On the Non-degeneracy of Unsatisfiability Proof Graphs Produced by SAT Solvers

Rohan Fossé(✉) and Laurent Simon(✉)

Labri, University of Bordeaux, UMR 5800, 33405 Talence Cedex, France
{rfosse,lsimon}@labri.fr

Abstract. Despite the important effort in developing fast and powerful SAT solvers, many aspects of their behaviors remains largely unexplained. We analyze the properties of learnt clauses derived by a typical Conflict Driven Clause Learning algorithm (CDCL) and study how they are linked to their ancestors, in the dependency graph generated by the resolution steps during conflict analysis and clauses minimizations. We show that all these graphs share a common structure: they are non k-degenerated with surprising large values, which mean they contain a very dense subgraph, the K-Core. We unveil the existence of large K-Cores, even on parallelized SAT solvers with clauses exchanges. We show that the analysis of the K-Core allows a good prediction of which literals will occur in future learnt clauses, until the very end of the computation. Moreover, we show that the analysis of the graph allows to identify a set of learnt clauses that will be necessary for deriving the final contradiction. At last, we demonstrate that the analysis of the dependency graph is possible with a reasonable cost in any CDCL.

1 Introduction

Since the introduction of the Conflict-Driven Clause Learning (CDCL) framework [13,14], the SAT technology has entered a new era. Solvers are now relying on *lookback* techniques rather than *lookahead* ones, which make the analysis of current methods harder. At the age of the Davis Putnam Logemann Loveland (DPLL) procedure [4,10], typically before the 2000's, the need for studying the behavior of algorithms was indeed less crucial. DPLL was a typical systematic backtrack search algorithm, relying on strong (costly and lookahead-based) heuristics for decisions: solvers were often spending most of their time at each node of the search tree computing their heuristics values to make careful decisions. As a consequence, the architecture of these solvers was mostly understood by the mathematical definition of the heuristics allowed to infer some general results on the size of the search tree [11], at least on random instances. On more structured problems, heuristics gave a very strong intuition explaining why the algorithm was working efficiently (*e.g.* branch on most frequent and balanced variables in shortest clauses, ...).

This work was supported by the French Project SATAS ANR-15-CE40-0017.

J. Hooker (Ed.): CP 2018, LNCS 11008, pp. 128–143, 2018.
https://doi.org/10.1007/978-3-319-98334-9_9

Within a few years, thanks to the introduction of the CDCL framework, the picture had considerably changed. The systematic backtrack search of the original DPLL is now ensured by the learning mechanism in CDCL, but aggressive clauses database cleanings [2,6], very reactive heuristics and very fast restarts [7] makes the final procedure very complex to analysis. More importantly, the state of the search in a CDCL is not static anymore (*i.e.* based on some counters analyzing the current formula at a point of the search tree). The state of the search is now based on the entire past of the solver. In addition, each component of so called "Modern" SAT solvers are tightly connected together and any intrusive change in any of them may have considerable side effects on other components. It is thus very difficult to study these solvers, and we argue that we need to consider them as *complex systems*, needing an important experimental study in order to understand their strengths and weaknesses.

This work can be viewed as extending a few previous works [1,8] that tried to connect theoretical measures with observed behavior of SAT solvers. However, no previous work has focused on some structural properties of generated proofs. In [1], they used a modified version of `satz` [10] to study the evolution of the space needed by the solver on a set of random and industrial problems. In another work [8], it was proposed to build a particular set of formulas (pebbling puzzles) to study the relationship between the minimal proofs for the initial formula and the behavior of CDCL solvers. The shape of proofs produced by CDCL solvers was also previously identified as a possible bottleneck for their efficient parallelization [9]. In some sense, we follow the same kind of idea in this paper but we focus on demonstrating a very particular structure of the proofs produced by sequential and parallel solvers. Our study is inspired by the work of [18] that already proposed to study the proofs, post-mortem, but by extending this work and focusing our study on the existence (and importance for the search) of a very dense subgraph (the K-Core) in all the proofs produced by CDCL SAT solvers.

Our hypothesis is that the existence of a K-Core strongly forces the search of the SAT solver, which implies that the study of K-Cores of dependency graphs is a necessary step towards a better understanding of SAT solvers. Our paper is thus an experimental paper, the aim of which is to report the existence – and the importance – of a dense subgraph in the dependency graph generated by SAT solvers and not (yet) to improve the performances[1] We organized our work as follows: after a few preliminaries presenting the essential notions of SAT solvers and K-Cores in graphs, we demonstrate the presence of K-Cores in dependency graphs produced by CDCL SAT solvers. Then, we show that the analysis of the dependency graph allows to make good predictions of the usefulness of learnt clauses. Then, we extend our findings in two directions. Firstly, we show that our results can be generalized to parallel proof. Secondly, we show that the analysis of the dependency graph can be done in any CDCL SAT solver with no additional memory required and at very small cost.

[1] For the reviewing process, figures are in colors. We plan to make two versions of the paper if accepted, with a black and white version of the figures for the proceedings.

