Using Community Structure to Detect Relevant Learnt Clauses *

Carlos Ansótegui[1], Jesús Giráldez-Cru[2], Jordi Levy[2], and Laurent Simon[3]

[1] DIEI, Universitat de Lleida carlos@diei.udl.cat
[2] Artificial Intelligence Research Institute, Spanish National Research Council (IIIA-CSIC) {jgiraldez,levy}@iiia.csic.es
[3] LaBRI, Université de Bordeaux lsimon@labri.fr

Abstract. Nowadays, Conflict-Driven Clause Learning (CDCL) techniques are one of the key components of modern SAT solvers specialized in industrial instances. Last years, one of the focuses has been put on strategies to select which learnt clauses are removed during the search. Originally, one need for removing clauses was motivated by the finiteness of memory. Recently, it has been shown that more aggressive clause deletion policies may improve solvers performance, even when memory is sufficient. Also, the utility of learnt clauses has been related to the modular structure of industrial SAT instances.
In this paper, we show that augmenting SAT instances with learnt clauses does not always make them easier for the SAT solver. In fact, it makes worse the solver performance in many cases. However, we identify a set of highly useful learnt clauses, and we show that augmenting SAT instances with this set of clauses contributes to improve the solver performance in many cases, especially in satisfiable formulas. These clauses are related to the community structure of the formula, and they can be computed in a fast preprocessing step. This would suggest that the community structure may play an important role in clause deletion policies.

1 Introduction

Modern CDCL SAT solvers have been shown to be very efficient at solving industrial, or real-world, SAT instances. They integrate four major components: conflict-driven clause learning [17], activity-based variable branching heuristics [13], lazy data structures [13], and restarts [8]. In [10], it is empirically shown that these four components contribute to such success, but clause learning is the most important. Most CDCL solvers learn just one clause each time a conflict is found for the partially computed assignment. It has been observed that not all learnt clauses have the same usefulness or relevance. Moreover, a clause may be relevant at a certain instant of the search, but it may become useless later. Clause removal policies were initially proposed with the objective of saving memory and speed up propagations by the solver [13, 7]. But the picture

* This work is partially supported by the CSIC project 201450E045, and the Ministerio de Economía y Competividad research project TASSAT2: TIN2013-48031-C4-4-P.

is more complex now. Since Glucose [2], aggressive clause removal policies are essential ingredients of CDCL solvers (more than 95% of the learnt clauses can be removed) and the initial arguments for clause database managements (unit propagation speed and memory issues) do not completely hold anymore. The intriguing question on how to predict efficiently and effectively the relevance of new learnt clauses is still open.

The structure of a SAT instance may be modeled as a graph with variables as nodes and clauses as edges. In [1], authors use this model to show that industrial SAT instances usually exhibit a clear community structure, i.e., high modularity. This means that we can find a partition of this graph into communities, with many edges between nodes of the same community and few edges connecting distinct communities. In [15], it is shown that the measure proposed in Glucose (i.e., the Literal Block Distance or *LBD*) can be strongly related to the community structure of the initial formula. However, this last result was just a one-way observation of the CDCL SAT solvers behavior: while LBD seems related to the number of communities in a learnt clause, it was not possible until now to exploit this correlation the other way, i.e., by using the community structure to guide the search in a CDCL SAT solver.

In this work, we show that community structure can be used to detect relevant learnt clauses. In particular, we present a technique that uses this structure to transform the formula adding learnt clauses, and hence guiding the search. This causality is much stronger than the previous observed correlation. Although we present our technique as a preprocessor for readability, our contribution is to give empirical evidence that the community structure can be used to generate relevant clauses, which is much stronger than identifying them (e.g., LBD is used to rank existing clauses). This would suggest that the community structure may play an important role in clause deletion policies.

Our preprocessor uses the community structure to split the instance into disjoint subformulas, and augments it with the learnt clauses of solving pairs of such subformulas. Intuitively, these clauses could be related to the notion of *glue clauses* used in Glucose. Our inspiration comes from the observation that clause learning destroys the (original) community structure of the instance. We give empirical evidence about the commonly accepted claim that having more learnt clauses does not always speed up the solving process. However, we show that augmenting the instance with our technique works experimentally. This is the case in several sets of industrial benchmarks and several CDCL SAT solvers. Notice that augmenting a formula with learnt clauses is against the common idea of preprocessing, which generally tries to reduce the instance.

The rest of this paper is structured as follows. After some preliminaries in Section 2, we review in Section 3 some observations about the effect of clause learning on the community structure of SAT instances. In Section 4, we provide some insights on the relevance of clauses learnt by a CDCL solver. In Section 5, we propose an algorithm that exploits the community structure to detect relevant clauses, and evaluate its performance in Section 6. We review some related works in Section 7, and we conclude in Section 8.

2 Preliminaries

The Boolean Satisfiability Problem (SAT) is the problem of determining if the variables of a propositional formula can be assigned in such a way that the formula is evaluated as `true`. A *literal* is either a Boolean variable x or its negation $\neg x$, a *clause* is a disjunction of literals, and a conjunctive normal form (*CNF*) instance is a conjunction of clauses.

