Clause Weighting Local Search for SAT

Clause Weighting Local Search for SAT

John Thornton
Institute for Integrated and Intelligent Systems, Griffith University
PMB 50, Gold Coast Mail Centre, 9726, Australia
email: j.thornton@griffith.edu.au

Abstract

This paper investigates the necessary features of an effective clause weighting local search algorithm for propositional satisfiability testing. Using the recent history of clause weighting as evidence, we suggest that the best current algorithms have each discovered the same basic framework, i.e. to increase weights on false clauses in local minima and then to periodically normalize these weights using a decay mechanism.

Within this framework, we identify two basic classes of algorithm according to whether clause weight updates are performed *additively* or *multiplicatively*. Using one of the best recently developed multiplicative algorithms (SAPS) and our own pure additive weighting scheme (PAWS), 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 pure versus flat random moves, deterministic versus probabilistic weight smoothing and multiple versus single inclusion of literals in the local search neighbourhood. In addition, we examined the effects of adding a threshold feature to multiplicative weighting that makes it indifferent to similar cost moves.

As a result of this investigation, we show that additive weighting can outperform multiplicative weighting on a range of difficult problems, while requiring considerably less effort in terms of parameter tuning. Our examination of the differences between SAPS and PAWS suggests that additive weighting does benefit from the random flat move and deterministic smoothing heuristics, whereas multiplicative weighting would benefit from a deterministic/probabilistic smoothing switch parameter that is set according to the problem instance. We further show that adding a threshold to multiplicative weighting produces a general deterioration in performance, contradicting our earlier conjecture that additive weighting has better performance due to having a larger selection of possible moves. This leads us to explain differences in performance as being mainly caused by the greater emphasis of additive weighting on penalizing clauses with relatively less weight.

1 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 recent algorithms:

the discrete Lagrangian method (DLM) [21] and scaling and probabilistic smoothing (SAPS) [7]. 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 effective weight differentials during the search.

The earliest clause weighting algorithms, such as Breakout [11], 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 [9]. 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 [21]. 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 ([14]) 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) [15], which in turn formed the basis of the scaling and probabilistic smoothing (SAPS) algorithm [7]. 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 violated clauses in a local minimum (rather than updating all clauses).

The basic idea of using weight penalties, or Lagrangian multipliers, to solve discrete optimization problems was originally developed in the Operations Research (OR) community [1], and has evolved into the area of subgradient optimization. These ap-

proaches have significant similarities to the weighting algorithms developed in the SAT community. However, as Schuurmans *et al.*, pointed out [15], the crucial difference is that OR techniques use linear penalty functions, whereas SAT algorithms use nonlinear *hinge* penalty functions that do not explicitly reward features or clauses that remain satisfied. In their analysis of ESG, Schuurmans *et al.*, further demonstrated that nonlinear penalty functions have the better performance in the SAT domain.

The important point for the current research is not only that the leading SAT clause weighting algorithms have converged on the same class of nonlinear hinge penalty functions, but also that they have converged on the same basic framework of weight control. One of the crucial steps from ESG to SAPS was the realization that weight normalization can be split into two phases: firstly penalizing false clauses in local minima and secondly periodically reducing weights according to a problem specific parameter. As the number of false clauses at any point during the search is relatively small compared to the total number of clauses, this splitting of the weight control allows for regular and fast weight increases, while the slower process of weight reduction occurs more infrequently, leading to significant gains in time performance. With this change, the weight update scheme of SAPS becomes 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. SAPS differs from DLM in using this parameter to probabilistically determine when weight is reduced, whereas DLM deterministically reduces weight after a fixed number of increases. Therefore, the remaining and crucial difference between the weighting mechanisms of SAPS and DLM is the use multiplicative as opposed to additive weight updates.

It is of interest to note that a third clause weighting algorithm, GLSSAT [10], employs 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. However, although GLSSAT performed well in comparison to Walksat, it could not match DLM on larger problems. Also, an earlier study [19] indicated that the basic approach of increasing weights on the least weighted false clauses is not as effective as increasing weights on all false clauses. For these reasons we decided to concentrate on SAPS and DLM, and leave a GLS type approach for future work.

The main aim of the study is to investigate whether an additive or multiplicative weight update scheme is better for satisfiability testing. The secondary aim is to discover whether the various sub-heuristics used in the two approaches provide a useful contribution to performance. 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 the same 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 the 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 detail the development of PAWS from SAPS and DLM, and then present the results and conclusions of our empirical study.

2 Clause Weighting Algorithms for SAT

Clause weighting local search algorithms for SAT follow the basic procedure of repeatedly flipping single literals that produce the greatest reduction in the sum of false clause weights. Typically, all literals are randomly initialized, and all clauses are given a fixed initial weight. The search then continues until no further cost reduction is possible, at which point the weight on all unsatisfied clauses is increased, and the search is resumed, punctuated with periodic weight reductions.

Clause weighting algorithms differ primarily in the schemes used to control the clause weights, and in the definition of the points where weight should be adjusted. Multiplicative methods, such as SAPS, generally adjust weights when no further improving moves are available in the local neighbourhood. This can either be when all possible flips lead to a worse cost, or when no flip will improve cost, but some flips will lead to equal cost solutions. As multiplicative real-valued weights have much finer granularity, the presence of equal cost flips is much more unlikely than for an additive approach, where weight is adjusted in integer units. This means that additive approaches frequently have the choice between adjusting weight when no improving move is available, or taking an equal cost (flat) move.

Following the DLM literature [16], we consider a *local minimum* to be a point or a connected area of equal cost moves where no further cost improvement is possible (i.e. the area is *surrounded* by cost increasing moves, and no combination of equal cost moves can ever escape). In this model, a *plateau* is a connected area of equal cost moves that eventually lead to one or more cost improving moves. An additive weighting algorithm, like DLM, will continually encounter situations where both equal cost and cost increasing moves are available, but is unable to distinguish between a plateau (where it is worth continuing the search) and a local minimum (where weight should be increased in order to escape).

Considerable effort has gone into developing strategies to help guide additive weighting over potential plateau areas. While this is described as plateau searching, it should be noted that such techniques search plateaus and local minima indifferently. It should also be noted that the SAPS' authors have developed a different terminology to describe local search landscapes (see [6]).

2.1 DLM and SAPS

DLM has been described as "ad hoc" [15] 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 [21]. 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 last version of DLM had 27 parameters, in practice only three of these require adjustment in the SAT domain.

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 cost improving move is available. This acts as an alternative to DLM's flat move heuristic. The second is that the set of local neighborhood 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 Walksat [4], which generally only require the tuning of a single noise parameter. To address this, a version of SAPS called Reactive SAPS (RSAPS) was developed [7] 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 *ad hoc* flat move heuristic, whereas SAPS can search purely on the basis of weight guidance (while taking the occasional random flip). From this it could be argued that multiplicative weighting is superior to additive weighting because it avoids the need for a flat move heuristic, i.e. by making finer weight distinctions between moves, the search space for a multiplicative method will contain far fewer and smaller plateau areas. However, this assumes that the overall performance of SAPS is at least

```
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 summed weight of false clauses 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)

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.

3 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 [7]. 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,[1] so to what extent is the superior time performance of SAPS based on the details of this implementation? Thirdly, does the success of SAPS depend on multiplicative weighting? i.e. can we obtain the same quality of guidance using additive weighting, avoiding the use of the multiplicative update parameters α and ρ? And finally, does additive weighting require a plateau searching strategy, with the associated tabu list length and flat move parameters, to compensate for the coarser grained nature of the additive weight updates?

To answer three these questions we developed a pure additive weighting scheme

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

```
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 summed weight of false clauses caused by flipping x_i
         if Δw < best then L ← x_ij and best ← Δw
         else if Δw = best then L ← L ∪ x_ij
      end for
      if best < 0 then randomly flip x_ij ∈ L
      else if best = 0 and probability ≤ P_flat then 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)

(PAWS),[2] which we embedded directly into the SAPS source code[3] (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.

PAWS takes a middle line between SAPS and DLM, firstly by doing away with DLM's plateau searching heuristic (and the associated θ_1 and θ_2 parameters) and replacing it with a random flip heuristic. Now, whenever PAWS encounters a situation where the best available move does not change the overall cost, it will either take this move with probability P_{flat} or it will increase weight. In contrast, DLM would always take the equal cost move unless it was on the tabu list (controlled by θ_2) or unless the maximum number of consecutive flat moves had already been taken (controlled by θ_1). PAWS retains DLM's preference for taking flat moves when no improving moves are available, by only selecting random moves from the domain of available flat moves. In contrast, when SAPS takes a random move (controlled by wp), it picks from the domain of all possible moves, regardless of cost. Finally, PAWS retains DLM's deterministic weight reduction scheme and the multiple inclusion of literals that appear in more than one false clause (whereas SAPS reduces weight probabilistically according to P_{smooth} and only includes unique literals in its candidate move list).

Figure 2 shows the complete PAWS procedure which is now controlled by two

[2] PAWS is a simplification and improvement over our earlier MAX-AGE algorithm, which was shown to be competitive with DLM on a range of larger SAT problems [18].

[3] http://www.cs.ubc.ca/ davet/dls4sat/software/saps-1.0.tar.gz

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 [4] (for example see Figure 4). The requirement to only tune a single parameter with a fairly stable relationship to cost gives PAWS a significant practical advantage over DLM and SAPS, which typically need considerably more effort to set up for a particular set of problems (see Section 4.4 for a further discussion of parameter tuning).

3.1 Differences between SAPS and PAWS

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

1. Multiple Inclusion (m): 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).
2. Random Flat (r): PAWS probabilistically takes a random flat move when no improving move is available (rather than allowing cost increasing moves).
3. Deterministic Smoothing (d): PAWS deterministically reduces weights after Max_{inc} number of increases (rather than reducing weights with probability P_{smooth}).

In order to distinguish the essential from the inessential features of the two approaches, we developed four SAPS variants based on the inclusion of the above heuristics:

1. SAPS+m: includes the multiple inclusion heuristic from PAWS.
2. SAPS+r: replaces the pure random move selection of SAPS with a random flat move selection. Hence SAPS+r will only (probabilistically) take a move in a local minimum if there is at least one move available that does not increase the weighted solution cost.[4]
3. SAPS+d: replaces the probabilistic smoothing of SAPS with a deterministic weight reduction scheme that smooths weights after a fixed number of weight increases.
4. SAPS+a: uses all three heuristics at once, i.e. multiple inclusion, random flat moves and deterministic smoothing. Hence SAPS+a is equivalent to PAWS except for the use of multiplicative weighting.

[4] In the original SAPS source code, the authors used a 0.1 threshold to distinguish an improving move from a zero cost move (see Figure 1). We therefore reused this value to define a flat move for SAPS+r as any move causing a weighted cost change within the range of ± 0.1.

We then developed four variants of PAWS that use the alternative SAPS heuristics:

1. PAWS-m: discards the multiple inclusion heuristic, and only considers distinct literals in move list L.
2. PAWS-r: discards the random flat move heuristic, and probabilistically selects a move in a local minimum without consideration of cost.
3. PAWS-d: uses probabilistic rather than deterministic weight reduction.
4. PAWS-a: uses all three of the above heuristics at once. Hence PAWS-a is equivalent to SAPS except for the use of additive weighting.

Finally, in our earlier work [17], we observed that the average length of move list L for PAWS tends to be longer than for SAPS. The explanation for this difference 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.

This led us to conjecture that differences in performance between SAPS and PAWS may be explained by differences in the number of moves available during the search. To test this, we developed a fifth variant of SAPS (SAPS+t) that includes a *threshold of indifference* between moves. This threshold is compared to an averaged flip cost, calculated by dividing the weighted cost change of a flip (Δw_{x_i} in Figure 1) by the current average clause weight. A flip is then included in list L if its cost change is within a threshold value of the best cost change available at that point in the search. This alters the SAPS move selection heuristic from Figure 1 as follows:

for each literal x_i in each false clause **do**
  $\Delta w_{x_i} \leftarrow$ (change in summed weight of false clauses caused by flipping x_i)/(average clause weight)
  if $\Delta w_{x_i} \leq best + threshold$ **then**
    if $\Delta w_{x_i} < best$ **then**
      $best \leftarrow \Delta w_{x_i}$
      remove all x_j from L where $\Delta w_{x_j} - best > threshold$
    end if
    $L \leftarrow L \cup x_i$
  end if
end for

In the following empirical study the threshold heuristic is added to the SAPS+a variant to make SAPS+t. Hence, SAPS+t is almost the same as PAWS, remaining

indifferent to finer move distinctions, but retaining a multiplicative clause weight ordering. In this way we can test our earlier conjecture that a longer L has a positive impact on performance, all else being equal [17].

