Engineering a Lightweight and Efficient Local Search SAT Solver

Adrian Balint and Uwe Schöning(✉)

Institute of Theoretical Computer Science, Ulm University,
89069 Ulm, Germany
{adrian.balint,uwe.schoening}@uni-ulm.de

Abstract. One important category of SAT solver implementations use stochastic local search (SLS, for short). These solvers try to find a satisfying assignment for the input Boolean formula (mostly, required to be in CNF) by modifying the (mostly randomly chosen) initial assignment by bit flips until a satisfying assignment is possibly reached. Usually such SLS type algorithms proceed in a greedy fashion by increasing the number of satisfied clauses until some local optimum is reached. Trying to find its way out of such local optima typically requires the use of randomness. We present an easy, straightforward SLS type SAT solver, called probSAT, which uses just one simple strategy being based on biased probabilistic flips. Within an extensive empirical study we evaluate the current state-of-the-art solvers on a wide range of SAT problems, and show that our approach is able to exceed the performance of other solving techniques.

1 Introduction

The SAT problem is one of the most studied $\mathcal{NP}$-complete problems in computer science. One reason is the wide range of SAT's practical applications ranging from hardware verification to planning and scheduling. Given a propositional formula in CNF with variables $\{x_1, \ldots, x_n\}$ the SAT-problem consists in finding an assignment for the variables such that all clauses are satisfied.

Stochastic local search (SLS) solvers operate on complete assignments and try to find a solution by flipping variables according to a given heuristic. Most SLS solvers are based on the following scheme: Initially, a random assignment is chosen. If the formula is satisfied by the assignment the solution is found. If not, a variable is chosen according to a (possibly probabilistic) variable selection heuristic, which is further called *pickVar*. The heuristics mostly depend on some score, which counts the number of satisfied/unsatisfied clauses, as well as other aspects like the "age" of variables, and others. It was believed that a good flip heuristic should be designed in a very sophisticated way to obtain a really efficient solver. We show in the following that it is worth to "come back to the roots" since a very elementary and (as we think) elegant design principle

© Springer International Publishing AG 2016
L. Kliemann and P. Sanders (Eds.): Algorithm Engineering, LNCS 9220, pp. 1–18, 2016.
DOI: 10.1007/978-3-319-49487-6_1

for the *pickVar* heuristic just based on probability distributions will do the job extraordinary well.

It is especially popular (and successful) to pick the flip variable from an unsatisfied clause. This is called *focused* local search in [14]. In each round, the selected variable is flipped and the process starts over again until a solution is eventually found.

Most important for the flip heuristic seems to be the *score* of an assignment, i.e. the number of satisfied clauses. Considering the process of flipping one variable, we get the *relative score change* produced by a candidate variable for flipping as: (*score after flipping* minus *score before flipping*) which is equal to *make* minus *break*. Here *make* means the number of newly satisfied clauses which come about by flipping the variable, and *break* means the number of clauses which become false by flipping the respective variable. To be more precise, we will denote $make(x, \alpha)$ and $break(x, \alpha)$ as functions of the respective flip variable x and the actual assignment α (before flipping). Notice that in case of focused flipping mentioned above the value of *make* is always at least 1.

Most of the SLS solvers so far, if not all, follow the strategy that whenever the score improves by flipping a certain variable from an unsatisfied clause, they will indeed flip this variable without referring to probabilistic decisions. Only if no improvement is possible as is the case in local minima, a probabilistic strategy is performed. The winner of the SAT Competition 2011 category random SAT, Sparrow, mainly follows this strategy but when it comes to a probabilistic strategy it uses a probability distribution function [2]. The probability distribution in Sparrow is defined as an exponential function of the score value. In this chapter we analyze several simple SLS solvers which are based only on probability distributions.

2 The New Algorithm Paradigm

We propose a new class of solvers here, called probSAT, which base their probability distributions for selecting the next flip variable solely on the make and break values, but not necessarily on the value of the $score = make - break$, as it was the case in Sparrow. Our experiments indicate that the influence of make should be kept rather weak – it is even reasonable to ignore make completely, like in implementations of WalkSAT [13]. The role of make and break in these SLS-type algorithms should be seen in a new light. The new type of algorithm presented here can also be applied for general constraint satisfaction problems and works as follows.

Algorithm 1. ProbSAT

Input : Formula F , $maxTries$, $maxFlips$
Output: satisfying assignment α or UNKNOWN

```
1  for i = 1 to maxTries do
2  |  α ← randomly generated assignment
3  |  for j = 1 to maxFlips do
4  |  |  if (α is model for F) then
5  |  |  |  return α
6  |  |  C_u ← randomly selected unsatisfied clause
7  |  |  for x in C_u do
8  |  |  |  compute f(x, α)
9  |  |  var ← random variable x according to probability f(x,α) / Σ_{z∈C_u} f(z,α)
10 |  |  α ← flip(var) in α
11 return UNKNOWN;
```

The idea here is that the function f should give a high value to variable x if flipping x seems to be advantageous, and a low value otherwise. Using f the probability distribution for the potential flip variables is calculated. The flip probability for x is proportional to $f(x, \alpha)$. Letting f be a constant function leads in the k-SAT case to the probabilities $(\frac{1}{k}, \dots, \frac{1}{k})$ morphing the probSAT algorithm to the random walk algorithm that is theoretically analyzed in [15]. In all our experiments with various functions f we made f depend on $break(x, \alpha)$ and possibly on $make(x, \alpha)$, and no other properties of x and α nor the history of previous search course. In the following we analyze experimentally the effect of several functions to be plugged in for f.

