On the Effect of Learned Clauses on Stochastic Local Search

Jan-Hendrik Lorenz(✉) and Florian Wörz(✉)

Institute of Theoretical Computer Science, Ulm University, 89069 Ulm, Germany
{jan-hendrik.lorenz,florian.woerz}@uni-ulm.de

Abstract. There are two competing paradigms in successful SAT solvers: Conflict-driven clause learning (CDCL) and stochastic local search (SLS). CDCL uses systematic exploration of the search space and has the ability to learn new clauses. SLS examines the neighborhood of the current complete assignment. Unlike CDCL, it lacks the ability to learn from its mistakes. This work revolves around the question whether it is beneficial for SLS to add new clauses to the original formula. We experimentally demonstrate that clauses with a large number of correct literals w. r. t. a fixed solution are beneficial to the runtime of SLS. We call such clauses *high-quality* clauses.

Empirical evaluations show that short clauses learned by CDCL possess the high-quality attribute. We study several domains of randomly generated instances and deduce the most beneficial strategies to add high-quality clauses as a preprocessing step. The strategies are implemented in an SLS solver, and it is shown that this considerably improves the state-of-the-art on randomly generated instances. The results are statistically significant.

Keywords: Stochastic Local Search · Conflict-Driven Clause Learning · Learned clauses

1 Introduction

The *satisfiability problem* (*SAT*) asks to determine if a given propositional formula F has a satisfying assignment or not. Since Cook's NP-completeness proof of the problem [21], SAT is believed to be computationally intractable in the worst case. However, in the field of applied SAT solving, there were enormous improvements in the performance of SAT solvers in the last 20 years. Motivated by these significant improvements, SAT solvers have been applied to an increasing number of areas, including bounded model checking [15,19], cryptology [23], or even bioinformatics [35], to name just a few. Two algorithmic paradigms turned out to be especially promising to construct solvers. The first to mention

The authors acknowledge support by the state of Baden-Württemberg through bwHPC.

F. Wörz—Supported by the Deutsche Forschungsgemeinschaft (DFG).

L. Pulina and M. Seidl (Eds.): SAT 2020, LNCS 12178, pp. 89–106, 2020.
https://doi.org/10.1007/978-3-030-51825-7_7

is *conflict-driven clause learning* (*CDCL*) [13,36,39]. The CDCL procedure systematically explores the search space of all possible assignments for the formula F in question, by constructing partial assignments in an exhaustive branching and backtracking search. Whenever a conflict occurs, a reason (a conflict clause) is learned and added to the original clause set [16,42]. The other successful paradigm is *stochastic local search* (see [16, Chapter 6] for an overview). Starting from a complete initial assignment for the formula, the neighborhood of this assignment is explored by moving from assignment to assignment in the space of solution candidates while trying to minimize the number of unsatisfied clauses by the assignment (or some other criterion). This movement is usually performed by *flipping* one variable of such a complete assignment. Both paradigms are described in Sect. 2 in more detail.

Besides the difference in the completeness of assignments considered during a run of those algorithms, another major difference between both paradigms is the completeness and incompleteness of the solvers (i. e., being able to certify both satisfiability and unsatisfiability, or not) [42]. CDCL solvers are complete, while SLS algorithms are incomplete. More interestingly for practitioners, perhaps, is the complimentary behavior these paradigms exhibit in terms of performance: CDCL seems well suited for application instances, whereas SLS excels on random formulas[1]. An interesting question thus is, if it is possible to combine the strength of both solvers or to eliminate the weaknesses of one paradigm by an oracle-assistance of the other. This challenge was posed by Selman et al. in [43, Challenge 7] (and again later in [30]):

> "Demonstrate the successful combination of stochastic search and systematic search techniques, by the creation of a new algorithm that outperforms the best previous examples of both approaches."

This easy to state question turns out to be surprisingly challenging. There have been some advances towards this goal, as we survey below. However, the performance of most algorithms that try to combine the strength of both paradigms, so-called *hybrid solvers*, is far from those of CDCL solvers (or other non-hybrids), especially on application instances [2] or even any wider range of benchmark problems [30]. In [27], the DPLL algorithm Satz [34] was used to derive implication dependencies and equivalencies between literals in WalkSAT [37].

The effect of a restricted form of resolution to clause weighting solvers was investigated in [1]. A similar approach was previously studied in [18], where new resolvent clauses based on unsatisfied clauses at local minima and randomly selected neighboring clauses are added.

Local Search over partial assignments instead of complete assignments extended with constraint propagation was studied in [29]. In case of a conflict, a conflict clause is learned, and local search is used to repair this conflict. A similar approach to construct a hybrid solver using SLS as the main solver and a

[1] It is, however, noteworthy that the winning solver in the random track of the SAT Competition 2018 was Sparrow2Riss, a CDCL solver.

complete CDCL solver as a sub-solver was also studied in [4,7], where the performance of the solvers HYBRIDGM, HYBRIDGP, and HYBRIDPP was empirically analyzed. The idea of these solvers is to build a partial assignment around one complete assignment from the search trajectory of the SLS solver. This partial assignment can then be applied to the formula resulting in a simpler one, which is solved by a complete CDCL solver.

A shared memory approach for multi-core processor architectures was proposed in [32]. In this case, DPLL can provide guidance for an SLS solver, being run simultaneously on a different core.

The solver HBISAT, introduced in [24], uses the partial assignments calculated by CDCL to initialize the SLS solver. The SLS solver sends unsatisfied clauses to the CDCL solver to either identify an unsatisfiable subformula of those clauses or satisfy them. This approach was later significantly improved by Letombe and Marques-Silva in [33].

Audemard et al. [2] introduced SATHYS, where both components cooperate by alternating between them. E. g., when CDCL chooses a variable to branch, its polarity is extracted from the best complete assignment found by the SLS solver. On the other hand, CDCL helps SLS out of local minima (i. e., CDCL is invoked conditionally in this solver).

Our Contribution. Our hybrid solver GAPSAT differs from the approaches described above in the sense that CDCL is used as a preprocessor for PROBSAT [11] (an SLS solver), terraforming the landscape in advance. This approach eliminates, in many cases, the possibility for PROBSAT to get stuck in local minima, eliminating the necessity of further, more complicated interactions between both paradigms.

