Improving implementation of SLS solvers for SAT and new heuristics for k-SAT with long clauses

Adrian Balint[1], Armin Biere[2], Andreas Fröhlich[2], and Uwe Schöning[1]

[1] Institute of Theoretical Computer Science, Ulm University, Ulm, Germany
{adrian.balint, uwe.schoening}@uni-ulm.de
[2] Inst. Formal Models and Verification, Johannes Kepler University, Linz, Austria
{armin.biere, andreas.froehlich}@jku.at

Abstract. Stochastic Local Search (SLS) solvers are considered one of the best solving technique for randomly generated problems and more recently also have shown great promise for several types of hard combinatorial problems. Within this work, we provide a thorough analysis of different implementation variants of SLS solvers on random and on hard combinatorial problems. By analyzing existing SLS implementations, we are able to discover new improvements inspired by CDCL solvers, which can speed up the search of all types of SLS solvers. Further, our analysis reveals that the multilevel break values of variables can be easily computed and used within the decision heuristic. By augmenting the probSAT solver with the new heuristic, we are able to reach new state-of-the-art performance on several types of SAT problems, especially on those with long clauses. We further provide a detailed analysis of the clause selection policy used in focused search SLS solvers.

1 Introduction

The Satisfiability problem (SAT) is one of the best known NP-complete problems. Besides its theoretical importance, it has also many practical applications in different domains such as software and hardware verification. Two of the best known solving approaches for the SAT problem are Conflict Driven Clause Learning (CDCL) and Stochastic Local Search (SLS). Solvers based on the CDCL solving approach have a very good performance on structured and application problems, and they are highly engineered for this type of problems. Stochastic Local Search solvers, on the other side, are good at solving randomly generated problems and several types of hard combinatorial problems.

The wide use of CDCL solvers in different applications has led to highly efficient implementations for these type of solvers. In contrast, SLS solvers do not have many practical applications, and their implementations have not been optimized as excessively as those of CDCL solvers. We think that SLS solvers have a high potential on hard combinatorial problems and that it is worth investing effort in improving algorithmic aspects of their implementation.

The contribution of this paper are three-fold. First we propose an improvement of the implementations techniques used in SLS solvers inspired from the CDCL solver NanoSAT. During the analysis of implementation techniques we discovered that the multilevel *break* value can be computed cheaply and can speed-up the search of SLS solvers when used in decision heuristics, this being our second contribution. Further we observed that the selection of unsatisfied clauses influences the performance of SLS solvers considerably. Our third contribution consists in the analysis of the clause selection heuristics.

1.1 Related Work

Implementation methods play a crucial role for SAT solvers and can influence their performance considerably. Fukunaga analyzed different implementations of common SLS solvers in [1]. He observed that during search the transition of clauses from having one satisfied to having two satisfied literals, or backwards occurs in structured problems very often. As a consequence, he proposed to introduce the 2-watched literals scheme for SLS solvers.

Multilevel properties of variables like the $make_l$ and $break_l$ values were first considered in [2], where the level one and two make value were combined as a linear function in order to break ties in WalkSAT. They were able to show that the performance of the WalkSAT solver can be considerably increased with this new tie breaking mechanism on 5-SAT and 7-SAT problems. The possible use of the $break_2$ value was also mentioned, but was not practically applied, probably due to the involved implementation complexity. In [3] the second level score value is being used ($score_2 = make_2 - break_2$) in the clause weighting solver CScoreSAT. The use of higher levels is considered but was not analyzed.

Within a caching implementation, the $make_2$ value can be computed with little overhead, while the $break_2$ value needs additional data structures. As one result of our implementation analysis, we have figured out that the $break_l$ value can be computed very easily within the non-caching implementations and consequently analyzed its role in the probSAT solver. The authors of [2] also mention that the role of *make* should not be neglected as was shown in [4]. This is of course true for their findings, but as the *make* and *break* value are complements of each other, it is sufficient to consider only the break value.

The importance of clause selection is strongly related to the class of GSAT solvers [5] and to those of weighted solvers [6,7,8]. Though, we are not aware of any analysis performed for focused random walk solvers like WalkSAT or probSAT. The pseudo breadth first search (PBFS) scheme was proposed in [9, p. 93] and only analyzed on a small set of randomly generated 3-SAT problems.

2 Implementations of SLS solvers

SLS solvers work on complete assignments as opposed to partial assignments used by CDCL solvers. Starting from a random generated assignment, an SLS solver selects a variable according to some heuristic ($pickVar$) and then changes

its value (a *flip*). This process is repeated until a satisfying assignment has been found or some limits have been reached.

The input to the SLS solver is a formula F in conjunctive normal form (CNF), i.e., a conjunction of clauses. A clause is a disjunction of literals, which are defined as variables or negation of these. An assignment α is called satisfying for formula F if every clause contains at least one satisfying literal. A literal l within a clause is satisfying if the value within the assignment α corresponds to its polarity (i.e. false for negated and true for positive literals).