2.1 An Exponential Function

First we considered an exponential decay, 2-parameter function:

$$f(x, \alpha) = \frac{(c_m)^{make(x,\alpha)}}{(c_b)^{break(x,\alpha)}}$$

The parameters of the function are c_b and c_m. Because of the exponential functions used here (think of $c^x = e^{\frac{1}{T}x}$) this is reminiscence of the way Metropolis-like algorithms (see [17]) select a variable. Also, this is similar to the Softmax probabilistic decision for actions in reinforcement learning [19]. We call this the *exp-algorithm*. The separation into the two base constants c_m and c_b will allow us to find out whether there is a different influence of the make and the break value – and there is one, indeed.

It seems reasonable to try to maximize *make* and to minimize *break*. Therefore, we expect $c_m > 1$ and $c_b > 1$ to be good choices for these parameters. Actually, one might expect that c_m should be identical to c_b such that the above formula simplifies to $c^{make-break} = c^{score}$ for an appropriate parameter c.

To get a picture on how the performance of the solver varies for different values of c_m and c_b, we have done a uniform sampling of $c_b \in [1.0, 4.0]$ and of $c_m \in [0.1, 2.0]$ for this exponential function and of $c_m \in [-1.0, 1.0]$ for the polynomial function (see below). We have then run the solver with the different parameter settings on a set of randomly generated 3-SAT instances with 1000 variables at a clause to variable ratio of 4.26. The cutoff limit was set to 10 s. As a performance measure we use PAR10: penalized average runtime, where a timeout of the solver is penalized with 10·(cutoff limit). A parameter setting where the solver is not able to solve anything has a PAR10 value of 100 in our case.

In the case of 3-SAT a very good choice of the parameters is $c_b > 1$ (as expected) and $c_m < 1$ (totally unexpected), for example, $c_b = 3.6$ and $c_m = 0.5$ (see Fig. 1 left upper diagram and the survey in Table 1) with small variation

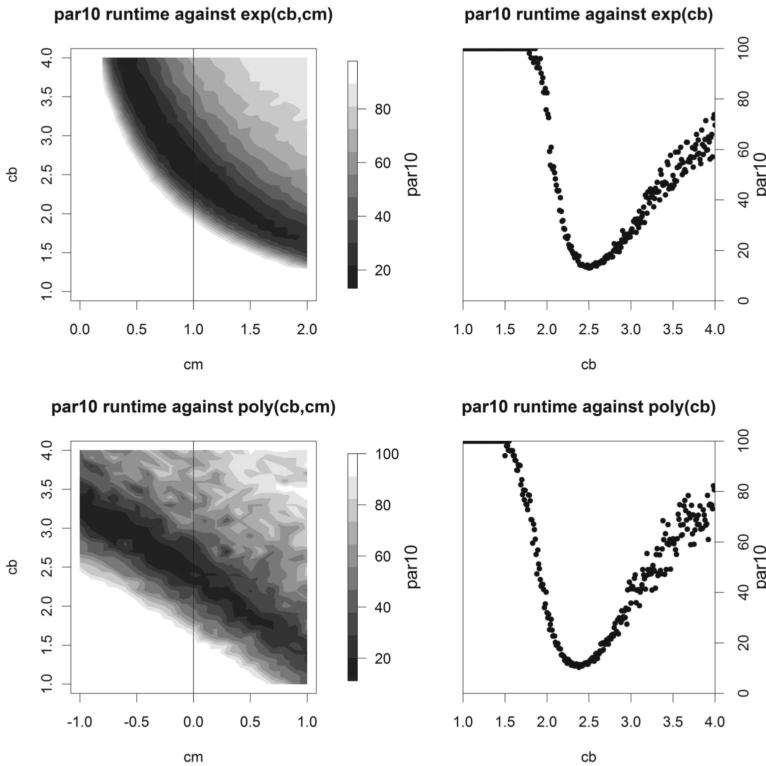

Fig. 1. Parameter space performance plot: The left plots show the performance of different combinations of c_b and c_m for the exponential (upper left corner) and the polynomial (lower left corner) functions. The darker the area the better the runtime of the solver with that parameter settings. The right plots show the performance variation if we ignore the make values (correspond to the cut in the left plots) by setting $c_m = 1$ for the exponential function and $c_m = 0$ for the polynomial function.

depending on the considered set of benchmarks. In the interval $c_m \in [0.3, 1.8]$ the optimal choice of parameters can be described by the hyperbola-like function $(c_b - 1.3) \cdot c_m = 1.1$. Almost optimal results were also obtained if c_m is set to 1 (and c_b to 2.5), see Fig. 1, both upper diagrams. In other words, the value of make is not taken into account in this case.

As mentioned, it turns out that the influence of make is rather weak, therefore it is reasonable, and still leads to very good algorithms – also because the implementation is simpler and has less overhead – if we ignore the make value completely and consider the one-parameter function:

$$f(x, \alpha) = (c_b)^{-break(x,\alpha)}$$

We call this the *break-only-exp-algorithm.*

2.2 A Polynomial Function

Our experiments showed that the exponential decay in probability with growing break value might be too strong in the case of 3-SAT. The above formulas have an exponential decay in probability comparing different (say) break values. The relative decay is the same when we compare $break = 0$ with $break = 1$, and when we compare, say, $break = 5$ with $break = 6$. A "smoother" function for high values would be a polynomial decay function. This led us to consider the following, 2-parameter function ($\epsilon = 1$ in all experiments):

$$f(x, \alpha) = \frac{(make(x, \alpha))^{c_m}}{(\epsilon + break(x, \alpha))^{c_b}}$$

We call this the *poly-algorithm.* The best parameters in case of 3-SAT turned out to be $c_m = -0.8$ (notice the minus sign!) and $c_b = 3.1$ (See Fig. 1, lower part). In the interval $c_m \in [-1.0, 1.0]$ the optimal choice of parameters can be described by the linear function $c_b + 0.9c_m = 2.3$. Without harm one can set $c_m = 0$, and then take $c_b = 2.3$, and thus ignore the make value completely.

