Speeding Up CDCL Inference With Duplicate Learnt Clauses

Stepan Kochemazov and **Oleg Zaikin** and **Alexander Semenov** and **Victor Kondratiev** [1]

Abstract. Conflict-driven clause learning (CDCL) is well-known to be the predominant SAT solving approach. Its main idea consists in using conflict clauses to guide the effective traversal of the complete search space. Despite the undoubted usefulness of this powerful mechanism, a CDCL solver may end up computing (exponentially) many conflict clauses. To resolve this issue, a number of efficient heuristics exist aiming at aggressive conflict clause filtering, which leads to some of the clauses being removed. Thus, when processing a particular instance, a solver may learn and remove the same clause multiple times. One might see it as an indication that such relearned clauses pose extra value. In the paper we show that extracting duplicate clauses and storing them indefinitely can be beneficial for the CDCL solver performance which is indicated by the fact that the family of solvers incorporating the corresponding heuristic won in the UNSAT and SAT+UNSAT tracks of the SAT Race 2019. We perform the detailed experimental evaluation of this heuristic on the instances from the SAT Competitions 2017 and 2018, and also SAT Race 2019 and show that it improves both PAR-2 and SCR scores.

1 INTRODUCTION

The success of state-of-the-art Conflict-Driven Clause Learning (CDCL) SAT solvers [17] largely comes from the careful handling of conflict clauses. They are used both to restrict the already processed parts of the search space, and to direct the subsequent search. In the recent two decades, new methods have been proposed for evaluating the quality of conflict clauses [1], preprocessing SAT formulas [3, 5, 7], and inprocessing the formulas during solving [9, 14, 16]. Informally, inprocessing and preprocessing aim to generate shorter and better representations for learnt clauses and for the clauses of an original formula, respectively. The important aspect of clause learning is the necessity to periodically reduce the learnt clauses database to maintain the time-memory trade-off: more learnt clauses give more information about the original problem, but sustaining them takes a lot of memory and slows down the Boolean constraint propagation.

The periodic purges of conflict clauses mean that some learnt clauses can be generated more than once in the course of solving SAT for some formula. To the best of our knowledge, this phenomenon while often acknowledged was not studied in any detail before. The goal of the present paper is to shed some light on this particular aspect of CDCL inner workings, in particular to show that the duplicate learnt clauses derived during CDCL inference can be put to good use. We propose a parameterized heuristic intended to (1) discover and (2) keep duplicate learnts in the solver's conflict database. To evaluate its effectiveness we incorporated the heuristic into the 3 solvers that won the SAT Competitions 2016 through 2018: MAPLECOMSPS, MAPLELCMDIST, MAPLELCMDISTCHRONOBT, and also into the COMINISATPS solver that can be viewed as their progenitor. Several common architectural features of these solvers exist that make the implementation of duplicate learnts heuristic easy and straightforward.

In the paper we use the mentioned solvers to perform the detailed experimental evaluation of the duplicate learnts heuristic on instances from the SAT Competitions 2017 and 2018 and that from the SAT Race 2019. Interestingly, the results show that the heuristic leads to stable improvement in performance of the solvers that incorporate learnt clause minimization [16] (also referred to as vivification [25] or distillation [11]). As for the solvers without clause vivification – the improvements are unstable. In the paper we study this phenomenon and provide some explanations regarding the possible reasons behind it. Note, that the MAPLELCMDISTCHRONOBT-DL family of solvers that incorporate the duplicate learnts heuristic into MAPLELCMDISTCHRONOBT became the winners of the SAT Race 2019. This fact can be viewed as a strong argument for the usefulness of the proposed heuristic.

The paper is organized as follows. In Section 2, we briefly touch on the related works crucial for the understanding of the paper. Section 3 contains the motivation behind the duplicate learnts heuristic and its detailed description. Section 4 covers the details regarding the implementation of the duplicate learnts heuristic in the considered CDCL solvers. In Section 5, the proposed heuristic is experimentally evaluated on instances from the SAT Competitions 2017-2018 and SAT Race 2019. It is followed by a discussion of the results and the description of the solvers submitted to the SAT Race 2019. Section 6 contains concluding remarks.

2 RELATED WORK

The description of technical details implies that the reader is familiar with the architecture of state-of-the-art CDCL solvers and possesses general knowledge about recent trends in practical SAT solving.

The seminal paper [18] paved the way for the Conflict-Driven Clause Learning SAT solvers by introducing the non-chronological backtracking and clause learning. It was later refined by the idea to periodically *restart* [10] the search by resetting the current decisions and starting the process anew. This idea together with the first methods for periodic purging of conflict database was introduced in [21]. Largely popularized by MINISAT [8], the CDCL concept became one of the cornerstones of effective complete SAT solvers. It revolves around *conflict* clauses, also referred to as *learnt clauses* or simply *learnts*. They are used both to limit the exploration of the decision

[1] Matrosov Institute for System Dynamics and Control Theory SB RAS, Irkutsk, Russia, email: veinamond@gmail.com, zaikin.icc@gmail.com, biclop.rambler@yandex.ru, vikseko@gmail.com

tree in the areas without solutions and to guide the search process in new directions via various heuristics. The most important question related to learnt clauses is how to judge one such clause to be better than the other? Because storing all of them indefinitely is out of the question: the resulting slowdown of constraint propagation nullifies all possible benefits of a larger conflict database. Moreover, it was shown that frequent purges of learnt clauses often result in better performance [1]. Unfortunately, there are no ways to know beforehand whether a learnt clause will be useful for future search or not. Thus, any methods for reducing the clause database have to rely on heuristics.

Probably the largest step in the practical analysis of learnt clauses was made in [1], where the concept of Literal Block Distance or `lbd` was introduced in the GLUCOSE solver. The `lbd` value is computed by counting the number of distinct decision levels for the literals of the conflict clause. The clauses with lower `lbd` are perceived to be more useful, than that with higher `lbd` values. Practically all modern CDCL solvers employ this measure to evaluate conflict clauses' worth.