An undirected weighted graph G is a pair $G = (V, w)$, where V is the set of nodes, and $w : V \times V \to \mathbb{R}^+$ is the edge-weight function that satisfies $w(x, y) = w(y, x)$.

The Variable Incidence Graph (VIG) of a SAT instance Γ is the graph whose nodes represent the variables of Γ, and there exists an edge between two variables if they both appear in a clause c. A clause with l literals results into $\binom{l}{2}$ edges. Thus, to give the same relevance to all clauses, edges have a weight $w(x, y) = \sum_{\substack{c \in \Gamma \\ x,y \in c}} 1/\binom{|c|}{2}$, where $|c| = l$ is the length of the clause c.

The *community structure* of a graph is usually measured using the notion of *modularity* [14]. Defined for a graph G and a partition P of its vertexes into communities, the modularity Q (see Eq. 1) measures the fraction of internal edges (edges connecting vertexes of the same community) w.r.t. a random graph with same number of vertexes and same degree. This avoids that the best partition is the one made up by an only community containing all vertexes.

$$Q(G, P) = \sum_{P_i \in P} \frac{\sum\limits_{x,y \in P_i} w(x, y)}{\sum\limits_{x,y \in V} w(x, y)} - \left(\frac{\sum\limits_{x \in P_i} deg(x)}{\sum\limits_{x \in V} deg(x)} \right)^2 \tag{1}$$

The modularity of a graph is the maximal modularity for any possible partition: $Q(G) = \max\{Q(G, P) \mid P\}$. This optimal modularity will be in the range $[0, 1]$. Computing the modularity of a graph is NP-hard [5]. Due to its complexity, instead of computing the (exact) modularity, most of methods in the literature approximate a lower-bound in the value of Q, trying to find a partition that maximizes this value. One of the most accurate and fastest algorithms is the Louvain method [4], extensively used to compute the modularity of large real-world networks.

In this work, we use the Louvain method to compute a partition of the formula into disjoint subformulas (i.e., sets of clauses). The cost of this algorithm depends on the number of nodes of the graph. We run this algorithm on the VIG, which is one of the graph representation of the formula with smallest number of nodes[4], and we assign each clause to the most frequent community among its variables (randomly assigned in case of ties). We have observed that this formula partitioning (using the VIG) is similar to the one obtained using other graph models, but its computation is much faster.

[4] In other models, clauses are represented as nodes in the graph.

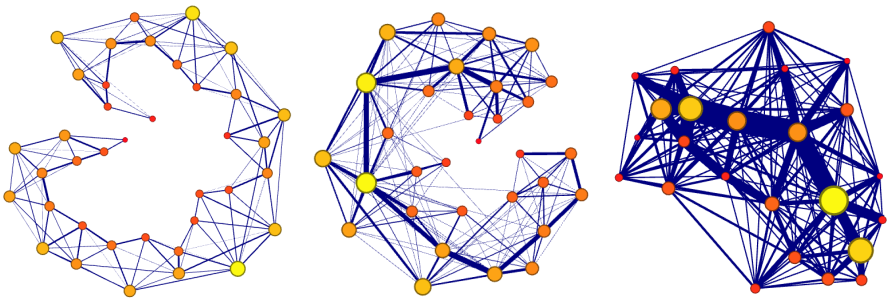

Fig. 1. Graph of communities of the instance `ibm-2002-22r-k60`: original formula (left), solved formula considering *small* learnt clauses (center), and solved formula considering *small* and *medium-sized* learnt clauses (right). Nodes and edges are accordingly scaled by community size and weight, respectively.

3 Clause Learning Destroys the Community Structure

In this section, we review some observations about the community structure of real-world SAT formulas, clause learning, and the relation between them.

Industrial SAT instances have been shown to have a very clear community structure, with modularity Q in the VIG higher to 0.7 in most of the cases. Recall that the maximum value of Q is 1. This means that we can find a partition of their variables into communities, such that clauses mainly constraint variables of the same community. However, this partition is destroyed by the addition of learnt clauses [1], as we will see in this section.

In order to represent how this (initial) community structure is destroyed by the effects of clause learning, we can use the graph of communities[5]. This graph is built as follows: all nodes of the VIG (variables) that belong to the same community are merged into a single node in the graph of communities, and weighted edges are updated accordingly.[6] In Fig. 1 (*left*), we represent the graph of communities of the industrial formula `ibm-2002-22r-k60`. This instance has a modularity $Q = 0.91$ and 35 communities. Glucose solved this formula keeping a total of 504964 learnt clauses. We can recompute the graph of communities after adding some of these learnt clauses to the original instance. In Fig. 1 (*center* and *right*), we represent the graph of communities after adding *small* learnt clauses (up to 10 literals), and *medium-sized* learnt clauses (up to 50 literals), respectively.[7] In these graphs of communities, the node size is scaled according to the number of variables that belong to each community. Also, edges are scaled by their weights. Notice that edges weights are computed using the weights of the

[5] We cannot directly represent the VIG due to its large number of nodes (variables).

