Post Mortem Analysis of SAT Solver Proofs

Laurent Simon*

University of Bordeaux,
Labri, UMR 5800, F-33405 Talence Cedex, lsimon@labri.fr

Abstract

Conflict-Driven Clause Learning algorithms are well known from an engineer point of view. Thanks to Minisat, their designs are well understood, and most of their implementations follow the same ideas, with essentially the same components. Same heuristics, fast restarts, same learning mechanism. However, their efficiency has an important drawback: they are more and more like complex systems and harder and harder to handle. Unfortunately, only a few works are focusing on understanding them rather than improving them. In most of the cases, their studies are often based on a generate and test pattern: An idea is added to an existing solver and if it improves its efficiency the idea is published and kept. In this paper, we analyse "*post-mortem*" the proofs given by one typical CDCL solver, Glucose. The originality of our approach is that we only consider it as a *resolution proofs* builder, and then we analyze some of the proof characteristics on a set of selected unsatisfiable instances, by shuffling each of them 200 times. We particularly focus on trying to characterize useless and useful clauses in the proof as well as proofs shapes. We also show that despite their incredible efficiency, roughly 90% of the time spent in a CDCL is useless for producing the final proof.

1 Introduction

Since the introduction of the Conflict-Driven Clause Learning (CDCL) framework [12, 14], the SAT technology has entered a new era. Solvers are now relying on *lookback* techniques rather than *lookahead* ones. In previous years, the Davis Putnam Logemann Loveland (DPLL) framework [4, 10] was dominating the field. It was typically a systematic backtrack, lookahead, search algorithm trying to reach a contradiction as early as possible in the tree. Thus, the solver spent most of its time at each node of the search tree computing its heuristics values to make careful decisions. As a consequence, the architecture of these solvers was mostly understood: 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, ...).

In a few years, the introduction of the CDCL framework has considerably changed the way SAT solvers are working. The systematic backtrack search of the original DPLL is now ensured by the learning mechanism in CDCL but aggressive clauses database cleanings [2, 6] with very fast restarts [7] makes this view obsolete. Moreover, the state of the search in a CDCL is not based anymore on a measure based on the status of the formula at the current search node. It is 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. Thus, it is very difficult to study these solvers,

*This work was supported by the French State, managed by the French National Research Agency (ANR) in the frame of the "Investments for the future" Programme IdEx Bordeaux - CPU (ANR-10-IDEX-03-02).

and we considering them as *complex systems* may be the right move to study them. In general, there is no other way to predict their behavior than running them.

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 is 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. Recently, the shape of proofs produced by CDCL solvers was identified as a possible bottleneck for their efficient parallelization [9]. In some sense, we follow the same kind of idea in this paper but only study basic shapes of the proof produced by sequential solvers, and ask ourselves some basic questions about it. For instance, how many clauses are useful for the proof? Are CDCL solvers producing useless clauses at some constant rate? How many initial clauses are natively used by CDCL solvers to derive the contradiction? Is there an observed relationship between widths of proofs and their sizes? We also try to see if there is a way to analyze, *post-mortem* the importance of clauses according to a set of measures (LBD, size, activity, age). All these questions are successively addressed in the next sections. We would like first to starts with some preliminaries and detailed motivations for this work.

2 Preliminaries and Motivations

This work does not claim to open ways for new families of CDCL solvers, or even lead to solvers direct improvements. Some of the reported numbers are just raw observations but reporting them may be interesting and sufficiently insightful. Most of the experiments are natural questions that may arise, but some parts of this paper may even raise more questions than give answers. However, we deeply think that understanding CDCL solvers is a very important question for the field, and this work aims at improving our knowledge.

2.1 What is a proof?

What we consider here as a proof is not the whole trace of the solver. We took here the standard, resolution-based, definition (as in [9]): A proof is a direct acyclic graphs (DAG), with input clauses as leaves, produced clauses as internal nodes and the empty clause at the top. Produced clauses are also totally ordered by the number of conflict they were produced at. We keep for each produced clause the set of its ancestors, *e.g.* the set of previous clauses that were used to derive it during a single conflict analysis. We also had to keep track of unary learnt clause (assigned at the decision level 0 in `Minisat`-like solvers [6] and then automatically (and lazily) removed during the remaining conflict analysis) and clause minimizations steps, occurring after the clause learning step, thus introducing possible additional resolution steps that are not obviously occurring in the main conflict analysis loop of a typical CDCL solver.

2.2 Useless clauses or *heuristics* clauses?

CDLC solvers are producing clauses. Some of them will occur in the final proof, some not. We call *useful* a clause necessary for the proof, and *useless*a clause that is not in the final proof. However, a number of precautions must be noticed here. The reader must understand that considering CDCL solvers as clause producers is only a degraded view of their efficiency. They rely on many components and, if we want to emphasize here a "proof producer" view

of them, it is also true that they may be very efficient, on some problems, by their efficient backtrack search mechanism only. We will not consider any branching mechanism or search space traversal by the solver here. All its past will be memorized by the set of clauses (with their ancestors). If this view does not take into account the whole complexity of CDCL solvers, it has the advantage of being precise and subject to a clear and formal analysis. Moreover, the notion of usefulness of a clause may be misleading. For instance, a clause ”useless” in the proof may be very important to guide the solver to the shortest contradictions (by unit propagation). However, we think that, (1) this view has the advantage to be simple and well defined and (2) it can be used as very good ”core sample” of all the past activities of any solver. Thus, it may be sufficient to draw some conclusion or precise some questions. More importantly, if a clause is not useful for any resolution step during conflict analysis or clause reduction, then a perfect branching heuristics could allow the solver to produce exactly the same proof, without any useless clause.

