

Using Cost Distributions to Guide Weight Decay in Local Search for SAT

John Thornton and Duc Nghia Pham

SAFE Program, Queensland Research Lab, NICTA and
Institute for Integrated and Intelligent Systems, Griffith University, QLD, Australia
{john.thornton,duc-nghia.pham}@nicta.com.au

Abstract. Although clause weighting local search algorithms have produced some of the best results on a range of challenging satisfiability (SAT) benchmarks, this performance is dependent on the careful hand-tuning of sensitive parameters. When such hand-tuning is not possible, clause weighting algorithms are generally outperformed by self-tuning WalkSAT-based algorithms such as AdaptNovelty⁺ and AdaptG²WSAT. In this paper we investigate tuning the weight decay parameter of two clause weighting algorithms using the statistical properties of cost distributions that are dynamically accumulated as the search progresses. This method selects a parameter setting both according to the speed of descent in the cost space and according to the shape of the accumulated cost distribution, where we take the shape to be a predictor of future performance. In a wide ranging empirical study we show that this automated approach to parameter tuning can outperform the default settings for two state-of-the-art algorithms that employ clause weighting (PAWS and gNovelty⁺). We also show that these self-tuning algorithms are competitive with three of the best-known self-tuning SAT local search techniques: RSAPS, AdaptNovelty⁺ and AdaptG²WSAT.

1 Introduction

One way to categorize the currently best performing satisfiability (SAT) local search algorithms is according to the method used to escape local minima. Firstly, there are those approaches that use randomized decision strategies, such as the WalkSAT family of algorithms [1] and the more recent G²WSAT algorithms [2]. Secondly, there are those that use weights to penalize local minima features, such as DLM [3], SAPS [4], GLSSAT [5], and PAWS [6]. To date, clause weighting algorithms have outperformed WalkSAT on many of the standard benchmark problems, but only when careful parameter tuning is allowed. Conversely, WalkSAT-based algorithms have consistently dominated the recent SAT competitions¹, where the hand-tuning of parameters is not possible because the details of the competition problems are not known in advance. This restriction more accurately reflects real-world situations where an answer is required

¹ <http://www.satcompetition.org/>

as quickly as possible, rather than needing to know how quickly we *could* have found an answer if we had known the optimal parameter settings in advance.

One of the main reasons for the success of WalkSAT algorithms is that their performance is primarily influenced by the value of a single *noise* parameter (noise controls the degree of randomness in each flip decision). While the best setting for this parameter varies widely from problem to problem, it can be effectively adapted during the search using a simple heuristic that measures the degree of search stagnation [7]. A similar heuristic was developed for the SAPS clause weighting algorithm [4] but this remains uncompetitive with the best WalkSAT techniques [8]

Of the other current clause weighting algorithms, the pure additive weighting scheme (PAWS) is probably the best candidate for the development of a parameter adapting heuristic because its performance depends on a single *MaxInc* parameter which controls the rate of decay of the clause weights [6]. However, despite considerable effort, no effective online method for adapting *MaxInc* has been discovered. More recently, the gNovelty⁺ algorithm has combined a WalkSAT-based heuristic with a clause weighting mechanism to produce the 2007 SAT competition random satisfiable category winner [8]. gNovelty⁺ uses the WalkSAT heuristic to adapt noise and has a second parameter to control the rate of clause weight decay. Although it is known that gNovelty⁺'s performance is affected by the setting of this second parameter, to date no method has been proposed to adapt it automatically.

In this paper we investigate an online method to automatically adapt clause weight decay using information extracted from distributions of false clause counts recorded at each flip. By accumulating distributions at various parameter settings, we can predict the best setting as the search continues. We have applied this method to both PAWS and gNovelty⁺ in order to improve their average case performance in comparison to the standard default weight decay settings.

In the remainder of the paper we provide more detail on existing approaches to parameter tuning and then provide an in-depth description of our new approach and how it has been incorporated into PAWS and gNovelty⁺. Using an empirical study, we compare the performance of the new approach against PAWS and gNovelty⁺, and against AdaptG²WSAT, AdaptNovelty⁺ and RSAPS. Finally we discuss the results and present our conclusions.