[6] The weight of the edge connecting communities A and B is the addition of the weights of the edges connecting one node from A and one node from B.

[7] As each clause of length l generates $\binom{l}{2}$ edges, it is hard to compute these graphs using *long* clauses.

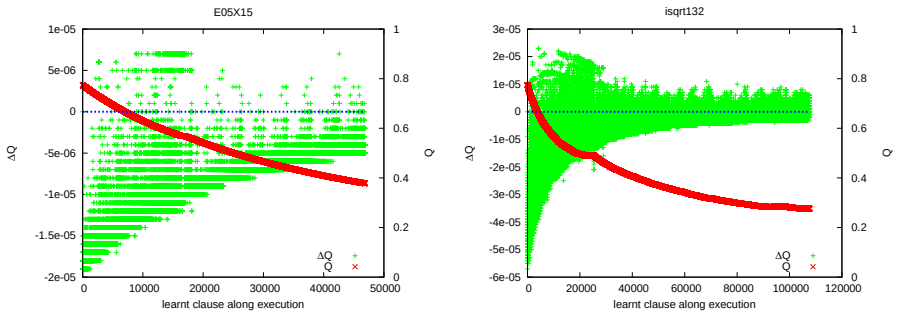

Fig. 2. Impact of adding learnt clauses on modularity, in instances `E05X15` (left) and `isqrt1_32` (right). Each point (x, y), with y measured in the left Y axis, represents a clause learnt at instant x and increasing Q on y. We also represent the evolution of the modularity Q (using the right Y axis).

VIG (i.e., taking into account the length of the clauses). As it is stated in [1], the community structure is clear in all of these three graphs. However, as we consider more learnt clauses, we can observe two phenomena. First, the number of communities (number of nodes in the graph of communities) decreases. This means that variables that originally belonged to distinct communities are now grouped into the same community. Second, the weight of the inter-communities edges increases. Therefore, from the two previous effects, we observe that the solver prefers to learn clauses containing variables of distinct (original) communities (also stated in [1]). This means that, in general, clause learning contributes to decrease the modularity.

A question now is: are there some learnt clauses that contribute to increase the modularity even when most of them do not? In order to answer this question, we can measure the increase of the modularity ΔQ that each learnt clause produces. Notice that ΔQ is positive when most of the new edges generated by such clause connect nodes (variables) of the same community. Otherwise, ΔQ is negative. After an extensive experimentation, we see that, in general, learnt clauses produce a very small decrease of the modularity (i.e., $\Delta Q < 0$, in most cases). In Fig. 2, we represent this analysis for the industrial instances `E05X15` and `isqrt1_32`. Each point (x, y), with y measured in the left Y axis, represents a clause learnt at instant x and increasing Q on y. We also represent (using the right Y axis) the value of the modularity Q using the original partition of variables, along the execution. We can see that, even when some learnt clauses contribute to increase the value of Q, most of them do not (i.e., $\Delta Q < 0$), and thus Q tends to decrease. Due to space limitations, we only represent this analysis in two benchmarks. However, we observed similar results in most industrial SAT instances studied. Therefore, we can conclude that, in general, learnt clauses contribute to destroy the (original) community structure of the formula. It is not due to some particular clauses but rather a general phenomenon of the learning mechanism.

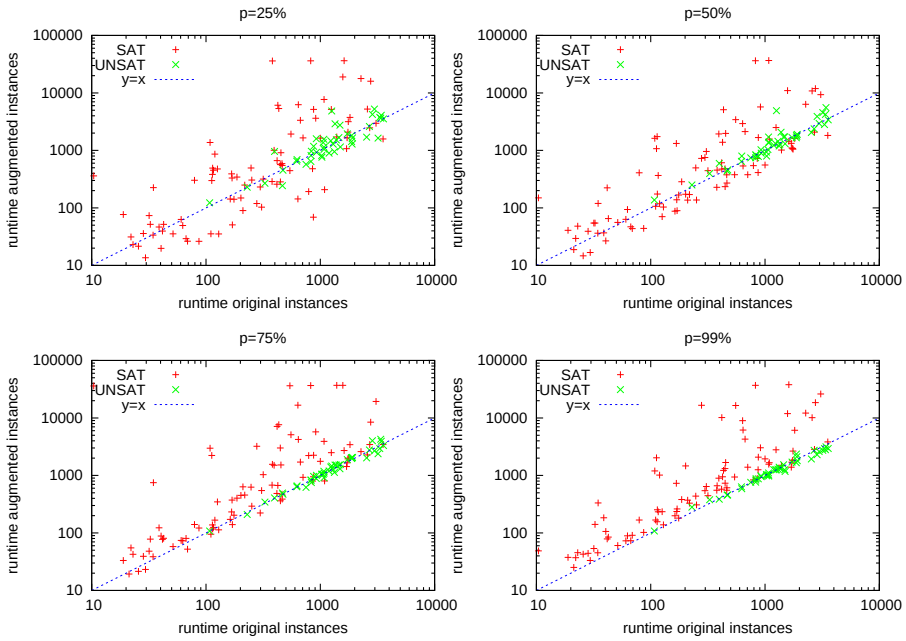

Fig. 3. Scatter plots of solving original instances (first step) versus generating and solving formulas augmented with learnt clauses (second plus third steps), at $p = 25\%, 50\%, 75\%$ and 99%.