2.3 90% of the time spent by a CDCL is useless

If a unit propagation is useful for a conflict analysis (thus a resolution is made upon the propagated literal), then this unit propagation will be removed right after, when backjumping (the propagated clauses is viewed at the last decision level). Thus, an interesting 1-to-1 matching easily arise: to be used for resolution, a clause has to be propagated, and a propagated clause used in a resolution step will have to be propagated again before being used in another resolution. A simple experiment can thus compare the number of propagations w.r.t the number of resolutions during conflict analysis. This experiment does not take into account the minimization stage but can already give a very good intuition of useful/useless unit propagations (used or not in a resolution step).

On our set of 60 selected instances (see 2.4 for their description), we observed that only 21% of the fired unit propagations by the solver are useful in resolution steps. Moreover, as we will show in this paper, only around 50% of the produced clauses are really useful. Thus, a perfect solver could see a $\times 10$ improvement, without any improvement on the final proof length, if all unit propagations would be used in a resolution step for producing a useful clause. This is an interesting new possible way of improving SAT solvers. They are often considered very efficient because of their Boolean Constraint Propagation (BCP) engines but, from a usefulness point of view, only 10% of the unit propagations are useful in the final derivation for the contradiction. Thus, if they are indeed efficient for fast (and blind) BCP, they cannot really be considered as efficient *useful* BCP engines. From a raw UNSAT proof point of view, 90% of the time spent by the solver is thus useless or can be only viewed as some kind of “heuristics”. However, clearly enough, identifying good resolutions to perform is very hard and it may be much more efficient to just blindly propagate as many literals as possible and only gather conflicting ones, rather than designing complex and costly heuristics.

2.4 Experiment set up

Tests were done upon the `Glucose` [2] solver. We used two clusters of computers. Each node of the first one had 12 cores and 98Gb memory (used for non-shuffled experiments). Each node of the second one had 8 cores and only 24Gb memory (used for shuffled experiments: a larger number of nodes was available). Because of memory issues, we only launched one solver per node (on 12/8 cores). We considered in this experiment 200 shuffling launches per instance, launched on the second cluster (even if 24Gb may not be sufficient to keep all the informations about the current proof) and we used a CPU timeout of 3600s (for the second cluster only,

no time out was used for the non shuffled instances). In order to report as many experiments as possible, we selected only 60 benchmarks over all the UNSAT problems found in the last competitions, and only considered them after `SatElite` preprocessing [5]. The strategy was to select, on the non shuffled problem, at least two benchmarks per family that needed less than one million conflicts to be solved on the original formula. In the same family, "harder" benchmarks were selected first (thus trying to limit the number of too easy problems).

Shuffling was done by randomly reordering clauses and variables[1]. Reports on shuffled instances are done by reporting the median number of all the 200 launches for each benchmark and for each considered statistics (average can be reported when mentioned).

Of course, we could have conducted our experiments on as many problems as possible, instead of shuffling a selection of them. However, our choice was motivated by the fact that (1) shuffling gives more strong statistical evidences, (2) selecting problems allows to balance the importance of different families of benchmarks (as we'll see, many results only holds for some families of problems) and (3) we can focus on reasonable problems. Extending our results to a larger set of problems is of course of interest. The complete list of used problems is:

`ProVE07-03, traffic_3b_unknown, q_query_3_144_lambda, traffic_kkb_unknown, rpoc_xits_07_UNSAT, q_query_3_145_lambda, velev-pipe-o-uns-1.1-6, goldb-heqc-dalumul, comb1.shuffled, velev-vliw-uns-4.0-9-i1, post-cbmc-aes-d-r2, post-cbmc-aes-ee-r2-noholes, rand_net60-40-10.shuffled, maxxororand032, hwmcc10-timeframe-expansion-k50-pdtpmsns2-tseitin, 9dlx_vliw_at_b_iq6.used-as.sat04-347, ibm-2002-25r-k10, isqrt2_32, SAT_dat.k80, goldb-heqc-frg1mul, velev-vliw-uns-2.0-uq5, isqrt1_32, jarvi-eq-atree-9, SAT_dat.k100, AProVE07-21, velev-pipe-o-uns-1.0-7, maxxor032, korf-15, manol-pipe-c10nidw, simon-s02-f2clk-50, k2fix_gr_rcs_w9.shuffled, hwmcc10-timeframe-expansion-k45-bobsm5378d2-tseitin, UCG-20-5p0, post-c32s-ss-8, manol-pipe-c8nidw, ibm-2002-22r-k60, cmu-bmc-longmult13, E02F17, cmu-bmc-longmult15, countbitsrotate016, c10idw_i, 6pipe_6_ooo.shuffled-as.sat03-413, mulhs008, velev-engi-uns-1.0-4nd, UTI-15-10p0, rbcl_xits_06_UNSAT, UTI-10-10p0, smtlib-qfbv-aigs-lfsr_008_063_080-tseitin, E05X15, een-pico-prop05-75, simon-s03-fifo8-400, schup-l2s-abp4-1-k31, cmu-bmc-barrel6, rand_net60-25-10.shuffled, UCG-15-5p0, smtlib-qfbv-aigs-lfsr_008_079_112-tseitin, sokoban-sequential-p145-microban-sequential.030-NOTKNOWN, dekker.used-as.sat04-989, ACG-10-10p0, hoons-vbmc-lucky7`

3 Size of proofs

We are interested in this section in very basic and natural questions, essentially around the size of the proofs. We first try to identify some relationship between the number of produced useful and useless clauses.