The COMINISATPS solver [24] introduced the three-tiered structure of the learnt clauses database where they are separated based on the `lbd` values and handled differently. The COMINISATPS-inspired special treatment of clauses with small `lbd` is currently employed by most CDCL SAT solvers. In particular, the winners of main tracks of the SAT Competitions 2016 through 2018 are all based on COMINISATPS.

There have been many attempts to explore different measures besides `lbd` (see, e.g., [2]) or to characterize clauses as useful or useless based on the analysis of SAT solver proofs [26] or community structure of a SAT instance [23]. Despite this fact, the majority of existing CDCL SAT solvers employ `lbd` together with some sort of *activity* to assess the value of learnt clauses and decide which of them should be removed during the next database purge.

Another very relevant and popular class of methods for handling conflict clauses is formed by the so-called *inprocessing* methods (see, e.g., [14]). They modify (minimize, simplify, etc.) the parts of formula during the solving. One of the most recent and prominent results in this direction was proposed in [16] and consists in a relatively straightforward method for learnt clauses minimization, also referred to as clause distillation [11] or vivification [25]. The important fact is that as a result of a clause vivification, the size of a clause can be (significantly) reduced resulting in lower `lbd` value, thus greatly improving the clause's value.

All the currently known methods for handling conflict clauses rely on periodic purges of their database. While it is evident that such purges might lead to learning some conflict clauses multiple times, the authors of the paper are not aware of this phenomenon having been studied before thus making the present research novel.

3 DUPLICATE LEARNTS

The idea behind the heuristic proposed below might seem a little audacious, but it actually follows from common sense. If a SAT solver repeatedly derives the same learnt clauses while exploring different paths of the search space, then the permanent addition of such clauses to the clause database may save the solver some work in the future and possibly even direct it to explore potentially more profitable regions of a decision tree. The natural question that comes to mind when speaking about duplicate learnts is "*how many of them are there?*". In fact, the wide adoption of unsatisfiability proofs [27] in contemporary SAT solvers makes it possible to answer this question with ease, for example as follows.

1. Launch a SAT solver on some instance with enabled proof construction (might want to interrupt on time limit).
2. Parse the proof file ignoring clause deletions.
3. Order each learnt clause in a specific order (say, ascending).
4. Count the occurrences of each learnt.

For example, during the first 2000 restarts (about 17 seconds) the MINISAT 2.2 solver [8] (the version from the SAT Competition 2018) on the pigeonhole formula [6] (with 12 pigeons and 11 holes) generates 905378 learnt clauses, of which 6353 are duplicates. Some learnts repeated up to 21 times, however the majority of them were met only two (5038), three (785) or four times (234). Note that because pigeonhole formulas are small, unsatisfiable and contain few variables, the number of duplicate learnts for them is much larger than for typical SAT instances. Nevertheless, in this instance the portion of re-learned conflict clauses in the conflict clauses database (for limited time period) is about 0.7 %. Thus, it is safe to assume that their number is actually pretty small, but since CDCL SAT solvers typically generate millions of learnt clauses, the duplicate ones are likely to be encountered regularly, albeit in small quantities. Also, we will show below that the learnt clause minimization technique [16] is responsible for the majority of duplicate learnts.

Let us now formulate the heuristic employing duplicate learnts.
The duplicate learnts heuristic: *Screen generated learnt clauses and add duplicates as permanent clauses.*

Below we consider the implementation in more detail, but first it is important to note several features of COMINISATPS-based solvers that allow us to implement the proposed heuristic in a natural and elegant manner.

3.1 On the architecture of COMINISATPS-based solvers

The winners of the main tracks of the SAT Competitions 2017 and 2018, MAPLELCMDIST [16] and MAPLELCMDISTCHRONOBT [22], are both variations of the MAPLECOMSPS solver [15] that won the main track of the SAT Competition 2016. MAPLECOMSPS is built upon COMINISATPS [24], based on MINISAT 2.2 [8] and GLUCOSE [1].

The peculiar feature of COMINISATPS and its derivatives that we will exploit below consists in the fact that they split all learnt clauses into three *Tiers* depending on their `lbd` and the `core_lbd_cut` parameter. The *Core* tier contains the learnts with `lbd` $\leq$ `core_lbd_cut`, and the corresponding clauses are stored indefinitely (they are never purged). The learnts with `core_lbd_cut` $<$ `lbd` ≤ 6 go into *Tier2* and all the remaining learnts – to the *Local* tier. The learnts purged during `reduceDB` from *Tier2* go to the *Local* tier. Meanwhile, *Local*-tier conflict clauses are deleted during `reduceDB`. Several other specific nuances regarding the handling of clauses from different tiers exist, but they are not important for the present research.

During the clause minimization (in MAPLELCMDIST and MAPLELCMDISTCHRONOBT) learnts can be reduced in size and therefore decrease their `lbd`. The outlined solvers apply vivification only to clauses from *Core* and *Tier2*. If the new value of `lbd` for a *Tier2* learnt satisfies the (previously unmet) condition for the *Core* then it is moved to that tier. Finally, the parameter `core_lbd_cut` is equal to 3 by default but is increased to 5 if after the first 100000 conflicts the *Core* tier contains fewer than 100 learnts.

The described setup is perfect for the proposed duplicate learnts heuristic: we can move the detected duplicate conflict clause into the *Core* tier regardless of its `lbd`.

3.2 Duplicate learnts heuristic: implementation details

Let us now give a detailed description of the proposed duplicate learnts (DL) heuristic that was briefly introduced above. The detection of duplicate learnts involves storing the database (in the form of a hash table) of all learnts generated during some period of time. Several observations should be taken into account in the implementation of the DL heuristic.

1. The higher the `lbd` of a learnt clause, the less the probability that it will be relearned later.
2. The number of times a learnt clause was generated matters and should be taken into account.
3. To avoid the uncontrolled growth of the hash table size it is necessary to limit it, e.g., by means of periodic purges.

The first point can be addressed naturally. It is reasonable to establish the `lbd_limit` and screen for duplicates only the learnt clauses for which `lbd` $\leq$ `lbd_limit`.