4 Empirical Study

4.1 Problem Set

In order to examine the relative performance of additive and multiplicative weighting, and the influences of the various SAPS and PAWS heuristics, we designed an experimental study using 29 benchmark problems that cover various dimensions of problem size, difficulty and structure.

Firstly, we took the problem set reported in the original study on SAPS [7], consisting of the median and hardest problems from several SATLIB problem classes. Secondly, to test performance on larger problem instances, we included the SATLIB ais12, logistics.d and bw-large.d blocks world problems, 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 selected 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 [13]. 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 (bin30-40), 30v10d80c (bin30-80), 50v15d40c (bin50-40) and 50v15d80c (bin50-80) problem sets from each of which the hardest problem was selected.[5]

4.2 Complete versus Local Search

One of the key motivations for the development of local search techniques for SAT is 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 less decisive. To clarify this issue we additionally tested our problem set using the well-known complete solvers, Satz (version 214) [8] and zChaff (version 2004.11.15) [12].

[5] Note that for all the larger randomly generated problems satisfiability was determined using our own local search algorithms with a cut-off of 100 million flips. Hence we may have rejected some harder satisfiable instances.

4.3 Testing for Significance

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 [3][6]:

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

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.

While the Wilcoxon test provides a good measure of overall performance, it can miss situations where one algorithm has a better probability of solving a problem within a certain time-range, even though its overall performance is relatively poor. In such circumstances a hybrid or portfolio approach [2] can produce better results, i.e. using the algorithm that has the greater solution probability in a given time-range. Hence, to test whether one algorithm clearly dominates another we have produced run-time distributions (RTDs) [5] to compare the best performing algorithm variants for each problem. RTDs are used to analyze local search performance of multiple runs on the same problem instance. By calculating and graphing the cumulative percentage of runs that have been solved over time, a picture of the overall behaviour of an algorithm on a problem can be obtained (see Figures 6, 5, 7 and 8). More importantly, if the RTD distribution of one algorithm dominates another on the same problem (i.e. at every time point it has solved a greater percentage of runs), then we can be more confident that the algorithm has the better performance. Conversely, if two RTDs cross (as in Figure 6), then we cannot safely conclude that one is uniformly better than another.

We therefore used a combination of the Wilcoxon test and an RTD analysis to assess whether there is a significant difference in algorithm performance according to the following rule: if the Wilcoxon rank-sum test is significant for $p < 0.05$ *and* the

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

RTD of the better algorithm dominates the other for all solution probabilities > 0.1 then the algorithm is classed as significantly better on the problem instance (one RTD dominates another when its distribution lies above the other, otherwise the RTDs will *cross*, e.g. see figure 7).

4.4 Parameter Setting

To make the empirical study feasible, we adopted a combination of exhaustive and local search strategies for setting the parameters of individual algorithm variants. In the exhaustive phase, we tested a range of parameter settings for the original SAPS and PAWS algorithms on each problem instance. As the issue of the number and sensitivity of parameters is important to our overall evaluation, we have taken a closer look at the parameter tuning process in the following two subsections:

4.4.1 Tuning SAPS

As discussed earlier (see Section 2), SAPS has four parameters: α, ρ, P_{smooth} and wp. In the original study [7], the SAPS authors fixed wp at 1% and P_{smooth} at 5% and manipulated α in the range of $1.1 \ldots 1.3$ and ρ in the range of $0.2 \ldots 0.9$. However, they acknowledged that "there are better parameter settings for almost all problem instances tested here. Determining these settings manually can be difficult and time-consuming." The attempt to reduce this difficulty led to the development of Reactive SAPS (RSAPS) [7]. Here, instead of fixing P_{smooth} and manipulating α and ρ, the authors fixed α, manually manipulated ρ and set P_{smooth} using an automated reactive mechanism.

As the problem set used in the current study contains several larger problems on which SAPS has not been previously tested, and also because the question of which SAPS parameters to fix and which to manipulate has yet to be settled, we decided to test the three main SAPS parameters on an expanded range of settings, varying α from $1.05 \ldots 2.00$ in steps of 0.05, ρ from $0.05 \ldots 1.00$ in steps of 0.05 and P_{smooth} from $4\% \ldots 8\%$ in steps of 1% (keeping wp fixed at 1%). For problems that PAWS solves in fewer than one million flips, we allowed 100 runs at each of the $20 \times 20 \times 5$ possible settings. For the remaining problems we allowed 10 runs at each setting and retested the best 10 of these at 100 runs. We then sorted the results for each problem instance according to the mean flip count and selected the best performing parameter setting for use in the main study.

We firstly allowed SAPS such a wide range of parameter values to ensure that the comparison with PAWS was not biased by a limited choice. Secondly, the experiment allows us to examine the range and sensitivity of the SAPS parameters. In Table 1, we show the mean flip counts of the best parameter settings for SAPS on each test set problem, in comparison to the recommended default settings of $\alpha = 1.3$, $\rho = 0.8$ and $P_{smooth} = 5\%$, with 100 runs on each problem and a cut-off of 20 million flips (50 million for bin50-40). These results show that using default parameter settings is not a practical approach, particularly on the larger, more difficult problems. For instance, the default settings were unable to solve any run on the g250.29, f1600-med and f1600-hard problems, and could only solve one out of 100 runs on g125.17. The results

	Tuned Settings			Tuned Results		Default Results		Wilcoxon		
Problem	α	ρ	P_{smooth}	success	mean flips	success	mean flips	speed-up	p-value	significant
bw_large.a	1.30	0.80	6	100	2,824	100	3,000	1.06	0.13066	×
bw_large.b	1.30	0.80	5	100	45,335	=	=	=	=	=
bw_large.c	1.10	0.60	7	100	2,103,352	75	7,034,958	3.34	0.00000	√
bw_large.d	1.05	0.80	5	100	2,398,205	5	19,589,233	8.17	0.00000	√
flat100-med	1.30	0.40	5	100	7,460	100	10,729	1.44	0.00000	√
flat100-hard	1.30	0.80	6	100	31,812	100	29,906	0.94	0.86585	×
flat200-med	1.30	0.40	5	100	83,558	100	168,030	2.01	0.00000	√
flat200-hard	1.30	0.40	5	100	3,397,088	100	3,837,537	1.13	0.15793	×
g125.17	1.20	0.05	5	97	4,187,750	1	19,953,867	4.76	0.00000	√
g250.29	1.15	0.10	6	90	4,622,915	0	20,000,000	4.33	0.00000	√
uf100-hard	1.30	0.80	5	100	4,250	=	=	=	=	=
uf250-med	1.30	0.40	6	100	7,050	100	13,584	1.93	0.00000	√
uf250-hard	1.30	0.70	5	100	223,593	100	254,243	1.14	0.00710	√
uf400-med	1.30	0.20	5	100	61,159	100	167,785	2.74	0.00000	√
uf400-hard	1.30	0.20	5	100	1,446,987	98	3,901,415	2.70	0.00000	√
f800-med	1.25	0.10	5	100	263,105	33	16,665,531	63.34	0.00000	√
f800-hard	1.25	0.30	5	100	1,754,017	17	18,593,591	10.60	0.00000	√
f1600-med	1.25	0.30	5	99	1,303,941	0	20,000,000	15.34	0.00000	√
f1600-hard	1.25	0.30	5	94	7,777,980	0	20,000,000	2.57	0.00000	√
ais10	1.30	0.90	4	100	18,085	100	20,339	1.12	0.00384	√
ais12	1.25	0.95	4	100	123,099	100	186,402	1.51	0.00000	√
logistics.c	1.30	0.90	5	100	8,436	100	9,399	1.11	0.00088	√
logistics.d	1.20	1.00	4	100	21,248	100	57,151	2.69	0.00000	√
par16-med	2.00	0.25	7	88	7,720,965	82	7,521,553	0.97	0.54918	×
par16-hard	1.40	0.90	4	86	9,725,495	78	9,825,138	1.01	0.12296	×
bin30-80	1.30	0.10	6	100	12,299	100	23,127	1.88	0.00000	√
bin30-40	1.25	0.50	6	100	19,711	100	36,826	1.87	0.00000	√
bin50-80	1.20	0.10	6	100	186,552	100	1,495,097	8.01	0.00000	√
bin50-40	1.25	0.25	5	99	11,562,103	70	25,607,766	2.21	0.00000	√

Table 1: SAPS parameter tuning comparison: SAPS defaults are α 1.30, ρ 0.80 and P_{smooth} 5%, rows with '=' values indicate the default and tuned settings were equal.

also show that the best performing algorithms have exploited nearly the full range of parameter settings with α varying between 1.05 (bw_large.d) and 2.00 (par16-med), ρ varying between 0.05 (g125.17) and 1.00 (logistics.d) and P_{smooth} varying between 4% (ais10) and 7% (bw_large.c). However, the larger values for α only appear on non-statistically significant results (par16-med and par16-hard[7]). Ignoring these two problems, α ranges more narrowly between $1.05 \ldots 1.40$.

Although the results show that a wide range of parameter values were used to obtain the best performance, we have yet to consider the sensitivity of individual parameters. It could be the case that one SAPS parameter dominates the others to the extent that the variations in the dominated parameters do not significantly affect performance. We can firstly reject the hypothesis that α is insignificant from the bw_large.c result. Here α is the only parameter varied from the default, and the result is a significant difference in performance. There are several similar examples of a significant difference obtained by only manipulating ρ from the default (i.e. flat200-med, uf200-hard, uf400-med, uf400-hard and logistics.c). Hence we can conclude that α and ρ are important parameters, with ρ showing a significant difference at a sensitivity of at least 0.01 (for logistics.c)

[7] Although not statistically different on flips the tuned par16 runs had better success rates.

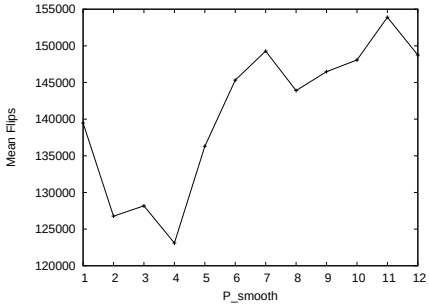

Figure 3: P_{smooth} settings for SAPS on ais12 with α 1.25 and ρ 0.95

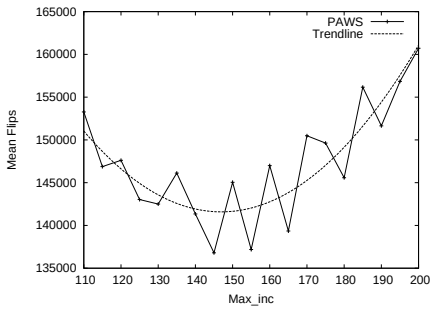

Figure 4: Max_{inc} settings for PAWS on ais12

and α showing a sensitivity of at least 0.15 (for bw_large.d).

However, as Table 1 does not show a significant difference arising from the manipulation of P_{smooth} alone, we decided to look at an individual problem (ais12) in more detail. From our initial parameter tests at 100 runs, the best setting for SAPS on ais12 was $\alpha = 1.25$, $\rho = 0.95$ and $P_{smooth} = 4\%$. In order to test sensitivity, we decided to manipulate P_{smooth} from $1\% \ldots 12\%$ in steps of 1% at $1,000$ runs per setting, keeping the other parameters fixed at their tuned values. The mean flip values for each of these settings are graphed in Figure 3, and show there is a relationship between P_{smooth} and performance on this problem. We performed a further Wilcoxon analysis between the 4% and 5% P_{smooth} settings and found a *nearly* significant time difference at $p = 0.06852$. A second comparison between the 4% and 6% settings did yield a significant difference at $p = 0.00013$.