3.1 Useless and useful produced clauses

The first fact to observe is certainly how many useless clauses are produced during the computation. We already mentioned that roughly 50% of clauses are useless. This is reported figure 1. This figure shows a scatter plot of useless clauses against useful clauses on the set of shuffled/original instances. We do not consider input clauses here. Despite some particular problems that have many more useful clauses than useless clauses, most of the problems seem to demonstrate that CDCL have a general tendency to produce as many useless clauses as useful. The inner subfigures confirms this. It shows the CDF plot of the ratio of useful clauses (Y-axis) in the final proof over all finished problems (X-axis). Half of the problem have at least a 0.5 ratio. Moreover, the shape of the CDF seems to confirm that the set of tested problems is well

[1]Reordering positions of literals in clauses has no impact, because `Glucose` is ordering clauses literals lexically

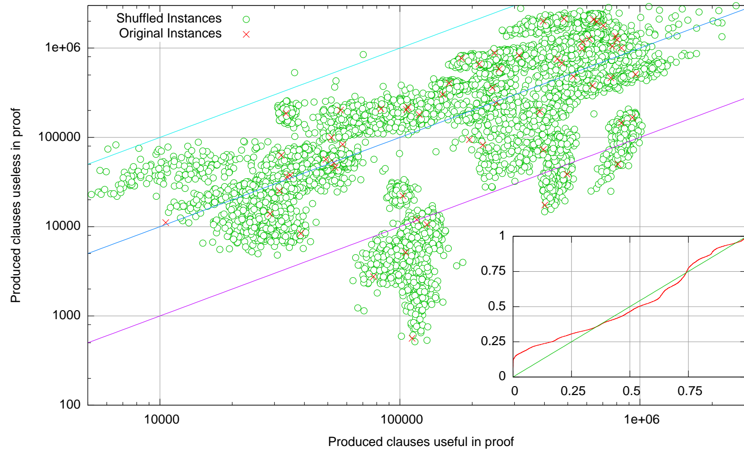

Figure 1: Useless and Useful clauses in `Glucose`, on shuffled problems. The inner subfigure is the CDF plot of the ratio of useful clauses (Y-Axis) in the final proof over all tested instances (Normalized, X-Axis), on the shuffled instances only.

chosen and particularly heterogeneous. We have a very nice increasing CDF, and thus touching very different kind of problems.

- Clouds of points (green points around red ones) seem to confirm that the ratio of useful over useless clauses is indeed a specific constant value for each kind of problem. Often, "clouds" of points are generally shuffled instances of the same original one;
- there seems to have a tendency for `Glucose` to produce more useless clauses than useful;
- if we get rid of some outliners, the ratio useless/useful seems also to reflect the charateristics of the solver (not only the problem).

3.2 Useless input clauses

We are interested here in the number of useless initial clauses. It is known that industrial problems are redundant and contain a lot of useless clauses for deriving the contradiction. In [3] it was shown that, "around 2/3 of the (industrial) instances have between 20% and 50% redundant clauses, the remaining ones have over 50% redundant clauses and close to 5% of the instances have in excess of 90% redundant clauses". The question is here to see how naturally the solver is focusing on a subpart of the original formula. It is for instance not really understood whether redundancies may help the solver or not. Let's look at figure 2. The outer figure gives an idea of how many initial clauses are really useful and useless in the proof. A new set of observations can be drawn from it:

- Once again, there is a clear tendency, on most of the examples, to show a relationship between the number of useless and useful clauses in the initial formula;
- this relationship seems highly related to each problem. Each "cloud" (or curved line here) is clearly associated with the same initial instance;
- interestingly enough, the inner subfigure, showing the ratio of useful clauses over all the clauses in the initial formula shows the same conclusions given in [3] and recalled above,

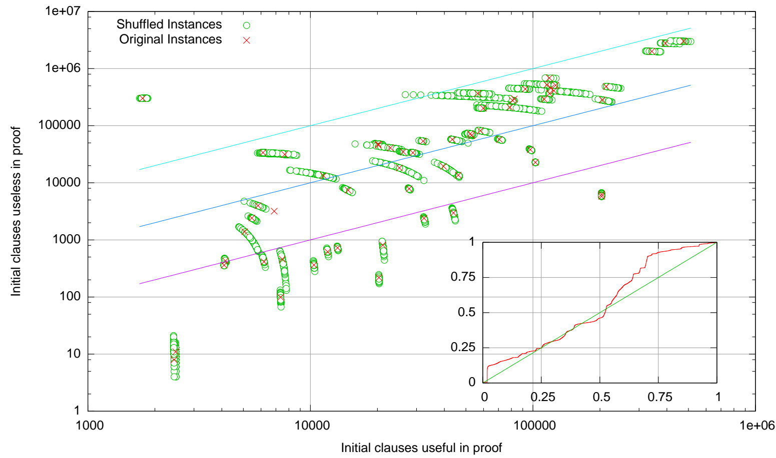

Figure 2: Useless and Useful initial clauses in `Glucose`, over initial clauses only. The inner subfigure is the CDF plot of the ratio of useful clauses in the initial formula.