4 On the Relevance of Learnt Clauses

In this section, we try to answer the following question: if we augment the original formula with a set of learnt clauses obtained from some CDCL solver, will this contribute to solve the formula faster? In order to answer this question, we first introduce the notion of *relevant* clauses.

Definition 1. *Given a SAT solver S, a formula Γ, and a set of clauses φ, we say that φ is* relevant *for Γ and S, if φ is a logical consequence of Γ and $\Gamma \cup \varphi$ is* easier to solve *for S than Γ.*

Notice that in this definition we neglect the time needed to compute φ. Obviously, previous definition is informal. In order to experimentally validate if a set of clauses is relevant, we have considered a significant set of industrial instances.

In a first experiment, we select the set of instances of the application track of the SAT Competition 2013 solved in less than one hour. Notice that this set contains both satisfiable and unsatisfiable instances. This experiment is divided in three steps. In all of them, we use the CDCL SAT solver MiniSAT [7].

First step: we compute the number of conflicts c needed to solve the formula in an arbitrary run.

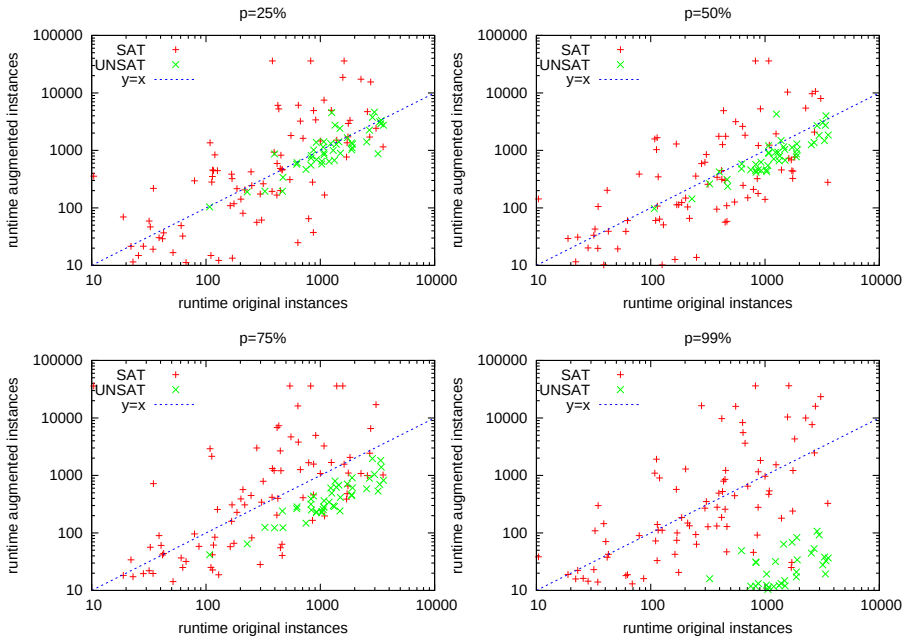

Fig. 4. Scatter plots of solving original instances (first step) versus solving formulas augmented with learnt clauses (third step), at $p = 25\%, 50\%, 75\%$ and 99%.

Second step: we repeat the same execution stopping the search after a certain number of conflicts $p \cdot c$ (where $0 < p < 1$), and we generate a new instance augmenting the original formula with the learnt clauses stored in the solver at that instant.

Third step: we solve the *augmented formula* generated in the previous step.

We could think that the third step is just the continuation of the second step due to a restart after $p \cdot c$ conflicts. But this is far from being true. First, CDCL SAT solvers do have more contextual information than learnt clauses, such as the activity counters, status of restarts, etc. It is also interesting to notice that the phase caching scheme [16] is not saved in the third step: a learnt clause could have been responsible for a propagation, and thus responsible for setting the phase caching scheme when backtracking, but this learnt clause could have been removed. Second, the learnt clauses used to generate the augmented formula will be treated as original clauses in the third step, i.e., they cannot be removed by the solver.

Since we limited the number of conflicts to $p \cdot c$ in the second step, you could expect to need around $(1-p) \cdot c$ conflicts to complete the search in the third step. Surprisingly, in our experiments, this is true when the instance is unsatisfiable, but not when it is satisfiable. If the formula is satisfiable, the aggregated runtime of generating the augmented formula (second step) and solving it (third step)

is usually higher than the runtime required to solve the original instance (first step).