2 Parameter Tuning and Performance Prediction

The literature on predicting algorithm performance can be divided along several axes depending on whether the prediction is based on: (i) an off-line training phase (e.g. [9]) or purely on feedback obtained while solving online instances (e.g. [10])(ii) measuring problem features (e.g. [11]) or on measuring an algorithm's past runtime performance (e.g. [12]) (iii) deciding between a portfolio of algorithms (e.g. [13]) or determining the parameter settings of a single algorithm (e.g. [14]) (iv) predicting performance on a per instance basis (e.g. [14]) or on

a per distribution basis (e.g. [11]) (v) using a high or low complexity prediction model (giving rise to so-called “low knowledge” approaches [15]).

While these distinctions cover a broad range of potential methods, there is considerable overlap across axes and between the kinds of machine learning technique that are effective. In relation to the current research, we are interested in using a “low knowledge” approach based on online feedback about runtime performance to predict parameter settings. Our aim is to improve the average performance of a candidate parametrized SAT algorithm in situations where we are only allowed a single run on a problem instance and where the problem characteristics are not known in advance.

Online SAT Implementations: The best known *online* self-tuning local search SAT algorithms have not explicitly predicted performance, but instead have exploited measures of search stagnation. For example, AdaptNovelty⁺ (the self-tuning version of Novelty⁺) adapts its noise parameter according to whether an improvement is observed in the overall best cost after a fixed number of flips, i.e.: if no improvement occurs, the value of the noise parameter is increased, thereby increasing the probability that non-greedy moves are accepted; otherwise, if a new minimum cost solution is found, then the noise value is immediately decreased. Hoos [7] demonstrated experimentally that this adaptive mechanism is effective both with Novelty⁺ and other WalkSAT variants. The same basic mechanism was also used to adapt the probability that clause weights will be multiplicatively reduced in the RSAPS algorithm [4] and, in earlier work, stagnation measures were used in reactive tabu search [16]. Again, referring to the SAT 2007 competition, the best individual local search algorithms (gNovelty⁺ and AdaptG²WSAT) both employ the AdaptNovelty⁺ self-tuning mechanism - making this the state-of-the-art for online adaptation (at least within the SAT local search community). However, in relation to the current research, stagnation measures have not proved effective for tuning the clause weight decay parameter of any clause weighting algorithm except RSAPS (and RSAPS is known to be uncompetitive with gNovelty⁺ or AdaptG²WSAT [8]).

Local Search Invariants: In their influential paper, McAllester et al. [17] reported an invariant statistical relationship in the cost distributions for a range of SAT algorithms on a selection of planning, graph colouring and hard random 3-SAT problems. Here, and in the rest of the paper, we define a *local search cost distribution* to be the distribution of the count of false clauses recorded at each flip during a sequence of local search steps. McAllester et al.’s invariant relationship was calculated as the mean divided by the standard deviation of the local search cost distribution recorded over a large number of runs for a particular algorithm, on both single instances and on groups of similar instances. We term this measure the *range* statistic, where the range specifies the distance of the mean search cost from the origin in standard deviation units. McAllester et al. found that the optimal setting for the noise parameters they were investigating consistently occurred at a value 10% greater than the noise value that minimized the range measure. They consequently conjectured that range could be an effec-

tive online *and* off-line performance predictor for tuning local search parameters.

“Low Knowledge” Algorithm Control: Outside the SAT domain there are several other approaches that attempt to predict performance on the basis of search behaviour (e.g. [10, 12]). The most interesting of these for current purposes is “low knowledge” algorithm control which uses reinforcement learning to dynamically allocate runtime slices to different algorithms as the search progresses. Each algorithm has a weight that is updated after a given number of iterations according to a reinforcement learning formula that takes the cost improvement per second as input, i.e. the faster an algorithm is descending in the cost space, the greater the increase in its weight and the larger the time slice it is allowed in the next series of iterations. In empirical tests, this method was able to exceed the average performance of the pure algorithms on which it was based. The “low knowledge” approach shows that an online application that only examines the current best cost can effectively allocate time slices between competing algorithms. The main differences between this work and our own, are that we need to decide between different parameter settings rather than different algorithms, and that we cannot accumulate knowledge between runs on different instances.