From the foregoing analysis we can conclude that each of the three main SAPS parameters can produce significant differences in performance on at least one of the test problems. However, this result must be qualified in several respects. Firstly, fine tuning parameters is unnecessary when comparing with another algorithm, if a coarsely tuned version still dominates. As the subsequent results show, SAPS does not clearly dominate PAWS, and so the fine tuning of parameters can be justified. Secondly, while the individual parameters are sensitive in isolation, this does not mean that one or two parameters could be fixed, and the free parameter(s) adjusted to achieve optimal performance (this assumes that different combinations of parameter settings could produce the same optimal performance). The earlier study on RSAPS [7] shows that holding α and ρ constant and manipulating P_{smooth} is not as effective as additionally allowing ρ to change. This indicates that at least two SAPS parameters need to be manipulated to achieve acceptable performance.

If we further consider the actual effects of α, ρ and P_{smooth}, it seems reasonable to assume that similar weighting behaviour could be achieved with combinations of different settings, i.e. α determines the amount of weight increase, ρ determines the amount of decrease and P_{smooth} determines how frequently a decrease occurs. Hence we could expect a smaller increase and larger decrease performed less frequently to be-

have similarly to a larger increase and a smaller decrease performed more frequently. In this case we would prefer the setting that reduces weight more infrequently, as it would be more time efficient. But the question remains as to how infrequently weight can be reduced without degrading performance. Certainly we know performance will degrade eventually, as the limit would be to never reduce weight, and the more infrequently we reduce weight the more insensitive the search becomes to local conditions.

In summary, we conjecture there may be discoverable relationships between α, ρ and P_{smooth} that could simplify the parameter tuning process. It may also be the case that a more fine grained tuning of one parameter could eliminate the need to tune another. We leave these questions for future research. In practical terms, the sensitivity of the SAPS parameters means we cannot be certain of obtaining the best performance without searching an extensive range of settings. While a particular parameter may not be sensitive on a particular problem, we are unfortunately unable to know this in advance.

4.4.2 Tuning PAWS

Tuning PAWS presented a relatively simpler problem. Keeping P_{flat} constant at 15%, we manipulated Max_{inc} from $5 \ldots 100$ in steps of 5, with 100 runs at each setting (as with SAPS we reduced the number of runs for the more difficult problems). We then graphed the mean flip performance against Max_{inc} and decided on an optimum setting by visual inspection. If the performance still appeared to be improving at $Max_{inc} = 100$, we tested PAWS with no weight decrease (i.e. $Max_{inc} = \infty$), and, if this proved better than $Max_{inc} = 100$, the ∞ value was accepted. Otherwise we continued with a further analysis from $105 \ldots 200$ in steps of 5 (this secondary analysis only proved necessary for the ais12 problem). Given an optimum point from graphical analysis, a final fine grained analysis was performed around this point, in the range of ± 5 in steps of ± 1. From this the best value was selected and used in the remainder of the study. As an example, the performance graph for PAWS on ais12 for the Max_{inc} range of $110 \ldots 200$ is shown in Figure 4 (with a trendline fitted). This gives a fairly typical picture of the behaviour of Max_{inc}, showing the presence of an unambiguous minimum flip value.

While Max_{inc} is sensitive to changes down to ± 1, especially for $Max_{inc} < 20$, the tuning process is considerably simpler than for SAPS, and contains less margin for error due to noise. This is because the single parameter allows for a simple graphical analysis and hence the identification of trends that are independent of noise. Conversely, tuning SAPS runs the risk of missing the best parameter settings, even when averaging over 100 runs.

4.4.3 Tuning the Variant Algorithms

After completing the above exhaustive parameter tuning exercises, we used the SAPS parameter settings to test the SAPS variants and the PAWS parameter settings to test the PAWS variants, with two qualifications:

1. Changing from deterministic to probabilistic smoothing or vice versa (i.e. for SAPS+d, SAPS+a, SAPS+t, PAWS-d and PAWS-a) requires a conversion of the

PAWS Max_{inc} and SAPS P_{smooth} parameters. This is achieved by dividing either parameter into 100. For example, if $Max_{inc} = 5$, this is converted to a P_{smooth} value of $100/5 = 20\%$, i.e. reducing weight at every 5^{th} local minimum is most closely approximated by a 20% chance of reducing weight at *each* local minimum. Similarly, P_{smooth} can be converted to a Max_{inc} value using the same procedure, i.e. $100/20 = 5$.

2. Although SAPS is usually run with a fixed wp probability of 1% and PAWS is run with a fixed flat move probability of 15%, it was not clear which probability value to use when converting between the two random move selection heuristics. We therefore ran versions of SAPS+r, SAPS+a, PAWS-r and PAWS-a using both settings and selected the best performing variant.

After these conversions, we tested all variants on the full problem set. The results of a similar experiment (excluding SAPS+a, SAPS+t and PAWS-a) were informally described in [17], where it was concluded that no particular variant produced an improvement over the base versions. For the current study, further parameter tuning was executed in the local neighbourhoods of the original conversions. From this we found that the α and ρ values for SAPS are fairly robust, as was P_{smooth} for SAPS+m and SAPS+d. However, our tests showed that the Max_{inc}/P_{smooth} conversions did not necessarily produce the best performance on all problems. Also, on several problems, the optimal Max_{inc} settings for PAWS-m and PAWS-r differed slightly from the original PAWS setting. Using these more refined settings we were able to produce considerably better performance across the range of SAPS and PAWS variants, as the following results show.

4.5 Results

Tables 2 to 7 divide the problem set results according to problem types, placing the instances in ascending order of size and/or difficulty within each table. For each problem we then report the performance of the original PAWS and SAPS algorithms and their variants, as described in Section 3.1. All results have a 20 million flip cut-off, except bin50-40 which has a 50 million cut-off, and the statistics refer to averages over 1,000 runs, except for those problems where at least one algorithm has an average flip cost greater than one million, in which case the average is over 100 runs[8].

In all six 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 '*'. The *intra* Wilcoxon column compares flips for the standard SAPS and SAPS+a heuristics for SAPS, and the standard PAWS and PAWS-a heuristics for PAWS. Hence the Wilcoxon intra column value of 0.3473f in the bw_large.a, SAPS+a row of Table 2 indicates that we can reject the hypothesis that SAPS+a has significantly better flip performance than SAPS on this problem (in all Wilcoxon statistics an 'f' refers to a comparison

[8] 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.

between flips and a 't' refers to a comparison between runtimes). Conversely, the intra column value of *0.0283f in the bw_large.d, PAWS row of Table 2 indicates that we can accept the hypothesis that PAWS has significantly better performance than PAWS-a on this problem. Additionally, the Wilcoxon *inter* column compares the basic SAPS and PAWS heuristics in terms of both flips and time. Hence the Wilcoxon inter column values of *0.0000f and *0.0000t in the bw_large.d, PAWS section of Table 2 indicate that we can accept the hypothesis that PAWS has better flip and time performance than SAPS on this problem. The additional RTD analysis described in Section 4.3 is shown in Table 8, where we present an overall comparison of the results.

Lastly, the table parameter column values for each variant are encoded using α, ρ, s and n, where α and ρ have their usual SAPS interpretation, but s and n have common definitions across both SAPS and PAWS variants, where s represents the smoothing parameter, which has a probabilistic interpretation for SAPS, SAPS+m, SAPS+r, PAWS-d and PAWS-a, and a deterministic interpretation for all other variants (see Section 4.4), and n represents a noise parameter which applies either as the probability of taking a pure random move for SAPS, SAPS+m, SAPS+d, PAWS-r and PAWS-a, or as the probability of taking a random flat move for all other variants.

In the following sub-sections we discuss the results for each of the six problem domain tables.

4.5.1 Blocks World Results

For the smaller bw_large.a and b problems (in Table 2), the SAPS variants generally have the better flip performance. However, this advantage does not carry over into the time domain, where PAWS is not significantly different from SAPS on bw_large.a and dominates on the three other problems. PAWS further dominates SAPS in terms of flips for bw_large.c and d. Hence, as problem size and difficulty increases, the PAWS variants also improve relative to SAPS, meaning PAWS has the overall advantage for this problem set.

In terms of individual variants, SAPS+a dominates the original SAPS, being significantly better on problems b and d, and slightly better on a and c. SAPS+a also challenges PAWS on bw_large.b, having a better flip count and roughly equal time performance. For the PAWS variants, there is a (non-significant) indication that PAWS-a does better on the smaller a and b problems, but standard PAWS becomes better on the larger problems, and is significantly better on bw_large.d.

4.5.2 Graph Colouring Results

As with the blocks world problems, SAPS starts well on the smaller graph colouring problems, having significantly better flip and time performance on the two flat-100 problems (see Table 3). However, as problem size increases, the relative performance of PAWS also improves, becoming significantly better than SAPS in terms of flips and time on flat200-med, g125.17 and g250.29, and roughly equal on flat200-hard.

The SAPS+a variant again looks better than standard SAPS, being significantly better on flat100-med, flat-200-med and g125.17, and *verging* on significantly better for the flat100-hard and flat200-hard problems. However, the situation is less clear

				Time (secs)		Flips		Wilcoxon	
Problem	Method	Parameters	%	median	mean	median	mean	intra	inter
bw_large.a	SAPS	$\alpha 1.3\rho 0.8 s6n1$	100	0.01	0.01	2,184	2,824		*0.0007f
	SAPS+m	$\alpha 1.3\rho 0.8 s6n1$	100	0.01	0.01	2,236	2,905		
	SAPS+r	$\alpha 1.3\rho 0.8 s6n1$	100	0.01	0.01	2,168	2,895		
	SAPS+d	$\alpha 1.3\rho 0.8 s16n1$	100	0.01	0.01	2,155	2,809		
	SAPS+a	$\alpha 1.3\rho 0.8 s16n1$	100	0.01	0.01	2,089	2,772	0.3473f	
	SAPS+t	$\alpha 1.3\rho 0.8 s16n15$	100	0.01	0.02	2,251	2,889		
	PAWS	$s34n15$	100	0.01	0.01	2,518	3,235		0.4570t
	PAWS-m	$s34n15$	100	0.01	0.01	2,273	3,003		
	PAWS-r	$s34n15$	100	0.01	0.01	2,403	3,067		
	PAWS-d	$s3n15$	100	0.01	0.01	2,369	3,169		
	PAWS-a	$s3n1$	100	0.01	0.01	2,453	3,118	0.2075f	
bw_large.b	SAPS	$\alpha 1.3\rho 0.8 s5n1$	100	0.20	0.26	34,488	45,335		0.4302f
	SAPS+m	$\alpha 1.3\rho 0.8 s5n1$	100	0.20	0.28	34,584	48,071		
	SAPS+r	$\alpha 1.3\rho 0.8 s5n1$	100	0.17	0.26	29,701	45,750		
	SAPS+d	$\alpha 1.3\rho 0.8 s30n1$	100	0.15	0.21	26,910	39,033		
	SAPS+a	$\alpha 1.3\rho 0.8 s30n1$	100	0.16	0.21	27,591	37,731	*0.0000f	
	SAPS+t	$\alpha 1.3\rho 0.8 s30n15$	100	0.19	0.27	26,508	38,524		
	PAWS	$s50n15$	100	0.16	0.21	33,480	45,501		*0.0000t
	PAWS-m	$s50n15$	100	0.15	0.20	30,832	43,418		
	PAWS-r	$s50n15$	100	0.16	0.21	32,977	44,635		
	PAWS-d	$s2n15$	100	0.15	0.21	32,104	44,458		
	PAWS-a	$s2n1$	100	0.15	0.21	32,133	44,109	0.2402f	
bw_large.c	SAPS	$\alpha 1.1\rho 0.6 s7n1$	100	17.63	26.45	1,366,319	2,103,352		
	SAPS+m	$\alpha 1.1\rho 0.6 s7n1$	100	18.53	30.96	1,448,924	2,370,600		
	SAPS+r	$\alpha 1.1\rho 0.6 s7n1$	100	21.21	30.02	1,669,114	2,264,986		
	SAPS+d	$\alpha 1.1\rho 0.6 s20n1$	100	13.80	16.68	1,366,083	1,671,323		
	SAPS+a	$\alpha 1.1\rho 0.6 s20n15$	100	12.72	17.20	1,224,860	1,664,822	0.1256f	
	SAPS+t	$\alpha 1.1\rho 0.6 s20n15$	82	15.00	29.34	1,471,762	4,665,851		
	PAWS	$s5n15$	100	4.74	6.87	798,389	1,181,032	0.3147f	*0.0001f
	PAWS-m	$s5n15$	100	4.64	6.67	786,344	1,143,014		*0.0000t
	PAWS-r	$s5n15$	100	5.84	7.40	957,610	1,247,581		
	PAWS-d	$s30n15$	100	7.26	9.04	1,246,342	1,581,417		
	PAWS-a	$s30n1$	100	4.41	6.96	742,669	1,206,099		
bw_large.d	SAPS	$\alpha 1.05\rho 0.8 s5n1$	100	29.22	37.87	1,868,733	2,398,205		
	SAPS+m	$\alpha 1.05\rho 0.8 s5n1$	100	25.11	36.64	1,512,079	2,213,055		
	SAPS+r	$\alpha 1.05\rho 0.8 s5n1$	100	30.92	47.27	1,884,327	2,819,920		
	SAPS+d	$\alpha 1.05\rho 0.8 s20n1$	100	21.68	29.00	1,210,114	1,660,640		
	SAPS+a	$\alpha 1.05\rho 0.8 s20n15$	100	20.01	25.53	1,152,146	1,536,322	*0.0002f	
	SAPS+t	$\alpha 1.05\rho 0.8 s20n15$	85	34.03	48.02	2,575,821	4,949,418		
	PAWS	$s4n15$	100	7.07	10.87	903,962	1,432,780	*0.0283f	*0.0000f
	PAWS-m	$s4n15$	100	7.50	10.13	1,007,744	1,324,650		*0.0000t
	PAWS-r	$s4n15$	100	8.24	10.45	1,058,403	1,349,599		
	PAWS-d	$s35n15$	100	7.72	12.67	930,462	1,594,599		
	PAWS-a	$s35n15$	100	9.65	15.75	1,175,815	1,956,037		