We examine the question, whether it is beneficial for SLS to add new clauses to the original formula in a *preprocessing step*, by invoking a complete CDCL solver. As it turns out, not all additional clauses are created equal. Experimentally, we demonstrate that adding clauses that contain a larger number of correct literals w. r. t. a fixed solution, drastically improves the performance of SLS solvers. However, clauses that only contain few correct literals w. r. t. the fixed solution can be deceptive for SLS. This effect can exponentially increase the runtime as measured in the number of flips of PROBSAT, a very simple SLS solver. In practice, one has to resort to known complete algorithms or proof systems to generate helpful clauses. We, in particular, investigate the effect of new clauses learned by CDCL and depth-limited resolution (Sect. 3). With the help of experiments, we conclude that CDCL (or resolution limited to depth 2) produces distinctively more helpful clauses for PROBSAT than resolution limited to depth 1, as was studied in the past [1]. We, therefore, focus our effort on CDCL as a clause learning mechanism for PROBSAT.

Motivated by these insights, we study the quality CDCL-learned clauses have for SLS in more detail. In training experiments that are described in Sect. 4, we systematically deduce parameter settings by statistical analysis that increase this quality. For example, shorter clauses learned by CDCL are more beneficial. As it turns out, however, the specific width depends on the underlying formula.

Another interesting observation is that the amount of added clauses has to be carefully restricted. Again, the specific restriction to use depends on the underlying formula.

To finally test the concrete effect of these ideas, we compared the performance of a newly designed solver with the winner of the 2018 random track competition, Sparrow2Riss [10]. Our observations were implemented in GapSAT, which forms a combination of Glucose and probSAT. A comprehensive experimental evaluation on 255 instances provides statistical evidence that the performance of our proposed solver GapSAT exceeds Sparrow2Riss' substantially. In particular, GapSAT was able to solve more instances in just 30 s than Sparrow2Riss in 5000 s. We present a summary of our experimental evaluation in Sect. 5.

2 Preliminaries

We briefly reiterate the notions necessary for this work. For a thorough introduction to the field, we refer the reader to [42]. A *literal* over a Boolean variable x is either x itself or its negation $\overline{x}$. A *clause* $C = a_1 \vee \cdots \vee a_\ell$ is a (possibly empty) disjunction of literals a_i over pairwise disjoint variables. A *CNF formula* $F = C_1 \wedge \cdots \wedge C_m$ is a conjunction of clauses. A CNF formula is a *k-CNF* if all clauses in it have at most k variables. An *assignment* α for a CNF formula F is a function that maps some subset of $\text{Vars}(F)$ to $\{0, 1\}$. Given a complete assignment α, the act of changing the truth value of precisely one variable of α is called a *flip*. *Resolution* is the proof system with the single derivation rule $\frac{B \vee x \quad C \vee \overline{x}}{B \vee C}$, where B and C are clauses.

CDCL. CDCL solvers, introduced in [13,36,39], construct a partial assignment. When some clause is falsified by the constructed assignment, the CDCL solver adds a new clause to the original formula F. This clause is a logical consequence of F. A more detailed description of CDCL can be found in [16,40,42]. Modern SAT solvers are additionally equipped with incremental data structures, restart policies [26], and activity-based variable selection heuristics (VSIDS) [39]. In this work, we use the CDCL solver Glucose [3] (based on MiniSat [22]).

probSAT. Contrary to CDCL-like algorithms, algorithms based on *stochastic local search* (*SLS*) operate on complete assignments for a formula F. These solvers are started with a randomly generated complete initial assignment α. If α satisfies F, a solution is found. Otherwise, the SLS solver tries to find a solution by repeatedly flipping the assignment of variables according to some underlying heuristic. That is, they perform a random walk over the set of complete assignments for the underlying formula.

In [11], the probSAT class of solvers was introduced. Over the last few years, probSAT-based solvers performed excellently on random instances: probSAT won the random track of the SAT competition 2013, dimetheus [9] in 2014 and 2016, YalSAT [14] won in 2017. Only recently, in 2018, other types of solvers significantly exceeded probSAT based algorithms. This performance is the reason for choosing probSAT in this study.

The idea behind the solver is that a function f is used, which gives a high probability to a variable if flipping this variable is deemed advantageous. A description of probSAT is given in Algorithm 1. This class of solvers is related to Schöning's random walk algorithm introduced in [41].

Input: Formula F, *maxFlips*, function f
$\alpha :=$ randomly generated complete assignment for F
for $i = 1$ **to** *maxFlips* **do**
  if α *satisfies* F **then return** "satisfiable"
  Choose a clause $C = (u_1 \vee u_2 \vee \cdots \vee u_\ell)$ that is falsified under α
  Choose $j \in \{1, \ldots, \ell\}$ with probability $\frac{f(u_j,\alpha)}{\sum_{u \in C} f(u,\alpha)}$
  Flip the assignment of the chosen variable u_j and update α

Algorithm 1. probSAT without restarts.

In [11], the *break-only-poly-algorithm* with $f(x,\alpha) := \big(\varepsilon + \text{break}(x,\alpha)\big)^{-b}$ was considered for 3-SAT, where $\text{break}(x,\alpha)$ is the number of clauses that are satisfied under α but will be falsified when the assignment of x is flipped. For $k \neq 3$, the *break-only-exp-algorithm* $f(x,\alpha) := b^{-\text{break}(x,\alpha)}$ was studied. Balint and Schöning [11] found good choices for the parameters of these two functions. In this work, we have adopted these parameter settings.

3 The Quality of Learned Clauses

In this section, we investigate the effect logically equivalent formulas have on the SLS solver probSAT. More precisely, we use a formula F as a base and add a set of clauses $S = \{C_1, \ldots, C_t\}$ to F to obtain a new formula $G := F \cup S$. In general, adding new clauses to a formula F does not yield a logically equivalent new formula G.