3 Tuning PAWS Online

The preceding analysis has yielded two promising avenues for further investigation: i) exploiting McAllester et al.’s range statistic as an online guide to parameter performance and ii) using the “low knowledge” approach of dividing online runtime resources according to the speed of descent of different parameter settings in the cost space. The challenge is that both these approaches have previously required multiple runs on the same problem or on distributions of similar problems before they can act as reliable guides. If we limit ourselves to looking at a single run, the stochastic nature of local search means we get little better than vague hints of which direction to move. In addition, the problem of tuning the PAWS *MaxInc* parameter is complicated by the large range of possible values (from 4 to 500) and the sensitivity of the parameter to small changes (for example, see [18]). To counteract these issues we developed two strategies. Firstly, we looked at changing the way that PAWS reduces weight to create a more robust parameter with a smaller range of possible values. And secondly, we broadened our view of the cost data available during each problem run to include the *shape* of the local search cost distribution.

The *MaxThres* Parameter: The effect of periodically reducing clause weights is to reduce the total number of clauses that have weight. An analysis of the runtime behaviour of PAWS on individual problems shows that each *MaxInc* setting yields a fairly stable mean number of false clauses. We therefore experimented with reducing weight whenever the number of false clauses exceeds a given threshold. This produced a new *MaxThres* parameter that exhibited sim-

ilar performance to *MaxInc* except that it proved more robust to small changes in its value.

The operation of *MaxThres* is detailed in the pseudocode of the UpdateClauseWeights function in Algorithm 1. This function is called whenever PAWS decides it has reached a local minimum and differs from the original PAWS only at lines 5 and 6. Previously, PAWS reduced weight at line 5 if *IncCounter* > *MaxInc* and omitted the while loop of line 6. Now the *MaxThres* parameter causes weight to be reduced when the number of weighted clauses ($|\mathcal{W}|$) and the number of false clauses ($|\mathcal{F}|$) both exceed *MaxThres* and only after at least *MinInc* consecutive weight increase phases have been completed (*MinInc* is fixed at 3). In addition, the while loop at line 6 ensures that each weight reduction phase reduces $|\mathcal{W}|$ to a value less than *MaxThres* (this step becomes necessary when evaluating the performance of different *MaxThres* values during the same run).

The new *MaxThres* parameter alters the behaviour of PAWS by tending to reduce weight relatively less frequently when there are fewer false clauses and relatively more frequently when there are more false clauses. This change produced small differences in performance in comparison with the original PAWS, but on the average the two approaches proved very similar. The main advantage of *MaxThres* is that we can (on average) obtain equivalent performance with the original PAWS while also reducing the set of parameter values from $\{4\ 5\ 6\ 7\ 8\ 9\ 10\ 11\ 12\ 13\ 15\ 20\ 25\ 30\ 35\ 40\ 50\ 75\ 100\ 125\ 250\ \infty\}$ for *MaxInc* to $\{25\ 50\ 75\ 125\ 250\ 500\ 750\ \infty\}$ for *MaxThres*.

Algorithm 1: UpdateClauseWeights

Input: $\mathcal{F} \leftarrow$ the set of currently false clauses; $\mathcal{W} \leftarrow$ the set of currently weighted clauses;
Output: updated membership of \mathcal{W} ; updated clause weights for $\mathcal{F} \cup \mathcal{W}$;

```

1 for each  $c_i \in \mathcal{F}$  do
2   Weight( $c_i$ )  $\leftarrow$  Weight( $c_i$ ) + 1;
3   if Weight( $c_i$ ) = 2 then  $\mathcal{W} \leftarrow \mathcal{W} \cup c_i$ ;
4 IncCounter  $\leftarrow$  IncCounter + 1;
5 if  $|\mathcal{W}| > \text{MaxThres}$  and  $|\mathcal{F}| > \text{MaxThres}$  and IncCounter > MinInc then
6   while  $|\mathcal{W}| > \text{MaxThres}$  do
7     for each  $c_i \in \mathcal{W}$  do
8       Weight( $c_i$ )  $\leftarrow$  Weight( $c_i$ ) - 1;
9       if Weight( $c_i$ ) = 1 then  $\mathcal{W} \leftarrow \mathcal{W} - c_i$ ;
10  IncCounter  $\leftarrow$  0;
```