Table 2: Blocks world planning problem results

for the largest g250.29 problem, where SAPS is significantly better than SAPS+a but has poorer success rate (90% versus 98%). For the PAWS variants, there is little difference on the smaller flat100 problems, but for all larger problems PAWS becomes significantly better.

Considering the standard SAPS and PAWS algorithms, we can conclude that PAWS has the better performance on this problem set, particularly as problem size grows. However, if we include consideration of the SAPS variants, then SAPS further dominates on both flat200 problems, at least in terms of flips. This is examined in more detail when we look at the overall results in Table 8.

4.5.3 Small Random 3-SAT Results

Repeating the blocks world and graph colouring pattern, SAPS begins well on the smaller problems, with significantly better flip and time performance on uf250-hard and significantly better flip performance uf100-hard, but is overtaken by PAWS on uf250-med and both larger uf400 problems (see Table 4).

				Time (secs)		Flips		Wilcoxon	
Problem	Method	Parameters	%	median	mean	median	mean	intra	inter
flat100-med	SAPS	$\alpha 1.3\rho 0.4 s5n1$	100	0.01	0.01	5,415	7,460		*0.0004f
	SAPS+m	$\alpha 1.3\rho 0.4 s5n1$	100	0.01	0.01	5,156	7,340		*0.0209t
	SAPS+r	$\alpha 1.3\rho 0.4 s5n1$	100	0.01	0.01	5,314	7,532		
	SAPS+d	$\alpha 1.3\rho 0.4 s20n1$	100	0.01	0.01	4,623	6,381		
	SAPS+a	$\alpha 1.3\rho 0.4 s20n1$	100	0.01	0.01	4,684	6,527	*0.0056f	
	SAPS+t	$\alpha 1.3\rho 0.4 s20n15$	100	0.01	0.02	5,182	7,425		
	PAWS	$s13n15$	100	0.01	0.01	6,402	8,628	0.4683f	
	PAWS-m	$s10n15$	100	0.01	0.01	5,747	7,883		
	PAWS-r	$s13n15$	100	0.01	0.01	6,078	8,117		
	PAWS-d	$s11n15$	100	0.01	0.02	6,409	9,024		
	PAWS-a	$s11n1$	100	0.01	0.02	6,207	8,676		
flat100-hard	SAPS	$\alpha 1.3\rho 0.8 s6n1$	100	0.04	0.06	21,965	31,812		*0.0010f
	SAPS+m	$\alpha 1.3\rho 0.8 s6n1$	100	0.04	0.06	20,938	30,288		*0.0164t
	SAPS+r	$\alpha 1.3\rho 0.8 s6n1$	100	0.04	0.05	22,422	30,669		
	SAPS+d	$\alpha 1.3\rho 0.8 s18n1$	100	0.04	0.05	21,449	30,026		
	SAPS+a	$\alpha 1.3\rho 0.8 s18n1$	100	0.04	0.05	20,888	29,321	0.0649f	
	SAPS+t	$\alpha 1.3\rho 0.8 s18n15$	100	0.05	0.08	19,828	29,952		
	PAWS	$s46n15$	100	0.04	0.06	26,065	36,178	0.2398f	
	PAWS-m	$s46n15$	100	0.04	0.06	25,626	37,882		
	PAWS-r	$s46n15$	100	0.05	0.07	27,191	39,825		
	PAWS-d	$s2n15$	100	0.04	0.06	25,039	35,993		
	PAWS-a	$s2n1$	100	0.04	0.06	27,046	37,880		
flat200-med	SAPS	$\alpha 1.3\rho 0.4 s5n1$	100	0.12	0.17	57,411	83,558		
	SAPS+m	$\alpha 1.3\rho 0.4 s5n1$	100	0.12	0.17	55,035	79,073		
	SAPS+r	$\alpha 1.3\rho 0.4 s5n1$	100	0.12	0.17	59,249	82,414		
	SAPS+d	$\alpha 1.3\rho 0.4 s20n1$	100	0.09	0.13	43,725	61,878		
	SAPS+a	$\alpha 1.3\rho 0.4 s20n1$	100	0.09	0.12	40,900	57,946	*0.0000f	
	SAPS+t	$\alpha 1.3\rho 0.4 s20n15$	100	0.16	0.23	48,850	71,190		
	PAWS	$s10n15$	100	0.10	0.13	48,990	67,781	*0.0000f	*0.0002f
	PAWS-m	$s9n15$	100	0.10	0.13	53,529	71,553		*0.0000t
	PAWS-r	$s11n15$	100	0.11	0.15	56,983	78,090		
	PAWS-d	$s13n15$	100	0.11	0.16	59,731	80,022		
	PAWS-a	$s13n1$	100	0.12	0.17	64,818	88,593		
flat200-hard	SAPS	$\alpha 1.3\rho 0.4 s5n1$	100	4.86	6.38	3,173,188	3,397,088		
	SAPS+m	$\alpha 1.3\rho 0.4 s5n1$	100	3.75	5.64	1,801,981	2,714,483		
	SAPS+r	$\alpha 1.3\rho 0.4 s5n1$	100	3.63	5.11	1,791,686	2,532,616		
	SAPS+d	$\alpha 1.3\rho 0.4 s20n1$	100	3.02	4.50	1,526,524	2,283,598		
	SAPS+a	$\alpha 1.3\rho 0.4 s20n1$	100	3.34	4.95	1,634,472	2,417,211	0.0525f	
	SAPS+t	$\alpha 1.3\rho 0.4 s20n15$	93	5.83	9.76	1,949,826	3,445,606		
	PAWS	$s74n15$	99	4.34	5.90	2,354,944	3,224,432	*0.0004f	0.3842f
	PAWS-m	$s74n15$	99	4.34	6.68	2,414,031	3,748,207		0.2121t
	PAWS-r	$s74n15$	100	5.58	6.70	3,009,177	3,621,447		
	PAWS-d	$s2n15$	98	6.52	9.45	3,376,852	5,001,280		
	PAWS-a	$s2n15$	99	7.83	9.93	4,245,350	5,425,641		
g125.17	SAPS	$\alpha 1.2\rho 0.05 s5n1$	97	55.59	81.65	2,772,017	4,187,750		
	SAPS+m	$\alpha 1.2\rho 0.05 s5n1$	99	59.86	86.35	2,940,185	4,253,806		
	SAPS+r	$\alpha 1.2\rho 0.05 s5n1$	99	54.59	81.10	2,683,624	3,998,689		
	SAPS+d	$\alpha 1.2\rho 0.05 s25n1$	89	55.85	104.55	2,564,038	5,457,198		
	SAPS+a	$\alpha 1.2\rho 0.05 s25n15$	100	35.10	52.35	1,845,627	2,641,413	*0.0104f	
	SAPS+t	$\alpha 1.2\rho 0.05 s25n15$	73	55.80	111.66	2,696,440	7,259,569		
	PAWS	$s4n15$	100	7.91	10.89	596,447	841,063	*0.0245f	*0.0000f
	PAWS-m	$s4n15$	100	7.68	9.17	542,355	668,972		*0.0000t
	PAWS-r	$s5n15$	100	11.93	15.23	821,849	1,057,978		
	PAWS-d	$s40n15$	100	8.76	13.27	644,027	986,354		
	PAWS-a	$s40n15$	100	11.86	17.15	849,986	1,235,643		
g250.29	SAPS	$\alpha 1.15\rho 0.1 s6n1$	90	100.14	219.92	563,152	4,622,915	*0.0000f	
	SAPS+m	$\alpha 1.15\rho 0.1 s6n1$	92	102.70	201.86	595,098	3,876,035		
	SAPS+r	$\alpha 1.15\rho 0.1 s6n1$	98	92.71	166.75	576,122	1,554,811		
	SAPS+d	$\alpha 1.15\rho 0.1 s30n15$	99	107.15	182.17	845,374	1,727,007		
	SAPS+a	$\alpha 1.15\rho 0.1 s33n15$	98	170.04	289.68	1,477,554	3,124,190		
	SAPS+t	$\alpha 1.15\rho 0.1 s33n15$	86	285.47	429.94	2,661,027	5,313,553		
	PAWS	$s4n15$	100	19.73	21.89	275,188	315,937	*0.0000f	*0.0000f
	PAWS-m	$s4n15$	100	20.33	23.92	252,243	302,942		*0.0000t
	PAWS-r	$s5n15$	100	34.22	45.43	348,796	483,978		
	PAWS-d	$s36n15$	100	29.66	38.17	364,024	444,394		
	PAWS-a	$s36n1$	100	47.22	60.43	502,662	633,402		

Table 3: Graph colouring problem results

Variant performance also follows the previous results, with PAWS consistently outperforming PAWS-a, and SAPS+a outperforming SAPS on all problems except uf100-hard. SAPS+a further dominates PAWS in terms of flips on uf400-hard, while achieving similar time performance (see Table 8 for more details).