Regarding the second point, the common sense suggests that the number of times a learnt clause was re-learned is important. In particular, if a clause was generated only two times – it is quite likely that the second occurrence is not due to a hidden trend. The more times it is repeated – the greater the probability that the inclusion of such a learnt into a permanent clause database can be beneficial. Hereinafter, by a *k-duplicate learnt* we mean a learnt clause for which the number of its occurrences in the hash table is equal to $k+1$ (it means that the learnt clause repeated k times). By a *duplicate learnt* we mean a k-duplicate learnt with $k \geq 1$. It is important to take into account the hash table because, as it will be detailed below, it is necessary to purge the majority of its entries from time to time. Thus, technically, a learnt clause can be derived more times than $k+1$ but be marked as k-duplicate by a hash table.

It is clear that the number of k-duplicates rapidly decreases with the increase of k. Thus, it is sensible to consider k-duplicates with relatively small k for inclusion into the *Core* tier. In our implementation, we use the parameter `min_dup_app` (minimum number of duplicate appearances). If we denote the `min_dup_app` value as mda then the DL heuristic works as follows: it puts all $(mda-1)$-duplicates into *Tier2* viewing them as candidates for inclusion into *Core*, and puts all mda-duplicates into *Core*.

As for the hash table growth, it is clear that storing the data in it indefinitely is out of the question, regardless of the `lbd_limit`'s value. While modern PCs are more than capable to hold the hash table containing the information on the appearances of learnt clauses for thousands of seconds, in the general sense it would be a bad practice. Also, from our empirical evaluation it turned out that the variants of the implementation where the hash table was not periodically purged showed worse performance. We limit the initial size of the duplicate learnts database by `db_size_limit` parameter and once the database size exceeds this value we purge from it all entries corresponding to k-duplicate learnts with $k <$ `min_dup_app`-1 and increase the value of `db_size_limit` by 10 %. The goal here is to preserve the data on clauses which have already been put into *Core* or will be put there if they appear during the search one more time.

It might seem that, since the $mda-$duplicate learnts go straight into the *Core* tier in COMINISATPS-based solvers, the separators between the *Core* and *Tier2* and between *Tier2* and *Local* tiers become more important, because there appears an additional source of core learnts. However, as it will be shown in Section 5, it turned out that in practice the amount of *Core* learnts that come from the DL heuristic is not large. Thus, it is not necessary to introduce any changes into the handling of the `core_lbd_cut` value. Note that in the SAT Race 2019 versions of the DL-based solvers the value of `core_lbd_cut` was reduced from 3 to 2 and the heuristic that increases its value to 5 depending on the size of the *Core* tier after first 100000 conflicts was disabled. In the experiments for the present paper we found that leaving the treatment of `core_lbd_cut` to the COMINISATPS variant works a bit better, thus the solvers we employed in the experiments slightly differ from that participated in the SAT Race 2019.

To sum things up, we propose to implement the DL heuristic with parameters: `lbd_limit`, `min_dup_app`, `db_size_limit`. The pseudocode depicting the interconnection between parameters is shown at Algorithm 1. The `isDuplicate` function is applied to each learnt clause generated during the conflict analysis and during learnt clause minimization of learnts from *Tier2* (because MAPLELCMDIST and MAPLELCMDISTCHRONOBT apply the minimization only to clauses from *Core* and *Tier2*). Essentially, it counts the number of occurrences of a learnt clause according to the hash table.

```
isDuplicate(learnt clause $L$)
    if $size(Hashtable) >$ db_size_limit then
        foreach $U$ in $Hashtable$ do
            if $Hashtable[U] <$ min_dup_app then
                Remove $U$ from $Hashtable$
        db_size_limit = db_size_limit × 1.1
    if $lbd(L) \leq$ lbd_limit then
        if $L$ in $Hashtable$ then
            $Hashtable[L] = Hashtable[L] + 1$
        else
            $Hashtable[L] = 1$
    return $Hashtable[L]$
```

Algorithm 1: Algorithm for processing duplicate learnts

It works as follows: a hash table ($Hashtable$ in Algorithm 1) is maintained that stores the information on the number of occurrences of learnts produced during the CDCL derivation. If a learnt clause satisfies the given criterion on its `lbd`, then the procedure updates and outputs the number of its occurrences in the hash table. If the result of `isDuplicate` for a learnt clause L is equal to `min_dup_app` then L is added to *Tier2* and if it is equal to `min_dup_app`+1, then it goes into the *Core* tier. Once the hash table size exceeds `db_size_limit`, all its entries for which the number of occurrences is $<$ `min_dup_app` are deleted and the value of `db_size_limit` is increased by 10%.

The hash table used in screening learnts for duplicates was implemented on top of the `unordered_map` class from the C++ Standard Template Library. The literals in each learnt clause are sorted in the ascending order before being processed by a hash table. In the next section let us cover the implementation details.

4 IMPLEMENTING DL HEURISTIC IN COMINISATPS-BASED SOLVERS

To evaluate the DL heuristic, we implemented it into several CDCL SAT solvers. We chose the ones that are based on COMINISATPS, since its tiered approach to storing learnts fits very well to the proposed heuristic. In the four years since the COMINISATPS was first introduced, the progress in SAT solving was quite significant, thus it is interesting how the DL heuristic will fare when implemented in top solvers from different years. Therefore we worked with COMINISATPS, MAPLECOMSPS, MAPLELCMDIST and MAPLELCMDISTCHRONOBT.

4.1 COMINISATPS

The COMINISATPS solver [24] debuted at the SAT Race 2015 and introduced several novel features that later were assimilated into other SAT solvers. In SAT Race 2019 [13], the majority of participants employed at least some of the techniques first proposed in [24].

COMINISATPS natively supports `lbd` and tiered structure of learnt clauses. It uses the Variable State Independent Decaying Sum (VSIDS) heuristic [21] to choose variables for branching, unlike the later solvers. Also it is completely deterministic: the solver frequently switches between fast glucose-like restarts and Luby restarts.

We refer to solvers with incorporated DL heuristic as to SOLVERNAME-DL, followed by the values of the DL parameters outlined in the previous section. COMINISATPS-DL does not employ the learnt clause minimization (it was proposed in 2017), thus its only source of duplicate learnts is the conflict analysis procedure.

