EPiC Series in Computing
Volume 73, 2020, Pages 1–11
LPAR23. LPAR-23: 23rd International Conference on Logic for Programming, Artificial Intelligence and Reasoning

Decision levels are stable: towards better SAT heuristics*

Robert Nieuwenhuis, Adrià Lozano, Albert Oliveras, and Enric Rodríguez-Carbonell

Technical University Catalonia (UPC), Barcelona, and Barcelogic.com
roberto@cs.upc.edu

Abstract

We shed new light on the *Literal Block Distance* (LBD) and *glue*-based heuristics used in current SAT solvers. For this, we first introduce the concept of *stickiness*: given a run of a CDCL SAT solver, for each pair of literals we define, by a real value between 0 and 1, how *sticky* they are, basically, how frequently they are set at the same decision level.

By means of a careful and detailed experimental setup and analysis, we confirm the following quite surprising fact: given a SAT instance, when running different CDCL SAT solvers on it, no matter their settings or random seeds, the stickiness relation between literals is always very similar, in a precisely defined sense.

We analyze how quickly stickiness stabilizes in a run (quite quickly), and show that it is stable even under different encodings of cardinality constraints. We then describe how and why these solid new insights lead to heuristics refinements for SAT (and extensions, such as SMT) and improved information sharing in parallel solvers.

1 Introduction

Propositional satisfiability (SAT) is a cornerstone problem. Since its early days it has played a fundamental role in the theory of Computer Science, as it was the first problem to be proved NP-complete [11]. While SAT is still an active research topic in theoretical areas such as proof complexity and computational complexity, most remarkably it has become part of today's industrial practice as well. Nowadays formulas with hundreds of thousands of variables and millions of clauses are routinely solved in a wide range of applications, e.g., electronic design automation [24], hardware and software verification [14, 16], bioinformatics [17], cryptography [25] and data mining [10], to name a few. Further, several extensions of SAT have also been devised to accommodate to different contexts. For example, in Satisfiability Modulo Theories (SMT) [7, 21], formulas are expressed in a fragment of first-order logic, which allows a higher-level language than propositional logic. Other satisfiability and optimization problems such as Pseudo-Boolean optimization [12] and Integer Linear Programming [20], as well as Constraint Programming based on Lazy Clause Generation [22] have been reviewed to incorporate the same underlying SAT-based techniques, including the ones discussed in this paper.

*Projects TIN2015-69175-C4-3-R (funded by FEDER/MINECO) and RTI2018-094403-B-C33 (funded by FEDER/Ministerio de Ciencia e Innovacion, Agencia Estatal de Investigacion, Spain).

This expansion of SAT and its extensions is thanks to the breathtaking improvements achieved in SAT solvers over the last 25 years. One of the most important enhancements is the *conflict-driven clause-learning* (CDCL) scheme [23]. In this scheme a backtracking search is conducted, where the current partial assignment is represented with a stack of literals (those that are true under the assignment). A *run* of the SAT solver consists of a sequence of stacks, starting with an empty stack, and leading to a stack describing a model of the formula if it is satisfiable, or to a special state indicating that the formula is unsatisfiable. The stack is enlarged by taking *decision* steps, in which a literal is decided to be true. Then the logical consequences of that decision are evaluated. To that end, efficient *unit propagation* techniques such as two-watched literals are employed [18]. Propagated literals are pushed onto the stack, and become part of the *decision level* of the last decided literal. Every time a falsified clause (i.e. a conflict) is identified, a *conflict analysis* is launched, which determines the number of the last decision levels that can be undone. As a by-product of conflict analysis, a new redundant clause called *lemma* is generated. This lemma is then *learned*, i.e., added to the clause database, with the aim of avoiding future similar conflicts. Since whenever a conflict occurs a lemma is produced, from time to time the clause database must be *cleaned up* to discard those lemmas that are unlikely to be useful in the future. Examples of cleanup policies are, for instance, to keep lemmas that are short (i.e., which contain few literals), or those which have proved to be useful up to this point (by means of a metric that measures the *activity* of a lemma, typically related to the number of times the lemma has been involved in a conflict analysis). Another successful cleanup policy is based on ranking lemmas by the number of different decision levels the variables in a clause belong to [5]. This value is called the *Literal Block Distance* (LBD) of a clause. Clauses with a low LBD score are preferred over clauses with a higher one. The rationale for this metric is as follows: The lower the LBD of a clause, the fewer the number of decision levels that are necessary for this clause to be unit propagated or falsified again during the search. In particular, interesting clauses to be kept under this policy are those with LBD equal to 2, which are called *glue clauses*.