				Time (secs)		Flips		Wilcoxon	
Problem	Method	Parameters	%	median	mean	median	mean	intra	inter
uf100-hard	SAPS	$\alpha 1.3 \rho 0.8 s5 n1$	100	0.01	0.01	2,857	4,250	0.4726f	*0.0444f
	SAPS+m	$\alpha 1.3 \rho 0.8 s5 n1$	100	0.01	0.01	3,041	4,344		0.2460t
	SAPS+r	$\alpha 1.3 \rho 0.8 s5 n1$	100	0.01	0.01	2,924	4,120		
	SAPS+d	$\alpha 1.3 \rho 0.8 s20 n1$	100	0.01	0.01	2,833	4,095		
	SAPS+a	$\alpha 1.3 \rho 0.8 s20 n1$	100	0.01	0.01	2,936	4,330		
	SAPS+t	$\alpha 1.3 \rho 0.8 s20 n15$	100	0.01	0.01	3,222	4,455		
	PAWS	$s40 n15$	100	0.01	0.01	3,282	4,579	*0.0280f	
	PAWS-m	$s40 n15$	100	0.01	0.01	3,124	4,641		
	PAWS-r	$s40 n15$	100	0.01	0.01	3,368	5,017		
	PAWS-d	$s3 n15$	100	0.01	0.01	3,370	4,650		
	PAWS-a	$s3 n1$	100	0.01	0.01	3,614	4,809		
uf250-med	SAPS	$\alpha 1.3 \rho 0.4 s6 n1$	100	0.01	0.02	4,895	7,050		
	SAPS+m	$\alpha 1.3 \rho 0.4 s6 n1$	100	0.01	0.02	4,799	6,939		
	SAPS+r	$\alpha 1.3 \rho 0.4 s6 n1$	100	0.01	0.02	4,628	6,353		
	SAPS+d	$\alpha 1.3 \rho 0.4 s19 n1$	100	0.01	0.01	4,282	5,972		
	SAPS+a	$\alpha 1.3 \rho 0.4 s19 n1$	100	0.01	0.02	4,519	6,442	0.0503f	
	SAPS+t	$\alpha 1.3 \rho 0.4 s19 n15$	100	0.02	0.02	4,647	6,197		
	PAWS	$s11 n15$	100	0.01	0.01	3,795	5,040	*0.0000f	*0.0000f
	PAWS-m	$s11 n15$	100	0.01	0.01	3,954	5,356		*0.0000t
	PAWS-r	$s11 n15$	100	0.01	0.01	3,733	5,183		
	PAWS-d	$s12 n15$	100	0.01	0.01	4,135	5,705		
	PAWS-a	$s12 n1$	100	0.01	0.01	4,358	5,993		
uf250-hard	SAPS	$\alpha 1.3 \rho 0.7 s5 n1$	100	0.41	0.56	160,710	223,593		*0.0000f
	SAPS+m	$\alpha 1.3 \rho 0.7 s5 n1$	100	0.39	0.58	149,149	223,794		*0.0000t
	SAPS+r	$\alpha 1.3 \rho 0.7 s5 n1$	100	0.39	0.55	156,802	220,161		
	SAPS+d	$\alpha 1.3 \rho 0.7 s20 n1$	100	0.34	0.53	140,166	215,087		
	SAPS+a	$\alpha 1.3 \rho 0.7 s20 n1$	100	0.36	0.52	140,695	206,023	*0.0412f	
	SAPS+t	$\alpha 1.3 \rho 0.7 s20 n15$	100	0.53	0.70	152,322	202,540		
	PAWS	$s18 n15$	100	0.52	0.79	213,393	320,273	*0.0008f	
	PAWS-m	$s17 n15$	100	0.63	0.94	262,147	394,199		
	PAWS-r	$s18 n15$	100	0.56	0.83	229,184	342,383		
	PAWS-d	$s7 n15$	100	0.55	0.78	222,563	316,954		
	PAWS-a	$s7 n1$	100	0.64	0.91	265,259	375,917		
uf400-med	SAPS	$\alpha 1.3 \rho 0.2 s5 n1$	100	0.12	0.17	42,514	61,159		
	SAPS+m	$\alpha 1.3 \rho 0.2 s5 n1$	100	0.12	0.17	41,799	59,483		
	SAPS+r	$\alpha 1.3 \rho 0.2 s5 n1$	100	0.13	0.17	45,721	61,997		
	SAPS+d	$\alpha 1.3 \rho 0.2 s20 n1$	100	0.09	0.14	33,420	50,452		
	SAPS+a	$\alpha 1.3 \rho 0.2 s20 n15$	100	0.09	0.13	31,938	47,701	*0.0000f	
	SAPS+t	$\alpha 1.3 \rho 0.2 s20 n15$	100	0.13	0.19	33,425	48,856		
	PAWS	$s9 n15$	100	0.08	0.10	28,601	38,363	*0.0000f	*0.0000f
	PAWS-m	$s9 n15$	100	0.07	0.10	27,945	39,660		*0.0000t
	PAWS-r	$s9 n15$	100	0.08	0.11	29,027	42,359		
	PAWS-d	$s12 n15$	100	0.09	0.12	32,760	44,729		
	PAWS-a	$s12 n1$	100	0.09	0.13	35,277	49,865		
uf400-hard	SAPS	$\alpha 1.3 \rho 0.2 s5 n1$	100	2.06	4.01	744,592	1,446,987		
	SAPS+m	$\alpha 1.3 \rho 0.2 s5 n1$	100	2.92	4.10	1,028,231	1,441,876		
	SAPS+r	$\alpha 1.3 \rho 0.2 s5 n1$	100	2.15	3.33	779,638	1,207,029		
	SAPS+d	$\alpha 1.3 \rho 0.2 s20 n1$	100	1.46	2.14	549,547	804,463		
	SAPS+a	$\alpha 1.3 \rho 0.2 s20 n1$	100	1.31	1.98	479,088	726,173	*0.0001f	
	SAPS+t	$\alpha 1.3 \rho 0.2 s20 n15$	96	2.09	5.13	555,764	1,496,309		
	PAWS	$s8 n15$	100	1.71	2.28	699,892	929,791	*0.0036f	*0.0178f
	PAWS-m	$s8 n15$	100	1.76	2.48	705,893	1,000,962		*0.0026f
	PAWS-r	$s8 n15$	100	2.08	2.76	857,409	1,154,580		
	PAWS-d	$s17 n15$	100	2.08	2.55	834,369	1,017,859		
	PAWS-a	$s17 n1$	100	2.83	3.88	1,116,114	1,537,465		

Table 4: Small random 3-SAT problem results

4.5.4 Large Random 3-SAT Results

These problems continue the small 3-SAT results from Table 4, and show the dominance of PAWS growing as problem size increases, with significantly better performance compared to all SAPS variants for all problems in terms of both flips and time (see Table 5).

PAWS remains dominant over PAWS-a, but PAWS-d also performs well on the three larger and more difficult problems. More interestingly, the previous dominance of SAPS+a over SAPS breaks down, with no significant difference on any problem except f1600-med where SAPS dominates.

				Time (secs)		Flips		Wilcoxon	
Problem	Method	Parameters	%	median	mean	median	mean	intra	inter
f800-med	SAPS	$\alpha 1.25\rho 0.1 s5n1$	100	0.59	0.91	169,562	263,105	0.3358f	
	SAPS+m	$\alpha 1.25\rho 0.1 s5n1$	100	0.74	0.96	207,760	270,976		
	SAPS+r	$\alpha 1.25\rho 0.1 s5n1$	100	0.76	0.95	221,127	272,112		
	SAPS+d	$\alpha 1.25\rho 0.1 s30n15$	100	0.54	1.02	160,086	307,293		
	SAPS+a	$\alpha 1.25\rho 0.1 s30n15$	100	0.70	0.99	213,200	284,172		
	SAPS+t	$\alpha 1.25\rho 0.1 s30n15$	100	1.11	1.39	228,023	282,938		
	PAWS	$s9n15$	100	0.26	0.36	82,392	115,451	*0.0007f	*0.0000f
	PAWS-m	$s9n15$	100	0.33	0.43	102,667	131,183		*0.0000t
	PAWS-r	$s9n15$	100	0.29	0.49	91,693	172,521		
	PAWS-d	$s16n15$	100	0.29	0.45	95,509	145,289		
	PAWS-a	$s16n1$	100	0.42	0.54	129,184	171,549		
f800-hard	SAPS	$\alpha 1.25\rho 0.3 s5n1$	100	4.88	6.12	1,414,621	1,754,017	0.2034f	
	SAPS+m	$\alpha 1.25\rho 0.3 s5n1$	100	5.35	6.45	1,494,538	1,804,377		
	SAPS+r	$\alpha 1.25\rho 0.3 s5n1$	100	3.92	5.54	1,113,578	1,576,366		
	SAPS+d	$\alpha 1.25\rho 0.3 s30n1$	100	5.91	7.50	1,739,737	2,184,186		
	SAPS+a	$\alpha 1.25\rho 0.3 s30n1$	100	5.49	7.25	1,557,723	2,039,950		
	SAPS+t	$\alpha 1.25\rho 0.3 s30n15$	99	8.14	11.16	1,545,959	2,135,779		
	PAWS	$s10n15$	100	2.58	3.18	897,696	1,087,076	0.2442f	*0.0011f
	PAWS-m	$s10n15$	100	3.04	4.36	916,292	1,334,897		*0.0000t
	PAWS-r	$s10n15$	100	3.53	4.40	1,199,636	1,607,297		
	PAWS-d	$s14n15$	100	2.27	3.09	753,345	1,035,762		
	PAWS-a	$s14n1$	100	2.80	4.11	867,340	1,277,586		
f1600-med	SAPS	$\alpha 1.25\rho 0.3 s5n1$	99	3.06	5.94	693,385	1,303,941	*0.0279f	
	SAPS+m	$\alpha 1.25\rho 0.3 s5n1$	100	2.58	4.96	538,407	1,033,478		
	SAPS+r	$\alpha 1.25\rho 0.3 s5n1$	100	3.30	5.12	715,152	1,098,818		
	SAPS+d	$\alpha 1.25\rho 0.3 s30n15$	99	4.80	8.11	1,086,758	1,920,641		
	SAPS+a	$\alpha 1.25\rho 0.3 s30n15$	100	4.49	8.07	1,036,529	1,810,566		
	SAPS+t	$\alpha 1.25\rho 0.3 s30n15$	92	5.64	16.70	896,631	2,742,393		
	PAWS	$s10n15$	100	0.98	1.74	284,591	548,322	*0.0006f	*0.0000f
	PAWS-m	$s10n15$	100	2.02	2.29	499,642	576,618		*0.0000t
	PAWS-r	$s11n15$	100	2.04	2.67	521,920	762,255		
	PAWS-d	$s14n15$	100	1.18	1.76	327,235	501,171		
	PAWS-a	$s16n1$	100	1.68	2.21	484,481	637,422		
f1600-hard	SAPS	$\alpha 1.25\rho 0.3 s5n1$	94	30.62	34.34	6,499,140	7,777,980		
	SAPS+m	$\alpha 1.25\rho 0.3 s5n1$	92	22.52	35.53	4,750,016	8,038,419		
	SAPS+r	$\alpha 1.25\rho 0.3 s5n1$	88	35.33	38.21	7,490,954	8,996,248		
	SAPS+d	$\alpha 1.25\rho 0.3 s30n1$	70	54.91	54.66	11,777,404	14,064,479		
	SAPS+a	$\alpha 1.25\rho 0.3 s28n15$	95	23.13	32.29	5,097,994	7,389,302	0.2850f	
	SAPS+t	$\alpha 1.25\rho 0.3 s28n15$	75	43.74	57.72	6,514,258	8,996,673		
	PAWS	$s11n15$	96	11.94	18.86	3,000,027	5,019,099	*0.0233f	*0.0001f
	PAWS-m	$s9n15$	94	11.72	17.15	3,764,003	5,826,437		*0.0000t
	PAWS-r	$s11n15$	95	16.37	21.59	4,778,339	6,348,370		
	PAWS-d	$s14n15$	100	9.35	16.02	2,709,427	4,711,993		
	PAWS-a	$s16n1$	95	16.20	19.86	4,651,466	6,053,078		

Table 5: Large random 3-SAT problem results

4.5.5 Structured DIMACS Results

These less related problems show PAWS doing significantly better on the par16 and logistics instances, but with SAPS pulling ahead on flip count for the ais problems (see Table 6). However, as the ais problem difficulty increases, there are signs that PAWS scales better, particularly in terms of time performance.

SAPS+a returns to its position of relative dominance over SAPS, although it only achieves a significant difference on logistics.c and ais12. PAWS also continues to dominate or roughly equal the performance of PAWS-a and its other variants.