The complexity of an SLS solver is completely dominated by two operations: $pickVar()$ and $flip(var)$. The complexity of the $pickVar$ method depends on the heuristic used by the SLS solver, while the complexity of the $flip(var)$ method depends on the computation of the information that has to be updated.

For the moment, we will restrict our analysis to simple SLS solvers like WalkSAT [5] and probSAT [4], and later extend it to other more complex solvers like Sparrow [10]. The information needed by WalkSAT and probSAT is identical. Both need the set of unsatisfied clauses under the current assignment and the $break$ value of variables from those clauses. The $break(x)$ value is the number of clauses that become unsatisfied after flipping x. Within their $pickVar$ method, both solvers randomly select an unsatisfied clause and then pick a variable from this clause according to their particular heuristic.

For each variable x in the selected clause, the probSAT solver uses a flip probability proportional to $cb^{-break(x)}$ or, alternatively, to $(break(x) + \epsilon)^{-cb}$. The WalkSAT solver randomly selects a variable x with $break(x) = 0$ if such a variable exists in the selected clause. Otherwise, it picks a variable randomly from selected clause with probability p and the best variable with respect to the $break$ value with probability $1 - p$.

In general, SLS solvers keep track of the transition of clauses from different satisfaction states. By applying an assignment α to the formula F, denoted by $F|\alpha$, we can categorize clauses according to their satisfaction status. If a clause has t true literals, we say that the clause is t-satisfied.

To maintain the set of unsatisfied clauses $falseClause$, an SLS solver keeps track of the clause transitions from 0-satisfied to 1-satisfied and back. For a variable x, the value of $break(x)$ is equal to the cardinality of the set of 1-satisfied clauses that contain x as a satisfying literal, because by flipping x these clauses will get 0-satisfied (unsatisfied). Further, $break(x)$ changes if and only if any of these clauses become 0-satisfied or 2-satisfied.

The $break$ value of variables can be either maintained incrementally (also called caching) or computed in every iteration (also called non-caching). The original implementation of WalkSAT and of probSAT uses the caching scheme while the WalkSAT implementation within the UBCSAT framework [11] is using the non-caching implementation.

To monitor the satisfaction status of clauses, SLS solvers use an array $trueLit$ that stores for each clause the number of true literals and maintains it incrementally. Besides the before mentioned data structures, an SLS solver additionally needs the occurrence list for each literal, which is the set of clauses where a literal

occurs. Further, for each 1-satisfied clause, also the satisfying variable is stored (this is also known as the critical variable $critVar$, or as the $watch1$ scheme).

Having introduced these data structures, we can now describe the standard implementation [1] of SLS solvers that use caching of the *break* value and of $falseClause$. The $flip(var)$ method using caching is described in Algorithm 1.

Algorithm 1: Variable flip including caching (see also [1]).

```
   Input: Variable to flip v
 1 α[v] = ¬α[v];                                   /* change variable value */
 2 satisfyingLiteral = α[v] ? v : ¬v;
 3 falsifyingLiteral = α[v] ? ¬v : v;
 4 for clause in occurrenceList[satisfyingLiteral] do
 5     if numTrueLit[clause] == 0 then             /* transition 0 → 1 */
 6         remove clause from falseClause ;
 7         break[v] + +;
 8         critVar[clause] = v
 9     else
10         if numTrueLit[clause] == 1 then         /* transition 1 → 2 */
11             break[critVar[clause]]--;
12     numTrueLit[clause]++;
13 for clause in occurrenceList[falsifyingLiteral] do
14     if numTrueLit[clause] == 1 then             /* transition 0 ← 1 */
15         add clause to falseClause ;
16         break[v]--;
17         critVar[clause] = v
18     else
19         if numTrueLit[clause] == 2 then         /* transition 1 ← 2 */
20             for var in clause do          /* find the critical variable */
21                 if var is satisying literal in clause then
22                     critVar[clause]=var;
23                     break[var]++;
24     numTrueLit[clause]--;
```

Except for transition $1 \leftarrow 2$, all others have constant complexity. Whenever the number of satisfied literals decreases from two to one, we have to search for the critical variable in the clause, thus rendering the complexity of this transition to be $O(len(C))$, where $len(C)$ denotes the length of the clause. This is the previous state-of-the-art and was first toroughly analyzed in [1].

With a simple trick, first introduced in the CDCL solver NanoSAT [12], we can reduce the complexity of this step to O(1). Instead of storing the critical variable for each clause, we store the XOR concatenation of all satisfying variables $trueVarX$. As alternative one can maintain the sum of the literals, using

addition and subtraction during updates, while for the XOR scheme only the XOR operation is needed.