Let us present these observations in detail. In Fig. 3, we present the scatter plot of the runtime of solving the original formula (first step) versus generating and solving the augmented formula (second plus third steps), with $p = 25\%, 50\%, 75\%$ and 99%, and distinguishing SAT and UNSAT instances. In unsatisfiable instances, there is almost no difference (i.e., almost all points are on the diagonal). On the contrary, in satisfiable formulas the differences are much bigger (almost all points are far from the diagonal). Moreover, as we increase p, solving original instances is faster than generating and solving their corresponding augmented formulas (almost all points are above the diagonal). In Fig. 4, we present the scatter plots of solving the original formula (first step) versus just solving the augmented formula (third step). Notice that in this case, we do not take into account the runtime needed to generate these augmented instances. However, even in this case, solving some satisfiable augmented instances takes more time than solving their corresponding original formulas.

We have observed that augmenting an instance with learnt clauses does not always contribute to make it easier, when the formula is satisfiable. Let us conjecture why. First, although adding learnt clauses helps to reduce the search space, there are other key components, such as the activity counters and the phase component caching. These heuristics are set to their *optimal*[8] values after a certain number of conflicts. The phase component caching may play a crucial role here, since the solver may use this information to keep the solution to a subproblem. Therefore, even if we have an oracle providing a set of learnt clauses, this does not mean that you will find a satisfying assignment faster. Also notice that the status of the activity counters cannot be reproduced from this set of learnt clauses. These counters depend on all clauses learnt during the execution of a solver, but some of them may have been removed, and therefore they do not belong to the provided set anymore. Second, in [18] it was shown that the runtime of solving unsatisfiable formulas is much more robust than for satisfiable ones. Shuffling the instance may have an important impact on satisfiable problems, but not on unsatisfiable ones: the effort to find the UNSAT answer (and the size its proof) are always of the same order. If we try to link our result to this work, we think a reasonable explanation is the following one. For satisfiable instances, the solver is mostly starting again the whole search, trying to *learn* the correct phase component caching values. In this case, adding learnt clauses can slightly help, but the overall process is dominated by the high discrepancy of CPU time needed for satisfiable problems when shuffling the instance. For unsatisfiable instances, this shows that the solver is *continuing* the same proof.

Therefore, even when adding learnt clauses does not always help in satisfiable instances, is it possible to find a set of highly useful clauses that makes these formulas easier? In the next section, we will show that we can use the community structure to identify some clauses that are indeed relevant for those instances, i.e., they help to solve satisfiable instances faster.

[8] In order to guide the search to a satisfying assignment.

Algorithm 1: Modularity-based SAT Instance Preprocessor (**modprep**)

```
   Input: SAT Instance Γ
   Output: SAT Instance Γ'
1  Γ' := Γ;
2  C := communityStructure(Γ);
3  foreach pair (c_i, c_j) of connected communities of C do
4  |  Solver s;
5  |  s.solve(c_i ∪ c_j);
6  |  if s == UNSAT then
7  |  |  return ∅;
8  |  Γ' := Γ' ∪ s.learntClauses
9  return Γ';
```

5 Detecting Relevant Learnt Clauses

Learnt clauses are redundant by definition, hence not strictly necessary. However, they can help to prevent exploring the same unsatisfiable subspaces during the search. Moreover, their role could be to *guide* the solver in building the UNSAT proof by resolution. It is essential here to see CDCL SAT solvers as a combination of backtrack search algorithms (where learnt clauses are used to prevent exploring the same search space) and resolution proof engines (where learnt clauses are used to derive new learnt clauses).

In the early versions of CDCL solvers, memory was an important issue [13, 7]. Therefore, some heuristics were proposed to remove useless clauses. Moreover, it is important to correctly manage the learnt clauses database in order to maintain a good unit propagation speed. More recently [2], some clause removal policies have been proposed. They aggressively remove most of the learnt clauses (95% of the learnt clauses can be removed). The proposed strategy is now one of the standards in CDCL engines. Thus, this policy is not only about maintaining good unit propagation rates, but also to guide the solver to some *easier* proofs. In Glucose, it was proposed to consider the number of decision levels occurring in a learnt clause as a measure of its quality (this was called Literal Block Distance, *LBD*, lower is better). The idea was that literals propagated at the same decision level were tightly connected and may often be propagated again and again together. Clauses of LBD 2 (called *glue clauses*) are kept forever in Glucose. Recently [15], it was shown that the LBD value was correlated to the number of communities of the clause. In this section, we show that community structure can be used to detect relevant learnt clauses.

In Alg. 1, we propose a technique presented as a preprocessing step, called *Modularity-based SAT Instance Preprocessor* (**modprec**). It augments the original formula with some learnt clauses based on its community structure. This algorithm proceeds as follows. First, it computes the community structure of the original formula (line 2), as described in Section 2. Recall that each community represents a set of clauses of the original instance. Then, for each pair of

connected communities[9], it creates a subformula containing both communities, and solves it (line 5). If this subformula is UNSAT, it returns the empty clause. Otherwise, the original instance is augmented with the clauses the solver learnt for solving such subformula (line 8). Finally, it returns the augmented instance.

Notice that the previous algorithm imposes a very strong condition, which is solving *all* subformulas between two connected communities and keeping *all* learnt clauses found in this process. This could be further refined. Moreover, this preprocessing step could be heuristically applied during the search in the flavor of inprocessing approaches [9].