Ignoring the make value (i.e. setting $c_m = 0$) brings us to the function

$$f(x, \alpha) = (\epsilon + break(x, \alpha))^{-c_b}$$

We call this the *break-only-poly-algorithm.*

2.3 Some Remarks

As mentioned above, in both cases, the exp- and the poly-algorithm, it was a good choice to ignore the make value completely (by setting $c_m = 1$ in the exp-algorithm, or by setting $c_m = 0$ in the poly-algorithm). This corresponds to the vertical lines in Fig. 1, left diagrams. But nevertheless, the optimal choice in both cases, was to set $c_m = 0.5$ and $c_b = 3.6$ in the case of the exp-algorithm (and similarly for the poly-algorithm.) We have $\frac{0.5^{make}}{3.6^{break}} \approx 3.6^{-(break+make/2)}$.

This can be interpreted as follows: instead of the usual $score = make - break$ a better score measure is $-(break + make/2)$.

The value of c_b determines the greediness of the algorithm. We concentrate on c_b in this discussion since it seems to be the more important parameter. The higher the value of c_b, the more greedy is the algorithm. A low value of c_b (in the extreme, $c_b = 1$ in the exp-algorithm) morphs the algorithm to a random walk algorithm with flip probabilities $(\frac{1}{k}, \dots \frac{1}{k})$ like the one considered in [15]. Examining Fig. 1, almost a phase-transition can be observed. If c_b falls under some critical value, like 2.0, the expected run time increases tremendously. Turning towards the other side of the scale, increasing the value of c_b, i.e. making the algorithm more greedy, also degrades the performance but not with such an abrupt rise of the running time as in the other case. These observations have also been made empirically by Hoos in [9], where he proposed to approximate the noise value from above, rather from below.

3 Experimental Analysis of the Functions

To determine the performance of our probability distribution based solver we have designed a wide variety of experiments. In the first part of our experiments we try to determine good settings for the parameters c_b and c_m by means of automatic configuration procedures. In the second part we will compare our solver to other state-of-the-art solvers.

3.1 The Benchmark Formulae

All random instances used in our settings are uniform random k-SAT problems generated with different clause to variable ratios, denoted with r. The class of random 3-SAT problems is the best studied class of random problems and because of this reason we have four different sets of 3-SAT instances.

1. 3sat1k [21]: 10^3 variables at $r = 4.26$ (500 instances)
2. 3sat10k [21]: 10^4 variables at $r = 4.2$ (500 instances)
3. 3satComp[1]: all large 3-SAT instances from the SAT Competition 2011 category random with variables range $2 \cdot 10^3 \dots 5 \cdot 10^4$ at $r = 4.2$ (100 instances)
4. 3satExtreme: $10^5 \dots 5 \cdot 10^5$ variables at $r = 4.2$ (180 instances)

The 5-SAT and 7-SAT problems used in our experiments come from [21]: 5sat500 (500 variables at $r = 20$) and 7sat90 (90 variables at $r = 85$). The 3sat1k, 3sat10k, 5sat500 and 7sat90 instance classes are divided into two equal sized classes called train and test. The train set is used to determine good parameters for c_b and c_m and the second class is used to report the performance. Further we also include the set of satisfiable random and crafted instances from the SAT Competition 2011.

[1] www.satcompetition.org.

3.2 Good Parameter Setting

The problem that every solver designer is confronted with is the determination of good parameters for its solvers. We have avoided to accomplish this task by manual tuning but instead have used an automatic procedure.

As our parameter search space is relatively small, we have opted to use a modified version of the iterated F-Race [5] configurator, which we have implemented in Java. The idea of F-race is relatively simple: good configurations should be evaluated more often than poor ones which should be dropped as soon as possible. F-Race uses a familywise Friedman test (see Test 25 in [18] for more details about the test) to check if there is a significant performance difference between solver configurations. The test is conducted every time the solvers have run on an instance. If the test is positive, poor configurations are dropped, and only the good ones are further evaluated. The configurator ends when the number of solvers left in the race is less than 2 times the number of parameters or if there are no more instances to evaluate on.

To determine good values for c_b and c_m we have run our modified version of F-Race on the training sets 3sat1k, 3sat10k, 5sat500 and 7sat90. The cutoff time for the solvers were set to 10 s for 3sat1k and to 100 s for the rest. The best parameter values returned by this procedure are listed in Table 1. Values for the class of 3sat1k problems were also included, because the preliminary analysis of the parameter search space was done on this class. The best parameter of the break-only-exp-algorithm for k-SAT can be roughly described by the formula $c_b = k^{0.8}$.

Table 1. Parameter setting for c_b and c_m: Each cell represents a good setting for c_b and c_m dependent on the function used by the solver. Parameter values close to these values have similar good performance.

	3sat1k	3sat10k	5sat500	7sat90
$exp(c_b, c_m)$	3.6 0.5	3.97 0.3	3.1 1.3	3.2 1.4
$poly(c_b, c_m)$	3.1 −0.8	2.86 −0.81	-	-
$exp(c_b)$	2.50	2.33	3.6	4.4
$poly(c_b)$	2.38	2.16	-	-

4 Empirical Evaluation

In the second part of our experiments we compare the performance of our solvers to that of the SAT Competition 2011 winners and also to WalkSAT [13]. An additional comparison to a survey propagation algorithm will show how far our probSAT local search solver can get.

Soft- and Hardware. The solvers were run on a part of the bwGrid clusters [8] (Intel Harpertown quad-core CPUs with 2.83 GHz and 8 GByte RAM). The operating system was Scientific Linux. All experiments were conducted with EDACC, a platform that distributes solver execution on clusters [1].