If there is only one satisfying variable per clause, $trueVarX(clause)$ will contain this variable. If there are two satisfying variables in a clause C, i.e. $trueVarX[C] = x_i \oplus x_j$, then we can obtain the second variable in constant many steps if we know the first variable. This is the case in the transition $1 \leftarrow 2$. Consequently, we can obtain x_j by removing x_i from $trueVarX[C]$, i.e., $x_j = x_i \oplus trueVarX[C]$. This new flip method is described in Algorithm 2

Algorithm 2: Variable flip with XOR caching

```
   Input: Variable to flip v
 1 α[v] = ¬α[v];                                   /* change variable value */
 2 satisfyingLiteral = α[v] ? v : ¬v;
 3 falsifyingLiteral = α[v] ? ¬v : v;
 4 for clause in occurrenceList[satisfyingLiteral] do
 5     if numTrueLit[clause] == 0 then             /* transition 0 → 1 */
 6         remove clause from falseClause ;
 7         break[v] + +;
 8         trueVarX[clause] = 0
 9     else
10         if numTrueLit[clause] == 1 then         /* transition 1 → 2 */
11             break[trueVarX[clause]]--;
12     numTrueLit[clause]++;
13     trueVarX[clause] ⊕= v;
14 for clause in occurrenceList[falsifyingLiteral] do
15     trueVarX[clause] ⊕= v;
16     if numTrueLit[clause] == 1 then             /* transition 0 ← 1 */
17         add clause to falseClause ;
18         break[v]--;
19     else
20         if numTrueLit[clause] == 2 then         /* transition 1 ← 2 */
21             break[trueVarX[clause]]++;
22     numTrueLit[clause]--;
```

The XOR scheme can be used in all types of solvers which need the break value and compute it incrementally. This is the case for almost all SLS solvers. Note, in CDCL solvers lazy data structures [13,14] have proven to be superior to schemes based on counting assigned literals. Counting, as well as the XOR scheme, both need full occurrence lists and their traversal for every assignment is too costly for those long clauses learned in CDCL [15].

To show the practical relevance of this implementation improvement, we have implemented the XOR scheme in probSAT and also in the way more complex SLS solver Sparrow.

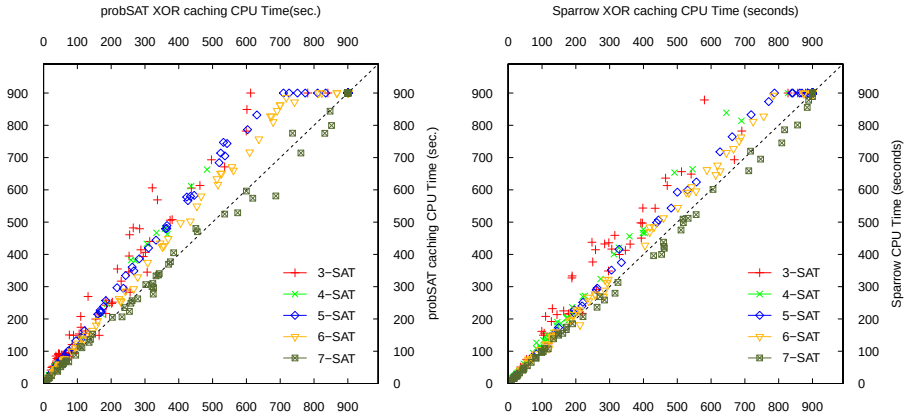

Fig. 1: Scatter plot between the caching implementation of probSAT (left side) and Sparrow (right side) and their respective XOR caching implementation on the set of random problems from the SAT Challenge 2012.

We have evaluated the standard and the XOR implementation on the random and hard combinatorial benchmarks of the SAT Challenge 2012 [3]. We have opted to use this benchmark set instead of the latest, because it contains more benchmarks and is designed for lower run times of 900 seconds. All solvers have been started with the same seed, i.e. their implementation variants are semantically identical and produce the exact same search trace. The only difference is the complexity of a search step, which will cause the runtime to vary.

The results of our evaluations on the random instances can be seen in Fig. 1. The solvers were evaluated on the same hardware as the SAT Challenge and we used a cutoff of 900 seconds. In most cases, the XOR implementation is faster than the standard implementation. The quantity of improvement of the XOR implementation over the standard implementation seems to be in inverse proportional to k (the clause length of the problem in uniform randomly generated problems), i.e. the best improvement could be achieved for 3-SAT problems, while the XOR implementation is actually slower for 7-SAT problems.

This phenomenon can be explained by analyzing the number of clause transitions from 2-satisfied to 1-satisfied. The more transitions of this kind take place during search, the better the XOR implementation, because this is the step where the XOR implementation reduces the complexity. During search on 3-SAT problems, almost 30% of the transitions are of the type $2 \rightarrow 1$. On 7-SAT problems, only 5% are of this kind. Further, the occurrence list of a variable in 7-SAT problems contains on average 600 clauses, which means that we have to perform as many XOR operations to update the $trueVarX$ values. The overhead introduced by the XOR operations cannot be compensated by constant complexity of the $2 \rightarrow 1$ transition, which occurs rarely.