Although we will show in next section that this approach works experimentally, we may wonder why these learnt clauses indeed improve the performance of the solver. It is worth noticing that, by construction, these learnt clauses are usually composed of at most 2 communities, and thus are clearly related to the notion of *glue clauses* aforementioned. In addition, as we showed in Section 3, learnt clauses contribute to destroy the original community structure. In order to do this, we first need to connect pairs of communities, then triples of communities, and so forth; since we learn clauses that connect all communities (i.e., the whole formula) and we derive the empty clause. Therefore, we do not want to erase the base of this process (clauses connecting pairs of communities). Notice that a solver not aware of the community structure may remove them, unless, as we do, these clauses are added in a preprocessing step as original clauses. i.e., the solver will not remove them.

In this work, we only consider learnt clauses connecting pairs of communities at the preprocessing step, and not triples or higher arities. This is because the combinatorial space for pairs can be managed efficiently by the SAT solver. For bigger arities, we would need some additional filtering criterion, or working on a parallel solver (discussed in Section 8).

6 Experimental Evaluation

In this section, we present an experimental evaluation of the modularity-based preprocessor presented in the previous section. All experiments were run in a cluster of 9 nodes IBM dx360 M2, each of them with 32GB of RAM and 2 processors Intel(R) Xeon(R) CPU L55202.27 GHz, limiting all experiments to a single core and to a maximum of 4GB of RAM. We use four representative CDCL SAT solvers: MiniSAT [7], Lingeling [3], Glucose [2], and MiniSAT-blbd [6]. MiniSAT is one of the most popular CDCL SAT solvers, while the three others were the best ranked solvers in the application track of the last SAT Competition 2014: Lingeling won both the UNSAT and the SAT+UNSAT tracks, MiniSAT-blbd won the SAT track, and Glucose was the second classified in the UNSAT track.

First, we evaluate how expensive is running the preprocessor described in Alg. 1 on a set of industrial SAT instances. We use the 300 application instances

[9] Two communities are connected if there exists at least one variable appearing in both of them.

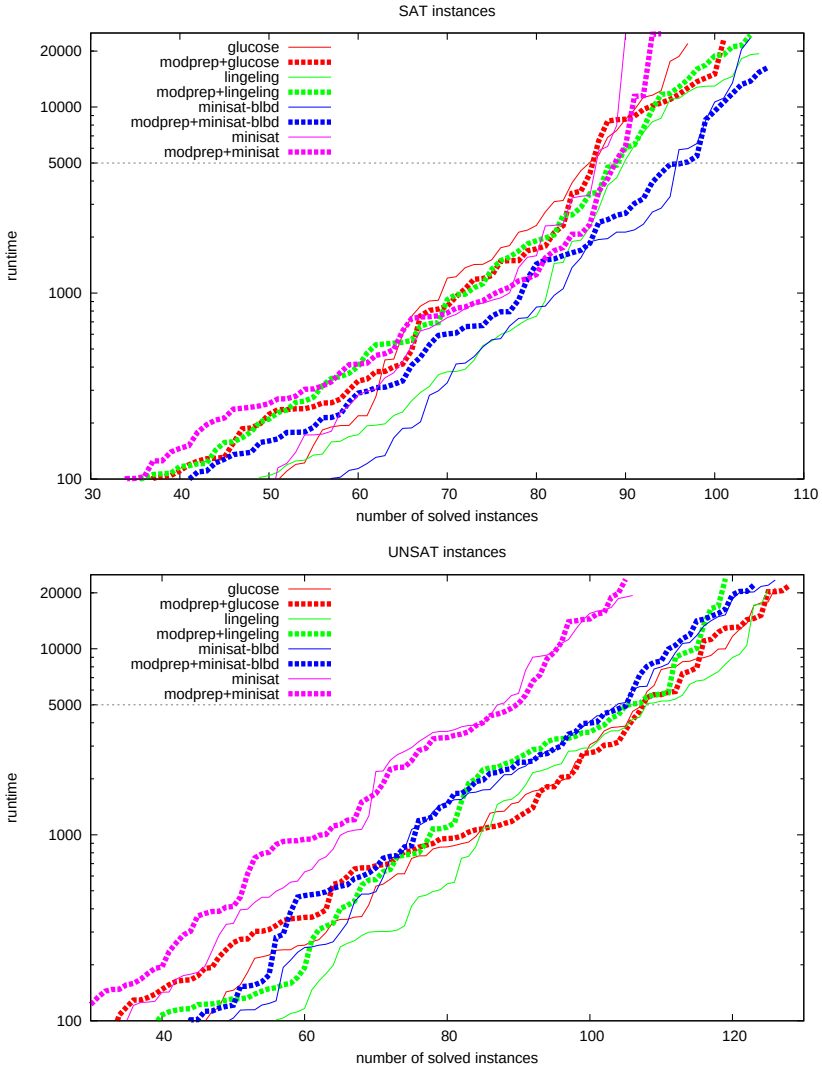

Fig. 5. Evaluation of application instances of the SAT Competition 2011, distinguishing satisfiable instances (top) and unsatisfiable instances (bottom), for Glucose, Lingering, MiniSAT-blbd, and MiniSAT; with and without using our preprocessor.

of the SAT Competition 2011. Notice that Alg. 1 can be split into two steps: i) partitioning the input formula into subformulas; and ii) solving them.