4.2 MAPLECOMSPS

The MAPLECOMSPS solver won in the main track of the SAT Competition 2016. Compared to COMINISATPS it has two major differences. The first is that it uses not only VSIDS to choose variables for branching, but also the so-called learning rate branching (LRB) [15]. The use of LRB made it possible to significantly improve the overall performance of SAT solvers. Another distinctive feature of MAPLECOMSPS is that it uses LRB + Luby restarts for the first 2500 seconds of the search and VSIDS + glucose restarts for the remaining time. The switch happens only between restarts. Note, that similar to COMINISATPS-DL, in MAPLECOMSPS-DL duplicate learnts can be produced only in the course of conflict analysis.

4.3 MAPLELCMDIST and MAPLELCMDISTCHRONOBT

The MAPLELCMDIST solver [16] won the main track in the SAT Competition 2017. Its main novelty compared to MAPLECOMSPS is the use of learnt clause minimization (LCM) [11, 25] as an inprocessing technique. Previously, LCM was employed only on the preprocessing stage. The learnt clause minimization turned out to be so effective that it is now used in most state-of-the-art CDCL solvers.

The MAPLELCMDISTCHRONOBT [22] is the evolution of MAPLELCMDIST that revived to some extent the chronological backtracking. This solver won the main track of the SAT Competition 2018. In [20], it was shown that incorporation of chronological backtracking into the CADICAL SAT solver significantly improved its performance, so it is likely that this technique will make its way into other solvers as well.

While chronological backtracking does not really interfere with the proposed DL heuristic, the learnt clause minimization certainly does. Let us give additional comments to the procedure. The clause vivification in the considered solvers is applied only to *Core* and *Tier2* learnts. It is invoked quite rarely, but has a significant effect on the search. The minimization procedure attempts to construct a shorter clause by testing whether each consecutive literal of an original clause can be safely removed. It does not always manage to reduce the size of a clause, but when it does, it often results in a reduced `lbd`.

Our implementation of the DL heuristic handles the learnt clause minimization by applying the `isDuplicate` function (see Algorithm 1) to each simplified clause from *Tier2* and processing it the way outlined in the previous section. An important nuance is that we apply `isDuplicate` to all *Tier2* clauses that undergone learnt clause minimization, regardless of whether their size was decreased or not. In our experiments this version showed better results than the (intuitively more natural) variant with screening only the reduced *Tier2* learnts. Note, that we use the same hash table for learnt clauses obtained through conflict analysis and through learnt clause minimization and do not distinguish the corresponding learnt clauses in any way. Thus, for example some learnt clause may first be generated through conflict analysis, then be removed during the database purge, then during learnt clause minimization phase some other learnt clause can be reduced to it, thus resulting in the second occurrence, etc.

5 COMPUTATIONAL EXPERIMENTS

This section describes the performed experimental evaluation comparing the performance of solvers implementing the proposed DL heuristic against their corresponding unmodified versions.

5.1 Experimental setup

In the experiments, for each of the four CDCL solvers mentioned in Section 4 (COMINISATPS, MAPLECOMSPS, MAPLELCMDIST, MAPLELCMDISTCHRONOBT) we made two DL-based modifications. In the first DL-based modification, DL-12-3-500K, the following values of the DL parameters (see Section 3) were used: `lbd_limit`=12, `min_dup_app`=3, and `db_size_limit`=500000. In DL-14-2-1M, the DL parameters had the following values: `lbd_limit`=14, `min_dup_app`=2, and `db_size_limit`=1000000. The main difference between them is that the DL-14-2-1M version processes much more potential duplicate learnts compared to DL-12-3-500K and admits larger number of them into the *Core* tier. This phenomenon is discussed in the remainder of this section in more detail.

As a test set we chose the instances from the main tracks of the SAT Competition 2017 [4], SAT Competition 2018 [12], and SAT Race 2019 [13]. Below we refer to these sets of instances as SC2017, SC2018, and SR2019, respectively.

All experiments were performed using the "Academician V.M. Matrosov" computing cluster of Irkutsk Supercomputer Center [19]. Each node of the cluster is equipped with two 18-core Intel Xeon E5-2695 CPUs and 128 GB DDR4 RAM. To ensure that all solvers worked in equal conditions, we singled out one cluster node and ran all experiments on it. The results presented below were obtained in parallel using 36 tasks per computing node at a time. Here by "task" a single run of a solver on some instance is meant.

Following the SAT Competition (and SAT Race) testing methodology, we ran all considered 12 solvers with the timeout of 5000 seconds, and compared them by PAR-2 (Penalized Average Runtime) score. Recall that PAR-2 is a sum of all runtimes for solved instances + 2$\times$timeout for unsolved instances. The lower PAR-2 indicates that the solver is better on average. We also show the SCR (Solution Count Ranking) scores since they complement PAR-2.

Table 1. Detailed statistics on solved instances from SC2017, SC2018, and SR2019. S and U stand for the number of solved satisfiable and unsatisfiable instances, respectively. The PAR-2 scores are reported in thousands. The best PAR-2 and SCR scores for SC2017, SC2018, and SR2019 are marked with bold.

	SC2017				SC2018				SR2019			
Solver	SCR	S	U	PAR-2	SCR	S	U	PAR-2	SCR	S	U	PAR-2
COMSPS	164	82	82	2018	206	114	92	2103	208	125	83	2140
COMSPS-DL-12-3-500k	168	83	85	1990	205	113	92	2117	206	126	80	2163
COMSPS-DL-14-2-1m	176	89	87	1910	200	108	92	2150	210	127	83	2139
MaCom	197	102	95	1715	212	116	96	2017	224	131	93	2012
MaCom-DL-12-3-500k	200	102	98	1703	208	114	94	2062	219	127	92	2059
MaCom-DL-14-2-1m	202	103	99	1680	214	119	95	2018	220	129	91	2029
MaLcm	206	100	106	1638	228	128	100	1885	226	131	95	1980
MaLcm-DL-12-3-500k	210	105	105	1607	236	134	102	1821	231	136	95	1928
MaLcm-DL-14-2-1m	211	108	103	1603	237	133	104	1805	234	137	97	1914
MaChr	212	98	114	1586	234	132	102	1834	231	137	94	1933
MaChr-DL-12-3-500k	**219**	105	114	**1530**	**244**	141	103	**1735**	**236**	139	97	**1872**
MaChr-DL-14-2-1m	**219**	107	112	1551	240	137	103	1783	235	138	97	1875