2 Preliminaries

We assume the reader familiar with SAT but let us just recall here the global schema of CDCL solvers [3,6,14]: a branch is a sequence of decisions (taken accordingly to the VSIDS heuristic), followed by unit propagations, repeated until a conflict is reached. Each decision literal is assigned at a distinct, increasing, decision level, with all propagated literals assigned at the same decision level. Each time a conflict is reached, a series of resolution steps, performed during conflict analysis, allows the solver to extract a new clause to learn. This clause is then added to the clause database and a *backjumping* is triggered, forcing the last learnt clause to be unit and then propagated. Solvers also incorporate other important components such as preprocessing [5], restarts and learnt clause database reduction policies. It was shown in [2] that the strategy based on Literal Block Distance (LBD) was a good way of scoring clauses. The LBD is computed during conflict analysis: it simply measures the number of distinct decisions levels occurring in the learnt clause.

Parallel SAT solvers are roughly independent SAT engines (duplicating all their clauses) that can exchange clauses after each conflicts following some politics. Classically, binary clauses and clauses of small LBD are exchanged between solvers, even if more sophisticated techniques have been proposed.

2.1 Dependency Graph Induced by the Proof

The idea behind our study relies on the earlier work of [18], by extending it in many ways. We base our study on the same notion of *dependency graph* produced by a CDCL solver (we took `Glucose` as a reference, which the author of [18] gave us). Each conflict generates a new node in the graph (nodes in the dependency graph either match an original or a learnt clause, and for simplicity we may refer to a node or a clause for the same object in the following), and an (oriented) edge is added from each learnt clause to all its ancestors: all the clauses viewed during conflict analysis or clause minimization, oriented from the learnt clause to its ancestor.

If the notion of Dependency Graph (DG) holds for SAT and UNSAT formulas, we focus in this article on UNSAT formulas only (except in Sect. 6), for which the DG can be considered as an UNSAT (resolution) proof. Such a proof is thus here a direct acyclic graphs (DAG), with a subset of original clauses as leaves, learnt clauses as internal nodes and the empty clause at the top. We added the ability of `Glucose` to produce a DG. For this, we had to consider the special case of unary clauses. In `Glucose` and `Minisat`, unary clauses are simulated by adding a virtual decision at level 0, forcing the assignment of the literal occurring in the unit-clause. Here, we had to keep unit clauses in memory too, in order to keep track of them during conflict analysis.

Learnt clauses are totally ordered by the number of conflicts they were produced at. In addition, for each clause, once the total DG is generated, we compute a set of features. First of all, its *usefulness*. We call *useful* a clause necessary of the proof, and *useless* if it's not. Useful clauses are in other words the clauses

connected with the top clause. Do notice that, with this definition of usefulness, we do not consider useful a clause that would have been crucial for propagating a literal during the search, if no resolution was done on this literal during any conflict analysis. The generalization of DGs to parallel SAT solvers proofs is easy, by simply considering clauses exchanged by SAT solvers as unique, to keep track of the origin of each clause (parallel proofs will be considered only in Sect. 5).

In graph theory, a **k-degenerate** graph is an undirected graph in which every subgraph has a vertex of degree at most k. Similarly, a k**-core** of a graph G is an undirected subgraph of G in which all vertexes have degree at least k [16]. Thus, a k-core can be seen as a certificate for the non k-degeneracy of the graph. We are here interested only in the maximal k-core of graphs, such that the graph is not k-degenerate but (k+1)-degenerate. We will use the notation K-core for the maximal k-core of the graph. As these notions are defined over directed graphs, we will simply consider DG as an undirected graph for computing them. We also defined the notion of *coreness* of a graph as the value of the largest k such that the graph is not k-degenerate.

2.2 Selection of UNSAT Problems

Modern SAT solvers can run for millions of conflicts, each one involving hundreds of resolution steps. They may thus quickly produce very large graphs preventing any costly analysis. In this article, we will consider two sets of problems. One suitable for in-depth analysis (described below), and one demonstrating that our findings on relatively small problems holds on larger classical problems (all problems from the last 5 competitions, from 2013 to 2017, see Sect. 6).

For the first part of our analysis, as aforementioned, we needed a set of not too hard, not too easy problems. For simplicity, we took the same approach as [18] and selected the same set of 60 problems from past competitions, selecting as many distinct series of problems as possible. The strategy to select the 60 UNSAT problems was to choose at least two benchmarks per family of problems that needed less than one million conflicts to be solved on the original formula (non shuffled, after preprocessing). In the same family, "harder" benchmarks were selected first (thus trying to limit the number of too easy problems). Once the benchmarks were selected, the `SatElite` preprocessing [5] was used on each of them. When mentioned, we also considered shuffled versions of these problems (random reordering of clauses order, literals positions in clauses and random renaming of literals). Without any other precisions, median values are considered when reporting statistics over shuffled problems. We used 50 shuffled instance per original problem. We used the exact same list of problems than in [18]. The interested reader can refer to this previous work for more details on the list of benchmarks.

Experiments were conducted on a cluster of Xeon E7-4870 processors from the *Mesocentre Aquitain de Calcul Intensif* with at least one hour CPU time (more CPU time will be allocated for simulating parallel solvers in the later).

