Neighbourhood Clause Weight Redistribution in Local Search for SAT

Abdelraouf Ishtaiwi, John Thornton, Abdul Sattar and Duc Nghia Pham

Institute for Integrated and Intelligent Systems
email: {a.ishtaiwi, j.thornton, a.sattar, d.n.pham}@griffith.edu.au

Abstract. In recent years, dynamic local search (DLS) clause weighting algorithms have emerged as the local search state-of-the-art for solving propositional satisfiability problems. This paper introduces a new approach to clause weighting, known as Divide and Distribute Fixed Weights (DDFW), that transfers weights from neighbouring satisfied clauses to unsatisfied clauses in order to break out from local minima. Unlike earlier approaches, DDFW continuously redistributes a fixed quantity of weight between clauses, and so does not require a weight smoothing heuristic to control weight growth. It also exploits inherent problem structure by redistributing weights between neighbouring clauses.
To evaluate our ideas, we compared DDFW with two of the best reactive local search algorithms, AdaptNovelty+ and RSAPS. In both these algorithms, a problem sensitive parameter is automatically adjusted during the search, whereas DDFW uses a fixed default parameter. Our empirical results show that DDFW has consistently better performance over a range of SAT benchmark problems. This gives a strong indication that neighbourhood weight redistribution strategies could be the key to a next generation of structure exploiting, parameter-free local search SAT solvers.

1 Introduction

The propositional satisfiability (SAT) problem is at the core of many computer science and artificial intelligence problems. Hence, finding efficient solutions for SAT has far reaching implications. In this study, we consider propositional formulae in conjunctive normal form (CNF): $\mathcal{F} = \bigwedge_m \bigvee_n l_{mn}$ in which each l_{mn} is a literal (propositional variable or its negation), and each disjunct $\bigvee_n l_{mn}$ is a clause. The problem is to find an assignment that satisfies $\mathcal{F}$. Given that SAT is NP complete, systematic search methods can only solve problems of limited size. On the other hand, relatively simple stochastic local search (SLS) methods have proved successful on a wide range of larger and more challenging problems [1].

Since the development of the Breakout heuristic [2], clause weighting dynamic local search (DLS) algorithms have been intensively investigated, and continually improved [3, 4]. However, the performance of these algorithms remained inferior to their non-weighting counterparts [5], until the more recent development of weight smoothing heuristics [6–9]), which currently represent the state-of-the-art for SLS methods on SAT problems. Interestingly, the two best performing DLS algorithms (SAPS [8] and PAWS [9]) have converged on the same underlying weighting strategy: increasing weights on

false clauses in a local minimum, then periodically reducing weights according to a problem specific parameter setting. PAWS mainly differs from SAPS in performing additive rather than multiplicative weight updates. A key weakness of these approaches is that their performance depends on problem specific parameter tuning. This issue was partly in the development of a reactive version of SAPS (RSAPS [8]) which used the same adaptive noise mechanism developed in AdaptNovelty+ [10].

The question addressed in the current study is whether there are alternative weighting schemes that can produce further performance gains in the SAT domain. In particular, we are interested in *weight redistribution* schemes, that move around a fixed quantity of weight between clauses. Such an approach offers the advantage of not explicitly reducing weights, thereby avoiding considerable computational overhead, and the need for a problem specific weight reduction parameter. Secondly, we are interested in exploiting structural information contained in the weight distributions between neighbouring clauses. As adding weight to a clause can only immediately affect those clauses with which it shares a variable, it appears promising to connect weighting decisions with the relative level of weight on neighbouring clauses. We combine both weight redistribution and consideration of neighbourhood relationships in the Divide and Distribute Fixed Weights (DDFW) algorithm, which implements weight redistribution between neighbouring clauses.

In the remainder of the paper we introduce DDFW in more detail, and provide an empirical comparison between DDFW, RSAPS and AdaptNovelty+.

2 Divide and Distribute Fixed Weights

DDFW introduces two new ideas into the area of clause weighting algorithms for SAT. Firstly, it evenly distributes a fixed quantity of weight across all clauses at the start of the search, and then escapes local minima by *transferring weight from satisfied to unsatisfied clauses*. The existing state-of-the-art clause weighting algorithms have all divided the weighting process into two distinct steps: i) increasing weights on false clauses in local minima and ii) decreasing or normalising weights on all clauses after a series of increases, so that weight growth does not spiral out of control. DDFW combines this process into a single step of weight transfer, thereby dispensing with the need to decide when to reduce or normalise weight. In this respect, DDFW is similar to the predecessors of SAPS (SDF [7] and ESG [11]), which both adjust *and* normalise the weight distribution in each local minimum. Because these methods adjust weight across all clauses, they are considerably less efficient than SAPS, which normalises weight after visiting a series of local minima.[1] DDFW escapes the inefficiencies of SDF and ESG by only transferring weights between pairs of clauses, rather than normalising weight on all clauses. This transfer involves selecting a single satisfied clause for each currently unsatisfied clause in a local minimum, reducing the weight on the satisfied clause by an integer amount and adding that amount to the weight on the unsatisfied clause. Hence DDFW retains the additive (integer) weighting approach of DLM [6] and PAWS, and

[1] Increasing weight on *false* clauses in a local minimum is efficient because only a small proportion of the total clauses will be false at any one time.

combines this with an efficient method of weight redistribution, i.e. one that keeps all weight reasonably normalised without repeatedly adjusting weights on all clauses.