The Competitors. The WalkSAT solver is implemented within our own code basis. We use our own implementation and not the original code (version 48) provided by Henry Kautz[2], because our implementation is approximately 1.35 times faster[3].

We have used version 1.4 of the survey propagation solver provided by Zecchina[4], which was changed to be DIMACS conform. For all other solvers we have used the binaries from the SAT Competition 2011[5].

Parameter Settings of Competitors. Sparrow is highly tuned on our target set of instances and incorporates optimal settings for each set within its code. WalkSAT [13] has only one single parameter, the walk probability wp. In case of 3-SAT we took the optimal values for $wp = 0.567$ which have been established in an extensive analysis in [11]. Because we could not find any settings for 5-SAT and 7-SAT problems we have run our modified version of F-Race to find good settings. For 5sat500 the configurator reported $wp = 0.25$ and for 7sat90 $wp = 0.1$. The survey propagation solver was evaluated with the default settings reported in [6] (fixing 5 % of the variables per step).

Results. We have evaluated our solvers and the competitors on the test set of the instance sets 3sat1k, 3sat10k, 5sat500 and 7sat90 (note that the training set was used only for finding good parameters for the solvers). The parameter setting for c_b and c_m are those from Table 1 (in case of 3-SAT we have always used the parameters for 3sat10k). The results of the evaluations are listed in Table 2.

On the 3-SAT instances, the polynomial function yields the overall best performance. On the 3-SAT competition set all of our solver variants exhibited the most stable performance, being able to solve all problems within cutoff time. The survey propagation solver has problems with the 3sat10k and the 3sat-Comp problems (probably because of the relatively small number of variables). The good performance of the break-only-poly-solver remains surprisingly good even on the 3satExtreme set where the number of variables reaches $5 \cdot 10^5$ (ten times larger than that from the SAT Competition 2011). From the class of SLS solvers it exhibits the best performance on this set and is only approx. 2 times slower than survey propagation. Note that a value of $c_b = 2.165$ for the break-only-poly solver further improved the runtime of the solver by approximately 30 % on the 3satExtreme set.

[2] http://www.cs.rochester.edu/u/kautz/walksat/.

[3] The latest version 50 of WalkSAT has been significantly improved, but was not available at the time we have performed the experiments.

[4] http://users.ictp.it/~zecchina/SP/.

[5] http://www.cril.univ-artois.fr/SAT11/solvers/SAT2011-static-binaries.tar.gz.

Table 2. Evaluation results: Each cell represents the PAR10 (Penalized average runtime with penalization factor 10 - every unsuccessful run is penalized with 10 times the maximum runtime.) runtime and the number of successful runs for the solvers on the given instance set. Results are highlighted if the solver succeeded in solving all instances within the cutoff time, or if it has the best PAR10 runtime. Cutoff times are 600 s for 3sat10k, 5sat500 and 7sat90 and 5000 s for the rest. The blank cells indicates that we have no parameter setting worth evaluating.

	3sat10k	3satComp	3satExtreme	5sat500	7sat90
$exp(c_b, c_m)$	46.6	93.84	-	12.49	201.68
	(998)	**(500)**		(10^3)	(974)
$poly(c_b, c_m)$	46.65	76.81	-	-	-
	996	**(500)**			
$exp(c_b)$	53.02	126.59	-	**7.84**	134.06
	(997)	**(500)**		(10^3)	(984)
$poly(c_b)$	**22.80**	**54.37**	1121.34	-	-
	(1000)	**(500)**	**(180)**		
Sparrow	199.78	498.05	47419	9.52	**14.94**
	(973)	(498)	(10)	(10^3)	(10^3)
WalkSAT	61.74	172.21	1751.77	14.71	69.34
	(995)	(499)	(178)	(10^3)	(994)
sp 1.4	3146.17	18515.79	**599.01**	5856	6000
	(116)	(63)	**(180)**	(6)	(0)

On the 5-SAT instances the exponential break-only-exp solver yields the best performance being able to beat even Sparrow, which was the best solver for 5-SAT within the SAT Competition 2011. On the 7-SAT instances though the performance of our solvers is relatively poor. We observed a very strong variance of the run times on this set and it was relatively hard for the configurator to cope with such high variances.

Overall the performance of our simple probability based solvers reaches state-of-the-art performance and can even get into problem size regions where only survey propagation could catch ground.

Scaling Behavior with the Number of Variables n. Experiments show that the survey propagation algorithm scales linearly with n on formulas generated near the threshold ratio. The same seems to hold for WalkSAT with optimal noise as the results in [11] show. The 3satExtreme instance set contains very large instances with varying $n \in \{10^5 \dots 5 \cdot 10^5\}$. To analyze the scaling behavior of probSAT in the break-only-poly variant we have computed for each run the number of flips per variable performed by the solver until a solution was found. The number of flips per variable remains constant at about $2 \cdot 10^3$ independent of n. The same holds for WalkSAT, though WalkSAT seems to have a slightly larger variance of the runtimes.

Results on the SAT Competition 2011 Satisfiable Random Set. We have compiled an adaptive version of probSAT and of WalkSAT, that first checks the size of the clauses (i.e. k) and then sets the parameters accordingly (like Sparrow does). We have ran these solvers on the complete satisfiable instances set from the SAT Competition 2011 random category along with all other competition winning solvers from this category: Sparrow2011, sattime2011 and EagleUP. Cutoff time was set to 5000 s. We report only the results on the large set, as the medium set was completely solved by all solvers and the solvers had a median runtime under one second. As can be seen from the results of the cactus plot in Fig. 2, the adaptive version of probSAT would have been able to win the competition. Interestingly is to see that the adaptive version of WalkSAT would have ranked third.