LBD-based cleanups have become standard in state-of-the-art SAT solving [15]. Although solvers initially compute the LBD of a lemma according to the state of the stack when the clause was generated, they follow different strategies for updating it. Some never recompute the LBD again; others, for example Glucose 1.0 [5], update the LBD when the clause is used in unit propagation, while others, e.g. Glucose 2.3 [8], do so only when the clause appears in a conflict analysis. Furthermore, SAT solvers also have distinct criteria when deciding which clauses should be kept according to their LBD. Given this diversity of techniques, it appears to be relevant to have a better understanding of LBD and its impact on the performance of SAT solvers.

1.1 Introducing Stickiness

Within a given (total or partial) run R of a CDCL SAT solver, for each variable x, we define $n_R(x)$ as the total number of decision levels containing x, that is, the number of times the literal x or the literal $\neg x$ is pushed on the solver's stack[1]. Similarly, for a literal l, we define $n_R(l)$ as the number of decision levels in R containing l. For a pair of variables (or literals) x and y, we define $n_R(x, y)$ to be the total number of decision levels in this run R containing both.

Now the *stickiness of x and y in R*, denoted by $stick_R(x, y)$ is the (conditional) probability that, if we pick a decision level of R containing one of x and y, that also the other one is

[1] We assume $n_R(x)$ is never 0, since we can eliminate from our analysis the (if any, rare) variables that never get assigned.

assigned at that same decision level:

$$stick_R(x,y) \quad = \quad \frac{n_R(x,y)/T}{n_R(x)/T + n_R(y)/T - n_R(x,y)/T} \quad = \quad \frac{n_R(x,y)}{n_R(x) + n_R(y) - n_R(x,y)}$$

(where the T denoted the total number of decision levels, or of decisions, in R).

For example, $stick_R(x,y)$ is 0 if x and y are never assigned together in the same decision level; it is 1 if they are always together when assigned; and if one of them is assigned 80 times, the other one 85 times, and together only 15 times, then it is $15/(80+85-15) = 0.1$. It is a quite demanding notion in the sense that it quickly drops; for example, if both are assigned the same number of times, of which 90% together, then it already drops to $90/(100+100-90) = 0.82$.

Given a run R on a given CNF over variables $\mathcal{X}$, its *stickiness function* $stick_R \colon \mathcal{X} \times \mathcal{X} \to [0 \dots 1]$ maps pairs of variables (or literals) to their stickiness: $(x,y) \mapsto stick_R(x,y)$. Now assume we have two different runs R and R' (of two possibly completely different CDCL solvers). The question arises: how can we quantify, again by a number between 0 and 1, the *similarity* $Sim(R,R')$ between the stickiness functions $stick_R$ and $stick_{R'}$?

Of course the function $stick_R$ can be seen as a (complete, undirected) weighted graph with vertices $\mathcal{X}$ and where $weight(x,y) = stick_R(x,y)$, and several (complex) notions for weighted graph similarity exist that do not make much sense for stickiness. As we will see, we need a simple, tailored one. A first similarity notion that comes to mind is:

$$\frac{\sum_{\{x,y\} \subseteq \mathcal{X}} 1 - |stick_R(x,y) - stick_{R'}(x,y)|}{\sum_{\{x,y\} \subseteq \mathcal{X}} 1}$$

which nicely gives results in $[0 \dots 1]$ and is closer to 1 if there are many pairs (x,y) which are similarly sticky in R and in R'. But it is not hard to see that on uniform random $stick_R$ and $stick_{R'}$ between 0 and 1 it will give 0.66 (since on average $|stick_R(x,y) - stick_{R'}(x,y)| = 0.33$), instead of 0, which is what one would like.