We compute the community structure as described in [1][10]. For this set of 300 application instances, this tool is able to correctly compute the community

[10] Tool available in `http://www.iiia.csic.es/~jgiraldez/software`.

structure of 298 instances. This process is, in general, very fast. The average, median and maximum runtimes are respectively 12.6, 4.3 and 294.5 seconds.

Then, we solve all subformulas using MiniSAT. This step is performed on the 298 industrial formulas, with an average, median and maximum runtime of 78.0, 21.8 and 975.8 seconds, respectively. The average, median, maximum and minimum number of clauses that our preprocessor learnt is 11243.9, 512, 794950 and 1 clauses, respectively. A natural question now is if the number of clauses learnt with this preprocessor depends on the solver used to solve such subformulas. We run again this step using Glucose instead that MiniSAT. Notice that Glucose uses a more aggressive clause removal policy. However, we observe that this solver learns, in general, a similar number of clauses as MiniSAT, and needs a similar runtime to solve these subformulas. This is because the input subformulas are, in general, very easy.

In the next experiment, we evaluate the performance of the mentioned solvers, with and without using the presented preprocessor (referred in the plots as *<solver>* and *modprep+<solver>*, respectively). In Fig. 5, we represent the plots of this evaluation (solvers with and without using the preprocessor) for the industrial instances of the SAT Competition 2011, distinguishing between satisfiable and unsatisfiable instances. We represent a cactus plot (i.e., the maximum runtime of solving a set of instances) with logarithmic Y axis. The timeout is set to 25000 seconds (the timeout usually used in competitions is 5000 seconds). We remark that the reported runtime when the preprocessor is used include the runtime of computing the community structure and the runtime of solving all subformulas. We observe that using our preprocessor with MiniSAT, Glucose or MiniSAT-blbd improves their performance in satisfiable instances. Moreover, in unsatisfiable instances, Glucose also improves its performance. Interestingly, for this timeout of 25000 seconds, enhancing a solver with our preprocessor results into the best choice for solving satisfiable instances (using MiniSAT-blbd) and unsatisfiable instances (using Glucose). More interestingly, the solver MiniSAT-blbd enhanced with our preprocessor also results into the best technique to solve satisfiable instances when a timeout of 5000 seconds is considered (similar to the timeout used in the competition). It is worth noting that, for very easy instances, the overhead of the preprocessor (i.e., computing the community structure and solving all subformulas) does not compensate.

We want to validate if the previous results also hold in a different set of industrial instances. We repeat the same experiment[11] for the set of 300 application instances of the SAT Competition 2014. In Fig. 6, we represent the cactus plot of this experiment, distinguishing between satisfiable and unsatisfiable instances. Again, we observe that Glucose and MiniSAT-blbd improve their performance in both satisfiable and unsatisfiable instances when the preprocessor is used. In fact, MiniSAT-blbd enhanced with our technique is the best solver in satisfiable instances. Interestingly, these solvers also improve their performance using a shorter timeout of 5000 seconds. For instance, in our cluster MiniSAT-blbd solves 97 SAT instances, while this solver enhanced with our preprocessor solves

[11] Excluding MiniSAT.