Thus, we observe two artificial models related to the *backbone* (see, e. g., [31]) and consider the 3-CNF case in the following. The backbone $\mathcal{B}(F)$ are the literals appearing in all satisfying assignments of F. In the first model, each new clause consists of one backbone literal $x \in \mathcal{B}(F)$ and two literals y, z such that their complements are backbone literals, i. e., y and z do not occur in any solution. We call this the *deceptive model*. In the second model, each new clause has one backbone literal and two randomly chosen literals. This is the *general model*.

Figure 1 displays the effect of both models on probSAT. On the left is the deceptive model. Generally, a large number of deceptive clauses have a harmful effect on the runtime of probSAT. That is, the average runtime of probSAT increases exponentially with the number of added clauses.

The right-hand side of Fig. 1 shows the general model. Here, we can observe a strong, positive effect on the behavior of probSAT. The average runtime of probSAT improved by two orders of magnitude by adding 200 new clauses

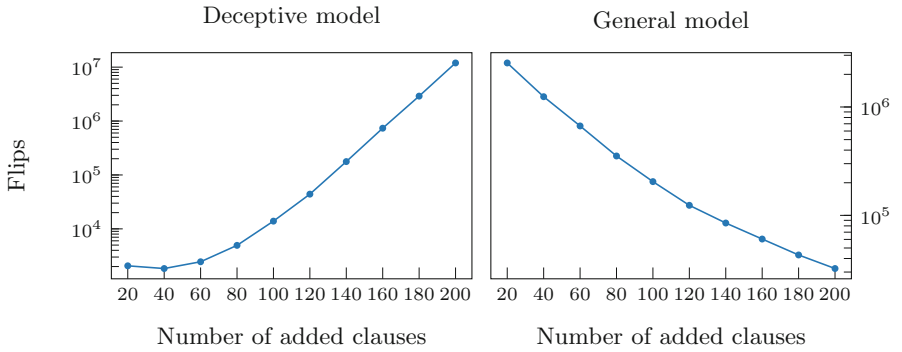

Fig. 1. On the left, the effect of *deceptive* clauses is displayed on an instance with 100 variables and 423 clauses. On the right, the effect of *general* clauses is displayed on an instance with 500 variables and 2100 clauses. The x-axes denote the number of additional clauses, and the y-axes denote the average runtime of 100 runs of probSAT as measured in the number of flips. Both y-axes are scaled logarithmically.

generated by the general model. Even though Fig. 1 depicts the data of only one instance, the general shape of the plot is similar on all tested instances.

Clauses generated by the deceptive model seem to give rise to new local minima which are far away from the solutions. Once probSAT is stuck in such a local minimum, the break-value makes it unlikely that probSAT escapes the region of the local minimum. On the other hand, the prevalence of correct literals in the general model seems to guide probSAT towards a solution. Due to this interpretation, we call clauses that have a high number of correct literals w. r. t. a fixed solution *high-quality clauses*. The view that clauses with few correct literals have a detrimental effect on local search solvers is also supported by the literature [28,42].

From the considerations described above, it should be evident that it is crucial which clauses are added to the formula. Clearly, neither the deceptive nor the general model can be applied to real instances: The solution space would have to be known in advance to generate the clauses. In contrast, approaches like resolution and CDCL can be applied to real instances. All clauses which can be derived by resolution are already implied by the original formula. Accordingly, adding such a clause to the original formula yields a logically equivalent formula. Similarly, clauses learned by a CDCL algorithm can be added to obtain a logically equivalent formula.

In the following, we compare two models based on resolution and one model based on CDCL. In particular, let F be a formula and let $B, C \in F$ be clauses such that there is a resolvent R. We call R a *level 1* resolvent. Secondly, let D, E be clauses such that there is a resolvent S and let D or E (or both) be level 1 resolvents. We call S a *level 2* resolvent. As a representative for CDCL solvers, we use Glucose [3].

Let F be a 3-CNF formula with m clauses. New and logically equivalent formulas F_1, F_2, and F_C are obtained in the following manner.

F_1 Randomly select at most $m/10$ level 1 resolvents of maximum width 4 and add them to F.
F_2 Randomly select at most $m/10$ level 2 resolvents of maximum width 4 and add them to F.
F_C Randomly select at most $m/10$ learned clauses with maximum width 4 from Glucose (with a time limit of 300 s) and add them to F.

The average behavior of probSAT over 1000 runs per instance on the instance types F_1, F_2, and F_C is observed. We use a small testbed of 23 uniformly generated 3-CNF instances with 5000 to 11 600 variables and a clause-to-variable ratio of 4.267. The instances of type F_1 were the most challenging for probSAT; as a matter of fact, F_1 instances were considerably harder to solve than the original instances. On instances of type F_2, probSAT performed better, and on F_C, it was even more efficient. The t-test [44] confirms the observations: F_2 instances are easier on average than F_1 instances ($p < 0.01$), and F_C instances are easier than F_2 instances ($p < 0.05$). Sections 4 and 5 present an in-depth examination of the effect clauses of type F_C have on probSAT.

These results lead us to believe that level 1 clauses are of low quality while level 2 and CDCL clauses are generally of higher quality. It is impractical to confirm this suspicion on uniformly generated instances of the above-mentioned size. Hence, we use randomly generated models with hidden solutions [12] and judge the quality of learned clauses based on the hidden solution.

The SAT competition 2018 incorporated three types of models with hidden solutions. All three types are generated in a similar manner; they just differ in the choice of the parameters. Here, we compare the average quality of the new clauses on each of the three models. For each instance, the set of all level 1, level 2, and CDCL clauses is computed, and the quality is measured w. r. t. the hidden solution.

For the most part, the results confirm the observations from the uniformly generated instances: On all three models, level 2 clauses have a statistically significantly higher quality than level 1 clauses (t-test, all $p < 0.01$). On two of three domains, CDCL clauses have higher quality than level 2 clauses (t-test, both $p < 10^{-5}$), while level 2 clauses have higher quality on the remaining domain (t-test, $p < 10^{-8}$).

As a side note, CDCL is capable of learning unit and binary clauses. Nevertheless, this did not influence the quality of the clauses in any meaningful way: In the 120 test instances, only a single binary and no unit clause was learned.