Moreover our $stick_R$ behaves in a particular way; in practice it is 0 or close to 0 for almost all pairs (x,y) and, intuitively, for our notion of similarity between runs R and R' it is clearly more important that $stick_R(x,y) \approx stick_{R'}(x,y)$ if both are close to 1 than if both are close to 0. To overcome this, from now on we simply define:

$$Sim(R,R') = \frac{\sum_{\{x,y\} \subseteq \mathcal{X}} min(\ stick_R(x,y),\ stick_{R'}(x,y)\ )}{\sum_{\{x,y\} \subseteq \mathcal{X}} max(\ stick_R(x,y),\ stick_{R'}(x,y)\ )}$$

which still gives results in $[0 \dots 1]$ and closer to 1 if many pairs (x,y) are similarly sticky in R and in R'.

It is also a quite demanding notion in the sense that it quickly drops; for example, since most literal pairs have low stickiness, if a large majority have stickiness, say, 0.3 in one run and 0.1 in the other run, they decisively contribute pushing $Sim(R,R')$ to 0.33. In fact, we will see similarities close to 0 in practice if R and R' are runs of the same SAT solver, on two CNFs that are identical except for a random permutation of variable names. Therefore it is remarkable that we will also see similarities above 0.7 and up to above 0.9 between runs with different random seeds an even with different solvers.

1.2 Results and Perspectives

In Section 3 we do experiments confirming that under our notion of similarity, indeed the stickiness relations of unrelated runs is very low, even between runs of the same SAT solver, on two CNFs that are identical except for a random permutation of variable names.

Section 4 analyzes how stickiness evolves along a given run, running different instances on different well-known SAT solvers.

In Section 5 we show that, given a SAT instance, when running different CDCL SAT solvers on it, no matter their settings or random seeds, their stickiness relations are always highly similar.

In Section 6 we show that stickiness is highly similar even between runs on two different clause sets, if both come from different cardinality constraint encodings of the same problem instance (from Barcelogic's real-world sports scheduling customers).

How can our new insights help improving solvers? This is discussed in Section 7: heuristics refinements for SAT (and extensions, such as SMT) and improved information sharing in parallel solvers.

2 Benchmarks, solvers, and tools used along this paper

Instances: We report on experiments on real-world SAT instances, selecting 10 of the around 170 old problems of the 2019 SAT Race by taking all multiples of 17 from the ordered list (below we use abbreviated names): 9-50-sc2017, ctl-4291-567-2-unsat-sc2013, frb59-26-1.used-as.sat04-891-sc2011, le450-15b.col.15-sc2018, par32-1-c.shuffled-as.sat03-1531-sc2002, rpoc-xits-12-UNKNOWN-sc2009, tseitingrid7x160-shuffled-sc2016, 001-80-12-sc2014, smulo032-sc2012, AProVE07-26-sc2007. A few instances with too many variables ($> 100,000$) were discarded and then the next one in the order was taken. The same was done for one too easy instance with a very short runtime. Before any experiments, we first pre-processed all instances using Glucose.

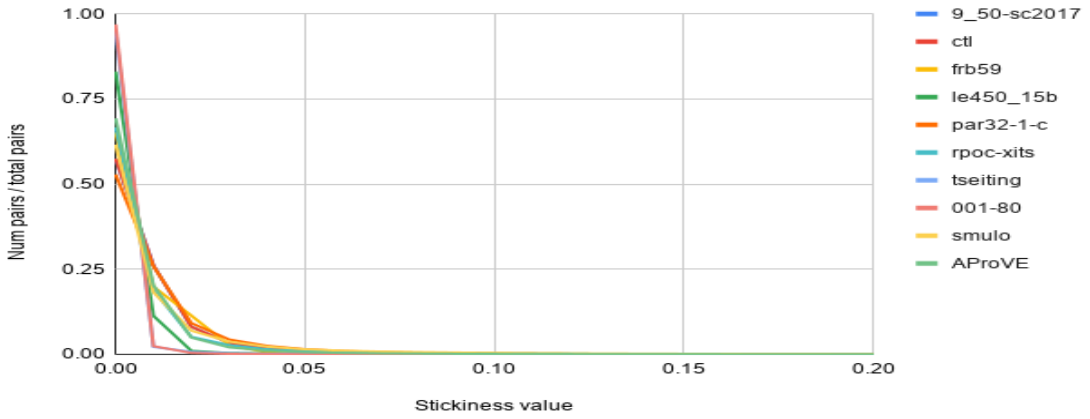
Solvers: We use the solver Cadical [9] in its three versions entered to the SAT Race 2019: version 1 (default), 2 (tuned for satisfiable problems), and 3 (tuned for unsatisfiable problems). We also used the latest version of the Glucose solver [6]. For all experiments we used 1-hour solver timeouts. Since Cadical does heavy inprocessing, eliminating variables, when comparing stickiness of two runs we only consider their intersection of variables (and of course renaming them back to their original numbers).

