*:9	100	0.30 0.40	96,574 131,303		$*0.0000t$ $*0.0000f$
f800-hard	SAPS	P:5 α:1.25 ρ:0.30	100	3.93 5.27	1,148,092 1,899,465	>3600	
	PAWS	*Max*:10	100	1.91 2.93	642,537 1,012,867		$*0.0000t$ $*0.0017f$
f1600-med	SAPS	P:5 α:1.25 ρ:0.30	100	3.59 5.70	791,484 1,222,877	>3600	
	PAWS	*Max*:10	100	1.17 1.60	338,792 511,318		$*0.0000t$ $*0.0000f$
f1600-hard	SAPS	P:5 α:1.25 ρ:0.30	92	29.01 34.67	6,444,467 7,620,710	>3600	
	PAWS	*Max*:11	99	11.55 17.05	3,003,227 4,542,798		$*0.0000t$ $*0.0001f$
par16-med	SAPS	P:7 α:2.00 ρ:0.25	85	14.52 22.01	6,682,149 8,020,488	1.52	
	PAWS	*Max*:36	98	6.27 8.01	3,193,260 4,063,500		$*0.0000t$ $*0.0000f$
par16-hard	SAPS	P:4 α:1.40 ρ:0.90	86	17.31 19.82	8,209,360 9,402,907	0.57	
	PAWS	*Max*:40	98	5.64 9.43	2,847,317 4,763,523		$*0.0000t$ $*0.0000f$
g125.17	SAPS	P:5 α:1.20 ρ:0.05	99	60.02 91.02	3,108,428 4,663,084	>3600	
	PAWS	*Max*:4	100	6.16 8.48	480,091 656,876		$*0.0000t$ $*0.0000f$
g250.29	SAPS	P:6 α:1.15 ρ:0.10	90	100.14 219.92	563,389 2,622,915	>3600	
	PAWS	*Max*:4	100	19.73 21.89	275,782 315,937		$*0.0000t$ $*0.0000f$
30v10d80c	SAPS	P:6 α:1.30 ρ:0.10	100	0.06 0.08	9,577 12,689	0.29	
	PAWS	*Max*:7	100	0.05 0.06	8,839 10,523		$*0.0059t$ $0.1495f$
30v10d40c	SAPS	P:6 α:1.25 ρ:0.50	100	0.10 0.12	17,628 21,376	0.35	
	PAWS	*Max*:7	100	0.09 0.12	15,926 23,944		$0.2245t$ $0.2318f$
50v15d80c	SAPS	P:6 α:1.20 ρ:0.10	100	2.24 3.53	134,936 229,951	>3600	
	PAWS	*Max*:5	100	1.28 1.64	124,941 158,479		$*0.0000t$ $*0.0280f$
50v15d40c	SAPS	P:5 α:1.25 ρ:0.25	99	96.84 149.05	7,579,338 11,748,482	>3600	$0.2535t$ $0.0735f$
	PAWS	*Max*:6	98	121.12 169.55	10,866,838 15,159,576		
Overall	SAPS		96.23	7.63 35.97	940,756 3,145,317		
	PAWS		99.46	2.93 18.40	427,173 2,373,633		$*0.0000t$ $*0.0000f$

Table 2: Results for harder problem set

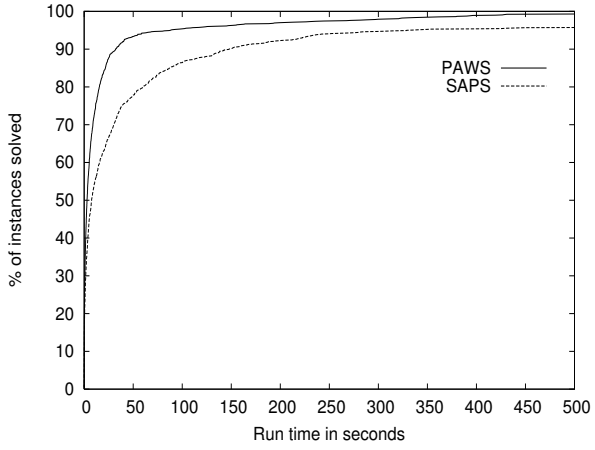

Figure 3: SAPS versus PAWS on the Table 2 problem set

Analysis and Conclusions

Our analysis of the three variants of each algorithm only showed an overall significant difference in performance ($P < 0.05$) on the Table 2 problems, where the exchange

of the deterministic reduction (PAWS-DR, SAPS+DR) and random flat (PAWS-RF, SAPS+RF) heuristics produced a significant worsening of performance for both SAPS and PAWS. Conversely, the multiple inclusion heuristic (MI) did not have a significant effect on the overall performance of either approach. Therefore, we conclude that neither algorithm significantly benefits from the inclusion of the other algorithm's secondary heuristics.