In conclusion, we conjecture that level 2 and CDCL clauses have higher quality than level 1 clauses. On the uniform random testbed, CDCL performs better than level 2 clauses; also, CDCL clauses have higher quality than level 2 clauses on two of the three hidden solution domains.

4 Training Experiments

In the previous section, we argued that adding supplementary clauses to an instance can have a positive effect on the behavior of probSAT. The focus of this section lies on the question which clauses and how many should be added.

Especially for 3-SAT instances, an initial guess might be adding all clauses acquirable by so-called ternary resolution [17]. Informally speaking, ternary resolution is the restriction of the resolution rule to ternary clauses such that the resolvent is either a binary or ternary clause. Ternary resolution is performed until saturation. In [8], the effect of (amongst other techniques) ternary resolution on another SLS solver, Sparrow (see [6]), is observed. They empirically show that ternary resolution has a negative effect on the performance on satisfiable hard combinatorial instances. Anbulagan et al. [1] study the effect of ternary resolution on uniform random instances. They found that SLS solvers do not benefit from ternary resolution. They even conjecture that ternary resolution has a harmful impact on the runtime of SLS solvers on uniform instances. We performed some experiments on our own and can confirm this suspicion for probSAT. On medium-sized uniform instances, ternary resolution slowed probSAT down by 0.5% on average. As a consequence, we focus on methods to improve the runtime behavior of probSAT with clauses learned by Glucose for the rest of this work.

The supplementary clauses are all learned by Glucose within a 300 second time window; we only distinguish the learned clauses by their width. The number of supplementary clauses is measured in percent of the number of original clauses. To put it differently, we are interested in the maximal length of the new clauses and what percentage of the modified formula should be new clauses. The results of this section are used to configure GapSAT.

Description of Training Experiments

We split the experiments into two phases. In the preliminary phase, promising intervals for the *maximal width* and the *maximal percentage* of new clauses are obtained. In the subsequent phase, the most advantageous parameter combination is sought. Hereafter, we describe the setup of the experiments and their results.

Training Data. We used a set of training instances $\mathcal{C}$, which is assembled as follows: All instances of the SAT Competitions random tracks[2] 2014 to 2017 were gathered. We filtered these instances by proven satisfiability: An instance was added to the training set $\mathcal{C}$ if and only if at least one participating solver showed satisfiability. Since not enough uniform random 3-SAT instances of medium size were in $\mathcal{C}$, we added all instances of this kind from the SAT Competition 2013 as well. In total, $\mathcal{C}$ consists of 377 instances which can be divided into the following three domains:

[2] See http://www.satcompetition.org/. In 2015 there was no random track.

- 120 randomly generated instances with a *hidden* solution [12],
- 149 uniformly generated random 3, 5, and 7-SAT instances of *medium size*. The clause-to-variable ratio is close to the satisfiability threshold [38].
- 108 uniform random 3, 5, and 7-SAT instances of *huge size*, i. e., with over 50 000 variables. The clause-to-variable ratio of each instance is somewhat far from the satisfiability threshold.

Training Setup. The experiments were performed on the bwUniCluster and a local server. Sputnik [45] helped to parallelize the trials. The setup of the computer systems is heterogeneous. Therefore, the runtimes are not directly comparable to another. Consequently, we do not use the runtimes for these experiments. Instead, the number of variable flips performed by probSAT is used, which is a hardware-independent performance measure.

In this section, we use a timeout of 10^9 flips for 3-SAT instances (5-SAT: $5 \cdot 10^8$; 7-SAT: $2.5 \cdot 10^8$). This timeout corresponds to roughly 10 min runtime on medium-sized instances on our hardware. Each instance from $\mathcal{C}$ is run 1000 times for each parameter combination. The primary performance indicator in this section is the number of timeouts per instance. Furthermore, the average *par2* value is sometimes used as a secondary performance indicator. The *par2* value is the *number of flips* if a solution was found or twice the timeout otherwise. For the rest of this work, probSAT refers to probSAT version SC13_v2 [5].

Results of Training Experiments

We conducted a thorough statistical analysis of the data that was obtained in the training experiments described above. We describe our findings in condensed form below. The main part of the remainder of this section is concerned with uniform, medium-sized instances. The results for uniform, huge instances, and instances with a hidden solution are briefly discussed at the end of this section.

3-SAT. We found that adding all clauses up to width 4 that Glucose could find within 300 s is the most beneficial configuration. Not limiting the number of added clauses is in stark contrast to the 5-SAT and 7-SAT cases. For 3-SAT, the relationship between the number of clauses and the performance is explored in Fig. 2. Each blue dot corresponds to one medium-sized 3-SAT instance from $\mathcal{C}$. We compare the average par2 value on the original instance with the average par2 value on the instance with all clauses up to width 4 added. Whenever the blue dot lies below the zero-baseline, then the performance of probSAT on the modified instance was better. The blue line is obtained by linear regression. By its slope, we can tell that, on average, adding more clauses is beneficial. The light blue area denotes the 95% confidence interval, which is calculated by bootstrapping [25]. The confidence interval shows that this relationship is unlikely to be due to chance. We conclude that the number of new clauses should not be limited for 3-SAT instances. On the other hand, the maximal width of the new clauses should be no more than four. Our experiments showed that adding longer clauses deteriorates the performance of probSAT.

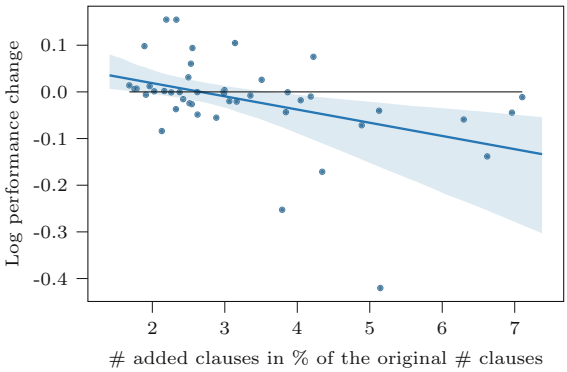

Fig. 2. In this plot, probSAT is compared on the original 3-SAT instances and the modified instances. The number of added clauses of the 3-SAT instances in % is on the x-axis. The y-axis is the logarithm of par2(modified)/par2(orig). The blue line denotes a linear regression fit, and the light blue area is the 95% confidence interval obtained by 1000 bootstrapping steps [25]. (Color figure online)