Trace files: We instrumented these solvers in order to, at each backjump, output to a trace file the decision levels that are popped from the stack. So these trace files contain one line per decision level: all its literals, including its decision literal, and also records the number of conflicts at each point. These traces become large: around 70 GB for a typical 1-hour Cadical run.

Counter files: Using a different program, each trace file is processed in order to extract the so-called "counter files" with the information, after each 100,000 conflicts, of $n_R(x)$ and $n_R(x, y)$ for variables and literals. Processing trace files efficiently is non-trivial: counting $n_R(x)$ is fast, but $n_R(x, y)$ obviously requires quadratic work in the size of each decision level with lots of random accesses to this quadratic number of counters (in fact $4n^2$ for literals if there are n variables; that is why we avoided instances with very large n). Along this paper, we consider stickiness between literals, which seems a more precise measure than stickiness between variables, although both behave very similarly in general.

Stickiness and similarity statistics: Finally, we made another little tool that, given one or two such counter files with quadratically many counters for each milestone (number of conflicts), efficiently extracts the stickiness information at each milestone, stickiness distributions, the similarity between two runs at a given milestone, etc.

Most pairs have very low stickiness: To get a first flavor of stickiness, here we give a graphical indication of how it is distributed (on average over the four solvers), for each one of the ten instances. Note that the x-axis has been cut at 0.2, because in the interval between 0.2 and 1.0 the lines become invisible.



3 Unrelated runs indeed have low similarity

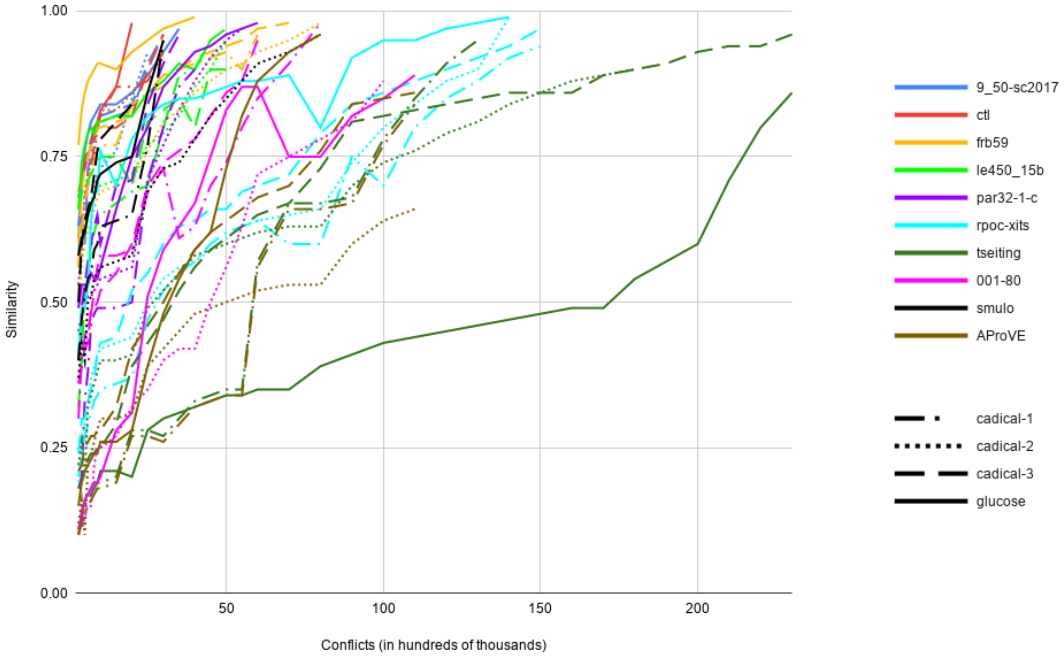
A first simple experiment we have done is to run each solver, with exactly the same settings and random seed, twice on each instance: one run R on the instance as it is, and one run R' on the same instance with randomly permuted variables. Indeed for these 40 pairs of runs, $sticky_R$ and $sticky_{R'}$ have a very low similarity, never higher than 0.1:

	cad1	cad2	cad3	glu
9-50-sc2017	0.04	0.09	0.02	0.01
ctl	0.03	0.07	0.10	0.04
frb59	0.01	0.07	0.07	0.08
le450-15b	0.05	0.02	0.05	0.07
par32-1-c	0.09	0.01	0.07	0.04
rpoc-xits	0.04	0.08	0.03	0.10
tseiting	0.07	0.07	0.08	0.09
001-80	0.03	0.04	0.06	0.10
smulo	0.06	0.03	0.09	0.01
AProVE	0.05	0.03	0.10	0.01