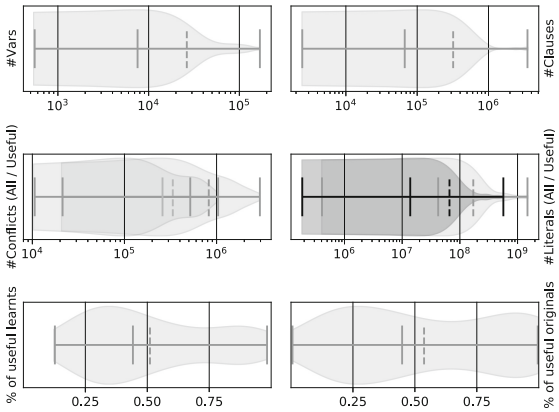


Fig. 1. Violin plots showing basic statistics on Dependency graph over the 60 problems. Dashed line is the mean value, middle line the median (over 50 shuffled instances). From left to right: Number of initial variables and clauses. Second line: number of internal nodes (or conflicts) and sum of learnt clause sizes. In light gray (more padded to the right) the total number and in darker gray the useful ones. Last line: percentage of useful learnt clauses and useful original clauses. All numbers are median values over shuffled and original problems.

2.3 Basic Dependency Graph Properties

Some important properties about DG were already identified in [18]. It was for instance observed that, on average, only 50% of learnt clauses were useful. Moreover, only 21% of clauses that were unit-propagated were seen in any conflict analysis. Given the fact that, in order to be used again in another conflict analysis, a clause must be unit-propagated again (due to the backjumping mechanism that unset all literals seen during last analysis), only a little more than 10% of unit propagations are used to derive a useful clause. If we approximate the time taken by a CDCL by the time taken by its unit-propagation engine (which is a reasonable assumption), we can thus observe that 90% of the time taken by a CDCL is "useless" (or only useful to update branching heuristics, ...). Understanding the characteristics of the Dependency Graph may be thus crucial to improve CDCL performances. Figure 1 shows some of the main characteristics of the formulas and the DG graphs we studied. It confirms the above conclusions made in [18]. We can also check that the formulas are indeed well chosen: they have different sizes and the median effort to solve them is around 500,000 conflicts, with a maximum of a few millions ones.

3 Characterization of K-Cores

To give a first intuition of the high density of DG generated by SAT solvers, we represent Fig. 2 a graphical representation of a very easy problem (less than 100,000 conflicts). Do notice the rotation of the graph around a very dense center:

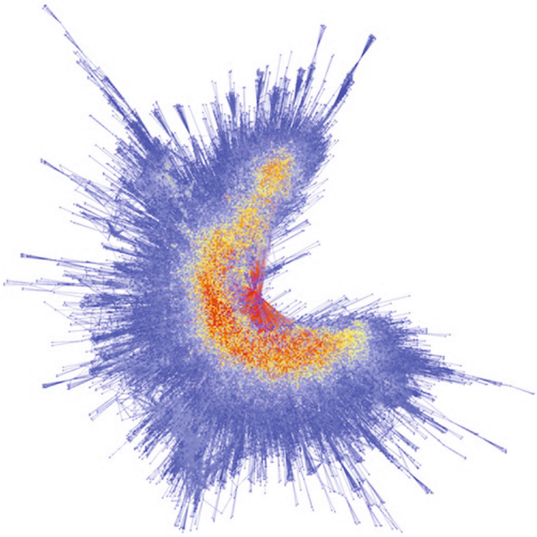


Fig. 2. Force-Directed layout of the Dependency Graph for the benchmark `een-pico-prop-05`. The color shows the *coreness* of the node.

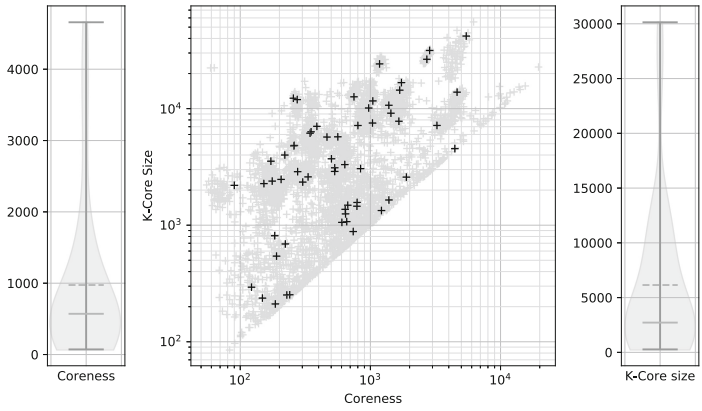


Fig. 3. Characteristics of K-Cores in the set of 60 problems. In the middle, darker plots are original problems, lighter shuffled problems. A log scale has been used to represent the wide range of obtained values. On left and right, we show the distribution of median values over shuffled instances.

the K-Core. Figure 3 shows the K-Core values (the coreness of the graph) and the sizes of the corresponding subgraph on the set of 60 problems, including shuffled ones. Let us first focus on the two violin plots. We can see (left) that the K-Cores can be very large. The median is larger than 500 and some problems can have K-Core values of a few thousands. On the right, we can see that the dense

subgraphs can be quite large too, containing typically more than 2,000 clauses. Now, if we focus on the scatter plot, it is striking to see how many problems are very close the $y = x$ line. Those problem have a K-Core size that is very close to the K-Core value, showing almost a clique of clauses as a K-Core.