Local Search Cost Distribution Shape: Given a single run, the information available to select a parameter setting is scarce and highly variable. Our first approach to ameliorate this situation was to set up a version of PAWS that progressively accumulates local search cost distributions for each parameter setting and allows the user to change parameter settings and graphically visualize the different cost distributions. From these observations it became clearer what *shape* of cost distribution was most often associated with the best parameter setting. The general rule of thumb is: select the distribution with the smallest mean, given the distribution has a roughly normal shape. As a result of extensive preliminary experimentation, we decided to use skewness and kurtosis statistics

as an additional guide for parameter setting. Skewness measures the degree of symmetry of a distribution (where a zero value indicates perfect symmetry) and is calculated as follows:

$$\frac{1}{n} \sum_{i=1}^n \left(\frac{x_i - \bar{x}}{\sigma} \right)^3$$

In the case of measuring the skewness of a local search cost distribution for a particular *MaxThres* value, n would be the number of flips taken at the selected *MaxThres* value, x_i the number of false clauses observed at flip i , and \bar{x} and σ the mean and standard deviation respectively of the distribution of x_i 's.

Kurtosis measures the degree of "peakedness" of a distribution, where a higher value indicates a sharper peak with flatter tails (in comparison to a standard normal distribution). We calculated kurtosis as follows:

$$\frac{1}{n} \sum_{i=1}^n \left(\frac{x_i - \bar{x}}{\sigma} \right)^4 - 3$$

Simulated Annealing: Having identified a few promising measures, we required a method to control the parameter value selection process during the lifetime of a single run. Inspired by the "low knowledge" approach, we used two interleaved searches on the same problem, one with a high *MaxThres* setting (750) and the other with a good low default setting (75), as follows: each search starts with its own copy of the same problem initialisation, and then pursues its own separate search trajectory; the two search procedures then compete for processor time according to a simulated annealing (SA) [19] schedule shown in Algorithm 2.

Algorithm 2: DecideUpperOrLowerSetting

```

Input: lowerThres ← lower MaxThres setting; upperThres ← upper MaxThres setting;
temp ← 1024; step ← 400; tempStep ← initial steps allocated to upper setting;
upperStep ← current steps allocated to upper setting;
lowerStep ← current steps allocated to lower setting;
1 if lowerStep < upperStep then tempStep ← tempStep + lowerStep;
2 else tempStep ← tempStep + upperStep;
3 while tempStep ≥ step do
4   | temp ← temp ÷ 2;
5   | step ← step × 2;
6 cost ← CostDifference(lowerThres, upperThres);
7 diff ← AbsoluteValue(cost);
8 uphillProb ← 50e-(diff/temp);
9 if probability ≤ uphillProb then
10  | if cost ≥ 0 then return lowerThres; else return upperThres;
11 else
12  | if cost ≤ 0 then return lowerThres; else return upperThres;

```

Here, SA is used to control a decision model that begins by randomly allocating time slices to the two search procedures and then, as the temperature decreases, biases decisions more and more towards respecting the CostDifference measure defined in Algorithm 3. This measure quantifies our notion of local search cost distribution shape. An important point to note here is that all statistics for each distribution (i.e. the mean, standard deviation, skewness and kurtosis) are reset each time the distribution reaches a solution that improves on the previously best minimum cost (for that distribution). This eliminates

the initial high variance phase of the search and avoids the distorting effects of outlying cost values. In addition, we ignore the sign of the skewness and kurtosis measures, taking their absolute value only (see *AbsSkew* and *AbsKurt* in Algorithm 3).

Algorithm 3: CostDifference(*thres1*, *thres2*)

```

1 minCostRatio ← 10 × (MinCost(thres1) ÷ (MinCost(thres1) + MinCost(thres2)));
2 rangeRatio ← 10 × (Range(thres1) ÷ (Range(thres1) + Range(thres2)));
3 skewRatio ← 10 × (AbsSkew(thres1) ÷ (AbsSkew(thres1) + AbsSkew(thres2)));
4 kurtRatio ← 10 × (AbsKurt(thres1) ÷ (AbsKurt(thres1) + AbsKurt(thres2)));
5 return 100 − ((9 × rangeRatio) + (7 × minCostRatio) + (2 × (skewRatio + kurtRatio)));
```

The DecideUpperOrLowerSetting procedure controls the PAWS *MaxThres* setting for the first 50,000 flips of the combined search trajectories. During this phase, the new PAWS will behave much like its predecessor (with *MaxInc* set to 10), except that it will “waste” a certain number of flips exploring the non-optimal distribution. Such exploration will help if the best setting is in the upper distribution, but otherwise it will degrade the relative performance.