4 How does stickiness evolve along a run of a solver?

The plot below shows how stickiness evolves along a run of a solver. For each of the 40 runs (10 instances identified by colors, and 4 solvers identified by different line types), we list the similarities between final stickiness of the run and stickiness after different numbers of conflicts.

As we can see, most runs quickly achieve similarities above 0.5, but not all. This gives some insight into the question of how stable the "traditional" LBD values of clauses are in a given run (see the discussion in Section 7 below):



5 Different runs and solvers, still highly similar stickiness

We first compare, on each instance, the stickiness between runs of the four solvers (again, 40 runs: 10 instances, 4 solvers). This gives $\binom{4}{2} = 6$ comparisons for each instance, totaling 60 comparisons (each solver with itself obviously has similarity 1).

	cad1-cad2	cad1-cad3	cad1-glu	cad2-cad3	cad2-glu	cad3-glu
9-50-sc2017	0.76	0.79	0.65	0.68	0.67	0.58
ctl	0.75	0.71	0.55	0.68	0.54	0.56
frb59	0.82	0.86	0.50	0.79	0.51	0.50
le450-15b	0.75	0.76	0.53	0.64	0.52	0.55
par32-1-c	0.77	0.60	0.55	0.64	0.66	0.53
rpoc-xits	0.68	0.70	0.53	0.63	0.54	0.53
tseiting	0.63	0.50	0.48	0.52	0.51	0.56
001-80	0.63	0.50	0.51	0.57	0.55	0.53
smulo	0.65	0.50	0.48	0.53	0.51	0.56
AProVE	0.68	0.50	0.52	0.54	0.51	0.61

We now run each instance with the same solver but with a different random seed, and compare both runs with this solver. Here we can see that, as expected, similarities are even higher than between different solvers:

	cad1	cad2	cad3	glu
9-50-sc2017	0.90	0.93	0.93	0.90
ctl	0.78	0.71	0.70	0.70
frb59	0.87	0.82	0.80	0.87
le450-15b	0.87	0.88	0.94	0.89
par32-1-c	0.92	0.86	0.85	0.90
rpoc-xits	0.77	0.75	0.76	0.79
tseiting	0.60	0.61	0.64	0.69
001-80	0.62	0.59	0.59	0.68
smulo	0.60	0.60	0.68	0.68
AProVE	0.68	0.66	0.73	0.69

6 Stickiness is stable even under different cardinality constraint encodings

Practical problems frequently involve arithmetic constraints. In particular, cardinality constraints, of the form $l_1 + \ldots + l_n \geq k$ (or $\leq$ or $=$). Solver performance depends significantly on how these constraints are encoded into CNF, as well as the syntactic properties of the CNF (see also Section 7). Interestingly, stickiness remains highly stable under different encodings of cardinality constraints.

This is not as surprising as it seems at first sight, because, if the considered encodings have the property that arc-consistency is preserved under unit propagation (as it is usually the case), then the decision levels will indeed tend to be similar, unless the solver decides frequently on the auxiliary variables (and different encodings have different types of auxiliary variables).

We took a (simplified version of) a real-world problem of one of Barcelogic's professional sport scheduling customers, a major European football league. The problem involves scheduling a double round-robin league, a combinatorial object with many exactly-one-constraints (each team one match per round, each match on exactly one round, etc.), as well as many other cardinality constraints (on this round, at most four Sunday matches, at least 5 Saturday matches, at most 4 top-50 matches per round, etc.).