4.5.6 Random CSP Results

Finally, Table 7 show the results for the random binary CSPs. These problems present a mixed picture, with SAPS showing better flip but equal time performance on bin30-40, and PAWS showing significantly better time and flip performance on bin30-80. For the larger problems, and unlike the other problem domains, SAPS and PAWS show roughly equivalent performance, with SAPS having an edge in terms of flips for bin50-40 and

				Time (secs)		Flips		Wilcoxon	
Problem	Method	Parameters	%	median	mean	median	mean	intra	inter
logistics.c	SAPS	$\alpha 1.3\rho 0.9 s5 n1$	100	0.04	0.05	6,954	8,436		
	SAPS+m	$\alpha 1.3\rho 0.9 s5 n1$	100	0.04	0.05	6,687	8,246		
	SAPS+r	$\alpha 1.3\rho 0.9 s5 n1$	100	0.04	0.05	6,512	8,328		
	SAPS+d	$\alpha 1.3\rho 0.9 s20 n1$	100	0.04	0.04	6,450	8,071		
	SAPS+a	$\alpha 1.3\rho 0.9 s20 n1$	100	0.04	0.05	6,397	8,028	*0.0378f	
	SAPS+t	$\alpha 1.3\rho 0.9 s20 n15$	100	0.05	0.06	6,740	8,610		
	PAWS	$s\infty n15$	100	0.02	0.03	5,229	6,771	*0.0280f	*0.0000f
	PAWS-m	$s\infty n15$	100	0.02	0.03	5,144	6,611		*0.0000t
	PAWS-r	$s\infty n15$	100	0.02	0.03	5,604	7,612		
	PAWS-d	$s0n15$	100	0.02	0.03	5,047	6,588		
	PAWS-a	$s0n1$	100	0.02	0.03	5,530	6,734		
logistics.d	SAPS	$\alpha 1.2\rho 1.0 s4 n1$	100	0.18	0.19	19,202	21,248		
	SAPS+m	$\alpha 1.2\rho 1.0 s4 n1$	100	0.19	0.20	18,269	20,904		
	SAPS+r	$\alpha 1.2\rho 1.0 s4 n1$	100	0.18	0.20	19,199	21,486		
	SAPS+d	$\alpha 1.2\rho 1.0 s23 n1$	100	0.18	0.19	18,721	21,384		
	SAPS+a	$\alpha 1.2\rho 1.0 s23 n1$	100	0.20	0.21	18,869	21,355	0.3690f	
	SAPS+t	$\alpha 1.2\rho 1.0 s23 n15$	100	0.36	0.38	21,794	24,005		
	PAWS	$s\infty n15$	100	0.12	0.14	18,330	22,632		0.0707f
	PAWS-m	$s\infty n15$	100	0.11	0.12	18,163	21,546		*0.0000t
	PAWS-r	$s\infty n15$	100	0.12	0.13	18,316	21,777		
	PAWS-d	$s0n1$	100	0.12	0.13	17,584	21,351		
	PAWS-a	$s0n1$	100	0.11	0.12	17,867	21,236	*0.0519f	
ais10	SAPS	$\alpha 1.3\rho 0.9 s4 n1$	100	0.06	0.10	11,708	18,085		*0.0182f
	SAPS+m	$\alpha 1.3\rho 0.9 s4 n1$	100	0.07	0.11	13,197	19,692		
	SAPS+r	$\alpha 1.3\rho 0.9 s4 n1$	100	0.07	0.10	13,225	18,442		
	SAPS+d	$\alpha 1.3\rho 0.9 s25 n1$	100	0.07	0.10	12,516	18,755		
	SAPS+a	$\alpha 1.3\rho 0.9 s25 n1$	100	0.06	0.09	12,086	17,299	0.3011f	
	SAPS+t	$\alpha 1.3\rho 0.9 s25 n15$	100	0.08	0.11	13,207	18,670		
	PAWS	$s52 n15$	100	0.06	0.09	13,661	19,594	0.0712f	0.4243t
	PAWS-m	$s52 n15$	100	0.07	0.09	14,227	20,086		
	PAWS-r	$s52 n15$	100	0.07	0.11	15,024	22,974		
	PAWS-d	$s2 n15$	100	0.07	0.09	14,081	19,892		
	PAWS-a	$s2 n1$	100	0.07	0.10	14,836	20,638		
ais12	SAPS	$\alpha 1.25\rho 0.95 s4 n1$	100	0.60	0.86	86,025	123,099		*0.0014f
	SAPS+m	$\alpha 1.25\rho 0.95 s4 n1$	100	0.67	0.96	93,867	133,992		
	SAPS+r	$\alpha 1.25\rho 0.95 s4 n1$	100	0.60	0.88	85,202	125,727		
	SAPS+d	$\alpha 1.25\rho 0.95 s30 n1$	100	0.60	0.86	88,482	127,737		
	SAPS+a	$\alpha 1.25\rho 0.95 s30 n15$	100	0.53	0.81	78,086	117,774	*0.0000f	
	SAPS+t	$\alpha 1.25\rho 0.95 s30 n15$	100	0.68	0.98	90,437	130,949		
	PAWS	$s148 n15$	100	0.57	0.80	102,774	142,979	0.2565f	0.0792t
	PAWS-m	$s148 n15$	100	0.52	0.79	94,512	143,541		
	PAWS-r	$s148 n15$	100	0.64	0.94	111,792	164,807		
	PAWS-d	$s1 n1$	100	0.60	0.85	102,253	145,982		
	PAWS-a	$s1 n1$	100	0.60	0.87	102,275	149,958		
par16-med	SAPS	$\alpha 2\rho 0.25 s7 n1$	88	12.32	20.98	5,589,195	7,720,965		
	SAPS+m	$\alpha 2\rho 0.25 s7 n1$	90	12.73	19.90	5,563,752	7,667,145		
	SAPS+r	$\alpha 2\rho 0.25 s7 n1$	89	10.19	20.95	4,624,300	7,340,485		
	SAPS+d	$\alpha 2\rho 0.25 s15 n1$	96	10.68	15.22	4,985,909	6,677,620		
	SAPS+a	$\alpha 2\rho 0.25 s15 n15$	95	11.01	14.48	4,885,148	6,595,288	0.3181f	
	SAPS+t	$\alpha 2\rho 0.25 s15 n15$	40	timed out	54.53	timed out	12,899,759		
	PAWS	$s36 n15$	97	5.32	8.77	2,646,531	4,496,763	0.0692f	*0.0004f
	PAWS-m	$s36 n15$	100	6.00	8.14	3,116,219	4,216,251		*0.0001t
	PAWS-r	$s36 n15$	99	6.70	8.98	3,316,710	4,517,832		
	PAWS-d	$s3 n15$	97	8.16	10.36	4,081,225	5,320,932		
	PAWS-a	$s3 n15$	98	7.30	10.31	3,835,814	5,512,200		
par16-hard	SAPS	$\alpha 1.4\rho 0.9 s4 n1$	86	13.78	18.71	6,454,597	9,725,495		
	SAPS+m	$\alpha 1.4\rho 0.9 s4 n1$	83	17.02	20.19	7,687,612	10,286,242		
	SAPS+r	$\alpha 1.4\rho 0.9 s4 n1$	87	12.34	17.37	5,781,604	8,950,757		
	SAPS+d	$\alpha 1.4\rho 0.9 s30 n1$	90	14.88	16.87	6,954,962	8,547,129		
	SAPS+a	$\alpha 1.4\rho 0.9 s25 n1$	94	12.86	16.11	5,751,190	7,597,764	0.0790f	
	SAPS+t	$\alpha 1.4\rho 0.9 s25 n15$	79	28.17	36.20	7,272,386	9,141,042		
	PAWS	$s40 n15$	98	6.79	9.51	3,379,909	4,809,418	0.1057f	*0.0000f
	PAWS-m	$s40 n15$	100	6.43	8.45	3,314,958	4,355,509		*0.0000t
	PAWS-r	$s40 n15$	98	6.67	8.90	3,312,261	4,493,758		
	PAWS-d	$s3 n15$	97	12.25	13.78	6,144,198	7,104,478		
	PAWS-a	$s3 n1$	99	7.67	10.79	4,072,925	5,761,065		

Table 6: Structured DIMACS problem results

PAWS being significantly better in terms of time on bin50-80.

As with the large 3-SAT problems, the SAPS+a variant no longer clearly dominates SAPS, showing roughly equivalent performance on bin50-40, slightly better performance on bin30-40, significantly better performance on bin30-80, but significantly worse performance on bin50-80. PAWS and PAWS-a show similar performance, with

				Time (secs)		Flips		Wilcoxon	
Problem	Method	Parameters	%	median	mean	median	mean	intra	inter
bin30-80	SAPS	$\alpha 1.3 \rho 0.1 s6n1$	100	0.06	0.08	8,661	12,299		
	SAPS+m	$\alpha 1.3 \rho 0.1 s6n1$	100	0.05	0.08	7,729	11,853		
	SAPS+r	$\alpha 1.3 \rho 0.1 s6n1$	100	0.05	0.08	7,940	12,242		
	SAPS+d	$\alpha 1.3 \rho 0.1 s20n1$	100	0.05	0.07	7,999	11,642		
	SAPS+a	$\alpha 1.3 \rho 0.1 s20n1$	100	0.05	0.07	7,511	10,593	*0.0009f	
	SAPS+t	$\alpha 1.3 \rho 0.1 s20n15$	100	0.05	0.08	7,652	10,594		
	PAWS	$s7n15$	100	0.04	0.06	7,576	10,633	*0.0449f	*0.0066f
	PAWS-m	$s7n15$	100	0.04	0.05	7,283	10,102		*0.0000t
	PAWS-r	$s9n15$	100	0.05	0.07	9,089	12,089		
	PAWS-d	$s15n15$	100	0.05	0.07	8,463	11,803		
	PAWS-a	$s17n1$	100	0.05	0.07	8,172	11,956		
bin30-40	SAPS	$\alpha 1.25 \rho 0.5 s6n1$	100	0.08	0.12	13,716	19,711		*0.0101f
	SAPS+m	$\alpha 1.25 \rho 0.5 s6n1$	100	0.09	0.12	14,470	20,149		0.4072t
	SAPS+r	$\alpha 1.25 \rho 0.5 s6n1$	100	0.08	0.12	14,031	20,330		
	SAPS+d	$\alpha 1.25 \rho 0.5 s17n1$	100	0.08	0.11	13,644	18,797		
	SAPS+a	$\alpha 1.25 \rho 0.5 s15n1$	100	0.08	0.11	12,741	19,119	0.1191f	
	SAPS+t	$\alpha 1.25 \rho 0.5 s15n15$	100	0.08	0.12	12,044	17,540		
	PAWS	$s7n15$	100	0.08	0.12	15,927	22,422	*0.0000f	
	PAWS-m	$s7n15$	100	0.08	0.13	15,779	24,321		
	PAWS-r	$s9n15$	100	0.10	0.14	18,746	27,309		
	PAWS-d	$s20n15$	100	0.08	0.12	15,798	23,287		
	PAWS-a	$s20n1$	100	0.10	0.14	19,432	27,610		
bin50-80	SAPS	$\alpha 1.2 \rho 0.1 s6n1$	100	1.81	2.92	119,552	186,552	*0.0181f	
	SAPS+m	$\alpha 1.2 \rho 0.1 s6n1$	100	2.08	3.82	130,022	224,231		
	SAPS+r	$\alpha 1.2 \rho 0.1 s6n1$	100	2.35	3.53	141,777	216,651		
	SAPS+d	$\alpha 1.2 \rho 0.1 s30n15$	100	2.58	3.81	202,745	297,099		
	SAPS+a	$\alpha 1.2 \rho 0.1 s30n1$	100	2.19	3.54	160,871	262,727		
	SAPS+t	$\alpha 1.2 \rho 0.1 s25n15$	98	2.12	7.09	141,226	604,380		
	PAWS	$s5n15$	100	1.44	1.85	128,837	168,402	0.4011f	0.3574f
	PAWS-m	$s5n15$	100	0.99	1.42	90,567	130,162		*0.0062t
	PAWS-r	$s5n15$	100	1.73	2.60	165,065	266,514		
	PAWS-d	$s30n15$	100	1.63	1.99	147,552	182,763		
	PAWS-a	$s30n15$	100	1.44	2.14	134,187	198,013		
bin50-40	SAPS	$\alpha 1.25 \rho 0.25 s5n1$	99	96.84	149.05	7,579,338	11,562,103		0.0735f
	SAPS+m	$\alpha 1.25 \rho 0.25 s5n1$	92	114.69	165.67	8,961,133	11,552,914		0.2535t
	SAPS+r	$\alpha 1.25 \rho 0.25 s5n1$	100	100.58	149.47	7,783,134	11,482,673		
	SAPS+d	$\alpha 1.25 \rho 0.25 s20n1$	99	81.24	120.76	6,450,368	9,449,164		
	SAPS+a	$\alpha 1.25 \rho 0.25 s20n1$	100	101.38	131.05	7,682,577	12,130,118	0.3305f	
	SAPS+t	$\alpha 1.25 \rho 0.25 s20n15$	37	timed out	334.81	timed out	32,454,528		
	PAWS	$s6n15$	98	121.12	169.55	10,866,838	14,848,547		
	PAWS-m	$s6n15$	91	155.61	209.73	13,644,648	17,170,359		
	PAWS-r	$s6n15$	100	114.85	194.17	10,487,116	17,164,797		
	PAWS-d	$s30n15$	100	84.64	119.13	7,591,267	13,659,181		
	PAWS-a	$s30n1$	99	100.44	126.90	8,923,186	14,874,496	*0.0419f	

Table 7: Random binary CSP problem results

PAWS dominating on the smaller bin30 problems, and PAWS-a matching PAWS on bin50-80 and dominating on bin50-40.