The left-hand side of Fig. 3 shows the overall performance of probSAT on instances with all clauses up to width 4. The y-axis denotes the difference of timeouts between the modified instance with all width 4 clauses and the original instance. Whenever the dot lies below the zero-baseline, the performance of probSAT was better on the modified instance. The color of the dots stands for the hardness as measured in the average par2 on the original instance. We can see that adding additional clauses has a positive effect, especially on hard instances with 4000 to 9000 variables. Nonetheless, the effect reverses for more than 9000 variables. Overall the results on 3-SAT instances are not statistically significant (t-test, $p = 0.0595$). However, we believe that the main reason for this is the bad performance of probSAT on the modified instances with more than 9000 variables. Furthermore, with a slightly larger sample size, the results might turn out to be statistically significant. We have used these observations in the configuration of GapSAT, as depicted in Fig. 5.

5-SAT. In preliminary experiments, we found that the maximal width of the new clauses should be in the interval $\{7, 8, 9\}$, and the maximal number of new clauses should be at most 15% of the original clauses. The effect of adding more clauses is especially pronounced: Adding more than 15% of the clauses diminished the performance of probSAT dramatically, in contrast to the 3-SAT case where more clauses turned out to be beneficial.

In the detailed phase of the experiments, we found that the best configuration is adding clauses up to width 8 and using a limit of at most 5% of the original clauses. The results of this parameter configuration are shown on the right-hand side of Fig. 3. Again, the performance of probSAT was better on the modified instances if the dot lies below the zero-baseline. The color of the dots describes the hardness of the instance. Overall, the modification has a favorable impact

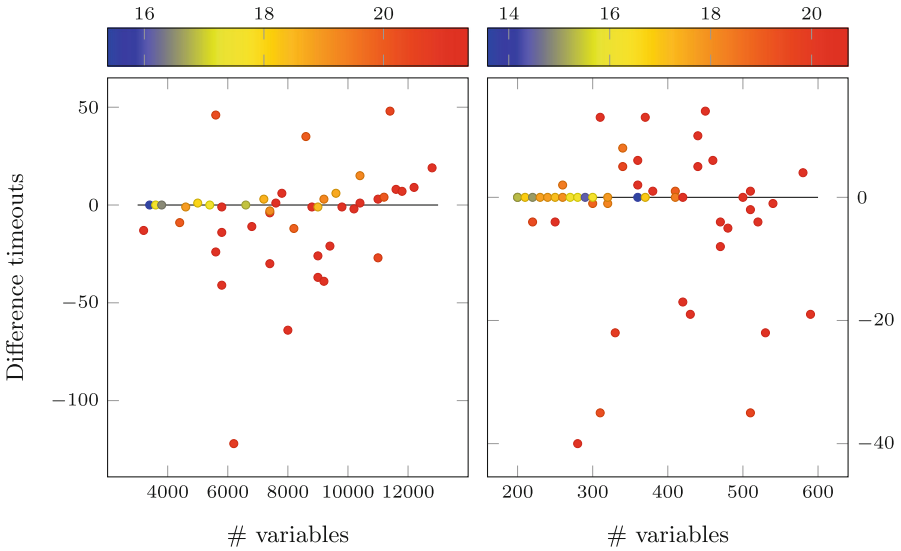

Fig. 3. Scatterplot of the number of variables of the 3-SAT (left) and 5-SAT (right) instance against difference in timeouts between probSAT on the original instance and the instance with the modified strategy. The color encodes the hardness of the original instance as measured in the logarithmic average par2 on 1000 runs of probSAT on the original instances. (Color figure online)

on the performance of probSAT over the full domain. The effect is statistically significant (t-test, $p = 0.0348$). Also, it appears to be increasing as the number of variables increase. However, we did not further investigate this relationship.

7-SAT. The preliminary experiments showed that the maximal width of the new clauses should be in the interval $\{9, 10, 11\}$. Moreover, similarly to the 5-SAT case, the number of new clauses should be limited. In the preliminary phase, we found that at most 3% of the original clauses should be added, otherwise the performance of probSAT decreases.

The detailed phase showed that clauses up to width 9 and a limit of at most 1% is the most advantageous combination. Figure 4 shows the results of this combination. The modified strategy was better on average if the corresponding dot lies below the zero-baseline. Again, we observe that the performance of probSAT clearly benefits from the modified instances, especially on hard instances (red dots). This observation is also confirmed by the t-test ($p = 0.0062$). Additionally, similar to 5-SAT instances, the effect seems to increase as the number of variables increase.

Hidden Solution. In our training set $\mathcal{C}$, all instances with a hidden solution are 3-SAT instances with few variables (at most 540). The results are similar in nature to those discussed in the paragraph about uniform medium 3-SAT

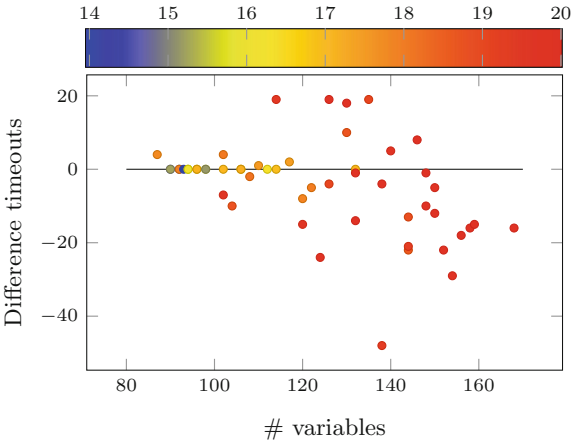

Fig. 4. Scatterplot of the number of variables of the 7-SAT instances against difference in timeouts between probSAT on the original instance and the instance with modified strategy. The color encodes the hardness of the original instance as measured in the logarithmic average par2 on 1000 runs of probSAT on the original instances. (Color figure online)

instances. That is, the addition of new clauses of maximal width 4 and no limit on the number of new clauses is generally beneficial.