We ran the four solvers on the problem encoded: A) using direct encodings without auxiliary variables (30,685 variables and more than 4 million clauses), B) using our standard sophisticated cardinality encodings (47,681 variables and some 600 thousand clauses) (a combination of techniques [4, 1]). Note: since version A) is hard to solve we had to work with a simplified version of the original problem. In version B), the first 30,365 variables are the ones of version A). For each solver, the similarity on these common literals of the stickiness relations of version A) and version B) is as follows:

cad1	cad2	cad3	glu
0.65	0.74	0.68	0.73

7 How can our new insights help improving solvers?

As said, LBD-based cleanups have become standard in state-of-the-art SAT solving. It seems that our notion of stickiness, along with our analysis of its stability along runs, and its similarity between different runs with different solvers, provides interesting new insights in this context. In fact, at a 2019 Dagstuhl seminar discussion, one of the inventors of LBD, Laurent Simon, expressed his opinion that LBD values were probably meaningful only during the same run.

Stable LBD: Some state-of-the-art solvers forever keep the LBD score of a clause c according to the stack when c was generated, a snapshot of that moment; others update it from time to time according to a later stack, which may again be an imprecise snapshot. Our results seem to indicate that it could make sense to collect stickiness information during the run and consider clause deletion criteria based on a *stable LBD* notion: the LBD of c would be the smallest k such that the literals of c can be split into k disjoint subsets, where pairs of literals inside each subset are highly sticky, say, ≥ 0.95. We are currently implementing this idea in order to experimentally assess it.

Stickiness in portfolio solvers: Most current parallel SAT solvers are based on a portfolio approach: running several different CDCL solvers in parallel, with different randoms seeds and/or settings, in the hope that one of them gets lucky. One key question is: which information to share between nodes? Experts seem to agree on sharing units and (sometimes) binary clauses, but not on sharing glue or other low-LBD clauses. Our results seem to explain why the latter might be useful, especially when a notion such as the aforementioned stable LBD is used. We are also working on an experimental assessment of this latter idea.

8 Relationship with syntactic properties of CNFs

In contrast with our semantic and dynamic analysis of the relationship between variables, several studies based on syntactic and static analysis of CNFs exist. They are mostly based on the *Variable Incidence Graph (VIG)* of a CNF F. In this weighted graph, every variable in F corresponds to a node and, for every clause with k variables, we create $\binom{k}{2}$ edges connecting all pairs of variables in the clause. The weight of an edge takes into account how many clauses contain both variables and which is their length. More formally :

$$w(x,y) = \sum_{\substack{C \in F \\ x,y \in C}} \frac{1}{\binom{|C|}{2}}$$

It has been shown [2] that VIGs constructed from industrial SAT instances are highly *modular* in the following sense: one can partition the vertices into *communities* in such a way that there is a very large amount of edges connecting vertices within the same community, but a low number of edges between vertices of different communities.

These results are purely syntactic, i.e., they do not take into account the behavior of SAT solvers when solving the formula. Indeed, using our "picky" notion of similarity, we found zero similarity between the VIGs and the graphs of our stickiness relation, that express semantic information of a concrete run. Even by using an oracle that adjusts all non-zero weights of the VIG to the weight in our graphs, this lack of similarity could not be fixed. One could still argue, though, that very different graphs might still have similar community structures.

However, despite the purely syntactic nature of VIGs, researchers have tried to prove that modularity is indeed related to the runtime behavior of SAT solvers, via the notion of LBD. One example of this line of research is the paper [19]. It uses a simplified version of the aforementioned VIG, where now weights are simply 0 or 1 depending on whether there is a clause containing both variables. This seems to ignore important information and can lead to unexpected situations. For example, adding to the CNF a single clause which is the disjunction of all literals would render this variant of VIG a complete (and hence also completely meaningless) graph. More realistically, a different encoding of, say, the (omnipresent) at-most-one constraint $l_1 + \ldots + l_n \leq 1$ also leads to completely different VIGs: under the quadratic pairwise encoding

the literals $l_1 \dots l_n$ form a clique in the VIG, whereas for the ladder encoding [13] *none* of them is connected and, in fact, l_1 and l_n are at distance n in the VIG.