[3] `http://baldur.iti.kit.edu/SAT-Challenge-2012/`

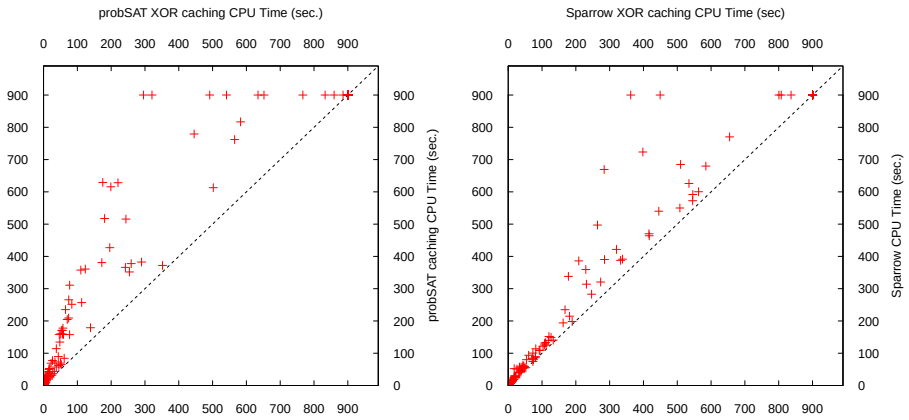

Fig. 2: Scatter plot between the caching implementation of probSAT (left side) and Sparrow (right side) and their respective XOR caching implementation on the set of hard combinatorial problems from the SAT Challenge 2012.

Hard combinatorial problems and structured problems are known to contain many $2 \rightarrow 1$ transitions [1]. We evaluated the different implementations on the hard combinatorial satisfiable problems from the SAT Challenge 2012. The XOR implementation of probSAT and of Sparrow is always faster than their standard implementation as can be seen from Fig. 2. Moreover, the XOR implementation of probSAT and Sparrow are able to solve four and six additional instances, respectively. The speed-up with respect to solving time was on some instances more than a factor of two. During the search on these instances, almost 50% of the steps were $2 \rightarrow 1$ transitions.

3 Incorporating multilevel break

Most modern SLS solvers use the *make* and *break* value of variables or the combination of both, e.g. given by the *score*. For a given variable x, the *score* is defined by $score(x) := make(x) - break(x)$. Solvers like probSAT or WalkSAT only use the *break* value within their decision heuristic. Within the non-caching implementations of theses solvers, the *break* value is computed from scratch. If we want to compute the *break* value of a literal l, occuring within a non-satisfied clause, we have to traverse the occurrence list of $\neg l$ (the clauses that contain l as a satisfying literal) and count the number of clauses with $numTrueLit = 1$. The *break* value is defined as the number of clauses that change their satisfiability status from 1-satisfied to 0-satisfied. In the following, we will also refer to this *break* value as the $break_1$ value. Accordingly, we can define the $break_l$ value of a variable x as the number of clauses that are l-satisfied and will become $l-1$ satisfied when x is flipped. The $make_l$ value can be defined similarly. Within the non-caching implementations, we can compute the $break_l$ value with little

overhead. Instead of counting only the number of 1-satisfied clauses, we also count the number of l-satisfied clauses.

The question that arises now, is how to integrate these $break$ values into the heuristic of the solvers. In [2], the $make_1$ and $make_2$ values were used in the form $lmake = w_1 \cdot make_1 + w_2 \cdot make_2$. The $lmake$ value was used as a tie breaker in the WalkSATlm solver. In [3] the $score_2 = make_2 - break_2$ value is being used in the solver CScoreSAT within the score function $score = score_1 + \lfloor score_2 \rfloor$.

Within the probSAT solver, new variable properties can be included quite easily by incorporating the property in the probability distribution. We propose the following inclusion of higher level $break$ values into the probability distribution function of probSAT:

$$p(x) = cb^{-break(x)} \quad \longrightarrow \quad p(x) = \prod_l {cb_l}^{-break_l(x)}$$

$$p(x) = (1 + break(x))^{-cb} \quad \longrightarrow \quad p(x) = \prod_l (1 + break_l(x))^{-cb_l}$$

The constants cb_l specify the influence of the $break_l$ value within the probability distribution. To figure out which values to take for the cb_l variables, we used an automated algorithm configurator. More specifically, we used a parallel version of the SMAC configurator [16], which is implemented in the EDACC [17] configuration framework. The instances of interest are the random k-SAT problems ($k \in \{3, 5, 7\}$), resulting in three scenarios. For each scenario, we have performed two types of configuration experiments. First, we allowed the configuration of the cb_1 and cb_2 parameters (i.e. only include the $break_1$ and $break_2$ value). In the second experiment, we allowed the configuration of all cb_l. For the configuration instances, we have used the train sets of the benchmark sets used in [18] and [4], which are two independent set of instances. Each set contains 250 instances of 3-SAT ($n = 10.000$, $r = 4.2$), 5-SAT ($n = 500$, $r = 20$) and 7-SAT ($n = 90$, $r = 85$) problems. Each configuration scenario was allowed to use up to $5 \cdot 10^5$ seconds and we optimized the PAR10 statistics which measures the average runtime and counts unsuccessful runs with ten times the time limit.