3.1 Evolution of K-Cores Along the Computation

Let us now study how the K-Core evolves along the computation. Figure 4 shows its characteristics at two points of the search w.r.t its final values. For this, we computed the K-Core values considering (1) only the first 20,000 conflicts and (2) only the first half of the conflicts. The first set of points (after 20,000) conflicts shows that the computed values can be very far from the final values. However, it is very encouraging to notice that, at half of the run, the values are really close to the final values, showing that a study of the K-Core at this point of the computation may probably provide good informations about the final K-Core.

One of the main hypothesis in this work is that the K-Core is strongly forcing the search of the SAT solver into a close search space. To illustrate this, let us discuss now the results shown Fig. 4-right. The figure reports the CDF of the distance of a certain percentage of learnt clauses to the K-Core, in terms of resolutions. Clauses in the K-Core are at distance 0. A learnt clause obtained by resolution with at least a clause of distance n is at most at distance $n + 1$. Similarly, a clause used in conflict analysis to produce at least a clause of distance n is, also, at most at distance $n+1$ (we thus consider distances on the undirected DG). On Fig. 4-right, each curve correspond to a CDF plot. The CDF for P% shows how many problems (x) have at least P% of its learnt clauses at distance smaller than y. We considered here all the problems (original and learnt). We can see that, in the very large majority of the cases, at least half of the learnt clauses are at distance 5 or less. It is also striking to notice that, on our selection

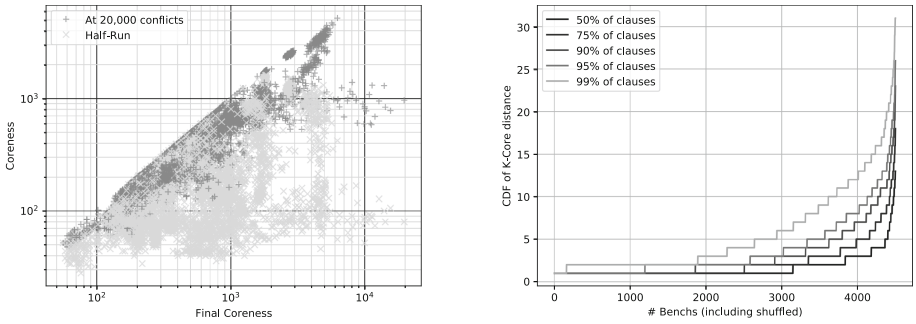


Fig. 4. (Left). Evolution of the K-Core characteristics after 20,000 conflicts and at the middle of the run (in terms of conflicts). (Right) Distance of the clauses w.r.t. the K-Core. Original and 50 shuffled versions of the 60 problems are considered. A very large fraction of the learnt clauses are often very close to the K-Core.

of problems, half of the problems have 99% of the learnt clauses at distance 5 or less. A distance of 15 resolutions seems also to be a very good bound in the majority of the cases. Most of the learnt clauses are very close to the K-Core in terms of resolutions.

3.2 K-Cores Structure

Let us now say a few words on the structure of the K-Core by reading the Fig. 5. First of all, let us point out that it is composed by original and learnt clauses, with a majority of learnt clauses, but with more than 30% of original clauses (see the median and mean values of the top-right violin plot). More surprisingly, despite its central role in the creation of all the learnt clauses (see section above), it is not entirely composed of useful clauses (see top-left violin plot): only around 75% of its clauses are used to derive the final contradiction. Let us now focus on the sizes of clauses in K-Core (see the 4 bottom plots of Fig. 5). By construction, original clauses are often limited in size (SAT encodings may prevent very large clauses to be built, and it is common to have a very large majority of binary clauses), but also contain a few large clauses. However, it is striking to see how short original clauses are in the K-Core (the median of the clauses are binary or ternary clauses only). This is emphasized with the comparison of the same statistics values, but not restricted to original K-Core clauses: the plot in darker gray shows these values computed on all the original clauses of each problem. We can thus see that original K-Core clauses are indeed short, compared to

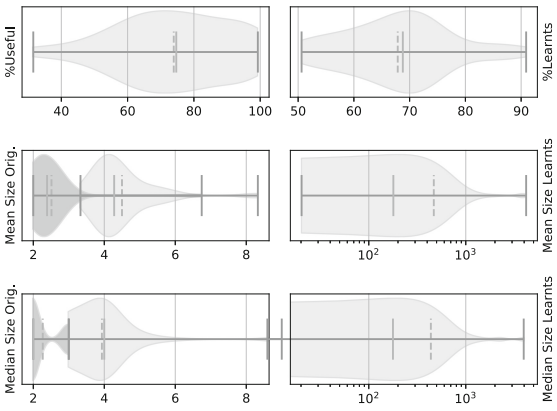


Fig. 5. The two top violin plots represents the percentages of useful clauses (left) and learnt clauses (right). The 4 others violin plots summarize the sizes of the clauses in the K-Core, split between original and learnt clauses. Darker gray violins (padded at the left) are the same values on the whole original problem (instead of original clauses in K-Core in light gray). A log scale has been used for reporting sizes of learnt clauses in the K-Core. All numbers are median values over 50 shuffled and original problems over our set of 60 problems.

the formula. As opposite, we had to use a log scale to represent the very large discrepancy in the size of K-Core learnt clauses. Here, clauses larger than 100 are frequent.