A conclusion of [19] states that in conflict clauses c there is a high similarity between a) the number of VIG communities they involve $\#com(c)$ and b) their LBD, $LBD(c)$. A detailed analysis of [19] reveals, however, that what is shown is something else. Namely, that for many SAT instances the set S of data points $|LBD(c) - \#com(c)|$ from the first 100,000 lemmas c generated in a given run has a low standard deviation: usually less than 0.1, implying that a very large majority of the data points are at distance less than 0.1 from the average (this seems even more surprising since the data points are integer and the average needs not be integer). Moreover this seems beside our point: if all data points were exactly, say, 1000, this standard deviation would even be 0, even though the $LBD(c)$ and $\#com(c)$ would be very different (at distance 1000) for each lemma c. To know more about how similar $LBD(c)$ and $\#com(c)$ are, we would be more interested in the average, median or the quartiles of S.

Finally, the argued relationship between LBDs and communities, and the belief that lemmas with low LBDs are key in the performance of SAT solvers, have pushed researchers to look for preprocessing methods that infer clauses that involve few VIG communities [3]. The hope is that these clauses would correspond to lemmas with low LBD learned during a run of a SAT solver and hence will speed up SAT solvers. In [3], modest gains are obtained for satisfiable problems.

9 Conclusion

All in all, since the stickiness relation (and, possibly, graphs derived from it) seems to capture interesting properties about solving SAT instances (more than purely syntactic graphs), we believe that these new concepts can lead to further insights and tools to be exploited in the future development of SAT solving technology, and carry over to extensions such as SMT [7, 21], Integer Linear Programming [20], Lazy Clause Generation [22], or Pseudo-Boolean solving [12], among others.

References

[1] Ignasi Abío, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. A parametric approach for smaller and better encodings of cardinality constraints. In Christian Schulte, editor, *Principles and Practice of Constraint Programming - 19th International Conference, CP 2013, Uppsala, Sweden, September 16-20, 2013. Proceedings*, volume 8124 of *Lecture Notes in Computer Science*, pages 80–96. Springer, 2013.

[2] Carlos Ansótegui, Maria Luisa Bonet, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Community structure in industrial SAT instances. *J. Artif. Intell. Res.*, 66:443–472, 2019.

[3] Carlos Ansótegui, Jesús Giráldez-Cru, Jordi Levy, and Laurent Simon. Using community structure to detect relevant learnt clauses. In Marijn Heule and Sean A. Weaver, editors, *Theory and Applications of Satisfiability Testing - SAT 2015 - 18th International Conference, Austin, TX, USA, September 24-27, 2015, Proceedings*, volume 9340 of *Lecture Notes in Computer Science*, pages 238–254. Springer, 2015.

[4] Roberto Asín, Robert Nieuwenhuis, Albert Oliveras, and Enric Rodríguez-Carbonell. Cardinality networks: a theoretical and empirical study. *Constraints*, 16(2):195–221, 2011.

[5] Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern SAT solvers. In Craig Boutilier, editor, *IJCAI 2009, Proceedings of the 21st International Joint Conference on Artificial Intelligence, Pasadena, California, USA, July 11-17, 2009*, pages 399–404, 2009.

[6] Gilles Audemard and Laurent Simon. On the glucose SAT solver. *Int. J. Artif. Intell. Tools*, 27(1):1840001:1–1840001:25, 2018.

[7] Clark W. Barrett, Roberto Sebastiani, Sanjit A. Seshia, and Cesare Tinelli. Satisfiability modulo theories. In Armin Biere, Marijn Heule, Hans van Maaren, and Toby Walsh, editors, *Handbook of Satisfiability*, volume 185 of *Frontiers in Artificial Intelligence and Applications*, pages 825–885. IOS Press, 2009.

[8] Anton Belov, Daniel Diepold, Marijn J.H. Heule, and Matti Järvisalo, editors. *Proceedings of SAT Competition 2014: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. University of Helsinki, Finland, 2014.

[9] Armin Biere. CaDiCaL at the SAT Race 2019. In Marijn Heule, Matti Järvisalo, and Martin Suda, editors, *Proc. of SAT Race 2019 – Solver and Benchmark Descriptions*, volume B-2019-1 of *Department of Computer Science Series of Publications B*, pages 8–9. University of Helsinki, 2019.

[10] Abdelhamid Boudane, Saïd Jabbour, Lakhdar Sais, and Yakoub Salhi. Sat-based data mining. *Int. J. Artif. Intell. Tools*, 27(1):1840002:1–1840002:24, 2018.

[11] Stephen A. Cook. The complexity of theorem-proving procedures. In Michael A. Harrison, Ranan B. Banerji, and Jeffrey D. Ullman, editors, *Proceedings of the 3rd Annual ACM Symposium on Theory of Computing, May 3-5, 1971, Shaker Heights, Ohio, USA*, pages 151–158. ACM, 1971.