Overall the results indicate that additive weighting tends to perform better than multiplicative weighting on larger and more difficult problems. The most obvious difference between the two schemes is that multipliers create finer distinctions between clause weights: as multiplicative weights are real-valued, the previous history of clause weighting will be retained in small differences, even after smoothing. Hence, in longer term searches, we would expect clause weights to become more and more distinguished, making it increasingly unlikely that any two flips will evaluate to the same cost. Conversely, additive weighting changes clause weights by simply adding or subtracting one, and most weights are returned to a base weight of one at some point in the search. Hence longer term residual weight is eliminated and the likelihood that different flips will evaluate to the same cost remains relatively high, meaning additive weighting will generally have a greater number of possible best cost moves to select from. We confirmed this by measuring the mean number of candidate moves in list L for SAPS and PAWS for a single run of each algorithm on each test problem, eliminating any duplicate flips from the PAWS list. For all instances, except bw_large.b/c and g125/g250, the SAPS list tends to a length of one, whereas the *smallest* average list length for PAWS was 2.05 (ais10), with an average list length ratio of PAWS to SAPS of 3.3

We therefore conjecture that this difference in the available number of moves is important for longer term searches and gives additive weighting the greater freedom of movement needed to navigate difficult cost surfaces (i.e. cost surfaces that produce ambiguous clause weight guidance). However, the fact that SAPS is better on the most difficult problem (50v15d40s), shows this rule cannot be automatically generalized and that for some problems other, as yet unidentified, features are important.

However, the overall case for preferring additive over multiplicative weighting is compelling: firstly, the average flip performance of PAWS does not differ significantly from SAPS for Table 1 and strongly dominates SAPS on the more difficult problems of Table 2 (i.e. those beyond the reach of Satz). Secondly, additive weighting is more time efficient than multiplicative due to using integer rather than real-valued clause weights (the average flips/sec for PAWS on the complete problem set was 148,899 versus 114,665 for SAPS, remembering both algorithms are running within the same software architecture). And finally the search space of possible parameter settings is at least an order of magnitude less for PAWS than for SAPS (Max_{inc} was tested on a domain of less than 100 distinct values ranging from 3 to 75 in steps of one, whereas the search space of α, ρ and P_{smooth} was approximately $20 \times 20 \times 5$).

In summary, this paper balances much of the recent work on clause weighting that has concentrated on multiplicative updates, showing that additive weighting can be faster, simpler in terms of parameter tuning, and more applicable to larger problems beyond the reach of complete search methods. However, multiplicative weighting still has the better performance in several problem domains, and in future work it would be worth identifying the problem characteristics and search behaviors that favor a multiplicative approach.

References

Gibbons, J., and Chakraborti, S. 1992. *Nonparametric Staistical Inference*. Statistics: Textbooks and Monographs. New York: Marcel Dekker, Inc. 241–251.

Hoos, H. 2002. An adaptive noise mechanism for WalkSAT. In *Proceedings of the Nineteenth National Conference on Artificial Intelligence (AAAI-02)*, 655–660.

Hutter, F.; Tompkins, D.; and Hoos, H. 2002. Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In *Proceedings of the Eighth International Conference on the Principles and Practice of Constraint Programming (CP'02)*, 233–248.

Li, C., and Anbulagan. 1997. Look-ahead versus look-back for satisfiability problems. In *Proceedings of the Third International Conference on the Principles and Practice of Constraint Programming (CP'97)*, 341–355.

McAllester, D.; Selman, B.; and Kautz, H. 1997. Evidence for invariance in local search. In *Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI-97)*, 321–326.

Mills, P., and Tsang, E. 1999. Guided local search applied to the satisfiability (SAT) problem. In *Proceedings of the 15th National Conference of the Australian Society for Operations Research (ASOR'99)*, 872–883.

Morris, P. 1993. The Breakout method for escaping local minima. In *Proceedings of the Eleventh National Conference on Artificial Intelligence (AAAI-93)*, 40–45.

Prestwich, S. 2003. Local search on SAT-encoded CSPs. In *Proceedings of the Sixth International Conference on Theory and Applications of Satisfiability Testing (SAT-03)*.

Schuurmans, D., and Southey, F. 2000. Local search characteristics of incomplete SAT procedures. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, 297–302.

Schuurmans, D.; Southey, F.; and Holte, R. 2001. The exponentiated subgradient algorithm for heuristic Boolean programming. In *Proceedings of the Seventeenth International Joint Conference on Artificial Intelligence (IJCAI-01)*, 334–341.

Wu, Z., and Wah, B. 2000. An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In *Proceedings of the Seventeenth National Conference on Artificial Intelligence (AAAI-00)*, 310–315.