Two conclusions can be drawn from this experiment. Firstly, it is interesting to notice that some clauses of the K-Core, despite its central role in the learning mechanism, are useless. This is potentially an interesting point for improving SAT solver performances: identifying these clauses could potentially help the solver not generating useless clauses. Secondly, the fact that the K-Core contains very large clauses casts also a new light on detecting important clauses. We also observed that K-Core clauses are not necessarily clauses of small LBD (experiment not reported here), and thus there is a high chance that they could be removed during the clause database reduction. In order to explain this apparent paradox, we make the hypothesis that these clauses stay because of an implementation trick in the clause database reduction: in `Glucose`, only clauses of small LBD are kept but things are a little bit more complicated. In fact, the clause database reduction is generally triggered right after a regular backjump after a conflict analysis, but not after a restart. During the reduction, clauses that are currently unit-propagated are not removed. Thus, very large clauses that are very likely to be unit-propagated are also likely to be kept, which is probably the case of K-Core clauses.

4 On Predictions Based on Dependency Graph Analysis

The existence of large K-Cores is something that is not uncommon in real-life graphs, in which its analysis can even be used to detect, for instance in social graphs, to efficiently detect the most important nodes [17], *i.e.* nodes that have the most influence on other nodes. We tried a few measures and report here two interesting results we obtained. The first one tries to identify which clauses will be useful. The second measure tries to identify literals that will occur frequently in learnt clauses until the very end of the computation. In both experiments, we report the analysis of the DG after 20,000 conflicts and at half-run, by simply removing from DG all the nodes and edges added after the limit.

4.1 Predicting Useful Clauses

In this experiment, we used a flow algorithm on the DG, without considering the K-Core. We initialize all root nodes (clauses without descendants) with a constant weight and propagate the weight of each node to its ancestors by dividing the weight equally between them (following a topological order). For each node, the idea is to measure the clauses that occurs in the maximal number of paths to a root node (the graph can have many root nodes before the end).

The results we obtained are summarized Fig. 6 (left and right). Figure 6-left shows the results of our prediction by ranking 10,000 clauses according to the above flow values. Of course the darker curves (finished rune) give good results, even if they must not be considered has having any practical interest (it is easy

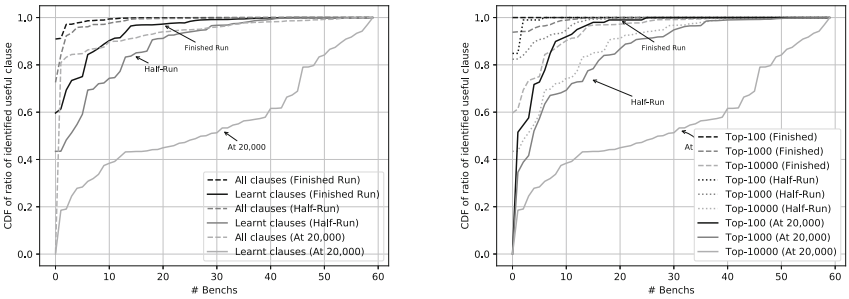


Fig. 6. (Left): Fraction of clauses that have been predicted as useful and that are useful. "All clauses" includes 10,000 clauses that can be original or learnt. "Learnt clauses" restricts the computation to only learnt clausesd. A prediction is made after 20,000 conflicts, at half-run, and at the end of the run. Reported values are median over original and 50 shuffled problems. (Right): Other parameters for the same experiment. Top-10000 curves are also represented on the left figure.

at the end to detect useful clauses with no errors by another method). However, we can see that a simple flow algorithm, with no knowledge of which root node is the contradiction, can already identify useful clauses with a good precision.

The second observation follows the bottom light gray curve (at 20,000 conflicts), which clearly gives very bad results for identifying useful learnt clauses. This is in fact not surprising given the fact that we try to identify 10,000 good learnt clauses after only 20,000 conflicts. Here, our technique is possibly not better than a random guess. However, if we try to predict 10,000 useful clauses including original ones, we see that we correctly guess 80% of the 10,000 clauses, probably because we mostly bet on original clauses. This is already encouraging: we can identify useful original clauses at the very beginning of the computation. We can imagine, for instance, a search procedure that focuses on working on these clauses in priority. More importantly, we can see that, at half of the computation, results are very good: on half of the problems, we correctly guess at least 90% of useful learnt clauses over 10,000 guesses. This is even better when considering original clauses too.

Let us now focus on Fig. 6-right. In this figure, we try to identify 100, 1000 or 10,000 clauses. What this figure shows is that, if detecting 10,000 clauses can somehow give bad results, we can identify, with a great confidence, 100 learnt clauses that will be useful in the future, even after only 20,000 conflicts. At half-run the prediction is even better (see the dotted CDF line). We have around 95% good classification for 100 learnt clauses.

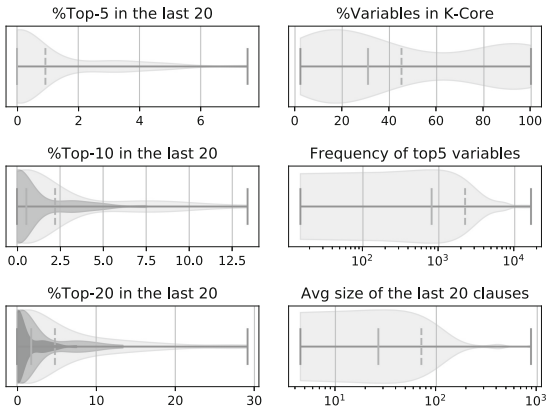
Let us temperate our findings. In fact, despite the importance of being able to detect a useful clause early in the computation (for instance for splitting the search space), it did not allowed us to improve `Glucose` yet. This is probably due to the fact that useful clauses may still have be deleted with no harm. It is for instance possible that only a descendant of the clause is needed for the remaining

computation, and thus our attempts to protect this clauses for further deletion did not helped. We are however still hopping that this kind of detection could help solvers to work on only part of the initial formula.