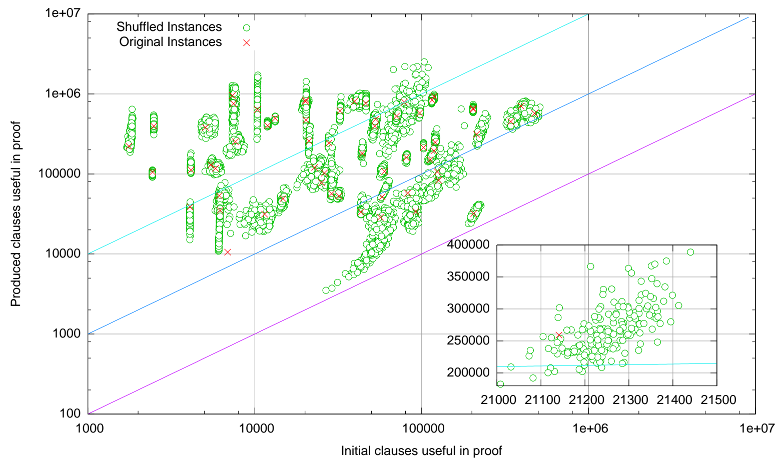

Figure 3: Useful clauses among input clauses and size of the final proof (produced clauses only) in `Glucose`. The inner subfigure is zooming on a small subpart of the outter plot. It shows that "vertical lines" of green points may be in fact increasing.

at least the same order of magnitude ratios. However, here, `Glucose` was not designed to focus on a Minimal Unsatisfiable Set (MUS) of the original formula. It seems to *naturally* focus on a MUS (but of course less strictly, current MUS extractors would converge very quickly if this was strictly true).

3.3 Choice of input clauses and final size of proofs

The above experiments suggested that, depending on the shuffling, the solver may find distinct subsets of initial clauses to derive the final contradiction. A natural question is now to check whether this choice may have a direct impact on the final proof size. We here simply check the possible relationship between the size of the initial set of clauses used by the solver and the size of the proof (produced clauses only). This is reported figure 3 and seems to show that the relationship is unclear. Most of the "clouds" of green points seem to be vertical lines. However, some problems clearly show increasing clouds and we cannot detect any apparent decreasing. Moreover, the log scale may reduce the visual impact of small increasing in X values. Thus, we zoomed on a very small portion of the plot, and shown, in the inner plot, that, on a non log scale, what may appear as a vertical line can be an increasing one. Of course, these results are preliminary only and a more detailed analysis of this relationship should be conducted. But all seems to confirm the conjecture that small set of initial clauses leads to smaller proofs, and thus better performances. However, what is the cause and what is the consequence in this observation is not clear. It could be interesting to investigate this observation with recent techniques for MaxSAT solving [13], where a community-based selection of input clauses is first performed.

3.4 On the impact of shuffling

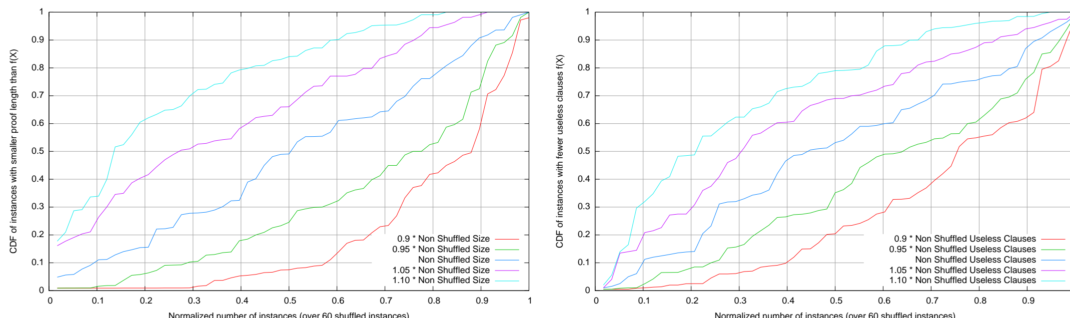

Figure 4: Ranking of the performances of `Glucose` over the original instance in comparison with all shuffled instances.

As already pointed out on the two figures 1 and 2, there seems to have some "clouds" (or curved lines) of points, each of them generally associated with the same benchmark. This suggests that shuffling could noticeably change the size of the final proof. It may be thus interesting to see how much shuffling may impact it. It is already known that we may observe an important discrepancy in the solving time for SAT instances, and results for UNSAT instances are more robust. What about proof sizes?

Figure 4-left reports the following experiment. Given a problem P, we recorded the size of its proof size(P), considering only produced clauses, on the original problem. Then we "rank" it over all the 200 shuffled runs for the same benchmark P. This first experiment is shown on the left, with the middle curve named "Non Shuffled Size". What is striking here is that there is no general rule about winning or loosing in proof size when shuffling. Half of the benchmarks (0.5 on the X-axis) are ranked 0.5 (see Y axis). The curve is dramatically linear. So if you

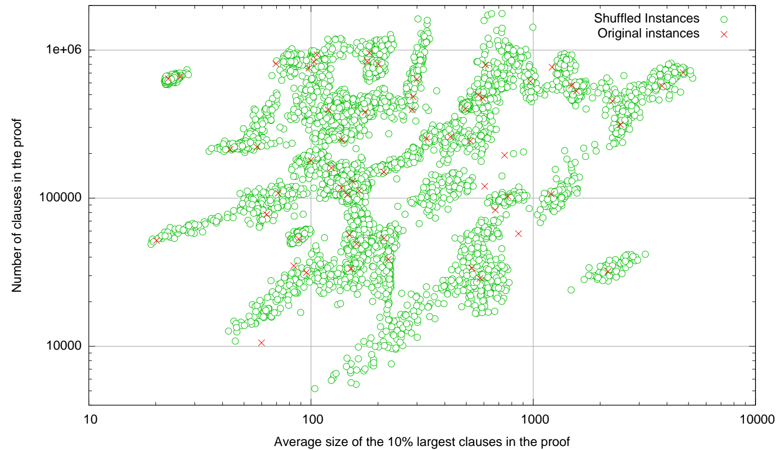

Figure 5: Width of proofs versus their sizes on all the tested problems.

shuffle an instance, it's a fifty-fifty chance to win or loose. Now let us look at how much you can win by shuffling your instance. The other curves on the same figure shows the ranking when considering C.size(P) instead of size(P). It shows that, when winning, the win is not so important: 0.9*size(p) is in the top 10% of half of the runs. This was already suggested by the "clouds" shown in the previous pictures. This result is somehow surprising given previous observations (on mixed SAT and UNSAT) problems that shown that high discrepancies can be observed. In a sense, this observation questions the validity of portfolio approaches for UNSAT problems (of course, they have more complex diversification strategies than just shuffling the original instance).