Huge Instances. The huge instances in the training set $\mathcal{C}$ often have several million original clauses. In contrast, only a few new clauses are learned during the preprocessing time. Consequently, the effect of additional clauses is negligible. The preprocessing step should, therefore, be avoided on these instances.

Description of GapSAT

The name GapSAT stands for *Glucose assisted probSAT*, hinting towards the combination of probSAT as the core solver, that is being helped by a Glucose preprocessing phase. The exact functioning principle of GapSAT is depicted in the flowchart of Fig. 5.

As was noticeable in Fig. 3, if the 3-CNF formula contained more than approx. 9000 variables, the act of adding new clauses slows down probSAT. Furthermore, on huge instances, the preprocessing step yields no advantage. Thus, for over 9000 variables, the strategy of GapSAT falls back to just probSAT on the original formula. Otherwise, in each case, a short run of probSAT is used to filter out very easy to solve instances. The runtime is limited by the number of flips. If the instance could not be solved, we employ the strategy (depending on the maximal clause width in the formula) that was deemed most promising in the evaluations described in the previous subsection. That is, we first let Glucose extract clauses. The runtime of glucose was limited by 300 s in all cases. In the 5-SAT and 7-SAT case, Glucose could finish earlier, if the restrictions on

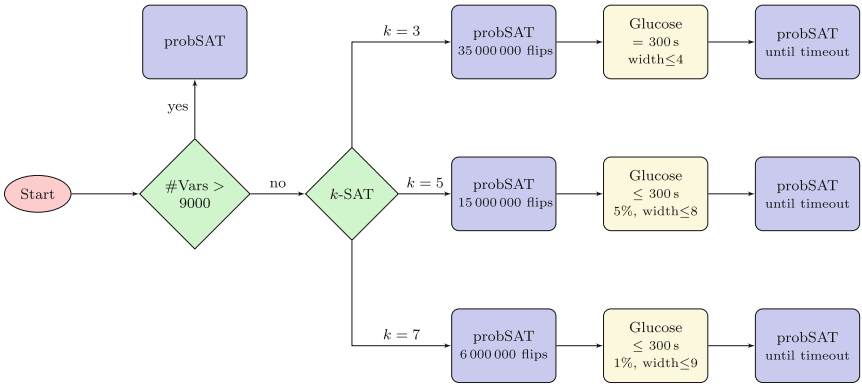

Fig. 5. Flowchart description of GapSAT.

the number of added clauses were met. We again emphasize that Fig. 2 explains the difference between the 3-SAT case when compared to the 5-SAT and 7-SAT case. Not restricting the number of learned clauses in the 3-SAT case turned out to be the superior strategy in our training experiments. One should further observe that Glucose has the possibility to solve the instance during its runtime. If this was not successful, probSAT is restarted on the formula, that was modified by running Glucose and adding the clauses corresponding to the strategy as developed in the previous subsection. It is noteworthy that GapSAT does not use any additional preprocessing techniques. We refer to Sect. 6 for a further discussion of that point.

5 Experimental Evaluation

In the following, the performance of GapSAT is evaluated. We compare GapSAT with the winner of the random track at the SAT competition 2018, Sparrow2Riss, and with the original version of probSAT.

All experiments were executed on a computer with 32 Intel Xeon E5-2698 v3 CPUs running at 2.30 GHz. We set the time limit to 5000 s and used no memory limit. The benchmarks consist of all 255 instances of the random track at the SAT competition 2018. Unlike the experiments in Sect. 4, the performance of each solver is measured based on its par2 value w. r. t. the runtime *in seconds*. In the following, the *score* denotes the sum of the par2 values over all instances.

Table 1. GapSAT, Sparrow2Riss, and probSAT are compared based on the number of solved instances and the corresponding score.

	# solved	score
probSAT	133	1 234 986.01
Sparrow2Riss	189	672 335.89
GapSAT	**223**	**347 156.40**

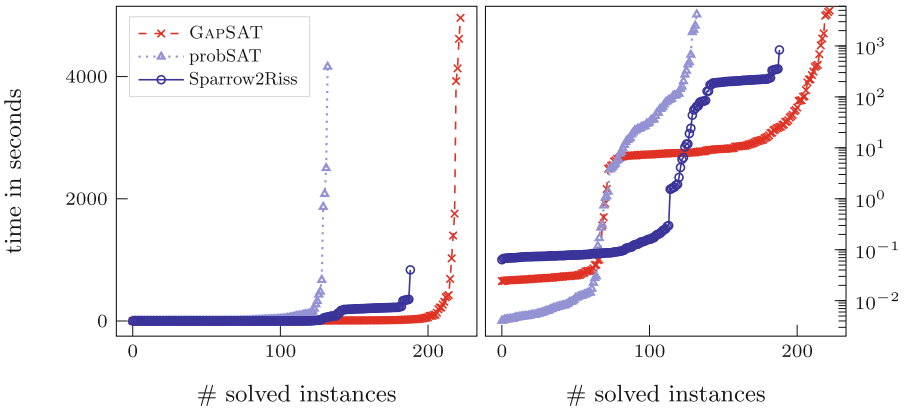

Fig. 6. Cactus plot comparing probSAT, Sparrow2Riss, and GapSAT on the instances of the random track of the SAT competition 2018. On the left, the plot is linearly-scaled; on the right, it is logarithmically-scaled.

As can be observed in Table 1, GapSAT solved substantially more instances than probSAT and Sparrow2Riss. The score of GapSAT is nearly halved compared to the score of Sparrow2Riss. Figure 6 demonstrates that GapSAT is especially efficient within the first few seconds. GapSAT solved more instances within 30 s than Sparrow2Riss solved within the standard timeout of 5000 s is a case in point. This behavior can be observed in the logarithmically-scaled part of Fig. 6. Furthermore, there are no instances that could be solved with Sparrow2Riss, but not with GapSAT within 5000 s.

We used statistical testing to evaluate the performance of GapSAT compared to Sparrow2Riss and probSAT. The t-test [44] shows that the score of GapSAT is better than both other solvers. We also used the Wilcoxon signed-rank test [46] to show that the median runtime of GapSAT is superior to Sparrow2Riss and probSAT. All results are statistically significant, with p-values less than 10^{-9}. Cohen's d value [20] is 0.39 for the comparison with Sparrow2Riss and 0.73 for the comparison with probSAT.