5 Analysis

Table 8 gives an overall comparison of the results from Tables 2 to 7, identifying the best variant for each algorithm on each problem, and giving a Wilcoxon and RTD analysis of the comparative time performance of these best variants. As discussed in Section 4.3, one variant is only considered significantly better than another if the Wilcoxon rank sum test is significant *and* it has a dominating RTD.

Table 8 also provides statistics on the relative average lengths of list L for SAPS, SAPS+t and PAWS, and a comparison of the relative SAPS and PAWS flip rates. As the flip rates and list lengths remained stable across problem variants, we only report the statistics for the base versions of SAPS and PAWS (with the exception of SAPS+t list lengths which were affected by the threshold heuristic). We also show the Satz and zChaff solution times for each problem in seconds (as discussed in Section 4.2).

	Best Time Variant							List Length			Flips per sec	
Problem	SAPS	PAWS	Overall	Wilcoxon	RTD	Satz	zChaff	SAPS	SAPS+t	PAWS	SAPS	PAWS
bw_large.a	SAPS+a	PAWS-m	PAWS	0.0021	$\surd$	0.08	0.01	1.4843	1.9527	2.8450	222,967	253,184
bw_large.b	SAPS+a	PAWS-m	no sig diff	0.2124	$\times$	0.26	†0.01	1.1457	1.7786	3.1005	176,073	209,888
bw_large.c	SAPS+a	PAWS-m	PAWS	0.0000	$\surd$	2.15	†0.53	1.0185	3.2655	4.0911	79,529	171,719
bw_large.d	SAPS+a	PAWS-m	PAWS	0.0000	$\surd$	660.60	†2.01	1.0654	5.7542	4.9923	63,320	131,772
flat100-med	SAPS+d	PAWS-m	SAPS	0.0003	$\surd$	0.01	0.01	1.1612	2.0977	3.4890	546,523	578,702
flat100-hard	SAPS+a	PAWS-d	SAPS	0.0012	$\surd$	0.02	†0.01	1.0405	2.1411	3.0929	564,152	592,997
flat200-med	SAPS+a	PAWS	no sig diff	0.0355	$\times$	0.12	0.30	1.0617	3.0782	5.2708	483,891	519,477
flat200-hard	SAPS+d	PAWS	no sig diff	0.2747	$\times$	†0.03	0.57	1.0016	3.2044	3.4551	497,716	541,442
g125.17	SAPS+a	PAWS-m	PAWS	0.0000	$\surd$	$>$ 1hr	$>$ 1hr	1.0045	3.2766	3.5996	51,289	77,222
g250.29	SAPS+r	PAWS-m	PAWS	0.0000	$\surd$	$>$ 1hr	$>$ 1hr	1.0358	5.3512	4.7199	11,927	14,439
uf100-hard	SAPS+d	PAWS	no sig diff	0.0850	$\surd$	0.03	0.01	1.0269	1.4609	1.9685	433,712	454,314
uf250-med	SAPS+d	PAWS	PAWS	0.0002	$\surd$	1.25	4.63	1.0793	1.7371	3.1550	391,263	406,842
uf250-hard	SAPS+a	PAWS	SAPS	0.0000	$\surd$	†0.32	93.62	1.0027	1.7448	3.3338	397,796	407,566
uf400-med	SAPS+a	PAWS	PAWS	0.0000	$\surd$	57.81	$>$ 1hr	1.2044	2.0989	4.0647	358,898	379,945
uf400-hard	SAPS+a	PAWS	no sig diff	0.1778	$\times$	178.92	$>$ 1hr	1.0011	2.0151	3.3460	361,168	407,892
f800-med	SAPS	PAWS	PAWS	0.0000	$\surd$‡	$>$ 1hr	$>$ 1hr	1.0211	2.8490	5.2173	289,413	321,861
f800-hard	SAPS+r	PAWS-d	PAWS	0.0001	$\surd$‡	$>$ 1hr	$>$ 1hr	1.0032	2.8492	4.1244	286,740	342,084
f1600-med	SAPS+m	PAWS	PAWS	0.0000	$\surd$	$>$ 1hr	$>$ 1hr	1.0155	4.2052	4.2509	217,590	314,260
f1600-hard	SAPS+a	PAWS-d	PAWS	0.0000	$\surd$	$>$ 1hr	$>$ 1hr	1.0030	4.2985	7.9250	215,021	257,173
logistics.c	SAPS+a	PAWS-d	PAWS	0.0000	$\surd$	0.45	0.08	2.5263	2.9066	3.7940	179,318	247,502
logistics.d	SAPS+m	PAWS-a	PAWS	0.0000	$\surd$	505.20	0.19	17.7148	17.8127	18.2289	110,813	167,203
ais10	SAPS+a	PAWS	no sig diff	0.3532	$\times$	0.06	0.09	1.0130	1.2968	2.0533	187,028	209,968
ais12	SAPS+a	PAWS	no sig diff	0.3518	$\times$	0.17	2.73	1.0036	1.3062	1.7517	143,347	179,228
par16-med	SAPS+a	PAWS-m	PAWS	0.0003	$\surd$‡	1.66	†0.49	1.0007	3.7710	5.3424	337,548	499,311
par16-hard	SAPS+a	PAWS-m	PAWS	0.0000	$\surd$‡	†0.56	1.71	1.0005	3.8163	5.6082	468,056	496,370
bin30-80	SAPS+a	PAWS-m	PAWS	0.0000	$\surd$	0.26	28.48	1.0363	1.6553	2.8070	153,169	187,080
bin30-40	SAPS+d	PAWS	no sig diff	0.1263	$\times$	0.33	†0.02	1.0220	1.7828	3.3194	170,857	189,825
bin50-80	SAPS	PAWS-m	PAWS	0.0000	$\surd$	$>$ 1hr	$>$ 1hr	1.0091	1.6773	3.4783	63,909	90,890
bin50-40	SAPS+d	PAWS-d	no sig diff	0.4766	$\times$	$>$ 1hr	$>$ 1hr	1.0002	1.4881	5.1387	78,822	89,409

Table 8: Overall problem comparison. Key: †indicates the Satz or zChaff run-time dominates all other methods; $\surd$ indicates the run-time distribution (RTD) of the overall best variant dominates the other best variant; ‡ indicates the RTD domination is not perfect, some cross-over at solution probability < 0.1; $\times$ indicates significant cross-over of RTDs at solution probability > 0.1

As there are considerable differences between the average flip rates for SAPS and PAWS on nearly all problem instances, in the following analysis we limit the comparison between SAPS and PAWS to their relative run-time distributions. However, as flip rates are fairly stable between variants of the same algorithm class, we generally consider flip distributions when comparing particular variants.

5.1 PAWS versus SAPS

The first striking feature of Table 8 is the dominance of the PAWS variants on the overall problem set. Of the 29 problem instances, PAWS is significantly better on 17 instances, SAPS is significantly better 3 instances, with no significant difference on the remaining 9 instances. For the 17 instances on which PAWS is classed as better, in 13 cases the RTDs are clearly dominant, and in 4 cases there is some crossing at a solution probability of less than 10% (marked with a ‡ in Table 8). To give an idea of these

distributions, Figures 7 and 8 show two of the RTDs that cross at less than 10%, Figure 5 shows a clearly dominant RTD and Figure 6 shows clearly crossing distributions.

The three instances on which SAPS does dominate are of relatively small size and can each be solved within 0.32 seconds by Satz or zChaff, and for those problems which the complete solvers find challenging (i.e. take longer than one second to solve), SAPS equals the performance of PAWS on only two instances: uf400-hard and bin50-40. In this context, bin50-40 presents an interesting case, as it has the longest solution times and highest flip count within the problem set, so any conclusion of the superiority of PAWS on larger problems must necessarily be qualified. Also, as with all empirical evaluations of stochastic local search algorithms, our conclusions cannot be reliably generalized beyond the given problem set. Having said this, the results do indicate that additive weighting has better general time performance than any of the multiplicative alternatives considered.

5.2 PAWS Variants

An examination of the relative performance of each PAWS variant in Table 8 shows that standard PAWS is better on 12 instances, PAWS-m is better on 11 instances, PAWS-d is better on 5 instances and PAWS-a is better on 1 instance (but only marginally). This firstly indicates that PAWS-r and PAWS-a do not perform well, and by implication that the random flat move heuristic is playing an important role in the performance of PAWS (i.e. both PAWS-r and PAWS-a have had the random flat move heuristic removed).

Considering the flip count statistics of PAWS in relation to PAWS-d, there are several problems where PAWS-d has considerably worse performance, e.g. bw_large.c, flat200-med, g250.29 and par16-hard, whereas on the five problems where PAWS-d has the best performance, the mean flip count in comparison to PAWS differs by less than 10%. A further run-*length* distribution (RLD) analysis [5] comparing flip performance on these problems confirmed that PAWS-d does not clearly dominate PAWS on any problem instance (an RLD analysis differs from the RTD analysis only in considering flip instead of time performance). Hence there is strong evidence suggesting that deterministic smoothing performs better than probabilistic smoothing for PAWS.

Lastly, the nearly equal first status of PAWS and PAWS-m (on a simple count of the problems on which they do better) suggests that they have roughly equal overall performance. However, a closer analysis of the flip counts for each problem shows there are several problems on which PAWS has considerably better mean flip performance (uf250-med, f800-hard, f1600-hard and bin50-40) and a similar number on which PAWS-m appears to dominate (par16-med, par16-hard and bin50-40). We therefore performed another RLD analysis on these problems, which showed a significant dominance only on bin50-40 (in favour of PAWS) and bin50-80 (in favour of PAWS-m). As there was no significant difference on any other problem, this suggests the multiple inclusion heuristic has a minimal effect on the overall performance of PAWS.

We therefore conclude, on the basis of the experimental evidence, that the PAWS deterministic smoothing and random flat move heuristics do contribute positively to the performance of additive weighting, and that the multiple inclusion heuristic has no significant effect either positively or negatively.