4.2 Detecting Future Learnt Clauses

In the above study, we were not able to detect which clauses will be important for the future of the search. In the following experiment, we try to guess which variables will occur in the last learnt clauses, just before deriving the final contradiction. For this, we propose to study the variables occurring in the K-Core at half of the computation. One first idea could be to consider all K-Core's variables, but, as we can see on the top-right violin plot Fig. 7, some problems can have large K-Cores, containing all their variables. We thus propose to simply consider only the most frequent variables in the K-Core (the 5, 10 and 20 most frequent variables, respectively). The middle-right violin plot shows how frequent the top-5 variables are (median larger than 800) in K-Cores.

Let us now comment the left part of Fig. 7. These plots report the percentage of top variables we observe in the very last learnt clauses of the computation. The top plot shows the percentage of the top-5 variables that occurs in the last 20 clauses, w.r.t the total number of literals in the last 20 clauses (some clauses can be very long, see the bottom-right plot). We can see that, on average, 1% of the literals in the last 20 learnt clauses are from the top-5 variables in the K-Core. Numbers are of course better if we consider the top-10 and top-20 variables (on the violin plots below, we also plots the above distribution plots to emphasize the differences in the prediction with 5, 10 and 20 variables). We already observe a pretty good prediction, given the very large number of variables of our problems (median at 8,000 variables, see Fig. 1). We can also report our predictions by the



	20	50	100	1000
5	27	37	45	53
10	42	47	50	54
20	49	51	51	55

(See caption for table description)

Fig. 7. (Left) Results of our prediction for literals occurring in the last learnt clauses. (Right) Over 60 problems, how many problems have at least one of the top-Y variables (rows) in the last X learnt clauses (columns)

lens of the table, right side of the same figure. We can see that for 27 problems over the 60 ones, at least one of the top-5 variables we identified occurred at least once in the last 20 clauses. At least a variable from the top-20 occurs in the 20 last learnt clauses, for 49 problems over the 60 (median values on all original and shuffled problems).

We did not have enough space to report another interesting experiment we also performed, but we noticed that, generally, the top variables are assigned by learning a unit clause only at the very end of the computation.

5 Analysis of Parallel Proofs

As we mentioned it in the first sections of this paper, sequential CDCL solvers are not well understood. The question is even more critical for parallel SAT solvers. They are generally a portfolio of SAT solvers exchanging clauses, based on some criterion (size, LBD are the most common ones). Clauses are thus shared amongst threads but no study have ever been conducted on the impact of parallelizing search over the final proof. Here, we study how the proof evolves w.r.t. the number of threads. For this, we simulated a parallel SAT solver on the top of our `Glucose` solver, hacked to handle dependency graphs. In our simulation, each simulated thread successively generates a clause by conflict analysis and offer to share it. Before each conflict, each thread can thus pick any last learnt clause by the other threads. In our setting, we imported clauses that had size strictly smaller than 8 and LBD strictly smaller than 4. We keep track of the origin of each imported clause. In the DG, imported clauses are not duplicated.

Let us now describe Fig. 8. On the two topmost sub-figures, each plot is a violin plot reporting a value for 1, 2, 4, 8, 12, 16, 20 and 24 cores, based on runs on the original problems only. We can firstly verify that parallelizing does not change the property of the proof: we have large K-Cores with a high coreness value. It can be observed that the K-Core seems to increase in comparison to the sequential version. Another finding is the evolution of the depth of the proof (initial clauses are of depth 0). Clearly enough, the depth is smaller as the number of cores increases. On the second sub-figure, we can see that, as opposite, the size of the proof tends to increase: proofs are shallower but larger as the number of threads increases. It is also interesting to measure how much effort is wasted when going parallel. "Useless learnts" reports the number of learnt clauses that do not occur in the proof. Clearly enough, the more we add threads, the more we produce useless clauses. This is emphasized by the "Efficiency" plot, that report the percentage of useful learnt clauses over the total number of learnt clauses. This clearly confirm the loss of efficiency in parallel solvers.

At last, we study, thanks to the bottom sub-figure of Fig. 8, the composition of the K-Cores in terms of clause origins (which thread produced the clause) with 24 cores. For each run, we measured the percentage of clauses from each thread in the final K-Core, and sorted it. Then, on the figure, we represent each curve in a cumulative way, normalized to 1.0. The result is in fact simple to read. Most of the problems have a K-Core composed by clauses from all the threads.