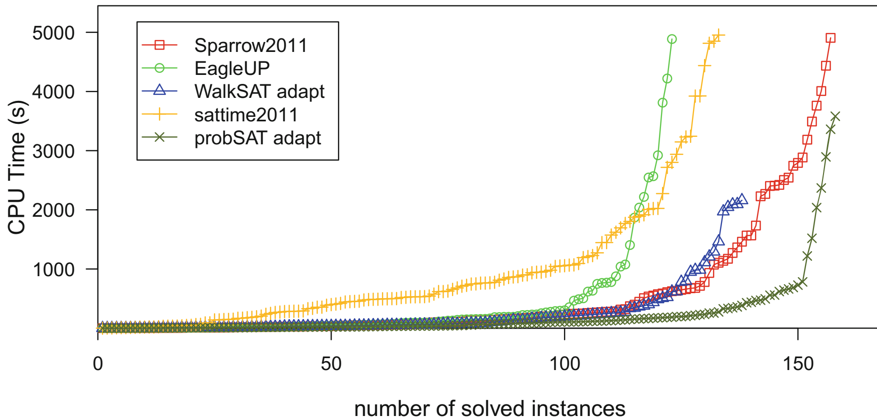

Fig. 2. Results on the "large" set of the SAT Competition 2011 random instances represented as a cactus plot. The x-axis represents the number of problems a solver was able to solve ordered by runtime; the y-axis is the runtime. The lower a curve (low runtimes) and the more it gets to the right (more problems solved) the better the solver.

Results on the SAT Competition 2011 Satisfiable Crafted Set. We have also run the different solvers on the satisfiable instances from the crafted set of SAT Competition 2011 (with a cutoff time of 5000 s). The results are listed in Table 3. We have also included the results of the best three complete solvers from the crafted category. probSAT and WalkSAT performed best in their 7-SAT break-only configuration solving 81 respectively 101 instances. The performance of WalkSAT could not be improved by changing the walk probability. probSAT though exhibited better performance with $c_b = 7$ and a switch to the polynomial break-only scheme, being then able to solve 93 instances. With such a high c_b value (very greedy) the probability of getting stuck in local minima is very high. By adding a static restart strategy after $2 \cdot 10^4$ flips per variable probSAT was then able to solve 99 instances (as listed in the table).

Table 3. Results on the crafted satisfiable instances: Each cell reports the number of solved instances within the cutoff time (5000 s). The first line shows the results on the original instances and the second on the preprocessed instances.

	Sattime	Sparrow	WalkSAT	probSAT	MPhaseSAT (complete)	clasp (complete)
Crafted	**107**	104	101	99	93	81
Crafted pre	86	97	**101**	95	98	80

The high greediness level needed for WalkSAT and probSAT to solve the crafted instances indicates that this instances might be more similar to the 7-SAT instances (generally to higher k-SAT). A confirmation of this conjecture is that Sparrow with fixed parameters for 7-SAT instances could solve 103 instances vs. 104 in the default setting (which adapts the parameters according to the maximum clause length found in the problem). We suppose that improving SLS solvers for random instances with large clause length would also yield improvements for non random instances.

To check whether the performance of SLS solvers can be improved by preprocessing the instances first, we have run the preprocessor of lingeling [4], which incorporates all main preprocessing techniques, to simplify the instances. The results unluckily show the contrary of what would have been expected (see Table 3). None of the SLS solvers could benefit from the preprocessing step, solving equal or less instances. These results motivated the analysis of preprocessing techniques in more detail, which was performed in [3]. It turns out that bounded variable elimination, which performs variable elimination through resolution rules up to certain bound is a good preprocessing technique for SLS solvers and can indeed improve the performance of SLS solvers.

Results on the SAT Challenge 2012 Random Set. We have submitted the probSAT solver (the adaptive version) to the SAT Challenge 2012 random satisfiable category. The results of the best performing solvers can be seen as a cactus plot in Fig. 3. probSAT was the second best solver on these instances been only outperformed by CCAsat.

While the difference to all other competitors is significant in terms of a Mann-Whitney-U test, the difference to CCAsat is not.

Results on the SAT Competition 2013 Satisfiable Random Set. We have also submitted an improved version of probSAT to the SAT Competition 2013 to the Random Satisfiable category. The implementation of probSAT was improved with respect to parameters, data structure and work flow.

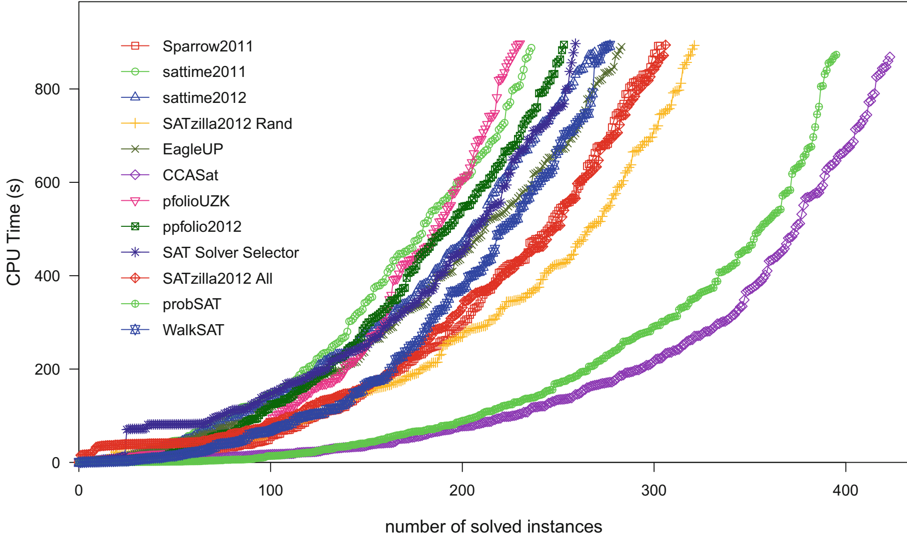

Fig. 3. Results of the best performing solvers on the SAT Challenge 2012 random instances as a cactus plot. For details about cactus plot see Fig. 2.