[12] Niklas Eén and Niklas Sörensson. Translating pseudo-boolean constraints into SAT. *J. Satisf. Boolean Model. Comput.*, 2(1-4):1–26, 2006.

[13] Ian P. Gent and Peter Nightingale. A new encoding of alldifferent into sat, 2004.

[14] Aarti Gupta, Malay K. Ganai, and Chao Wang. Sat-based verification methods and applications in hardware verification. In Marco Bernardo and Alessandro Cimatti, editors, *Formal Methods for Hardware Verification, 6th International School on Formal Methods for the Design of Computer, Communication, and Software Systems, SFM 2006, Bertinoro, Italy, May 22-27, 2006, Advanced Lectures*, volume 3965 of *Lecture Notes in Computer Science*, pages 108–143. Springer, 2006.

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

[16] Franjo Ivancic, Zijiang Yang, Malay K. Ganai, Aarti Gupta, and Pranav Ashar. Efficient sat-based bounded model checking for software verification. *Theor. Comput. Sci.*, 404(3):256–274, 2008.

[17] Inês Lynce and João Marques-Silva. SAT in bioinformatics: Making the case with haplotype inference. In Armin Biere and Carla P. Gomes, editors, *Theory and Applications of Satisfiability Testing - SAT 2006, 9th International Conference, Seattle, WA, USA, August 12-15, 2006, Proceedings*, volume 4121 of *Lecture Notes in Computer Science*, pages 136–141. Springer, 2006.

[18] Matthew W. Moskewicz, Conor F. Madigan, Ying Zhao, Lintao Zhang, and Sharad Malik. Chaff: Engineering an efficient sat solver. In *DAC*, pages 530–535. ACM, 2001.

[19] Zack Newsham, Vijay Ganesh, Sebastian Fischmeister, Gilles Audemard, and Laurent Simon. Impact of community structure on SAT solver performance. In Carsten Sinz and Uwe Egly, editors, *Theory and Applications of Satisfiability Testing - SAT 2014 - 17th International Conference, Held as Part of the Vienna Summer of Logic, VSL 2014, Vienna, Austria, July 14-17, 2014. Proceedings*, volume 8561 of *Lecture Notes in Computer Science*, pages 252–268. Springer, 2014.

[20] Robert Nieuwenhuis. The intsat method for integer linear programming. In Barry O'Sullivan, editor, *Principles and Practice of Constraint Programming - 20th International Conference, CP 2014, Lyon, France, September 8-12, 2014. Proceedings*, volume 8656 of *Lecture Notes in Computer Science*, pages 574–589. Springer, 2014.

[21] Robert Nieuwenhuis, Albert Oliveras, and Cesare Tinelli. Solving SAT and SAT Modulo Theories: from an Abstract Davis-Putnam-Logemann-Loveland Procedure to DPLL(T). *Journal of the ACM*, 53(6):937–977, November 2006.

[22] Olga Ohrimenko, Peter J. Stuckey, and Michael Codish. Propagation = lazy clause generation. In Christian Bessiere, editor, *Principles and Practice of Constraint Programming - CP 2007, 13th International Conference, CP 2007, Providence, RI, USA, September 23-27, 2007, Proceedings*, volume 4741 of *Lecture Notes in Computer Science*, pages 544–558. Springer, 2007.

[23] João P. Marques Silva and Karem A. Sakallah. Grasp: A search algorithm for propositional satisfiability. *IEEE Trans. Computers*, 48(5):506–521, 1999.

[24] João P. Marques Silva and Karem A. Sakallah. Boolean satisfiability in electronic design automation. In Giovanni De Micheli, editor, *Proceedings of the 37th Conference on Design Automation, Los Angeles, CA, USA, June 5-9, 2000*, pages 675–680. ACM, 2000.

[25] Mate Soos, Karsten Nohl, and Claude Castelluccia. Extending SAT solvers to cryptographic problems. In Oliver Kullmann, editor, *Theory and Applications of Satisfiability Testing - SAT 2009, 12th International Conference, SAT 2009, Swansea, UK, June 30 - July 3, 2009. Proceedings*, volume 5584 of *Lecture Notes in Computer Science*, pages 244–257. Springer, 2009.