5.2 Results

The detailed statistics for SC2017, SC2018, and SR2019 are presented in Table 1. It shows (1) the total number of instances solved (SCR), (2) the number of satisfiable and (3) unsatisfiable instances solved, as well as (4) PAR-2 score for each solver on a set of instances combined from SC2017, SC2018, and SR2019. In the table, COMSPS stands for COMinisatPS; MaCom for MapleCOMSPS; MaLcm for MapleLCMDist; and MaChr for MapleLCMDistChronoBT. The results are also presented in graphical form in Figure 2. In particular, Figures 2a, 2b, 2c, and 2d show the runtimes of the considered solvers on instances from SC2017, SC2018, SR2019, and the combined set, respectively.

5.3 Discussion

It is clear that for both MapleLCMDist and MapleLCMDistChronoBT the DL versions show stable and significant speedup on all three considered sets of benchmarks. In particular, MapleLCMDistChronoBT-DL-14-2-1m managed to solve 17 more instances (compared to MapleLCMDistChronoBT) on the combined set of instances, while MapleLCMDistChronoBT-DL-12-3-500k solved 22 more instances. The PAR-2 scores are also significantly better.

An interesting detail is that while MapleLCMDist showed significantly worse results compared to MapleLCMDistChronoBT on all considered sets, the performance of both versions of MapleLCMDist-DL is comparable to that of MapleLCMDistChronoBT. In particular, on SC2018 and SR2019 two variants of MapleLCMDist-DL outperform MapleLCMDistChronoBT. In a way, the DL heuristic makes it possible to gain the speedup comparable to the one provided by chronological backtracking on a wide set of benchmarks.

Overall, the best performance over all sets of test instances (SC2017, SC2018, and SR2019) was achieved by MapleLCMDistChronoBT-DL-12-3-500k. To see where does the performance increase come from, we analyzed the results of the solver on families of benchmarks. From the analysis of SC2017, SC2018, and SR2019 we outlined 11, 23 and 10 families of benchmarks, respectively, in each set of instances. Thus, in total we considered 44 benchmark families. Figure 1 shows the scatter plot that compares PAR-2 scores on all 44 families obtained

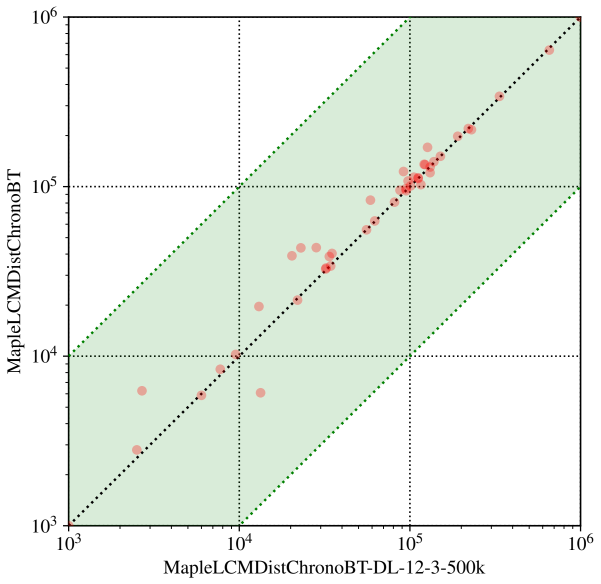


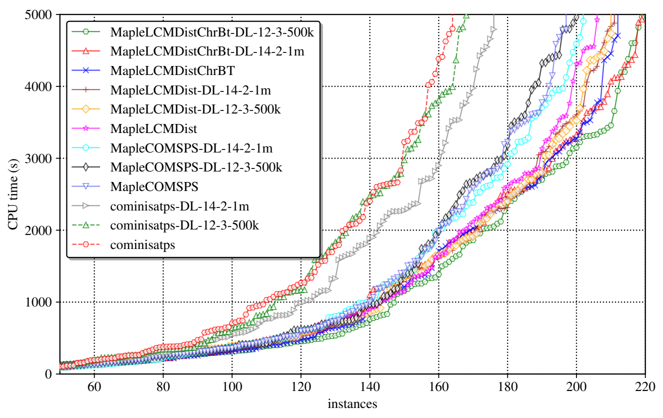
Figure 1. Scatter plot depicting PAR-2 scores obtained for 44 families of benchmarks from SC2017, SC2018, and SR2019

by MapleLCMDistChronoBT-DL-12-3-500k and MapleLCMDistChronoBT. One can see a steady increase in performance of MapleLCMDistChronoBT-DL-12-3-500k over the wide range of benchmark families. In particular, it turned out that MapleLCMDistChronoBT-DL-12-3-500k had better PAR-2 on 31 families out of 44 (i.e. on about 70 % of families). It means that the proposed DL heuristic is applicable to a wide area of problems. We show 8 families on which the speedup was greater than 1.2x in Table 2. The families are sorted in descending order by their speedup.

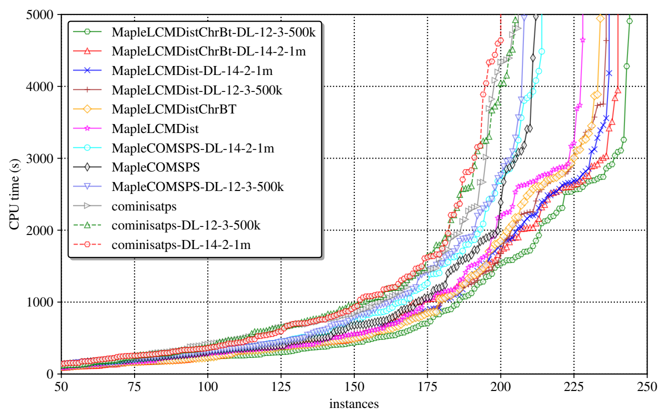
On the other hand, DL heuristic only slightly improves the performance of COMinisatPS and MapleCOMSPS. Sometimes the DL variants manage to solve more instances and do it faster but on other instances the addition of the DL heuristic results in decreased SCR and PAR-2. In particular, DL-versions of COMinisatPS significantly outperform the original solver on SC2017, but are slightly worse on SC2018. For the case of MapleCOMSPS the similar pic-

Table 2. Families of benchmarks from SC2017, SC2018, and SR2019 on which MAPLELCMDISTCHRONOBT-DL-12-3-500K had more than 1.2x speedup (by PAR-2) compared to MAPLELCMDISTCHRONOBT. MACHR stands for MAPLELCMDISTCHRONOBT, MACHR-DL12 stands for MAPLELCMDISTCHRONOBT-DL-12-3-500K.