3.1 Configuration of $break_l$

For 3-SAT problems, the configurator reported a cb_2 value of one, meaning that it could not improve upon the default configuration, which is not using the $break_2$ value. For the 5-SAT problems, the best configuration had $cb_1 = 3.7$, $cb_2 = 1.125$, and for 7-SAT problems it was $cb_1 = 5.4$, $cb_2 = 1.117$. The low values of cb_2, when compared to those of cb_1 shows that the $break_2$ values have a lower importance than $break_1$ or that the $break_2$ values are considerably larger than those of $break_1$. This motivates the analysis of the $break_2$ values within the search of the solver.

Similar as in the configuration of $break_2$, for 3-SAT problems the configurator could not find better configurations than the default configuration (that ignores

the $break_2$ and $break_3$ values). The best configuration found for 5-SAT problem had the parameters:

$$cb_1 = 3.729 \quad cb_2 = 1.124 \quad cb_3 = 1.021 \quad cb_4 = 0.990 \quad cb_5 = 1.099$$

The best configuration for 7-SAT had the parameters:

$$cb_1 = 4.596 \;\; cb_2 = 1.107 \;\; cb_3 = 0.991 \;\; cb_4 = 1.005 \;\; cb_5 = 1.0 \;\; cb_6 = 1.0 \;\; cb_7 = 1.0$$

The value of the constants seems to be very low, but the high $break_l$ values that occur during search probably require such low cb_l values.

3.2 Results

We compare the $break_2$ and the $break_l$ implementations with the XOR implementation of probSAT and with the solvers WalkSATlm [2] and CScoreSAT [3], two solvers that established the latest state-of-the-art results in solving 5-SAT and 7-SAT problems. The binaries of the latter two solvers were the ones submitted to the SAT Competition 2013. We evaluate the solvers on the Competition benchmarks from 2011, 2012 and 1013, which all together represent a very heterogeneous set of instances, as they were generated with different parameters. An additional benchmark set of randomly generated 5-SAT problems with $n = 4000$ and a clause to variable ratio of $r = 20$ shall show where the limits of our new approach lies. The results of our evaluation can be seen in Tab. 1.

	probSAT$_x$		probSAT$_2$		probSAT$_l$		WalkSATlm		CScoreSAT	
	#sol.	time	#sol.	time	#sol.	time	#sol.	time	#sol.	time
SC11-5-SAT	32	333s	40	34s	**40**	**19s**	39	108s	39	118s
SC12-5-SAT	67	578s	103	246s	**107**	**207s**	82	427s	77	465s
SC13-5-SAT	7	809s	5	836s	**7**	**808s**	6	817s	5	824s
5sat4000	0	900s	36	470s	**41**	**382s**	8	827s	0	900s
SC11-7-SAT	8	721s	10	521s	12	495s	11	571s	**14**	**437s**
SC12-7-SAT	49	633s	67	548s	69	488s	66	520s	**73**	**462s**
SC13-7-SAT	19	666s	17	671s	**20**	**652s**	16	673s	18	652s

Table 1: The evaluation results on different 5-SAT and 7-SAT benchmarks sets. For each solver the number of solved instances (#sol.) and the average run time (time) is reported (unsuccessful runs are also counted in the run time). Bold values represents the best achieved results for that particular instance class.

By only incorporating the $break_2$ values in the probability distribution improves the performance of the probSAT solver considerably. On the SC12-5-SAT instances probSAT$_2$ is almost twice as fast as probSAT in terms of solved instanced and also in terms of runtime. It also dominates the WalkSATlm solver, which was shown to be the state-of-the-art solver for 5-SAT problems in [2]. The

	$break_1$		$break_2$		$break_3$		$break_4$		$break_5$		$break_6$		$break_7$	
k-SAT	th.	pr.	th.	pr.	th.	pr.	th.	pr.	th.	pr.	th.	pr.	th.	pr.
3-SAT	1.6	1.7	3.2	3.7	1.6	1.6								
4-SAT	2.4	2.2	7.1	7.4	7.1	7.2	2.4	2.3						
5-SAT	3.1	2.8	12.5	13.1	18.6	19.5	12.5	12.7	3.1	3.2				
6-SAT	3.9	3.6	19.7	19.4	39.4	38.9	39.4	38.9	19.7	19.5	3.9	3.9		
7-SAT	4.6	4.5	27.9	28.8	69.7	72.2	93.0	96.7	69.7	72.9	27.9	29.2	4.6	4.9

Table 2: The theoretical (th.) expected number of l-satisfied clauses that contain an arbitrary variable x in a randomly generated k-SAT formula and the average $break_l$ values (pr.) encountered during search for variables within a randomly picked unsatisfied clause (value in brackets).

probSAT$_2$ solver achieves better results also on the 7-SAT instances. One exception are the SC13 problems, that have been generated exactly on the threshold. For these problems probSAT$_2$ is worse than probSAT.

By allowing also higher level break values to influence the probability distribution further improvements can be achieved. The probSAT$_l$ solver achieves the best results on all 5-SAT and on SC13-7-SAT problems. On the remaining 7-SAT problems it is only slightly worse than the CScoreSAT solver but still better than the WalkSATlm solver.

It is also worth mentioning that our non-caching implementation is on average about 30% slower than the caching implementation of the WalkSATlm and CScoreSAT solver in terms of flips per second performed by the solver.