The parameters of probSAT have been set as follows:

k	fct	cb	ϵ
3	poly	2.06	0.9
4	exp	2.85	-
5	exp	3.7	-
6	exp	5.1	-
≥ 7	exp	5.4	-

where k is the size of the longest clause found in the problem during parsing. These parameter values have been determined in different configuration experiments.

All array data structures where ended by a sentinel[6] (i.e. the last element in the array is the stop value; in our case we have used 0). All for-loops have been changed into while-loops that have no counter but only a sentinel check, allowing us to save several memory dereferences and variables. As most of the operations performed by SLS solvers are loops over some small sized arrays, this optimization turns out to improve the performance of the solver between 10 %–25 % (dependent on the instances).

[6] We would like to thank Armin Biere for this suggestion.

Compared to the original version the version submitted to the competition is not selecting an unsatisfied clause randomly but will iterate through the set of unsatisfied clauses with the flip counter (i.e. instead of `c=rand() modulo numUnsat` we use `c=flipCounter modulo numUnsat`). This scheme will reduce the probability of undoing a change right in the next step. This small change seems to improve in some cases the stagnation behavior of the solver giving it a further boost[7].

To measure the isolated effect of the different changes we have performed a small experiment on the 3sat10k instance set. We start with the version that was submitted to the SAT Challenge 2012 with new parameters (sc12(1)), then we add the code optimizations (sc12(2)) and finally we remove the random selection of a false clause (sc13). A further version was added to this evaluation that does not cache the break values, but computes them on the fly. This version is denoted with (nc) in the table and was analyzed only after the competition. The results of the evaluation are listed in Table 4.

Table 4. The results of the evaluation of different implementation variants of the probSAT solver on the 3sat10k instance set. The last column shows the speed up with respect to the last row. Time is measured in seconds.

	Solver	Total CPU time	Average CPU time	Median CPU time	Average speedup
1	probSAT sc13 (nc)	4356.0729	17.4243	7.886	2.01x
2	probSAT sc13	4696.9674	18.7879	8.499	1.86x
3	probSAT sc12(2)	7632.1326	30.5285	10.695	1.15x
4	probSAT sc12(1)	8781.8255	35.1273	12.489	-

The code optimizations yielded an average speedup of 15 %, while the removal of random clause selection is further improving the performance by around 70 %. Further adding on the fly computation of the break values yields a twofold speedup compared to the original version with new parameters.

probSAT sc13 was submitted to SAT Competition 2013[8]. The results of the best performing solvers submitted to SAT Competition 2013 can be seen as a cactus plot in Fig. 4. probSAT is able to outperform all its competitors. The instances used in SAT Competition 2013 contained randomly generated instances on the phase transition point for $k = 3, \dots, 7$ and also a small set of huge instances (in terms of number of variables). The last were intended to test the robustness of the solvers. probSAT turns out to be a very robust solver, being able to solve many of the huge instances 18 out of the 26 that have been solved by some solver (out of a total of 36). From the set of phase transition instances

[7] This might also be the case for the WalkSAT solver.

[8] The code was compiled with the Intel®Compiler 12.0 with the following parameters: *-O3 -xhost -static -unroll-aggressive -opt-prefetch -fast*.

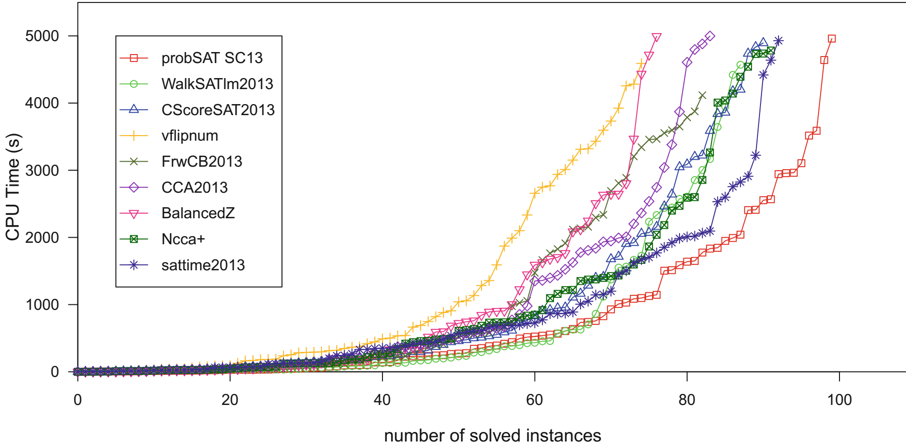

Fig. 4. Results of the best performing solvers on the SAT Competition 2013 random satisfiable instances.

probSAT solved 81 out of 109 that could be solved by any solver. Altogether this shows that the solving approach (and the parameter settings) used by probSAT has an overall good performance.

5 Comparison with WalkSAT

In principle, WalkSAT [13] also uses a certain pattern of probabilities for flipping one of the variables within a non-satisfied clause. But the probability distribution does not depend on a single continuous function f as in our algorithms described above, but uses some non-continuous if-then-else decisions as described in [13].

In Table 5 we compare the flipping probabilities in WalkSAT (setting the wp parameter i.e. the noise value to $wp = 0.567$) with the break-only-poly-algorithm (with $c_b = 2.06$ and $\epsilon = 0.9$) using several examples of $break$ values combinations that might occur within a 3-CNF clause.