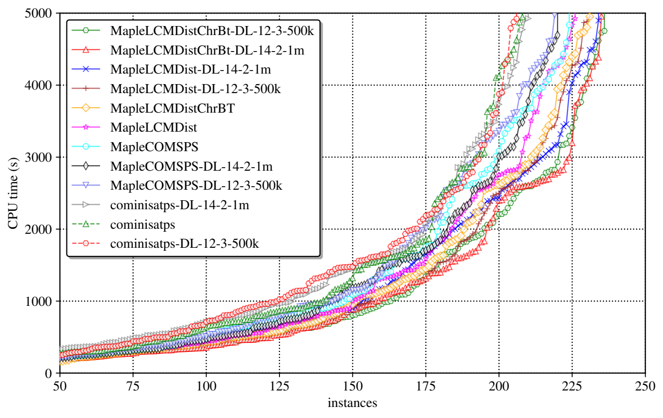
Family description	Source	PAR-2 on MACHR	PAR-2 on MACHR-DL12	PAR-2 Speedup
Subshape in Grid	SC2017	6 249	2 690	2.32x
SHA-1 Pre-image Attack	SR2019	39 077	20 353	1.92x
Searching for unit-distance graph	SC2018	43 526	22 961	1.9x
SATcoin — Bitcoin mining via SAT	SC2018	43 665	28 259	1.55x
Cryptanalysis of keystream generators	SR2019	19 631	13 037	1.5x
Latin squares	SC2017	83 211	58 695	1.42x
Polynomial multiplication	SC2017	122 741	91 712	1.34x
Pigeonhole principle	SC2018	170 451	126 822	1.34x



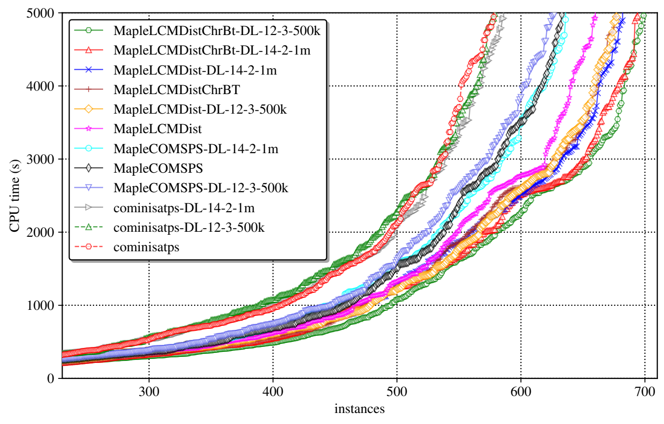
(a) Instances from SC 2017



(b) Instances from SC 2018



(c) Instances from SR 2019



(d) All instances from SC 2017, SC 2018, and SR 2019

Figure 2. Comparison of the considered solvers on instances from SC 2017, SC 2018, and SR 2019

ture is observed but with slowdown of the DL-versions on SR2019. The largest distinction between COMINISATPS, MAPLECOMSPS and MAPLELCMDIST, MAPLELCMDISTCHRONOBT is that the latter solvers employ the learnt clause minimization. Below we show that it positively influences the DL heuristic.

5.4 On the explanation of DL effectiveness

In order to better understand the reasons behind apparent effectiveness of the DL heuristic, we gathered additional CPU-independent information. In particular, we modified MAPLELCMDISTCHRONOBT-DL-12-3-500K and MAPLELCMDISTCHRONOBT-DL-14-2-1M so that they counted and periodically outputted the DL-related statistics, and ran them on the combined set of instances from SC2017, SC2018, and SR2019. We measured the number of 2-duplicates and (for 12-3-500K) 3-duplicates. In addition to that we measured the time overhead of the DL-heuristic, i.e. what percentage of the total runtime does it take to check whether learnt clauses are duplicates or not. Finally, we counted the number of hash table purges.

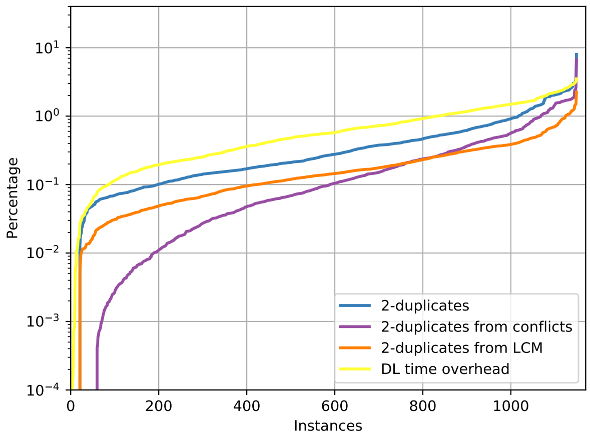


Figure 3. Statistics on the number of duplicates and DL heuristic time overhead for MAPLELCMDISTCHRONOBT-DL-14-2-1M. The vertical axis is in logarithmic scale.