The second and more original idea developed in DDFW, is the exploitation of neighbourhood relationships between clauses when deciding which pairs of clauses will exchange weight. We term clause c_i to be a neighbour of clause c_j, if there exists at least one literal $l_{im} \in c_i$ and a second literal $l_{jn} \in c_j$ such that $l_{im} = l_{jn}$. Furthermore, we term c_i to be a *same sign* neighbour of c_j if the sign of any $l_{im} \in c_i$ is equal to the sign of any $l_{jn} \in c_j$ where $l_{im} = l_{jn}$. From this it follows that each literal $l_{im} \in c_i$ will have a set of same sign neighbouring clauses $C_{l_{im}}$. Now, if c_i is false, this implies all literals $l_{im} \in c_i$ evaluate to false. Hence flipping any l_{im} will cause it to become true in c_i, and also to become true in all the same sign neighbouring clauses of l_{im}, i.e. $C_{l_{im}}$. Therefore, flipping l_{im} will ***help*** all the clauses in $C_{l_{im}}$, i.e. it will increase the number of true literals, thereby increasing the overall level of satisfaction for those clauses. Conversely, l_{im} has a corresponding set of opposite sign clauses that would be *damaged* when l_{im} is flipped. The DDFW heuristic adds weight to each false clause in a local minimum, by taking weight away from the most weighted same sign neighbour of that clause. However, the weight on a clause is not allowed to fall below $W_{init} - 1$, where W_{init} is the initial weight distributed to each clause at the start of the search. If no neighbouring same sign clause has sufficient weight to give to a false clause, then a non-neighbouring clause with sufficient weight is chosen randomly. Lastly, if the donating clause has a weight greater than W_{init} then it donates a weight of two, otherwise it donates a weight of one. The program logic for DDFW is otherwise based on PAWS, and is shown below:

Algorithm 1 DDFW($\mathcal{F}$, W_{init})

```
randomly instantiate each literal in F;
set the weight w_i for each clause c_i ∈ F to W_init;
while solution is not found and not timeout do
    find and return a list L of literals causing the greatest reduction in weighted cost Δw when flipped;
    if (Δw < 0) or (Δw = 0 and probability ≤ 15%) then
        randomly flip a literal in L;
    else
        for each false clause c_f do
            select a satisfied same sign neighbouring clause c_k with maximum weight w_k;
            if w_k < W_init then
                randomly select a clause c_k with weight w_k ≥ W_init;
            end if
            if w_k > W_init then
                transfer a weight of two from c_k to c_f;
            else
                transfer a weight of one from c_k to c_f;
            end if
        end for
    end if
end while
```

The intuition behind the DDFW heuristic is that clauses that share same sign literals should form alliances, because a flip that benefits one of these clauses will always benefit some other member(s) of the group. Hence, clauses that are connected in this way will form groups that tend towards keeping each other satisfied. However, these groups

are not closed, as each clause will have clauses within its own group that are connected by other literals to other groups. Weight is therefore able to move between groups as necessary, rather than being uniformly smoothed (as in existing methods).[2]

3 Analysis of Results and Conclusions

	DDFW		AdaptNovelty+		RSAPS		DPLL
Problem	Success	Mean Time	Success	Mean Time	Success	Mean Time	Time
bw_large.a	**100**	**0.00**	100	0.01	100	0.01	0.01
bw_large.b	**100**	**0.04**	100	0.13	100	0.06	*0.01
bw_large.c	**100**	**0.49**	61	7.50	84	19.85	0.53
bw_large.d	**100**	**1.31**	18	19.96	4	109.00	2.01
ais10	100	1.10	100	2.00	**100**	**0.02**	0.06
logistics.c	100	0.67	100	0.08	**100**	**0.01**	0.08
flat100-med	100	0.01	**100**	**0.00**	100	0.00	0.01
flat100-hard	100	0.03	100	0.03	**100**	**0.02**	*0.01
flat200-med	100	0.11	**100**	**0.08**	100	0.13	0.12
flat200-hard	**100**	**0.99**	37	4.32	78	5.04	*0.03
uf100-hard	100	0.00	100	0.00	**100**	**0.00**	0.01
uf250-med	100	0.02	**100**	**0.00**	100	0.02	1.25
uf250-hard	100	0.65	97	1.09	**100**	**0.18**	0.32
uf400-med	**100**	**0.06**	100	0.11	100	0.13	57.81
uf400-hard	**100**	**0.57**	45	12.30	100	4.07	178.92
f800-med	100	0.97	**100**	**0.25**	16	15.20	timed out
f800-hard	**100**	**2.81**	72	3.70	8	15.50	timed out
f1600-med	**100**	**3.44**	95	1.88	0	timed out	timed out
f1600-hard	**100**	**17.38**	96	18.86	70	16.70	timed out
par16-med	**100**	**96.13**	49	53.30	91	11.70	*1.52
par16-hard	**100**	**93.13**	21	26.30	71	20.60	*0.57
30v10d80c	100	1.52	**100**	**0.01**	100	0.15	0.26
30v10d40c	100	2.86	**100**	**0.02**	100	0.12	0.02
50v15d80c	100	130.00	**100**	**0.45**	47	60.66	timed out
50v15d40c	**100**	**529.76**	98	169.55	3	57.70	timed out