Even though the probabilities look very similar, we think that the small differences renders our approach to be more robust. Further, probSAT has the PAC property [10, p. 153]. In each step every variable has a probability greater zero to be picked for flipping. This is though not the case for WalkSAT. A variable occurring in a clause where an other variable has a score of zero can not be chosen for flipping. There is no published example for which WalkSAT gets trapped in cycles. Though, during a talk given by Donald Knuth in Trento at the SAT Conference in 2012 where he presented details about his implementation of WalkSAT, he mentioned that Bram Cohen, the designer of WalkSAT, has provided such an example.

Table 5. Probability comparison of WalkSAT and probSAT: The first columns show some possible *break* value combinations that occur within a clause in a 3-SAT formula during the search. For the different solvers considered here the probabilities for each of the 3 variables to be flipped are listed.

Breaks			WalkSAT			Break-only-poly		
0	0	0	0.33	0.33	0.33	0.33	0.33	0.33
0	0	1	0.5	0.5	0	0.45	0.45	0.10
0	1	1	1.0	0	0	0.70	0.15	0.15
0	1	2	1.0	0	0	0.76	0.16	0.07
0	2	2	1.0	0	0	0.85	0.07	0.07
1	1	1	0.33	0.33	0.33	0.33	0.33	0.33
1	1	2	0.41	0.41	0.17	0.41	0.41	0.18
1	2	2	0.53	0.23	0.23	0.54	0.23	0.22
1	2	3	0.53	0.23	0.23	0.61	0.25	0.14

6 Implementation Variations

In the previous sections we have compared the solvers based on their runtime. As a consequence the efficiency of the implementation plays a crucial role and the best available implementation should be taken for comparison. Another possible comparison measure is the number of flips the solver needs to perform to find a solution. From a practical point of view this is not optimal. The number of flips per second (denoted with $flips/sec$) is a key measure of SLS solvers when it comes to compare algorithm implementations or two different similar algorithms. In this Section we would like to address this problem by comparing two different implementations of probSAT and WalkSAT on a set of very large 3-SAT problems.

All efficient implementations of SLS solvers are computing the scores of variables from scratch only within the initialization phase. During the search of the solver, the scores are only updated. This is possible because only the score of variables can change that are in the neighborhood of the variable being flipped. This method is also known as *caching* (the scores of the variables are being cached) in [10, p. 273] or *incremental* approach in [7].

The other method would be to compute the score of variables on the fly before taking them into consideration for flipping. This method is called *non-caching* or *non-incremental* approach. In case of probSAT or WalkSAT only the score of variables from one single clause has to be computed as opposed to other solvers where all variables from all unsatisfied clauses are taken into consideration for flipping.

We have implemented two different versions of probSAT and WalkSAT within the same code basis (i.e. the solvers are identical with exception of the *pickVar* method), one that uses caching and one that does not. We have evaluated the

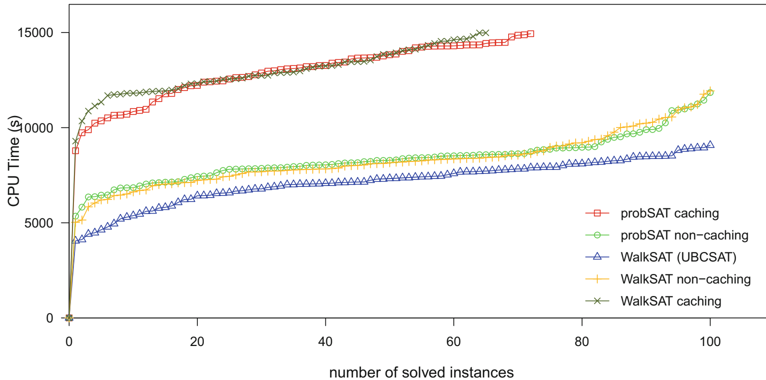

Fig. 5. Comparison of the different implementation variants of probSAT and WalkSAT on extreme large 3-SAT problems (within the same code basis), with and without caching of the break values. We also evaluate the best known WalkSAT implementation (non-caching) from UBCSAT as a reference.

four different solvers on a set of 100 randomly generated 3-SAT problems with 10^6 variables and a ratio of 4.2. The results can be seen in Fig. 5.

Within the time limit of $1.5 \cdot 10^4$ s only the variants not using caching were able to solve all problems. The implementation with caching solved only 72 (probSAT) respectively 65 instances (WalkSAT). Note that all solvers started with the same seed (i.e. they perform search on the exactly same search trajectory). The difference between the different implementations in terms of performance can be explained by the different number of $flips/sec$. While the version with caching performs around $1.4 \cdot 10^5$ flips/sec the version without caching is able to perform around $2.2 \cdot 10^5$ flips/sec. This explains the difference in runtime between the two different implementations. Similar findings have also been observed in [20, p. 27] and in [7].

The advantage of non-caching decreases with increasing k (for random generated k-SAT problems) and becomes even a disadvantage for 5-SAT problems and upwards. As a consequence the latest version of probSAT uses caching for 3-SAT problems and non-caching for the other types of problems.

7 Conclusion and Future Work

We introduced a simple algorithmic design principle for a SLS solver which does its job without heuristics and "tricks". It just relies on the concept of probability distribution and focused search. It is though flexible enough to allow plugging in various functions f which guide the search.

Using this concept we were able to discover a non-symmetry regarding the importance of the *break* and *make* values: the *break* value is the more important one; one can even do without the *make* value completely.

We have systematically used an automatic configurator to find the best parameters and to visualize the mutual dependency and impact of the parameters.