Binary Search: After the 50,000 flip threshold, both the upper and lower search trajectories are allowed to explore other *MaxThres* settings within a lower range of {25 50 75 125} and an upper range of {250 500 750 ∞}. This procedure takes the form of a binary search, such that after every search step of 100 flips (where the value of *MaxThres* remains fixed) the DecideUpperOrLowerSetting function determines which half of the parameter space will be used next. Then we use the DecideSetting and FindBestCost functions to further subdivide the parameter space into a single setting. For example, if DecideUpperOrLowerSetting selects lower, then we will call:

DecideSetting(FindBestCost(25, 50), FindBestCost(75, 125))

Otherwise we will call:

DecideSetting(FindBestCost(250, 500), FindBestCost(750, ∞))

The DecideSetting function follows the simulated annealing approach of DecideUpperOrLowerSetting with two changes to reflect the finer grain of the decision. Firstly, the annealing function has a consistently higher probability of returning an uphill move, replacing line 8 of Algorithm 2 with:

$$uphillProb \leftarrow 30e^{-\left(\frac{diff}{temp}\right)} + 20$$

Secondly, the annealing schedule is only reduced according to the number of steps taken since the last minimum cost was discovered for each distribution, replacing lines 1-2 from Algorithm 2 with:

```

if (lowerStep < upperStep) then tempStep ← lowerStep;
else tempStep ← upperStep;
```

Finally, we limit the parameter search space on problems with more than 50,000 clauses to only consider 25 or 50 in the lower distribution (the upper distribution parameter range remains unchanged). This reflects empirical observations showing that larger problems tend to have smaller optimal *MaxThres* settings.

4 Experimental Study

To test the new self-tuning version of PAWS (which we term iPAWS), we selected a range of *random* problems from the SAT competition satisfiable benchmarks and a range of *structured* problems from the SATLIB library. The SAT competition problems are to give an idea of potential performance in the competition’s satisfiable random category (this is the category where SAT local search algorithms have consistently outperformed complete search techniques), while the other problems allow comparison with the existing SAT literature. Firstly, to form the SATLIB structured benchmark, we took two “flat” graph colouring problems (flat200-median and hard²), two blocksworld planning problems (bw_large.c and d), two logistics planning problems (logistics.c and d), three all interval series problems (ais10, 12 and 14), five hard quasigroup problems (qg1-08, qg2-08, qg5-11, qg6-09, qg7-13), five 16-bit parity learning problems (par16-1-c to par16-5-c), four large graph colouring problems (g125.17, g125.18, g250.15, g250.29), four circuit synthesis formula problems (2bitadd_11, 2bitadd_12, 3bitadd_31, 3bitadd_32) and four Beijing scheduling problems (enddr2-1, enddr2-8, ewddr2-1 and ewddr2-8). Secondly, to form the random benchmark set, we randomly selected 12 k3-SAT instances, 12 k5-SAT instances and 10 k7-SAT instances from the large satisfiable random problems used in the 2005 SAT competition.

To obtain a measure of the improvement in iPAWS over the original PAWS we included a manually tuned PAWS (PAWS(t)) with *MaxInc* optimized for each problem category and a default-valued PAWS (PAWS(d)) with *MaxInc* fixed at 10. We also included the best-known self-tuning SAT local search algorithms: AdaptNovelty⁺ [7], AdaptG²WSAT [2] and RSAPS [4]. Both AdaptNovelty⁺ and AdaptG²WSAT are included for their class leading performance in the recent SAT competitions and RSAPS is included to compare iPAWS with another clause weighting adaptive algorithm. Finally, we included gNovelty⁺, the winner of the random satisfiable category of the 2007 SAT competition, and arguably the best general purpose SAT local search solver currently available [8].

iNovelty⁺: As discussed in the introduction, gNovelty⁺ also performs clause weighting and has a parameter that can control the rate of clause weight decay. Experimental observations have shown that gNovelty⁺’s performance can be significantly enhanced by treating this parameter as a binary switch that either allows weight to accumulate without reduction, or turns off clause weighting altogether [8]. To investigate the applicability of the iPAWS weight decay heuristic to other clause weighting algorithms, we decided to partially implement the iPAWS tuning process into gNovelty⁺. This involved controlling the binary decision about whether or not to accumulate weight in gNovelty⁺ in the same way that iPAWS decides whether to use an upper or lower setting of *MaxThres*. More specifically, gNovelty⁺ was adapted to match iPAWS so that it runs two separate searches on same problem starting point, one with weight accumulation

² We take all designations of median and hard problems from [4]

turned on and the other with it turned off. Each search then competes for processor time using the `DecideUpperOrLowerSetting` procedure defined in Algorithm 2. We term this new weight-tuning version of `gNovelty+` as `iNovelty+`.