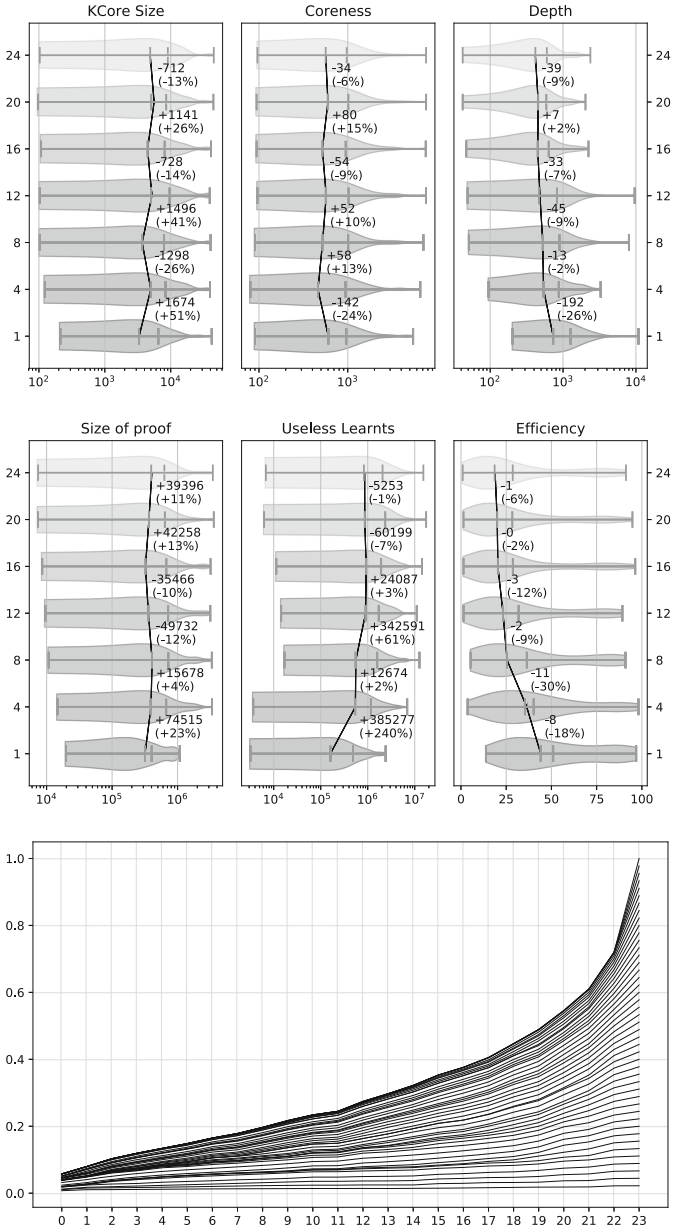


Fig. 8. (Two top figures) Characteristics of K-Cores when using from 1 to 24 threads. Changes in medians are kept track for each violin plot. Experiments are done on the 60 original problems, not shuffled. (Bottom) Cumulative plot of sorted origin of clauses in a 24-threads parallel solver over the 60 original problems

Some threads have a stronger presence in the Kcore, but not really significantly, except for one thread, especially on a small set of problems (most of the curves are almost "parallel" except for a few curves at the top of the plot: a perfect parallel curve to the curve below means that all threads are equally represented in the K-Core). Our conclusion is that, somehow surprisingly, the K-Core is composed of all the cores, despite the strong bottleneck in clauses exchanges. Even in a parallel setting, the presence of a K-Core strongly guide the search.

6 Fast Dependency Graph Analysis

Until know, we used a modified version of `Glucose` that kept track of all the dependencies (even for removed clauses). The memory consumption of our technique is thus not suitable for improving any CDCL performances: it is simply too costly to maintain all the informations. The goal of this short section is to show that is possible to analyze a simplified version of the DG in a regular CDCL with almost no cost. The idea is to built the DG just once, by using vivification on all the learnt clauses in memory at the time we want to build the DG [12,15] (of course, a lot of learnt clauses have been removed). This step, performed once in our setting after 110,000 conflicts, allows us to find reasons for each learnt clauses in memory (this is the same method that is somehow used when rebuilding DRAT proofs [19]). Of course we cannot find the original dependencies but, as we can see Fig. 9 we were able to find very similar results as previously reported (large coreness of the DG), despite the partial information we have about the current proof (limited to the set of current learnt clauses). Let us point out here that we were able to compute this values for all the problems from the 5 last SAT competitions (2013–2017 included). Do notice also that we included in this experiment SAT and UNSAT problems, as well as problems which timed out after the computation of the K-Core. This last

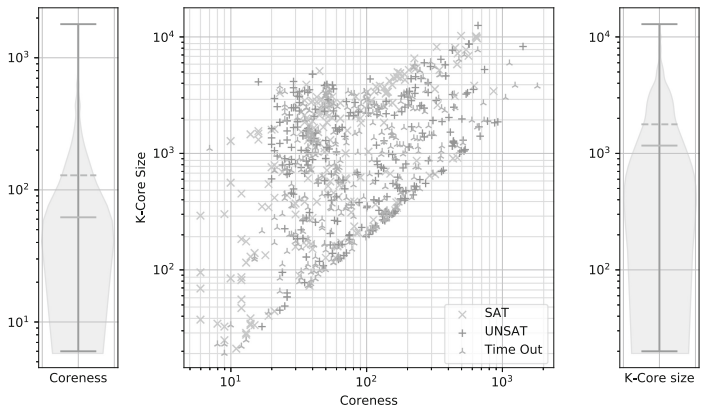


Fig. 9. Characteristics of K-Cores computed with vivification on the remaining learnt clauses after 110,000 conflicts, for all the problems from competitions 2013 to 2017.

experiment clearly demonstrates that the presence of K-Cores is a strong and general characteristics of DG, for SAT and UNSAT problems (do notice also that, on many problems, we also observed a K-Core size of the coreness value, showing a number of problems with large cliques in their DG).