On figure 4-right, we conducted exactly the same experiment, but with useless clauses. It may be interesting to see if sometimes a solver is producing more or less useless clauses for the final proof. The experiment suggests exactly the same conclusion as above. Shuffling is not a good strategy to win a lot. However, clearly enough, these results are on all the problems. There can be families of problems with particular behaviors.

3.5 Width and Size of proof

It is well known that, from a theoretical point of view, the width of a minimal proof (the length of the larger clause) is related to its size (its number of clauses). Thus, we tried to see the correlation between the average size of the 10% larger clauses in the proof (no input clauses) and the number of clauses in the proof (no input clauses too). The general result is given figure 5. Even if the reader is encouraged to see some kind of increasing set of points on this figure, it is hardly possible to be convinced in the general case. It seems that this could be true for some problems but not all. Let us focus on Figure 6 on the very convincing benchmark `fifo8-400`, submitted in the SAT 2002 competition by Emmanuel Dellacherie. It is a typical industrial model checking instance. On this particular problem, the relationship between the average size of the top 10% of the clauses in the proof and the total length of the proof is really clear. However, as mentioned above, systematically checking this property on all the benchmarks has to be done and one weakness of our study is the lack of very hard problems (involving more

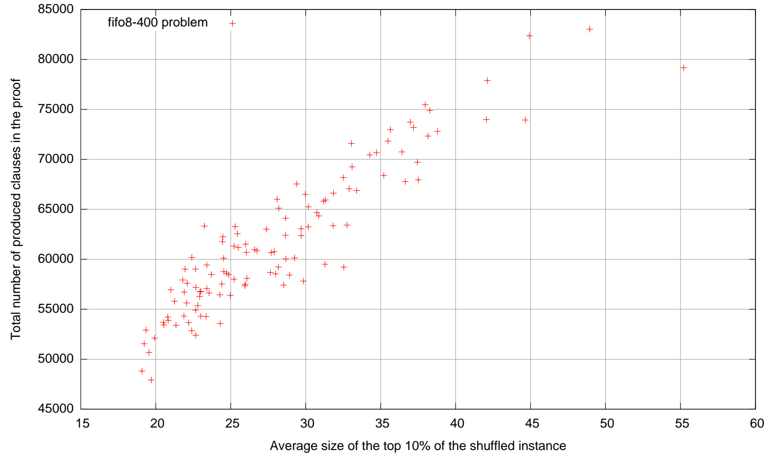

Figure 6: Width of proofs versus their sizes, focusing on the `fiifo8-400` problem.

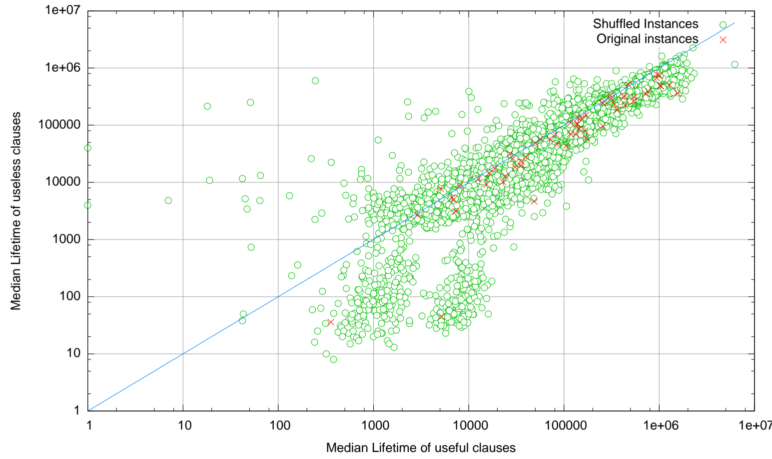

Figure 7: So called "lifetime" of clauses for producing useful clauses versus useless clauses.

than one million conflicts on the original instance).

4 On predicting the quality of learnt clauses

In this section we try to measure, post mortem, if any distinction can be made between good (useful) and bad (useless) clauses.

4.1 No obvious locality in learnt clause usefulness

In this experiment, we wanted to see if some locality was observed in the resolution proof, meaning that the solver may spend some time generating a proof for a subproblem then going

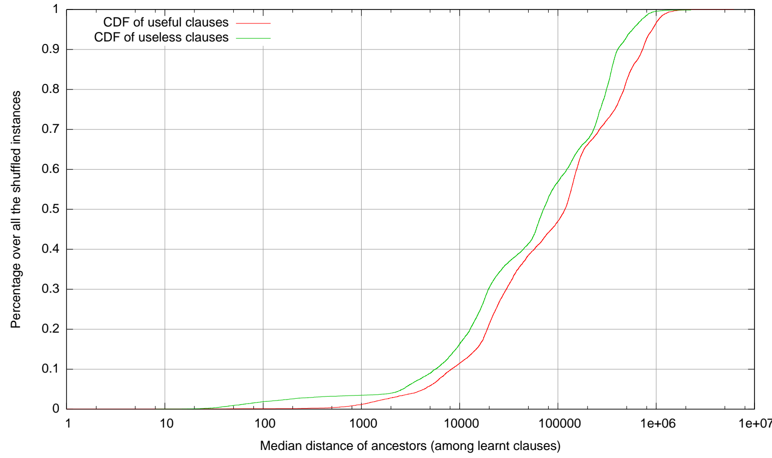

Figure 8: CDF of the lifetime of clauses.