4.1 Results

To obtain an overall measure of performance, we adapted the SAT competition scoring metric (see <http://www.satcompetition.org/2007/rules07.html>) to suit the situation of our allowing each algorithm 100 runs on each instance with a 600 second cutoff for each run. This required us to divide the SAT competition *solution purse* up in proportion to the number of successful runs for solver s on problem p , as follows:

$$\text{solutionAward}(p, s) \leftarrow \frac{1000 \times \text{successCount}(p, s)}{\sum_{i=1}^n \text{successCount}(p, i)}$$

where `successCount` counts the number of runs where a solution has been found within the standard timeout of 600 seconds. Similarly, the SAT competition *speed purse* is divided as follows:

$$\text{speedAward}(p, s) = \frac{1000 \times \text{speedFactor}(p, s)}{\sum_{i=1}^n \text{speedFactor}(p, i)}$$

where `speedFactor`(p, s) calculates the sum, in seconds, of: $600 \div (1 + \text{solution time})$ for each successful run of algorithm s on problem p . Finally, the SAT competition *series purse* allocates 3000 points for a problem series containing 5 or more problems (i.e. the parity, quasigroup, k3-SAT, k5-SAT and k7-SAT series), otherwise it allocates 1000 points. Here the points are divided equally amongst all algorithms that can find any solution to any problem within a given series. The final measure for each solver is then calculated as the sum of the three purses.

Structured Benchmarks: Table 1 shows the scores for the structured benchmark problems using the SAT competition measure, both overall and on a per series basis. These results clearly show the `gNovelty+` variants outperform the other solvers regardless of the parameter tuning methods employed. Within the `gNovelty+` solvers, `iNovelty+` produced a slight improvement over `gNovelty+(d)` but this almost entirely rests on `iNovelty+`'s ability to beat the default algorithm on the parity series. There are other signs of improvement on the `bw_large` and large graph colouring problems, but on the remaining problems the improvements have not outweighed the overhead of performing two interleaved searches.

The `iPAWS` results show that the self-tuning heuristic has produced a more sizable improvement in comparison to the default `PAWS(d)`, raising the overall score from 4,656 to 7,940. In particular, `iPAWS` has performed better than `PAWS(d)` on all the structured series except `bw_large` (where performance is still similar), having the largest improvement on the parity problems. Comparing `iPAWS` with the optimally tuned `PAWS(t)` shows that `iPAWS` sometimes gains an advantage from being able to adapt to individual problems (particularly in the `bitadd` and `qg` series). Overall, however, the prior tuning of `PAWS(t)` outperforms `iPAWS`, particularly on the `bw_large` problems.

Looking at `PAWS` in comparison with `AdaptG2WSAT`, `AdaptNovelty` and `RSAPS` shows that the improvement on `PAWS(d)` is enough to move `PAWS`

from being the worst performing solver, to being the best (excepting gNovelty⁺). While the difference between iPAWS and AdaptG²WSAT is slight, the relative effect of the iPAWS self-tuning method is impressive, especially considering the extra effort needed to perform an interleaved search on two problem instantiations, of which one is necessarily exploring the worse half of the parameter space.

Solver	bitadd	ais	bw_large	e*ddr	flat	g*	logistics	par16	qg	Total
gNovelty ⁺ (t)	1,477.2	916.1	820.7	1,333.9	594.7	1,199.4	571.9	1,855.3	2,911.5	11,680.6
gNovelty ⁺ (d)	1,477.2	916.1	609.1	1,333.9	594.7	875.4	571.9	466.6	2,447.9	9,292.7
iNovelty ⁺	1,440.0	830.3	677.8	1,156.0	530.2	1,072.7	566.4	1,296.4	1,941.0	9,510.7
PAWS (t)	380.1	822.5	835.8	1,166.9	569.2	1,298.2	569.2	2,685.4	805.6	9,132.8
PAWS (d)	380.1	694.8	324.4	377.5	528.2	578.7	535.3	407.4	829.7	4,656.1
iPAWS	640.2	824.3	310.8	971.9	537.1	1,021.8	561.9	2,011.2	1,060.6	7,940.0
AdaptG ² WSAT0	1,283.5	631.6	546.6	787.6	576.8	1,252.3	496.6	1,529.4	749.4	7,853.7
AdaptNovelty ⁺	1,348.9	505.1	491.7	708.5	507.7	1,216.1	555.9	1,392.4	743.3	7,469.6
RSAPS	572.8	859.3	383.2	1,163.9	561.4	485.4	570.8	1,355.9	1,511.0	7,463.8