3.3 Theoretical distribution of $break_l$ values

Given a randomly generated formula with constant clause length k and $m = r_k n$ many clauses (where the ratios $r_3 = 4.2$, $r_4 = 9.5$, $r_5 = 20$, $r_6 = 42$, and $r_7 = 85$ have been used) we are interested in the distribution and the expectation of the $break_l$ value for a given variable. The probability that a random clause contains this variable (in the correct polarization) is $k/(2n)$. Furthermore, for having l satisfied literals in such a random clause we need that among the remaining $k-1$ literals exactly $l-1$ of them are satisfied. This happens with probability $\binom{k-1}{l-1}/2^{k-1}$. Therefore, the $break_l$ value is binomially distributed as $Bin(m,p)$ where $m = r_k n$ and $p = \frac{k}{2n} \cdot \frac{\binom{l-1}{k-1}}{2^{k-1}}$. The expectation of the $break_l$ value is

$$r_k n \cdot \frac{k}{2n} \cdot \frac{\binom{l-1}{k-1}}{2^{k-1}} = \frac{r_k \cdot k \cdot \binom{k-1}{l-1}}{2^k}$$

Table 2 lists these values for common k-SAT problems.

Comparing these two values (the theoretical $break_l$ values and the actually observed values) there is a very good agreement. Interestingly SLS solver take into consideration only the transitions $0 \leftarrow 1$ (break) and $0 \rightarrow 1$ (make). SLS

solvers are optimizers that try to minimize the number of 0-satisfied clauses to zero, and thus only these two transitions play a role.

As we can see from Tab. 2 by far the largest $break_l$ values occur in the middle range, $l \approx k/2$. Therefore, the cb_l values even when they are relatively close to 1.0 should not be disregarded since they are raised to the $break_l$-th power, and so play an important role for the success of the algorithm. These seems to open up opportunities for further algorithmic improvements.

4 Clause selection

While there has been lots of work on different variable picking schemes, comparatively few work has considered clause selection so far. In a certain way, clause weighting schemes [6,7,8] in combination with GSAT algorithms [5] relate to this because they influence the likelihood of a clause to become satisfied in the current step. However, this effect is rather indirect, and when it comes to WalkSAT algorithms, most implementations simply select a clause randomly.

We propose to question this selection and analyze several alternative schemes. For this, it is important to be aware of the data structures and algorithms that are used to save possible candidate clauses (i.e. the unsatisfied ones in a *falseClause* container). In most implementations, all unsatisfied clauses are stored in a list which is then updated in each iteration. The update procedure consists of removing newly satisfied clauses (the *make-step*) and adding newly unsatisfied clauses (the *break-step*).

The list itself is usually implemented as an array with a non-fixed length m'. If a new clause is added in the break-step, m' is increased and the clause is simply put at the last position. Whenever a clause is removed in the make-step, the element from the last position is used to replace it and m' is decreased. This is an easy but very efficient implementation used in most SLS solvers. To select a random clause during the clause selection phase, one can use a random number $r \in \{0, \ldots, m'\}$ and choose the clause at index r. We will call this approach RS, which is short for *random selection*.

One alternative way to select a clause is by using the current flip count j instead of choosing a clause at a random index. This was first proposed in [9, p. 93] and was the standard implementation of probSAT in the SAT Competition 2013. This version of probSAT selects the clause at index $j \bmod m'$ in each step. Interestingly, this version of probSAT, denoted with *pseudo breadth first search* (PBFS), performed much better than the original one from [4] (c.f. Fig. 3) which used random selection. Note, that under the (unrealistic) assumption of using real random numbers, this almost trivial change actually renders the original state-less implementation of probSAT to rely on the search history now, more precisely on the number of flipped clauses sofar.

Considering the success of DLS solvers compared to GSAT, it is not surprising that clause selection policies can have a big influence on the performance of WalkSAT solvers as well. Nevertheless, it is not fully clear why the particular

approach of using the flip count provides this remarkable increase in performance. We therefore analyze this heuristic in more detail.

It is easy to see, that the candidate clauses are traversed in the same order in which they are contained in the array. As already described, new clauses are added to the end of the list in the break-step. This approach is somehow similar to the behavior of a queue. On the other hand, it is obviously not a real queue because the flip count is used for clause selection. Also, a certain shuffling takes place whenever clauses are removed in the make-step.

As long as a clause is selected randomly, the order of the candidate list does not matter. For PBFS, the situation is quite different. Once we start selecting clauses according to their position in the array, where exactly they are put becomes important. For example, there is a difference in whether the make-step or the break-step is performed first. Assume the list of unsatisfied clauses includes $[C_0, C_1, C_2]$ in the given order in iteration 0. C_0 will be selected. Further, assume that by flipping a variable in C_0, both C_0 and C_1 will get satisfied while previously satisfied clauses C_3, C_4, and C_5, will become unsatisfied. If the make-step is performed first, the list will change to $[C_2]$ and then to $[C_2, C_3, C_4, C_5]$ after the break-step. Therefore, C_3 will be selected in iteration 1 while C_2 is "skipped". If the break-step is performed first, the list will contain $[C_0, C_1, C_2, C_3, C_4, C_5]$ and finally become $[C_5, C_4, C_2, C_3]$ after the make-step. This leads to C_4 being selected in iteration 1. Obviously, this will also have an influence on the selection process in all following iterations.