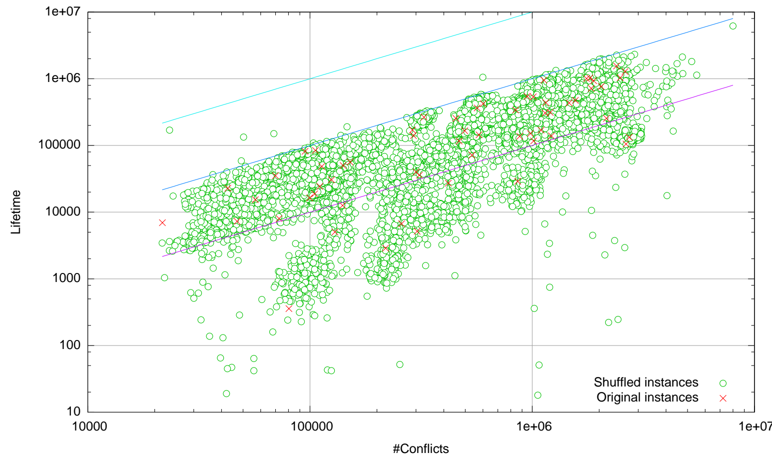

Figure 9: Median lifetime of clauses used for generating good clauses.

to another subproblem. If this is true, most of the ancestors of all the clauses will be "local", intuitively. Thus, we recorded the difference between the age of each clause (the conflict number at which the clause was learnt) and the age of each of its direct ancestors (among learnt clauses only). For each shuffled run, we report the median value of the age differences of all the ancestors, of all learnt clauses (no aggregation is done on each clause: if we have N conflicts and R resolutions by conflict, we give the median value of the N*R values). We call this value the "lifetime" of clauses for this run (an ancestor clause has to be kept at least this number of conflict times (in 50% of the time) to be able to derive the considered clause). We have done the same experiment by computing a value for all the useful clauses (one median value) and all the useless clauses (another median value). Figure 7 reports this experiment as a scatter plot between the two values, hoping to show whether there are more "locality" for good clauses or

bad clauses. The result is that good clauses tend to have a greater median value, even if the difference tends to be very small. This tight but significant difference is confirmed by the CDF plot figure 8 and may be of course simply explained by the fact that `Glucose` tends to remove bad clauses early in the search. The second thing that one may observe on this figure is that there is no much differences between the locality of bad and good clauses. More importantly, Figure 7 suggests that old clauses are always used for resolutions as the solver progresses in its proof construction. This is confirmed by the figure 9, where we can see that the lifetime of clauses for good clauses productions is clearly related to the total number of conflicts. As a conclusion, the strategy to identify good and bad clauses cannot be related to the age of a clause. A lot of older and older clauses are necessary. However, here, we should take some precaution. We are not claiming that keeping old clauses is the best possible way of building the shortest proof. We just observe that current CDCL solvers need old clauses for deriving their proof.

4.2 Accuracy of different prediction strategies

To give a simple intuition of the efficiency of different ways of characterizing good/bad clauses during the run, we consider here a simple experiment. Once good clauses have been formally identified (they occur on the proof), we can analyze whether an automated process would correctly classify any clause as good or bad.

Thus, we consider in this part three classifiers with a parameter L:

- A clause is good iff it has an initial LBD score of L;
- A clause is good iff it has a final LBD score of L;
- A clause is good iff it has a size of L.

This set of classifiers are very static, but should have the advantage of being simple and precisely defined. We collected the *accuracy* value of our classifier on all the shuffled instances and represented the CDF distribution of these values over the normalized set of instances, for each classifier, and for a set of parameters. Intuitively, the accuracy is the probability that a given clause was correctly classified.

Figure 10 shows a set of CDF plots for the initial LBD classifier. We see that by statically considering clauses of initial LBD of 10 or 14 we already have a not so trivial classifier that shows an accuracy of more than 60% for 2/3 of the problems. More interestingly, the final LBD score is even more discriminant for classification. Figure 11 shows an even more interesting set of not so bad classifications. The accuracy is above 50% for more than 85% of the test set as soon as we use a static limit of 10 for the final LBD score. Let us recall here that this does not reflect the quality of prediction of `Glucose`: during run time, `Glucose` can automatically adapt its LBD score for classification. Another interesting remark is the increasing of the accuracy between initial/final LBD for LBD scores of 2 and 4. This means that improving the LBD score of a clause to 2 or 4 increase its probability to be on the final proof.

Now let's look at another classifier, based on the size of clauses: as reported figure 12, we observe weaker performances. More importantly, we see that all strategies considering clauses sizes of below 40 or 22 are really bad. Of course, the whole experiment may be biased by the fact that we use `Glucose`, and this solver is somehow focusing on clauses of small LBD values. They have more chances to stay in the clause database, thus more chances to be useful for the proof. An additional set of experiments is needed here to study the impact of distinct strategies of clause database reductions.