The instances of the random track at the SAT competition 2018 can be split into three domains. Some instances are generated uniformly at random with a medium number of variables and a clause-to-variable ratio close to the satisfiability threshold [38]. Similarly, there are uniform random instances with a huge number of variables but with a clause-to-variable ratio not too close to the phase transition. Finally, there are randomly generated instances with a hidden solution [12]. Table 2 shows the performance of all three solvers on each domain. GapSAT was the fastest solver on all three domains. It should be stated that the performances of GapSAT and probSAT are interchangeable on huge, uniform instances since the differences are just due to random noise on this domain. Lastly, the average time needed to learn the new clauses was 103.17 s. That said,

the actual time to perform the clause learning process is much shorter on most instances: The median time is just 7.89 s.

Table 2. GapSAT, Sparrow2Riss, and probSAT are compared on three domains based on the score.

	Hidden	Medium	Huge
probSAT	872 938.74	137 396.83	224 650.43
Sparrow2Riss	8 589.12	171 492.91	492 253.86
GapSAT	**851.36**	**127 982.19**	**218 322.85**

We conclude that future generations of local search solvers should incorporate some kind of clause learning mechanisms, for example, as a prepossessing step as used by GapSAT.

6 Conclusion and Outlook

In this work, a novel combination of CDCL as a preprocessing step and local search as the main solver is introduced. We empirically show on several domains that short clauses learned by CDCL have a high number of correct literals w. r. t. a fixed solution. Consequently, these new clauses guide local search solvers towards a solution. Using this knowledge, we design a new SAT solver GapSAT which uses the CDCL solver Glucose in a preprocessing step to find new clauses. It then proceeds to use probSAT on the modified formula to find a solution. We show that GapSAT improves the state-of-the-art on randomly generated instances.

The GapSAT solver can be improved even further: Besides the techniques described in this paper, no preprocessing steps are performed. We believe that further, finely tuned preprocessing may help to increase the performance of GapSAT on instances where it struggled to find a solution. When tuning GapSAT, we used the original settings of probSAT (i. e., we use the parameters from [5]). The only tuned parameters are the number of new clauses and their length. An interesting direction for further research is to obtain even better performance by simultaneously tuning these parameters together with the probSAT settings. Furthermore, we argued that the clauses which are added to the formula have a substantial effect on the performance of SLS algorithms. Even though clauses learned by Glucose have good properties on average, it would be beneficial to devise a clause selection heuristic for local search algorithms. If clauses having a negative impact on local search can be avoided, then the overall performance of solvers like GapSAT should improve significantly. Another general question that could be investigated is about the clauses from MiniSAT, that is being used in Glucose. Clauses learned by Glucose are generated by conflict analysis but may depend on clauses generated by the MiniSAT preprocessing. It

may be the case that short clauses are missed because they are only considered inside the solver, but never by the learning mechanism (when generated in the preprocessing step).

Supplementary Material. The source code of GapSAT and all evaluations are available under https://doi.org/10.5281/zenodo.3776052.

References

1. Anbulagan, A., Pham, D.N., Slaney, J.K., Sattar, A.: Old resolution meets modern SLS. In: Proceedings of the 20th National Conference on Artificial Intelligence and the 17th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 2005), pp. 354–359 (2005)
2. Audemard, G., Lagniez, J.-M., Mazure, B., Saïs, L.: Boosting local search thanks to CDCL. In: Fermüller, C.G., Voronkov, A. (eds.) LPAR 2010. LNCS, vol. 6397, pp. 474–488. Springer, Heidelberg (2010). https://doi.org/10.1007/978-3-642-16242-8_34
3. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 399–404 (2009)
4. Balint, A.: Engineering stochastic local search for the satisfiability problem. Ph.D. thesis, University of Ulm (2014)
5. Balint, A.: Original implementation of probSAT (2015). https://github.com/adrianopolus/probSAT
6. 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). https://doi.org/10.1007/978-3-642-14186-7_3
7. Balint, A., Henn, M., Gableske, O.: A novel approach to combine a SLS- and a DPLL-solver for the satisfiability problem. In: Kullmann, O. (ed.) SAT 2009. LNCS, vol. 5584, pp. 284–297. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-02777-2_28
8. Balint, A., Manthey, N.: Boosting the performance of SLS and CDCL solvers by preprocessor tuning. In: POS@ SAT, pp. 1–14 (2013)
9. Balint, A., Manthey, N.: Dimetheus. In: Proceedings of SAT Competition 2016: Solver and Benchmark Descriptions, vol. B-2016-1, pp. 37–38. Department of Computer Science Series of Publications B, University of Helsinki (2016)
10. Balint, A., Manthey, N.: SparrowToRiss 2018. In: Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions, vol. B-2018-1, pp. 38–39. Department of Computer Science Series of Publications B, University of Helsinki (2018)
11. Balint, A., Schöning, U.: Choosing probability distributions for stochastic local search and the role of make versus break. In: Cimatti, A., Sebastiani, R. (eds.) SAT 2012. LNCS, vol. 7317, pp. 16–29. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-31612-8_3
12. Balyo, T., Chrpa, L.: Using algorithm configuration tools to generate hard SAT benchmarks. In: Proceedings of the 11th Annual Symposium on Combinatorial Search (SoCS 2018), pp. 133–137. AAAI Press (2018)
13. Bayardo Jr., R.J., Schrag, R.: Using CSP look-back techniques to solve real-world SAT instances. In: Proceedings of the 14th National Conference on Artificial Intelligence (AAAI 1997), pp. 203–208. AAAI Press (1997)