Table 1. Scoring of solvers’ performance on structured problems

Series	gNovelty ⁺	PAWS (d)	iPAWS	AdaptG ² WSAT0	AdaptNovelty ⁺	RSAPS
k3-SAT	7,468.6	11,364.3	1,476.5	2,618.7	4,071.9	0.0
k5-SAT	5,892.8	742.3	7,052.4	7,174.3	5,633.6	504.6
k7-SAT	4,424.2	2,966.6	3,600.6	4,973.8	5,335.8	1,699.0
Total	17,785.7	15,073.1	12,129.5	14,766.9	15,041.3	2,203.6

Table 2. Scoring of solvers’ performance on random problems

Random Benchmarks: Table 2 shows the results for the random benchmark problems (again using the SAT competition metric). Here we have reproduced the parameter settings for PAWS ($MaxInc = 10$) and gNovelty⁺ that were used in the original SAT competitions. In this case, gNovelty⁺ used a heuristic that sets its clause weight decay parameter to 1.0 for all 5-SAT and 7-SAT problems (turning clause weighting off) and to 0.4 for all 3-SAT problems (reducing weight after an increase with probability 0.4) [8]. This heuristic means gNovelty⁺ is no longer using a fixed parameter value for these problems and makes a direct comparison with iNovelty⁺ unfair. We therefore propose that iNovelty⁺ use the gNovelty⁺ heuristic whenever it detects uniform 3, 5, and 7-SAT problems (making it equivalent to gNovelty⁺ on these problems).

The results again show gNovelty⁺ outperforming all other solvers by a sizeable margin, with PAWS(d) and AdaptNovelty⁺ coming in a close second and third respectively, followed by AdaptG²WSAT, iPAWS and finally RSAPS (after a large gap). Looking in more detail, we can see the relatively poor performance of iPAWS is due to the 3-SAT problems, otherwise it outperforms PAWS(d) on both the 5-SAT and 7-SAT series. If we allow the gNovelty⁺ heuristic to be legitimate, then a similar heuristic applied to iPAWS could switch it to perform a default PAWS search on all 3-SAT problems. In this case, iPAWS would defeat both PAWS(d) and gNovelty⁺ on the random benchmarks.

5 Discussion and Conclusions

The primary aim of this paper was to develop an effective online method to tune the PAWS weight decay parameter within single runs on single problems. The secondary aim was to explore the use of this method within another clause weighting algorithm. The results have shown the new iPAWS heuristic to be effective across a range of structured problems and across two of the three classes of uniform random SAT problems. We have also proposed a simple heuristic to improve iPAWS on the 3-SAT benchmarks.

However, our attempt to extend the iPAWS approach to gNovelty⁺ did not produce such dramatic improvements. We propose two reasons for this. Firstly, gNovelty⁺ already uses an adaptive online mechanism to tune the Novelty noise parameter. It may be that the two adaptive mechanisms do not interact to good effect. A promising area of future research would therefore be to use an iPAWS approach to simultaneously tune the gNovelty⁺ noise and weight decay parameters. This also suggests using the local search cost distributions to tune the noise parameters of other WalkSAT algorithms, to test if this would be more effective than using the current stagnation measures. Secondly, the relatively small improvement of iNovelty⁺ over gNovelty⁺(d) could also be explained by iNovelty⁺ not having the advantage of being able to do a more refined search of the parameter space. As the gNovelty⁺ weight decay parameter has so far proved relatively insensitive to intermediary settings between 0 and 1 - with the exception of preferring a 0.4 setting on 3-SAT problems and a 0.1 setting on the parity 16 problems - this suggests the gNovelty⁺ parameter may not be suitable for an iPAWS-type heuristic.