7 Conclusion

Our experimental report points out a hidden structure behind the learning mechanism of CDCL SAT solvers. We demonstrated that all the proof graphs generated by SAT solvers share a common important characteristics: dependency graphs are non-degenerated with high values. This shows that there is a very dense subgraph that seem to play an important role in the search. In order to emphasize this role, we demonstrated that, by analyzing the dependency graph at half of the run, we were already capable of identifying a set of learnt clauses that will be necessary for deriving the final contradiction, with high confidence. The analysis of the K-Core also allowed us to identify a very small set of variables that will occur in the very last learnt clauses. We also demonstrated that most of the learnt clauses are very close to the K-Core in terms of resolution steps. We demonstrated that, even in parallel, a K-Core composed by clauses of all the threads is also dominating the proof. Additionally, parallel proofs are generally larger, even if they are shallower.

At last, we shown that the analysis of the dependency graph can be done in any CDCL solver at minimal cost. It is sufficient to analyze the current set of learnt clauses to build a dependency graph that also exhibit a strong K-Core. This last experiment also demonstrates that the existence of K-Cores also holds for SAT and UNSAT problems. We hope that our work will cast new research directions on the reasons why SAT solvers are so efficient. We hypothesis that one of the reasons for their efficiency is their ability to produce and handle proofs with these properties. In some sense, this proves that CDCL proofs are really far away from tree-like resolutions (non-degenerate graphs are clearly far from trees). We also think that our analysis will allow to efficiently reduce the proof sizes generated by SAT solvers. A short proof for UNSAT can be essential in many applications.

References

1. Ansótegui, C., Bonnet, M.L., Levy, J., Many, F.: Measuring the hardness of sat instances. In: Proceedings of AAAI, pp. 222–228 (2008)
2. Audemard, G., Simon, L.: Predicting learnt clauses quality in modern sat solvers. In: IJCAI (2009)
3. Darwiche, A., Pipatsrisawat, K.: Complete Algorithms, Chap. 3, pp. 99–130. IOS Press (2009)
4. Davis, M., Logemann, G., Loveland, D.: A machine program for theorem proving. JACM **5**, 394–397 (1962)

5. Eén, N., Biere, A.: Effective preprocessing in SAT through variable and clause elimination. In: Bacchus, F., Walsh, T. (eds.) SAT 2005. LNCS, vol. 3569, pp. 61–75. Springer, Heidelberg (2005). https://doi.org/10.1007/11499107_5
6. 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
7. Huang, J.: The effect of restarts on the efficiency of clause learning. In: IJCAI 2007, pp. 2318–2323 (2007)
8. Järvisalo, M., Matsliah, A., Nordström, J., Živný, S.: Relating proof complexity measures and practical hardness of SAT. In: Milano, M. (ed.) CP 2012. LNCS, pp. 316–331. Springer, Heidelberg (2012). https://doi.org/10.1007/978-3-642-33558-7_25
9. Katsirelos, G., Sabharwal, A., Samulowitz, H., Simon, L.: Resolution and parallelizability: barriers to the efficient parallelization of sat solvers. In: Proceedings of AAAI (2013)
10. Li, C.M., Anbulagan, A.: Heuristics based on unit propagation for satisfiability problems. In: Proceedings of IJCAI, pp. 366–371 (1997)
11. Li, C.M., Gérard, S.: On the limit of branching rules for hard random unsatisfiable 3-sat. In: Proceedings of ECAI, pp. 98–102 (2000)
12. Luo, M., Li, C.M., Xiao, F., Manya, F., Lu, Z.: An effective learnt clause minimization approach for CDCL SAT solvers. In: Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence (IJCAI-17), pp. 703–711 (2017)
13. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)
14. Moskewicz, M., Madigan, C., Zhao, Y., Zhang, L., Malik, S.: Chaff: engineering an efficient SAT solver. In: Proceedings of DAC, pp. 530–535 (2001)
15. Piette, C., Hamadi, Y., Sas, L.: Vivifying propositional clausal formulae. In: ECAI, pp. 525–529 (2008)
16. Seidman, S.B.: Network structure and minimum degree. Soc. Netw. **5**(3), 269–287 (1983)
17. Shin, K., Eliassi-Rad, T., Faloutso, C.: Graph mining using k-core analysis - patterns, anomalies and algorithms. In: IEEE 16th International Conference on Data Mining ICMD, pp. 469–478 (2016)
18. Simon, L.: Post mortem analysis of sat solver proofs. In: Berre, D.L. (ed.) POS-14. Fifth Pragmatics of SAT Workshop. EPiC Series in Computing, vol. 27, pp. 26–40. EasyChair (2014). https://doi.org/10.29007/gpp8, https://easychair.org/publications/paper/N3GD
19. Wetzler, N., Heule, M.J.H., Hunt, W.A.: DRAT-trim: efficient checking and trimming using expressive clausal proofs. In: Sinz, C., Egly, U. (eds.) SAT 2014. LNCS, vol. 8561, pp. 422–429. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-09284-3_31