14. Biere, A.: Cadical, lingeling, plingeling, treengeling, YalSAT entering the SAT competition 2017. In: Proceedings of SAT Competition 2017 - Solver and Benchmark Descriptions B-2017-1, pp. 14–15 (2017)
15. Biere, A., Cimatti, A., Clarke, E.M., Strichman, O., Zhu, Y.: Bounded model checking. Adv. Comput. **58**, 117–148 (2003)
16. Biere, A., Heule, M., van Maaren, H., Walsh, T. (eds.): Handbook of Satisfiability. Frontiers in Artificial Intelligence and Applications, vol. 185. IOS Press (2009)
17. Billionnet, A., Sutter, A.: An efficient algorithm for the 3-satisfiability problem. Oper. Res. Lett. **12**(1), 29–36 (1992)
18. Cha, B., Iwama, K.: Adding new clauses for faster local search. In: Proceedings of the 13th National Conference on Artificial Intelligence and 8th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1996), pp. 332–337 (1996)
19. Clarke, E.M., Biere, A., Raimi, R., Zhu, Y.: Bounded model checking using satisfiability solving. Formal Methods Syst. Des. **19**(1), 7–34 (2001). https://doi.org/10.1023/A:1011276507260
20. Cohen, J.: Statistical Power Analysis for the Behavioral Sciences. Routledge (2013)
21. Cook, S.A.: The complexity of theorem-proving procedures. In: Proceedings of the 3rd Annual ACM Symposium on Theory of Computing (STOC 1971), pp. 151–158 (1971)
22. Eén, N., Sörensson, N.: An extensible SAT-solver. In: Giunchiglia, E., Tacchella, A. (eds.) SAT 2003. LNCS, vol. 2919, pp. 502–518. Springer, Heidelberg (2004). https://doi.org/10.1007/978-3-540-24605-3_37
23. Eibach, T., Pilz, E., Völkel, G.: Attacking bivium using SAT solvers. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 63–76. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79719-7_7
24. Fang, L., Hsiao, M.S.: A new hybrid solution to boost SAT solver performance. In: Proceedings of the Design, Automation and Test in Europe Conference and Exposition (DATE 2007), pp. 1307–1313 (2007)
25. Givens, G.H., Hoeting, J.A.: Computational Statistics, vol. 703. Wiley (2012)
26. Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: Proceedings of the 15th National Conference on Artificial Intelligence and 10th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1998), pp. 431–437 (1998)
27. Habet, D., Li, C.M., Devendeville, L., Vasquez, M.: A hybrid approach for SAT. In: Van Hentenryck, P. (ed.) CP 2002. LNCS, vol. 2470, pp. 172–184. Springer, Heidelberg (2002). https://doi.org/10.1007/3-540-46135-3_12
28. Hirsch, E.A.: SAT local search algorithms: worst-case study. J. Autom. Reason. **24**(1–2), 127–143 (2000). https://doi.org/10.1023/A:1006318521185
29. Jussien, N., Lhomme, O.: Local search with constraint propagation and conflict-based heuristics. Artif. Intell. **139**(1), 21–45 (2002). (Preliminary version in IAAI 2000)
30. Kautz, H., Selman, B.: Ten challenges *Redux*: recent progress in propositional reasoning and search. In: Rossi, F. (ed.) CP 2003. LNCS, vol. 2833, pp. 1–18. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45193-8_1
31. Kilby, P., Slaney, J., Thiébaux, S., Walsh, T., et al.: Backbones and backdoors in satisfiability. AAAI **5**, 1368–1373 (2005)
32. Kroc, L., Sabharwal, A., Gomes, C.P., Selman, B.: Integrating systematic and local search paradigms: a new strategy for MaxSAT. In: Proceedings of the 21st International Joint Conference on Artificial Intelligence (IJCAI 2009), pp. 544–551 (2009)

33. Letombe, F., Marques-Silva, J.: Improvements to hybrid incremental SAT algorithms. In: Kleine Büning, H., Zhao, X. (eds.) SAT 2008. LNCS, vol. 4996, pp. 168–181. Springer, Heidelberg (2008). https://doi.org/10.1007/978-3-540-79719-7_17
34. Li, C.M., Anbulagan, A.: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), pp. 366–371 (1997)
35. Lynce, I., Marques-Silva, J.: SAT in bioinformatics: making the case with haplotype inference. In: Biere, A., Gomes, C.P. (eds.) SAT 2006. LNCS, vol. 4121, pp. 136–141. Springer, Heidelberg (2006). https://doi.org/10.1007/11814948_16
36. Marques-Silva, J.P., Sakallah, K.A.: GRASP–a new search algorithm for satisfiability. In: Proceedings of the IEEE/ACM International Conference on Computer-Aided Design (ICCAD 1996), pp. 220–227 (1996)
37. McAllester, D.A., Selman, B., Kautz, H.A.: Evidence for invariants in local search. In: Proceedings of the 14th National Conference on Artificial Intelligence and 9th Innovative Applications of Artificial Intelligence Conference (AAAI/IAAI 1997), pp. 321–326 (1997)
38. Mertens, S., Mézard, M., Zecchina, R.: Threshold values of random K-SAT from the cavity method. Random Struct. Algorithms **28**(3), 340–373 (2006)
39. Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of the 38th Design Automation Conference (DAC 2001), pp. 530–535 (2001)
40. Pipatsrisawat, K., Darwiche, A.: On the power of clause-learning SAT solvers with restarts. In: Gent, I.P. (ed.) CP 2009. LNCS, vol. 5732, pp. 654–668. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-04244-7_51
41. Schöning, U.: A probabilistic algorithm for k-SAT based on limited local search and restart. Algorithmica **32**(4), 615–623 (2002). https://doi.org/10.1007/s00453-001-0094-7. (Preliminary version in FOCS 1999)
42. Schöning, U., Torán, J.: The Satisfiability Problem: Algorithms and Analyses. Mathematics for Applications (Mathematik für Anwendungen), vol. 3. Lehmanns Media (2013)
43. Selman, B., Kautz, H.A., McAllester, D.A.: Ten challenges in propositional reasoning and search. In: Proceedings of the 15th International Joint Conference on Artificial Intelligence (IJCAI 1997), pp. 50–54 (1997)
44. Student: The probable error of a mean. Biometrika **6**(1), 1–25 (1908)
45. Völkel, G., Lausser, L., Schmid, F., Kraus, J.M., Kestler, H.A.: Sputnik: ad hoc distributed computation. Bioinformatics **31**(8), 1298–1301 (2015)
46. Wilcoxon, F.: Individual comparisons by ranking methods. Biometrics **1**(6), 80–83 (1945)