Furthermore, we observe a large variation regarding the running times even on the same input formula. Therefore the issue of introducing an optimally chosen restart point arises. Some initial experiments show that performing restarts, even after a relatively short period of flips (e.g. $20\,n$) gives favorable results on hard instances. It seems that the probability distribution of the number of flips until a solution is found, shows some strong heavy tail behavior (cf. [12,16]).

Finally, a theoretical analysis of the Markov chain convergence and speed of convergence underlying this algorithm would be most desirable, extending the results in [15].

Acknowledgments. We would like to thank the BWGrid [8] project for providing the computational resources. This project was funded by the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 302/9-1. We thank Daniel Diepold and Simon Gerber for implementing the F-race configurator and providing different analysis tools within the EDACC framework. We would also like to thank Andreas Fröhlich for fruitful discussions on this topic and Armin Biere for helpful suggestions regarding code optimizations.

References

1. Balint, A., Diepold, D., Gall, D., Gerber, S., Kapler, G., Retz, R.: EDACC - an advanced platform for the experiment design, administration and analysis of empirical algorithms. In: Coello, C.A.C. (ed.) LION 2011. LNCS, vol. 6683, pp. 586–599. Springer, Heidelberg (2011). doi:10.1007/978-3-642-25566-3_46
2. Balint, A., Fröhlich, A.: Improving stochastic local search for SAT with a new probability distribution. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 10–15. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14186-7_3
3. Balint, A., Manthey, N.: Analysis of preprocessing techniques and their utility for CDCL and SLS solver. In: Proceedings of POS2013 (2013)
4. Biere, A.: Lingeling and friends at the SAT competition 2011. Technical report, FMV Reports Series, Institute for Formal Models and Verification, Johannes Kepler University, Altenbergerstr. 69, 4040 Linz, Austria (2011)
5. Birattari, M., Yuan, Z., Balaprakash, P., Stützle, T.: F-Race and iterated F-Race: an overview. In: Bartz-Beielstein, T., Chiarandini, M., Paquete, L., Preuss, M. (eds.) Experimental Methods for the Analysis of Optimization Algorithms, pp. 311–336. Springer, Heidelberg (2010). http://dx.doi.org/10.1007/978-3-642-02538-9_13
6. Braunstein, A., Mézard, M., Zecchina, R.: Survey propagation: an algorithm for satisfiability. Random Structures & Algorithms **27**(2), 201–226 (2005)
7. Fukunaga, A.: Efficient implementations of SAT local search. In: Seventh International Conference on Theory and Applications of Satisfiability Testing (SAT 2004), pp. 287–292 (2004, this volume)
8. bwGRiD(http://www.bwgrid.de/): Member of the German D-Grid initiative, funded by the Ministry of Education and Research (Bundesministeriumfür Bildung und Forschung) and the Ministry for Science, Research and Arts Baden-Württemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg). Techical report, Universities of Baden-Württemberg (2007-2010)

9. Hoos, H.H.: An adaptive noise mechanism for WalkSAT. In: Proceedings of the Eighteenth National Conference in Artificial Intelligence (AAAI 2002), pp. 655–660 (2002)
10. Hoos, H.H., Stützle, T.: Stochastic Local Search: Foundations and Applications. Morgan Kaufmann, San Francisco (2005)
11. Kroc, L., Sabharwal, A., Selman, B.: An empirical study of optimal noise and runtime distributions in local search. In: Strichman, O., Szeider, S. (eds.) SAT 2010. LNCS, vol. 6175, pp. 346–351. Springer, Heidelberg (2010). doi:10.1007/978-3-642-14186-7_31
12. Luby, M., Sinclair, A., Zuckerman, D.: Optimal speedup of Las Vegas algorithms. In: ISTCS, pp. 128–133 (1993). http://dblp.uni-trier.de/db/conf/istcs/istcs1993.html#LubySZ93
13. McAllester, D., Selman, B., Kautz, H.: Evidence for invariants in local search. In: Proceedings of the Fourteenth National Conference on Artificial Intelligence (AAAI 1997), pp. 321–326 (1997)
14. Papadimitriou, C.H.: On selecting a satisfying truth assignment. In: Proceedings of the 32nd Annual Symposium on Foundations of Computer Science (FOCS 1991), pp. 163–169 (1991)
15. Schöning, U.: A probabilistic algorithm for k-SAT and constraint satisfaction problems. In: Proceedings of the Fourtieth Annual Symposium on Foundations of Computer Science (FOCS 1999), p. 410 (1999)
16. Schöning, U.: Principles of stochastic local search. In: Akl, S.G., Calude, C.S., Dinneen, M.J., Rozenberg, G., Wareham, H.T. (eds.) UC 2007. LNCS, vol. 4618, pp. 178–187. Springer, Heidelberg (2007). doi:10.1007/978-3-540-73554-0_17
17. Seitz, S., Alava, M., Orponen, P.: Focused local search for random 3-satisfiability. CoRR abs/cond-mat/0501707 (2005)
18. Sheskin, D.J.: Handbook of Parametric and Nonparametric Statistical Procedures, 4th edn. Chapman & Hall/CRC, Boca Raton (2007)
19. Sutton, R.S., Barto, A.G.: Reinforcement Learning: An Introduction. MIT Press, Cambridge (1998). http://www.cs.ualberta.ca/%7Esutton/book/ebook/the-book.html
20. Tompkins, D.A.D.: Dynamic local search for SAT: design, insights and analysis. Ph.D. thesis, University of British Columbia, October 2010
21. Tompkins, D.A.D., Balint, A., Hoos, H.H.: Captain jack: new variable selection heuristics in local search for SAT. In: Sakallah, K.A., Simon, L. (eds.) SAT 2011. LNCS, vol. 6695, pp. 302–316. Springer, Heidelberg (2011). doi:10.1007/978-3-642-21581-0_24