Overall, the iPAWS heuristic is complex (e.g. in comparison to AdaptNovelty) and relies on a large number of hand-tuned (but robust) settings. This would argue against it if there were an effective, simpler heuristic available. However, after extensive exploration, we were unable to find a more compact combination of measures that correlated well with an optimal weight decay setting *and* were reliable across a wide range of problem types. Also, despite the complexity of the implementation, the underlying principles remain quite simple, i.e. we use the statistical properties of local search cost distributions, accumulated for different parameter settings, to bias future parameter selection decisions, according to a simulated annealing schedule. Nevertheless, it would be worthwhile to search for a more principled way to fix (or remove) the various settings on which the algorithm depends - most obviously by using existing machine learning approaches (e.g. as in [14]).

In conclusion, the paper has introduced a new approach to tuning local search parameters online. The initial implementation has been the product of considerable trial and error and should not be considered definitive. Rather, it is intended to show that the underlying concept is workable and to act as a foundation for further investigation. Nevertheless, the initial results are encouraging, and the new iPAWS algorithm has been shown to be competitive with a range of the best known local search SAT solvers.

Acknowledgements

We thankfully acknowledge the financial support from NICTA and the Queensland Government. NICTA is funded by the Australian Government as represented by the Department of Broadband, Communications and Digital Economy and the Australian Research Council through the ICT Centre of Excellence Program.

References

1. Selman, B., Levesque, H., Mitchell, D.: A new method for solving hard satisfiability problems. In: Proceedings of AAAI-92. (1992) 440–446
2. Li, C.M., Huang, W.Q.: Diversification and determinism in local search for satisfiability. In: Proceedings of SAT-05. (2005) 158–172
3. Wu, Z., Wah, B.: An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In: Proceedings of AAAI-00. (2000) 310–315
4. Hutter, F., Tompkins, D., Hoos, H.H.: Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In: Proceedings of CP-02. (2002) 233–248
5. Mills, P., Tsang, E.: Guided local search applied to the satisfiability (SAT) problem. In: Proceedings of ASOR'99. (1999) 872–883
6. Thornton, J.R., Pham, D.N., Bain, S., Ferreira Jr., V.: Additive versus multiplicative clause weighting for SAT. In: Proceedings of AAAI-04. (2004) 191–196
7. Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Proceedings of AAAI-02. (2002) 635–660
8. Pham, D.N., Thornton, J.R., Gretton, C., Sattar, A.: Advances in local search for satisfiability. In: Proceedings of Australian AI-07. (2007) 213–222
9. Xu, L., Hutter, F., Hoos, H.H., Leyton-Brown, K.: The design and analysis of an algorithm portfolio for SAT. In: Proceedings of CP-07. (2007) 712–727
10. Gagliolo, M., Schmidhuber, J.: Dynamic algorithm portfolios. In: Proceedings of AI-MATH-06. (2006)
11. Hutter, F., Hoos, H.H., Stützle, T.: Automatic algorithm configuration based on local search. In: Proceedings of AAAI-07. (2007) 1152–1157
12. Birattari, M., Stützle, T., Paquete, L., Varrentrapp, K.: A racing algorithm for configuring metaheuristics. In: Proceedings of GECCO-02. (2002) 11–18
13. Gomes, C.P., Selman, B.: Algorithm portfolios. *Artificial Intelligence* **126** (2001) 43–62
14. Hutter, F., Hamadi, Y., Hoos, H.H., Leyton-Brown, K.: Performance prediction and automated tuning of randomized and parametric algorithms. In: Proceedings of CP-06. (2006) 213–228
15. Carchrae, T., Beck, J.C.: Low-knowledge algorithm control. In: Proceedings of AAAI-04. (2004) 49–54
16. Battiti, R., Protasi, M.: Reactive search, a history-sensitive heuristic for MAX-SAT. *ACM Journal of Experimental Algorithmics* **2**(Article 2) (1997)
17. McAllester, D.A., Selman, B., Kautz, H.A.: Evidence for invariants in local search. In: Proceedings of AAAI-97. (1997) 321–326
18. Thornton, J.: Clause weighting local search for SAT. *Journal of Automated Reasoning* **35**(1-3) (2005) 97–142
19. Kirkpatrick, S., Gelatt, C.D., Vecchi, M.P.: Optimization by simulated annealing. *Science* **220**(4598) (1983) 671–680