Thus, to complete our experiments, we report the comparison of the two last classifiers on the set of original instances, when `Glucose` does not perform any clause database cleaning. Due

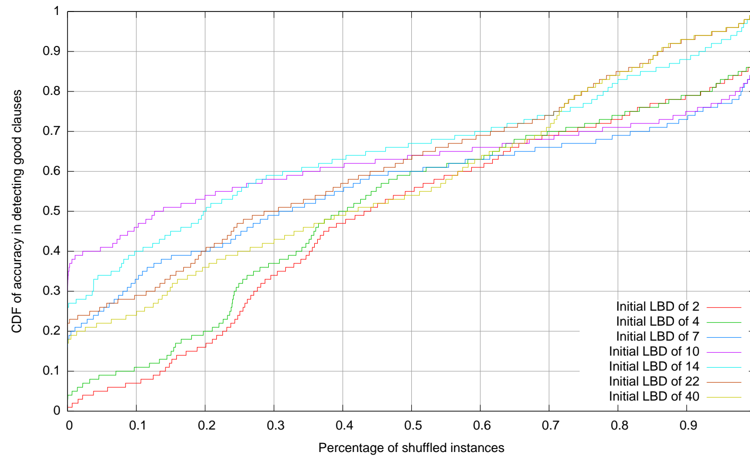

Figure 10: Accuracy when considering the initial LBD score of clauses.

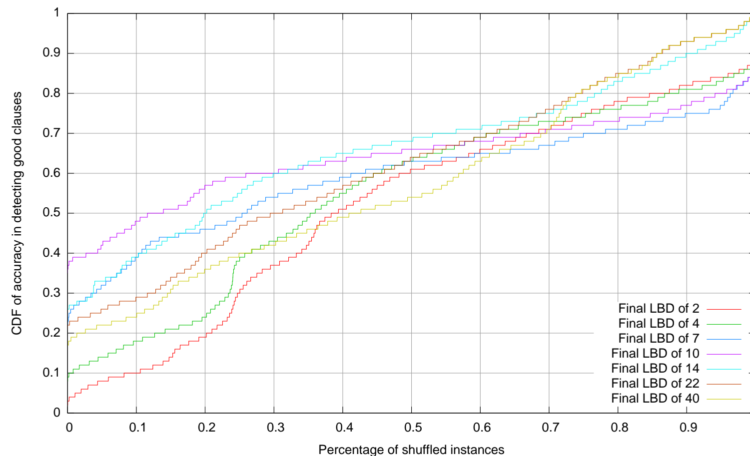

Figure 11: Accuracy when considering the final LBD score of clauses.

to a lack of time, we were not able to conduct the same exhaustive experiment as previous by shuffling the instances. This experiment is reported Figure 13 and clearly demonstrates that, in the very large majority of the cases (over all possible values for L), the final LBD is a better estimator than the size of the clause. The inner subfigure shows the CDF of the Y values and X values of the outter figure. It confirms the superiority of the LBD over the clause size classifiers. This also tends to show that the results obtained above may probably be generalized for other clause database cleaning strategies and seems to confirm that all the previous results are not only due to the cleaning strategies of `Glucose`.

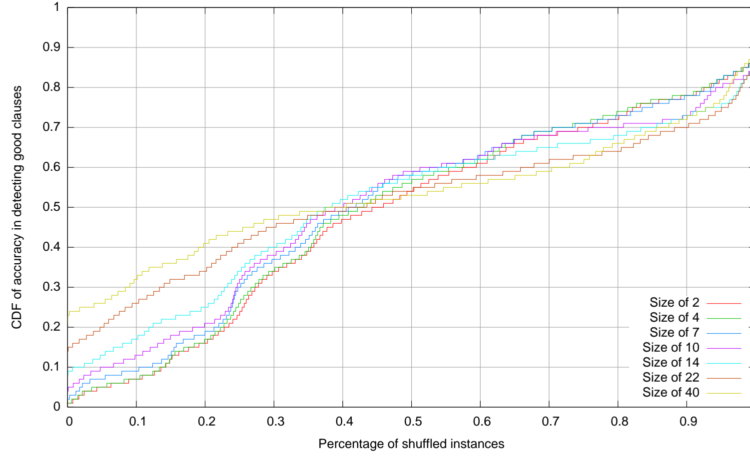

Figure 12: Accuracy when considering the size of clauses.

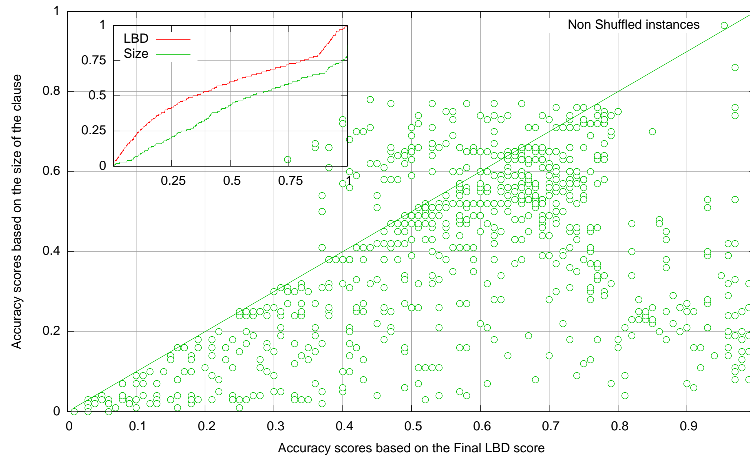

Figure 13: Accuracy of the Final LBD and Size classifiers when no database cleaning is performed, for only the 60 initial (non shuffled) instances, but with all possible L values in $\{1, 2, 3, 4, 5, 6, 7, 8, 10, 14, 18, 22, 30, 40\}$.