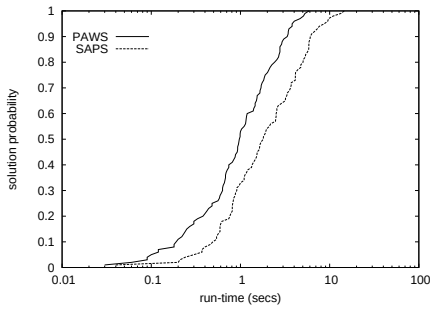

Figure 5: Run-time distribution comparing SAPS and PAWS-m on bin50-80

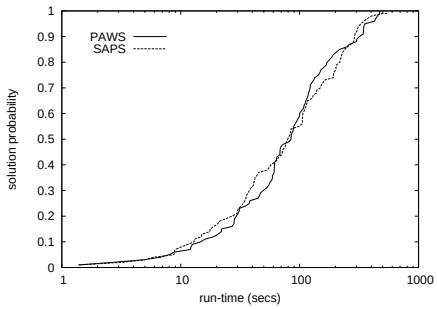

Figure 6: Run-time distribution comparing SAPS+d and PAWS-d on bin50-40

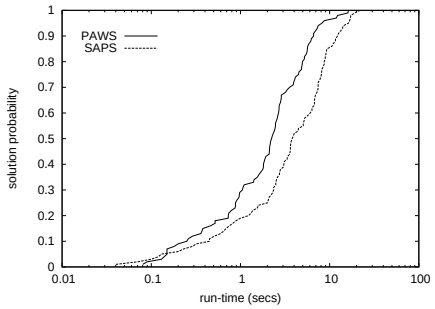

Figure 7: Run-time distribution comparing SAPS+r and PAWS-d on f800-hard

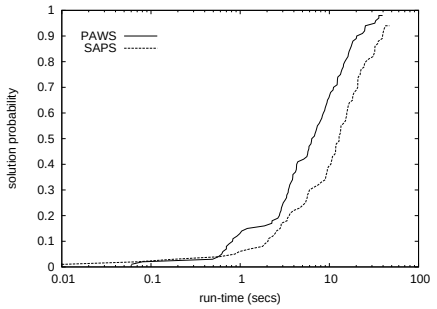

Figure 8: Run-time distribution comparing SAPS+a and PAWS-m on par16-hard

5.3 SAPS Variants

Again performing a count on Table 8 gives 17 problems for which SAPS+a is better, 6 problems for which SAPS+d is better, 2 problems for which SAPS+r is better, 2 problems for which SAPS+m is better, and 2 problems for which SAPS is better. The counts certainly suggest that SAPS benefits from the inclusion of the PAWS heuristics. However, a closer examination of the problem flip counts shows that it is hard to draw a general conclusion that fits all problem instances.

Firstly, SAPS+a and SAPS+d often clearly perform better than the other variants, while themselves having similar performance, i.e. on bw_large.b, c and d, flat100-med, flat200-med and hard, uf400-med and hard and par16-med and hard. However, there are other problems where SAPS does well and SAPS+a and SAPS+d do relatively worse, i.e. on f800-med and hard, f1600-med and bin50-80. Then there are problems where SAPS+d does badly relative to all other variants, i.e. g125.17 and f1600-hard, and other problems where SAPS+d does well and SAPS+a does poorly, i.e. g250.29 and bin50-40. Considering the other variants, SAPS+r only stands out on f800-hard and g250.29 and SAPS+m only stands out on f1600-med, otherwise their performance follows SAPS fairly closely. Hence, we consider that the +m and +r heuristics do not have a major effect on SAPS, at least in isolation. This result is further supported by the relatively insignificant effects that would be expected from these heuristics. Firstly, although SAPS+m biases the move choice towards literals that appear more than once in the false clause list, it does not override the move cost. Also, removing this heuristic from PAWS has already been shown above to have little effect. Secondly, the SAPS+r heuristic is only operational in situations where no improving move is available, and then only for 1% of the time. At this point it simply reduces the domain of choice from all possible moves, to moves that have a zero cost (i.e. within the threshold of ± 0.1). While this removes the chance of taking a cost increasing move, such moves will typically be quickly reversed in a local search. Also, in further work on SAPS, the removal of the random flip heuristic has been shown to have little noticeable effect [20]. Our results therefore support these findings.

This leaves SAPS+d and SAPS+a as the two candidate best SAPS variants. Of these SAPS+d has a slight advantage, firstly, because its worst performance is on problems for which SAPS is not competitive, and secondly because it represents a simpler change to SAPS, i.e. switching from probabilistic to deterministic smoothing. However, uniformly adopting deterministic smoothing would definitely degrade the performance of SAPS on a range of the larger randomly generated CSP and 3-SAT problems. We therefore conclude that the best overall performance could be obtained by adding an additional SAPS parameter that switches between deterministic and probabilistic smoothing. This extends the results presented in [20], where a deterministic version of SAPS was found not to differ from SAPS in performance on a range of the smaller problems already considered in this study.

5.4 The SAPS Threshold Heuristic

So far we have not considered the SAPS threshold variant, SAPS+t. This is because, while it can equal the flip performance of SAPS+a (on which it is based) for many

smaller problems, it produced some of the highest failure rates of any variant on several of the larger problems (uf400-hard, f1600-med and hard, par16-med and hard, and bin50-40). Also, due to the additional overhead of calculating an averaged flip cost, the time performance of SAPS+t was uniformly worse than SAPS+a. Hence, we can conclude that adding a threshold, at least to SAPS+a, does not improve the performance of multiplicative weighting.

In relation to the effect of the threshold heuristic on the candidate list lengths, Table 8 clearly shows the greater choice in candidate moves available to PAWS, and that, as solution times increase, the SAPS list length tends to zero. For the SAPS+t experiments we set the threshold value to 0.1, producing SAPS+t list lengths somewhere between those of SAPS and PAWS. Further experiments with larger threshold values did produce longer list lengths, but these changes uniformly caused SAPS+t performance to degrade. Hence we have no evidence to suggest that the superior performance of PAWS can be explained by its greater choice of moves. If this were the case, we would expect SAPS+t to have improved over SAPS+a, as SAPS+a *is* PAWS except that it uses multiplicative weighting. This refutes our earlier conjecture [17] and reopens the question of explaining the superior performance of PAWS, especially on the larger problems.

6 Conclusions

The aim of this study was to identify and analyze the key features required for an effective clause weighting local search algorithm. On the basis of the previous work, we observed that the best clause weighting algorithms use the same underlying strategy, i.e. to increase clause weights in a local minimum, and then to periodically reduce or smooth these weights to maintain a stable relative weight distribution that remains sensitive to local conditions in the search space. From this we identified the key distinguishing feature of current approaches, i.e. the use of additive or multiplicative clause weighting. We therefore set out to systematically investigate the performance of additive and multiplicative clause weighting on a range of SAT benchmark problems, and using a range of sub-heuristics.

Overall, our results indicate that additive weighting tends to perform better than multiplicative weighting, particularly on larger and more difficult problems. From our investigation into the various additive and multiplicative sub-heuristics, we came to the following conclusions:

- Some form of random plateau move heuristic is useful for additive weighting. This is less relevant to multiplicative weighting, possibly because the finer weight distinctions caused by multiplicative updates produce smaller plateau areas.
- Deterministic weight reduction appears generally helpful for additive weighting, but only assists multiplicative weighting on selected instances.
- The effect of multiple inclusion heuristic is not significant. Overall it had little effect on multiplicative weighting, and only made a small difference, both positively and negatively, to additive weighting performance.

- The threshold heuristic caused a fairly uniform deterioration in the performance of multiplicative weighting. This means the superior performance of additive weighting cannot obviously be explained by the greater choice of moves afforded by additive weight updates.

As the threshold heuristic failed to produce any improvement, we were led to develop a new conjecture to explain the relatively better performance of the additive approach:

Firstly, the study has shown that the differences in performance between the additive and multiplicative schemes cannot be explained by differences in the sub-heuristics used. If this were the case we would expect the performance of SAPS and PAWS to become equivalent with the right application of heuristics. However, regardless of the choice of sub-heuristic, additive weighting has shown the generally superior performance.

Secondly, our experiments with SAPS+t indicate that there is no causative link between the coarser weight distinctions of additive weighting and its better performance[9]. Hence, the overall outcome of the study suggests there is something inherent in additive weight updates that can improve the performance of clause weighting algorithms. By a process of elimination, the remaining distinction is the essential geometric nature of multiplicative weight updates, i.e. multiplicatively increasing weight will always cause those clauses with greater weight to have a greater relative increase in weight. Conversely, additive updates are more egalitarian, with each false clause getting an identical weight increase. The overall effect is that multiplicative weighting will raise the weight on a false clause more quickly, relative to other clauses with lesser weight, and will also reduce weight more quickly when a clause becomes true. Hence, a newly weighted clause will have less immediate effect on the search trajectory, and the basic ordering of clause weight importance will differ, i.e. in a multiplicative scheme, clauses that have been false for longer will have greater importance.

In general, therefore, additive weighting is a "blunter" instrument. For instance, most clause weights at any point in an additive search will have their weights set to one, whereas multiplicative weighting retains small real valued distinctions on nearly all clauses that have been false. Additive weighting is also less selective: it does not care how long a clause has been true or false, it still gets the same update. The conjecture of our study is therefore that this generally simpler behaviour explains the better performance of additive weighting on longer term searches. In particular, additive weighting provides a relatively greater emphasis on clauses that have recently become false and so is more responsive to the immediate situation. More generally, the efficiencies gained in performing simpler clause weight updates mean additive weighting can also dominate on smaller problems where multiplicative weighting otherwise has the advantage in terms of flips.

Overall the case for preferring additive over multiplicative weighting is compelling: firstly, the average flip performance of PAWS does not differ significantly from SAPS on smaller problems and strongly dominates SAPS on the more difficult problems (i.e. those beyond the reach of Satz or zChaff). Secondly, additive weighting is more time

[9]This must be qualified by the understanding that there are other possible threshold heuristics that may have better performance

efficient than multiplicative due to using integer rather than real-valued clause weights. This is shown by the consistently faster flip rates for PAWS on most problems (remembering SAPS and PAWS 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.

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, especially in terms of flips, and in future work it would be worth identifying the problem characteristics and search behaviors that favour a multiplicative approach.

References

[1] H. Everett. Generalized Lagrange multiplier method for solving problems of the optimal allocation of resources. *Operations Research*, 11:399–417, 1963.

[2] I. Gent, H. Hoos, P. Prosser, and T. Walsh. Morphing: Combining structure and randomness. In *Proceedings of the Sixteenth National Conference on Artificial Intelligence (AAAI-99)*, 1999.

[3] J. Gibbons and S. Chakraborti. *Nonparametric Statistical Inference*, pages 241–251. Statistics: Textbooks and Monographs. Marcel Dekker, Inc., New York, 1992.

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

[5] H. Hoos and T. Stutzle. Evaluating Las Vegas algorithms: Pitfalls and remedies. In *Proceedings of Fourteenth Conference on Uncertainty in Artificial Intelligence (UAI-98)*, pages 238–245, 1998.

[6] H. Hoos and T. Stützle. *Stochastic Local Search: Foundations and Applications*. Elsevier, New York, 2005.

[7] F. Hutter, D. Tompkins, and H. Hoos. 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)*, pages 233–248, 2002.

[8] C. Li and Anbulagan. 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)*, pages 341–355, 1997.

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

[10] P. Mills and E. Tsang. 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)*, pages 872–883, 1999.

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

[12] M. Moskewicz, C. Madigan, Y. Zhao, L. Zhang, and S. Malik. Chaff: Engineering an efficient SAT solver. In *Proceedings of the 39th Design Automation Conference (DAC 2001)*, pages 530–535, 2001.

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

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

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

[16] Y. Shang and B. Wah. A discrete Lagrangian-based global search method for solving satisfiability problems. *J. Global Optimization*, 12:61–99, 1998.

[17] J. Thornton, D. Pham, S. Bain, and V. Ferreira Jr. Additive versus multiplicative clause weighting for SAT. In *Proceedings of the 19th National Conference on Artificial Intelligence, AAAI-2004*, pages 191–196, 2004.

[18] J. Thornton, W. Pullan, and J. Terry. Towards fewer parameters for clause weighting SAT algorithms. In *Proceedings of the 15th Australian Joint Conference on Artificial Intelligence, AI-2002*, pages 569–578, 2002.

[19] J. Thornton and A. Sattar. On the behaviour and application of constraint weighting. In *Proceedings of the Fifth International Conference on the Principles and Practice of Constraint Programming, CP'99*, pages 446–460, 1999.

[20] D. Tompkins and H. Hoos. Warped landscapes and random acts of SAT solving. In *Proceedings of the 8th International Symposium on Artificial Intelligence and Mathematics, AIMA-04*, 2004.

[21] Z. Wu and B. Wah. 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)*, pages 310–315, 2000.