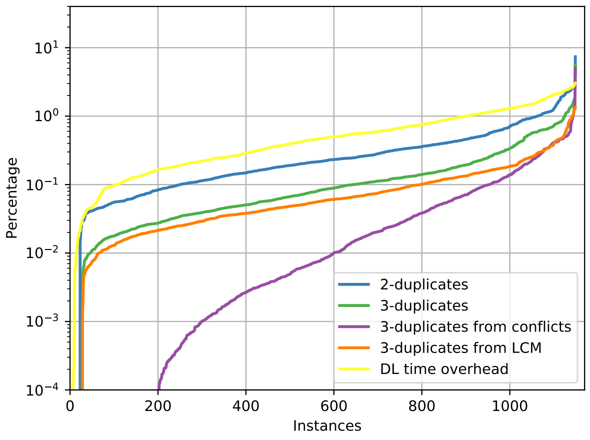


Figure 4. Statistics on the number of duplicates and DL heuristic time overhead for MAPLELCMDISTCHRONOBT-DL-12-3-500K. The vertical axis is in logarithmic scale

Collecting the data on database purges was the most straightforward part, and it turned out that on average there are about 7 purges per 5000 seconds for MAPLELCMDISTCHRONOBT-DL-14-2-1M and about 14 purges per 5000 seconds for MAPLELCMDISTCHRONOBT-DL-12-3-500K. Figures 3 and 4 show the remaining statistics we gathered using the verbose variants of MAPLELCMDISTCHRONOBT-DL-14-2-1M and MAPLELCMDISTCHRONOBT-DL-12-3-500K, respectively. They are made in the style of cactus plots, i.e. for each plot line the corresponding values are ordered in the ascending order independently from the data for the other plot lines. The plots use the logarithmic scale on the vertical axis and show the percentage of different kinds of duplicate learnts proportional to the total number of learnts screened for duplicates (e.g., the ones with `lbd` $\leq$ 12 for MAPLELCMDISTCHRONOBT-DL-12-3-500K and `lbd` $\leq$ 14 for MAPLELCMDISTCHRONOBT-DL-14-2-1M). Note, that in accordance with Algorithm 1, we periodically purge from the hash table all entries that correspond to k-duplicates with $k <$ `min_dup_app`, thus some learnt clauses may have repeated more times than it was measured (but not fewer). Recall that all the 2-duplicates in MAPLELCMDISTCHRONOBT-DL-14-2-1M and 3-duplicates in MAPLELCMDISTCHRONOBT-DL-12-3-500K go to the *Core* tier and remain there indefinitely. For them we additionally show at different plot lines the percentage of duplicate learnts that go to *Core* after having occurred during the conflict analysis or the learnt clause minimization.

From Figure 3 it can be seen that the amount of duplicates that go to the *Core* tier from the learnt clause minimization is quite significant and is on average higher than that from the conflict analysis. This situation is even more drastic when we look at 3-duplicates at Figure 4: their total number is significantly lower than that of 2-duplicates, and the vast majority of duplicate learnts that were repeated 3 times actually comes from LCM.

We investigated this issue a little bit deeper and it turned out, that the vivification procedure from [16] is made in a way that does not prevent it from constructing multiple duplicate learnts during one invocation of minimization procedure. Recall that in, say, MAPLELCMDISTCHRONOBT, the learnt clause minimization is applied only to *Core* and *Tier2* learnts. The procedure itself is ran once in a while between restarts. In its course, the minimization is applied to all corresponding learnt clauses that have not yet undergone one. Informally, let us refer to clauses *minimized* in one block as to belonging to the same minimization *phase*. Thus, for example, among 1000 core learnts minimized during one LCM phase, there can appear 100 identical learnts. The current implementation of the DL heuristic handles them in two different ways: first, it does not allow the solver to put duplicates of existing learnts to *Core* within the current minimization phase, thus improving the propagation effectiveness. Second, if sufficient number of learnt clauses from *Tier2* were minimized into identical shorter clause, then this clause is moved to *Core*, which may well be beneficial for the solver's performance.

Note that the peculiar symbiosis of DL and the learnt clause minimization, as well as the DL heuristic as a whole (but to a lesser extent) may indirectly affect the LRB and VSIDS scores of variables involved in corresponding learnt clauses, but we believe this influence to be negligible.

From Figures 3 and 4 it is clear that the time overhead of the DL heuristic (measured as the percentage of time spent on invoking the hash table aimed at detecting duplicate learnts to the time it took to solve an instance or to the time limit of 5000 seconds) is usually under 1 percent, thus the heuristic is quite cheap. It is cheaper for the 12-3-500K version since the hash table size is smaller and the number of duplicates is lower due to them having to be repeated more times to be noticed.

Another interesting observation from the presented figures is that there are instances, on which there are little to no duplicates, and instances that yield a lot of duplicates. We believe that further investigation of this feature, that seems to be tied to specific test families, will make it possible to improve the targeting and efficacy of the DL heuristic.

From our point of view, the presented data is sufficient to validate the relevance of the proposed heuristic and its potential for future research.

5.5 Participation in SAT Race 2019

Three solvers incorporating the DL heuristic participated in the SAT Race 2019. Among them, MAPLELCMDISTCHRONOBT-

DL-v2.1 and MapleLCMDistChronoBT-DL-v2.2 are slightly modified variants of MapleLCMDistChronoBT-DL-14-2-1m. In particular, in the SAT Race 2019 versions the value of `core_lbd_cut` parameter was fixed to 2. v2.2 version also used the value of `lbd` separating *Tier2* from *Local* set to 7 instead of 6. MapleLCMDistChronoBT-DL-v3 is a variant of MapleLCMDistChronoBT-DL-12-3-500k that employs periodic deterministic switching between LRB+Luby restarts and VSIDS + glucose restarts. The solver switches modes relatively rarely since it is tied to the number of propagations. All three solvers won the second place in the SAT track and the first place in the SAT+UNSAT track (their results were comparable to each other).

6 CONCLUSIONS

In the present paper, we introduced the idea that the duplicate learnts derived by a CDCL solver can be used to increase its performance. We implemented the corresponding technique in solvers that won the main track of the SAT Competitions from 2016 to 2018. It turned out, that it gives a significant speedup for winners of the SAT Competitions 2017 and 2018 that use learnt clause minimization. Also, the DL-based versions of MapleLCMDistChronoBT took the first place in the SAT+UNSAT track of the SAT Race 2019.

In the future we are planning to improve the handling of duplicate learnts and seek understanding why the proposed technique gives a particularly significant speedup on some families of benchmarks.