4.3 `Glucose` strategies

The previous experiment suggested to look at the proofs, depending on the clause database cleaning strategy. We thus compare here the number of useful/useless clauses in `Glucose` w.r.t. `Minisat`. We took `Glucose` and put the activity-based cleaning strategy of `Minisat` with Luby restarts. The surprising result is given figure 14. If the total number of produced clauses is in favor of `Glucose`, the number of produced useless clauses is larger for `Glucose` than `Minisat`. However, the size of the proofs obtained by `Glucose` remains smaller than `Minisat`. Thus it seems that, by throwing away a larger set of clauses, `Glucose` is able to focus on a smaller proof.

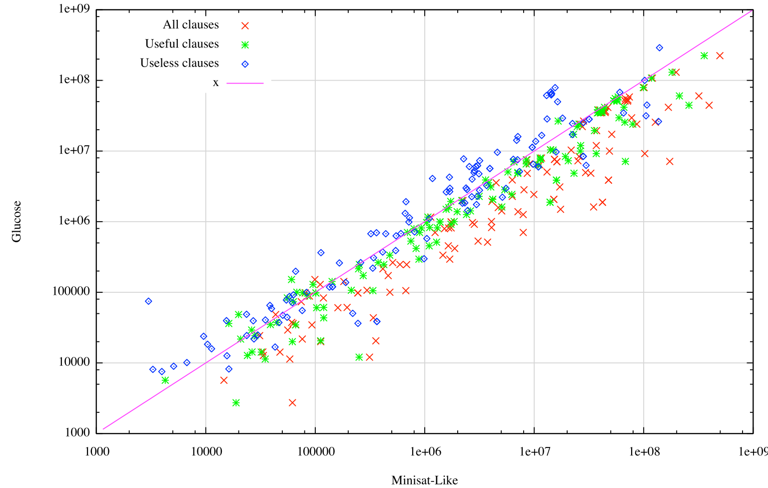

Figure 14: Useless and Useful clauses in `Glucose` *versus* `Minisat`. Here (and only here in the paper), the set of benchmarks is all the UNSAT problems of the last competitions (with no timeout but a memory limit of 80Gb).

It is however surprising to see how many more useless clauses `Glucose` is generating.

5 Conclusion

In this paper we studied some parameters of SAT proofs generated by `Glucose`, a typical CDCL solver based on `Minisat` [6]. We showed some new relationship between some of them, like the number of useful clauses in the initial formula and the size of the proof itself. We also demonstrated that the width a proofs have also an interesting impact on their size, in many cases. We also investigated a few classical scoring mechanisms for learnt clauses usefulness and demonstrated that, in most of the cases, the score used in `Glucose` has the best accuracy.

This paper also shows that 90% of the time spent in CDCL solvers on unsatisfiable instances could be considered as "heuristics" for generating the correct final proof. We think that this shows that there is still a huge possible improvement in SAT technologies, even if we restrict ourselves to current proofs qualities. We first have to understand what this underlying, hidden, heuristics are really doing. A lot of additional works are pending. We are investigating how greedy are CDCL solvers to build the proof (how many successive bad clauses are generated), for instance. It is also important to extend our work on harder proofs, and to be able to partition the set of problems according to proofs characteristics.

Acknowledgments

The author would like to thanks George Katsirelos, Ashish Sabarwhal and Horst Samulowitz for fruitful discussions. The anonymous reviewers of the POS workshop also gave very insightful comments on the first version of this paper.

References

[1] Carlos Ansótegui, Maria Luisa Bonnet, Jordi Levy, and Felip Many. Measuring the hardness of sat instances. In *Proceedings of AAAI*, pages 222–228, 2008.

[2] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *IJCAI*, 2009.

[3] Anton Belov, Mikolás Janota, Inês Lynce, and João Marques-Silva. On computing minimal equivalent subformulas. In *Principles and Practice of Constraint Programming (CP)*, pages 158–174, 2012.

[4] M. Davis, G. Logemann, and D. Loveland. A machine program for theorem proving. *JACM*, 5:394–397, 1962.

[5] Niklas Eén and Armin Biere. Effective preprocessing in SAT through variable and clause elimination. In *proceedings of SAT*, pages 61–75, 2005.

[6] Niklas Eén and Niklas Sörensson. An extensible SAT-solver. In *SAT*, pages 502–518, 2003.

[7] Jinbo Huang. The effect of restarts on the efficiency of clause learning. In *IJCAI'2007*, pages 2318–2323, 2007.

[8] Matti Järvisalo, Arie Matsliah, Jakob Nordström, and Stanislav Z̃ivný. Relating proof complexity measures and practical hardness of sat. In Michela Milano, editor, *Principles and Practice of Constraint Programming (CP)*, Lecture Notes in Computer Science, pages 316–331, 2012.

[9] George Katsirelos, Ashish Sabharwal, Horst Samulowitz, and Laurent Simon. Resolution and parallelizability: Barriers to the efficient parallelization of sat solvers. In *Proceedings of AAAI*, 2013.

[10] Chu Min Li and Anbulagan. Heuristics based on unit propagation for satisfiability problems. In *Proceedings of IJCAI*, pages 366–371, 1997.

[11] Chu Min Li and Sylvain Gérard. On the limit of branching rules for hard random unsatisfiable 3-sat. In *Proceedings of ECAI*, pages 98–102, 2000.

[12] Joao P. Marques-Silva and Karem A. Sakallah. Grasp: a search algorithm for propositional satisfiability. *Computers, IEEE Transactions on*, 48(5):506–521, May 1999.

[13] Ruben Martins, Vasco Manquinho, and Inês Lynce. Community-based partitioning for maxsat solving. In *Proceedings of SAT*, pages 182–191, 2013.

[14] Matthew Moskewicz, Connor Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient SAT solver. In *proceedings of DAC*, pages 530–535, 2001.