To distinguish between the different order of make-step and break-step, we will use indices *mf* and *bf* to denote *make-first* and *break-first*, respectively. If the index is omitted, we refer to the make-first implementation. In Fig. 3, the performance of PBFS_{mf} and PBFS_{bf} is compared. We can see that PBFS_{mf} outperforms PBFS_{bf} significantly. Actually, PBFS_{bf} is even slower than RS.

Again, it is not clear why this is the case. Adding clauses to the end in combination with the use of the modulo operator makes it hard to predict the effect on the clause selection order. The closer the index is to the end of the array, the sooner a newly added clause will be selected. Shuffling clauses in the break-step obfuscates the order of the clauses even more. However, better understanding the cause for this difference in performance might help us to further improve clause selection heuristics.

One of our conjectures was that maybe PBFS_{mf} resembles a true *breadth first search* (BFS) more closely, leading to the increased performance. We therefore implemented a real queue in our solver. The result of this BFS approach is also shown in Fig. 3. Interestingly, BFS performs much worse than PBFS_{mf} and even worse than RS and PBFS_{bf}. Apparently, the conjecture does not hold. Just for reference, we also implemented a stack for the clauses in order to simulate a *depth first search* (DFS). As expected, DFS performs very poorly and worse than all other approaches. This confirms that, although pure BFS does not work out, some kind of breadth first search seems to be beneficial.

Since a true BFS is too strict, we decided to use a modification inspired by the example given in the context of comparing PBFS_{mf} and PBFS_{bf}. As pointed

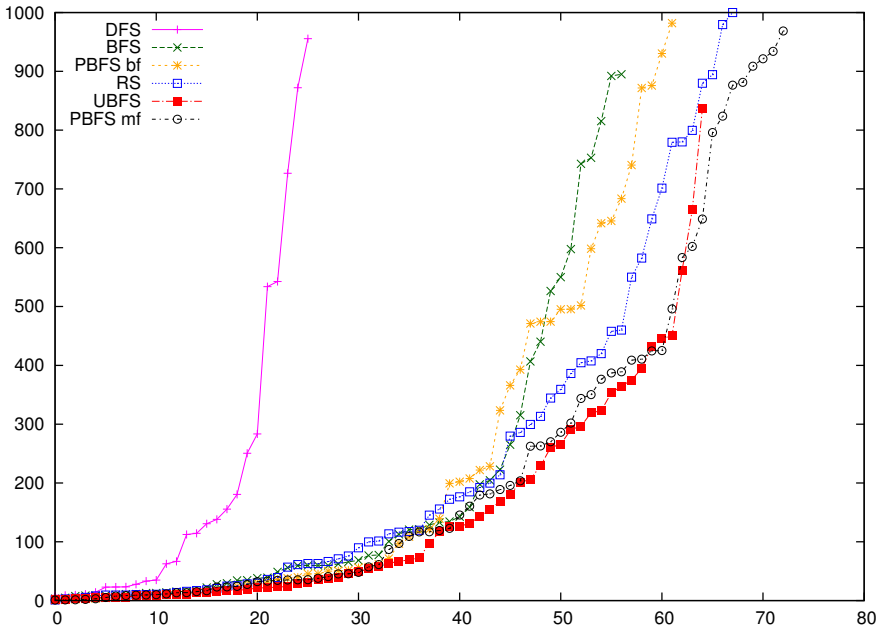

Fig. 3: Cactus plot showing the performance of a faithful reimplementation of probSAT in our internal Yals SAT solver with various different clause selection heuristics on the satisfiable instances from the SAT Competition 2013 hard combinatorial track (cutoff time ist 1000 seconds). The X-Axis and Y-Axis represent the number of solved instances and the runtime (in seconds), respectively.

out in connection with the (better-performing) make-first version, clauses in this kind of implementation sometimes are "skipped", i.e. they will only be visited in the next full cycle. We simulated this in an implementation called *unfair breadth first search* (UBFS) in the following way. The list of clauses is still saved in a queue. When the first element is touched, we select it with probability p_u. With probability $1 - p_u$, it is moved to the end of the queue and the second clause is picked instead. For our experiments, we set $p_u = \frac{1}{2}$. In Fig. 3, we can see that this version performs much better than the previous queue implementations and already comes close to the results of PBFS.

Finally, an additional remark considering the efficiency of the different data structures. While it is easy to implement the array structure for RS and PBFS efficiently, more care has to be taken when implementing a queue for the other approaches. In our implementation we used a dedicated memory allocator and moving garbage collector for defragmentation. This achieves, roughly the same time efficiency as using the original array structure, considering average flips per second but needs slightly more memory.