ACKNOWLEDGEMENTS

We thank anonymous reviewers for their thoughtful and constructive feedback that made it possible to significantly improve the quality of the present paper. We also express deep gratitude to Alexey Ignatiev for insightful comments and discussions of preliminary results of this study.

The research was funded by Russian Science Foundation (project No. 16-11-10046). Stepan Kochemazov was additionally supported by Council for Grants of the President of the Russian Federation (stipend no. SP-2017.2019.5).

References

[1] Gilles Audemard and Laurent Simon, ‘Predicting learnt clauses quality in modern SAT solvers’, in *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 399–404, (2009).

[2] Gilles Audemard and Laurent Simon, ‘Lazy clause exchange policy for parallel SAT solvers’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *Lecture Notes in Computer Science*, pp. 197–205, (2014).

[3] Fahiem Bacchus and Jonathan Winter, ‘Effective preprocessing with hyper-resolution and equality reduction’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pp. 341–355, (2003).

[4] Tomás Balyo, Marijn J. H. Heule, and Matti Järvisalo, eds. *Proceedings of SAT Competition 2017: Solver and Benchmark Descriptions*, volume B-2017-1 of *Series of Publications B*. Department of Computer Science, University of Helsinki, 2017.

[5] Armin Biere, ‘Preprocessing and inprocessing techniques in SAT’, in *Haifa Verification Conference (HVC)*, volume 7261 of *Lecture Notes in Computer Science*, p. 1, (2011).

[6] Stephen A. Cook and Robert A. Reckhow, ‘The relative efficiency of propositional proof systems’, *Journal of Symbolic Logic*, **44**(1), 36–50, (1979).

[7] Niklas Eén and Armin Biere, ‘Effective preprocessing in SAT through variable and clause elimination’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 3569 of *Lecture Notes in Computer Science*, pp. 61–75, (2005).

[8] Niklas Eén and Niklas Sörensson, ‘An extensible SAT-solver’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 2919 of *Lecture Notes in Computer Science*, pp. 502–518, (2003).

[9] Katalin Fazekas, Armin Biere, and Christoph Scholl, ‘Incremental inprocessing in SAT solving’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 11628 of *Lecture Notes in Computer Science*, pp. 136–154, (2019).

[10] Carla P. Gomes, Bart Selman, Nuno Crato, and Henry Kautz, ‘Heavy-tailed phenomena in satisfiability and constraint satisfaction problems’, *Journal of Automated Reasoning*, **24**(1), 67–100, (2000).

[11] HyoJung Han and Fabio Somenzi, ‘Alembic: An efficient algorithm for CNF preprocessing’, in *Design Automation Conference (DAC)*, pp. 582–587, (2007).

[12] Marijn J. H. Heule, Matti Järvisalo, and Martin Suda, eds. *Proceedings of SAT Competition 2018: Solver and Benchmark Descriptions*, volume B-2018-1 of *Series of Publications B*. Department of Computer Science, University of Helsinki, 2018.

[13] Marijn J. H. Heule, Matti Järvisalo, and Martin Suda, eds. *Proceedings of SAT Race 2019: Solver and Benchmark Descriptions*, volume B-2019-1 of *Series of Publications B*. Department of Computer Science, University of Helsinki, 2019.

[14] Matti Järvisalo, Marijn Heule, and Armin Biere, ‘Inprocessing rules’, in *International Joint Conference on Automated Reasoning (IJCAR)*, volume 7364 of *Lecture Notes in Computer Science*, pp. 355–370, (2012).

[15] Jia Hui Liang, Vijay Ganesh, Pascal Poupart, and Krzysztof Czarnecki, ‘Learning rate based branching heuristic for SAT solvers’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9710 of *Lecture Notes in Computer Science*, pp. 123–140, (2016).

[16] Mao Luo, Chu-Min Li, Fan Xiao, Felip Manyà, and Zhipeng Lü, ‘An effective learnt clause minimization approach for CDCL SAT solvers’, in *International Joint Conference on Artificial Intelligence (IJCAI)*, pp. 703–711, (2017).

[17] João P. Marques-Silva, Inês Lynce, and Sharad Malik, ‘Conflict-driven clause learning SAT solvers’, in *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, 131–153, (2009).

[18] João P. Marques-Silva and Karem A. Sakallah, ‘GRASP: A new search algorithm for satisfiability’, in *International Conference on Control, Automation and Diagnosis (ICCAD)*, pp. 220–227, (1996).

[19] Irkutsk Supercomputer Center of SB RAS. http://hpc.icc.ru.

[20] Sibylle Möhle and Armin Biere, ‘Backing backtracking’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 11628 of *Lecture Notes in Computer Science*, pp. 250–266, (2019).

[21] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik, ‘Chaff: Engineering an efficient SAT solver’, in *Design Automation Conference (DAC)*, pp. 530–535, (2001).

[22] Alexander Nadel and Vadim Ryvchin, ‘Chronological backtracking’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 10929 of *Lecture Notes in Computer Science*, pp. 111–121, (2018).

[23] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon, ‘Impact of community structure on SAT solver performance’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *Lecture Notes in Computer Science*, pp. 252–268, (2014).

[24] Chanseok Oh, ‘Between SAT and UNSAT: the fundamental difference in CDCL SAT’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 9340 of *Lecture Notes in Computer Science*, pp. 307–323, (2015).

[25] Cédric Piette, Youssef Hamadi, and Lakhdar Sais, ‘Vivifying propositional clausal formulae’, in *European Conference on Artificial Intelligence (ECAI)*, volume 178 of *Frontiers in Artificial Intelligence and Applications*, pp. 525–529, (2008).

[26] Laurent Simon, ‘Post mortem analysis of SAT solver proofs’, in *Pragmatics of SAT (POS)*, volume 27 of *EPiC Series in Computing*, pp. 26–40, (2014).

[27] Nathan Wetzler, Marijn Heule, and Warren A. Hunt Jr., ‘DRAT-trim: Efficient checking and trimming using expressive clausal proofs’, in *International Conference on Theory and Applications of Satisfiability Testing (SAT)*, volume 8561 of *Lecture Notes in Computer Science*, pp. 422–429, (2014).