Table 1. Comparison of runtimes with best local search performance in bold. The DPLL results are the best of either Satz or zChaff, with dominating DPLL times indicated with a '*'. DDFW was run with a fixed W_{init} value of 8. The problems are taken from the earlier PAWS study [9], where bw_large = blocks world planning, ais = all-interval-series, flat = graph colouring, f and uf = randomly generated hard 3-SAT problems, and the 30v and 50v problems are randomly generated hard binary CSPs, where v = number of variables, d = domain size and c = constraint density. All experiments were performed on a Sun supercomputer with 8 $\times$ Sun Fire V880 servers, each with 8 $\times$ UltraSPARC-III 900MHz CPU and 8GB memory per node. Problems with a mean flip count of less than one million were tested on 1,000 runs, otherwise tests were over 100 runs, with all runs having a 20 million flip cut-off, except 50v15d40c, which used 50 million.

[2] To the best of our knowledge the only other SAT local search techniques to exploit neighbourhood relationships were [3] and [12]. These approaches used opposite sign relationships to generate new clauses by resolution, and so are not directly related to the work on DDFW. DDFW's weight transfer approach also bears similarities to the operations research subgradient optimisation techniques discussed in [11].

The results in Table 1 show that overall DDFW dominates AdaptNovelty+ and RSAPS, having the best performance on 13 of the 25 problems, with AdaptNovelty+ having the better performance on 7 and RSAPS on 6 of the remaining problems. In addition, DDFW is the only method that achieved a 100% success rate over the whole problem set. As versions of AdaptNovelty+ have won the SAT 2004 and 2005 local search competitions, the superior performance of DDFW is a significant achievement. In further tests (not reported here), DDFW was not able to match the performance PAWS or SAPS on the Table 1 problem set, when problem specific parameter tuning was allowed. Nevertheless DDFW showed the best performance on default parameter settings, and, when tuning was allowed, it was significantly better on all bw_large problems and several graph colouring and random 3-SAT problems.

In conclusion, DDFW represents a powerful general purpose SAT solver for problem domains where extensive parameter tuning is not practical. The work on DDFW also represents a first step in the development of a weight redistribution approach to clause weighting, and shows a simple way that neighbourhood structure can be used to guide weight redistribution decisions.

In future work we consider it will be promising to extend a DDFW-like approach to handle MAX-SAT problems with hard and soft constraints. Here the natural division between mandatory and optional clause satisfaction can be exploited by redistributing weight from hard to soft clauses, and vice versa, according to whether all hard clauses are currently satisfied.

References

1. Hoos, H., Stulze, T.: Stochastic Local Search. Morgan Kaufmann, Cambridge, Massachusetts (2005)
2. Morris, P.: The breakout method for escaping from local minima. In: Proceedings of 11th AAAI. (1993) 40–45
3. Cha, B., Iwama, K.: Adding new clauses for faster local search. In: Proceedings of 13th AAAI. (1996) 332–337
4. Frank, J.: Learning short-term clause weights for GSAT. In: Proceedings of 15th IJCAI. (1997) 384–389
5. McAllester, D., Selman, B., Kautz, H.: Evidence for invariants in local search. In: Proceedings of 14th AAAI. (1997) 321–326
6. Wu, Z., Wah, B.: An efficient global-search strategy in discrete Lagrangian methods for solving hard satisfiability problems. In: Proceedings of 17th AAAI. (2000) 310–315
7. Schuurmans, D., Southey, F.: Local search characteristics of incomplete SAT procedures. In: Proceedings of 10th AAAI. (2000) 297–302
8. Hutter, F., Tompkins, D., Hoos, H.: Scaling and Probabilistic Smoothing: Efficient dynamic local search for SAT. In: Proceedings of 8th CP. (2002) 233–248
9. Thornton, J., Pham, D., Bain, S., Ferreira Jr., V.: Additive versus multiplicative clause weighting for SAT. In: Proceedings of 19th AAAI. (2004) 191–196
10. Hoos, H.H.: An adaptive noise mechanism for walk-sat. In: Proceedings of 19th AAAI. (2002) 655–660
11. Schuurmans, D., Southey, F., Holte, R.: The exponentiated subgradient algorithm for heuristic boolean programming. In: Proceedings of 17th IJCAI. (2001) 334–341
12. Pullan, W., Zhao, L.: Resolvent clause weighting local search. In: Proceedings of 17th Canadian AI. (2004) 233–247