Our results show that clause selection can have a large impact on the performance of WalkSAT solvers. There are still plenty of open questions. For example, it is not clear yet which is the optimal value for p_u. Also, it might be interesting to combine UBFS-like heuristics with clause weights, assigning different probabilities to clauses depending on different properties, such as its variable score distribution, the number of flips since the last time it was touched, the number of times it has been satisfied, or its length for non-uniform problems.

5 Summary and Future Work

We started with a detailed analysis of standard SLS implementations. This analysis revealed that there is room for improvement for the caching variants and that the $break_l$ property can be computed cheaply in the non-caching variants.

For caching implementations, we have shown that we can speed up the flip procedure of SLS solvers by using an XOR implementation as done in NanoSAT. This approach provides a great flexibility because it can be applied to all SLS solvers that use the *break* value within their heuristics and compute it in an incremental way. Our experimental results showed that the XOR implementation should be used for random k-SAT problems (except for 7-SAT problems) and especially for structured problems. For future work, it will be interesting to look at non-uniform problems more closely. As we have seen from random SAT, the clause length is the crucial factor for determining whether an XOR implementation helps to increase performance. In non-uniform formulas, it might be beneficial to combine XOR implementations with common ones and benefit from both advantages. For short clauses the XOR scheme should be used, while for long clauses the classical ones.

Within the non-caching implementations, it turned out that the multilevel $break_l$ value can be computed cheaply. By incorporating $break_l$ into the probSAT solver, we were able to discover better heuristics for 5-SAT and 7-SAT problems that establish new state-of-the-art results, especially on 5-SAT problems. We further extended our practical results by giving a detailed theoretical analysis regarding the distribution of different $break_l$ values in random formulas, providing a better understanding of their individual impact.

While most WalkSAT implementations simply select a random clause, we proposed several alternative ways. Our experimental results clearly show that selection heuristics influence the performance and picking a clause randomly is not the best choice. For structured problems the PBFS scheme is currently the best one and can be implemented very easily. Analyzing the impact of clause selection heuristics in more detail is a another promising research direction for future work. For example, clause weights might be one possible way to further improve the quality of clause selection heuristics.

Acknowledgments We would like to thank the BWGrid [19] project for computational resources. This work was supported by the Deutsche Forschungsgemeinschaft (DFG) under the number SCHO 302/9-1, and by the Austrian Science Fund (FWF) through the national research network RiSE (S11408-N23).

References

1. Fukunaga, A.S.: Efficient implementations of SAT local search. In: Proc. SAT'04. (2004)
2. Cai, S., Su, K., Luo, C.: Improving WalkSAT for random k-satisfiability problem with $k > 3$. In: Proc. AAAI'13. (2013)
3. Cai, S., Su, K.: Comprehensive score: Towards efficient local search for SAT with long clauses. In: Proc. IJCAI'13. (2013)
4. Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In: Proc. SAT'12. (2012) 16–29
5. McAllester, D.A., Selman, B., Kautz, H.A.: Evidence for invariants in local search. In: Proc. AAAI/IAAI'97. (1997) 321–326
6. Wu, Z., Wah, B.W.: An efficient global-search strategy in discrete lagrangian methods for solving hard satisfiability problems. In: Proc. AAAI/IAAI'00. (2000) 310–315
7. Hutter, F., Tompkins, D.A.D., Hoos, H.H.: Scaling and probabilistic smoothing: Efficient dynamic local search for SAT. In: Proc. CP'02. (2002) 233–248
8. Thornton, J., Pham, D.N., Bain, S., Jr., V.F.: Additive versus multiplicative clause weighting for SAT. In: Proc. AAAI'04. (2004) 191–196
9. Balint, A.: Engineering stochastic local search for the satisfiability problem. PhD thesis, Ulm University (2013)
10. Balint, A., Fröhlich, A.: Improving stochastic local search for SAT with a new probability distribution. In: Proc. SAT'10. (2010) 10–15
11. Tompkins, D.A.D., Hoos, H.H.: UBCSAT: An implementation and experimentation environment for SLS algorithms for SAT & MAX-SAT. In: Proc. SAT'04. (2004)
12. Biere, A.: The evolution from limmat to NanoSAT. Technical Report 444, Dept. of Computer Science, ETH Zürich (2004)
13. Zhang, H.: Sato: An efficient propositional prover. In: Proc. CADE'97. (1997) 272–275
14. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. DAC'01. (2001) 530–535
15. Biere, A.: PicoSAT essentials. JSAT **4**(2-4) (2008) 75–97
16. Hutter, F., Hoos, H.H., Leyton-Brown, K.: Sequential model-based optimization for general algorithm configuration. In: Proc. LION'11. (2011) 507–523
17. 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: Proc. LION'11. (2011) 586–599
18. Tompkins, D.A.D., Balint, A., Hoos, H.H.: Captain jack: New variable selection heuristics in local search for SAT. In: Proc. SAT'11. (2011) 302–316
19. BWGrid: (http://www.bw-grid.de), member of the German D-Grid initiative, funded by the Ministry for Education and Research (Bundesministerium für Bildung und Forschung) and the Ministry for Science, Research and Arts Baden-Württemberg (Ministerium für Wissenschaft, Forschung und Kunst Baden-Württemberg).