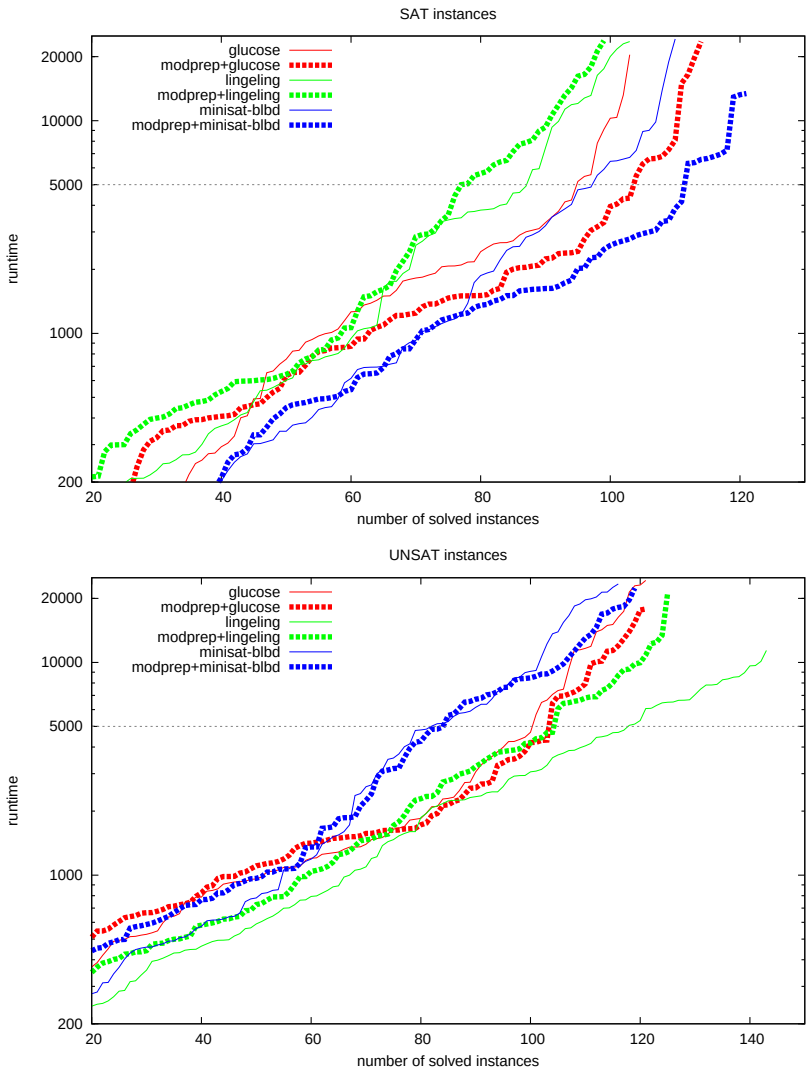

Fig. 6. Evaluation of application instances of the SAT Competition 2014, distinguishing satisfiable instances (top) and unsatisfiable instances (bottom), for Glucose, Lingering, and MiniSAT-blbd; with and without using our preprocessor.

111. This difference is significant in the context of competitions. Also, Glucose solves 194 SAT+UNSAT instances, while using our technique with this solver results into a total of 206 SAT+UNSAT solved instances. Again, this difference is significant. However, our preprocessor does not improve the performance of Lingeling.

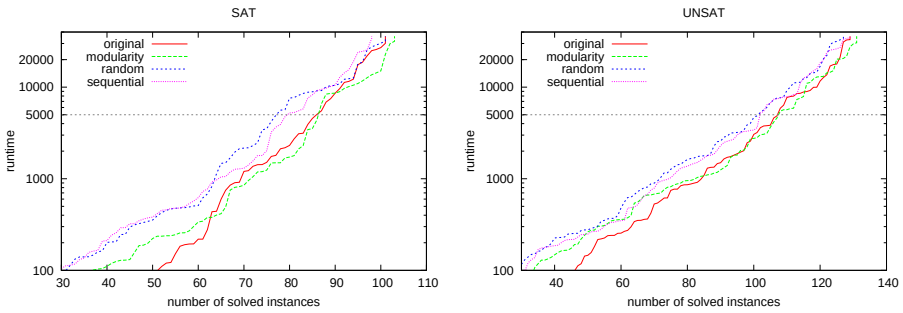

Fig. 7. Evaluation of random and sequential partitions, distinguishing between satisfiable (left) and unsatisfiable formulas (right), using the set of industrial instances of the SAT Competition 2011, and solved by Glucose.

Finally, we want to check if a random partition of the formula would have the same effect as the partition provided by the community structure. For every instance, we compute a random partition of the formula with the same number of components as in the community structure. Also, we compute a sequential partition, where all variables of a component have sequential indexes. Then, we repeat all the experimentation with these random and sequential partitions. In Fig. 7, we show the cactus plot of the results on the set of industrial instances of the SAT Competition 2011. As expected, none of these methods performs better than either our proposed technique or solving the original instances.

Notice that in the previous experiment, the average, median, maximum and minimum number of clauses learnt by our preprocessor was respectively 4015.12, 28, 209085 and 0 clauses using the random partition, and 35360, 987, 951839 and 0 clauses using the sequential partition. Recall that using the community structure, our preprocessor learnt in average, median, maximum and minimum a total of 11243.9, 512, 794950 and 1 clauses, respectively. Therefore, with random components the number of learnt clauses is smaller than using the community structure, whereas with sequential components this number is bigger. This suggests that the partition used to create the subformulas is more important than the number of clauses learnt by the preprocessor.

7 Related Work

A pioneering work on using community structure to speed-up solvers was presented in [12]. In particular, they proposed to solve Maximum Satisfiability formulas by partitioning them according to the community structure and adding incrementally to the MaxSAT solver the sets of clauses related to communities.

In [11], it is shown that learnt clauses are most likely to be composed by variables on the fringes between communities. Interestingly enough, this confirms that the learning scheme tends to destroy the community structure: adding clauses with internal variables of communities would increase the clustering into

communities. However, adding links between clusters by linking variables on their fringes seems to be more efficient.

As already mentioned, our work is also related to the work in [15]. Our current work contributes to confirm this by suggesting that good clauses are composed of variables from a few communities but, for the first time, it was possible to guide a CDCL SAT solver by the community structure of the formula. In particular, we think we were able to guide the solver to learn a set of initial *glue clauses*.

8 Conclusions and Future Work

In this paper, we use the community structure of industrial SAT instances to identify a set of highly useful learnt clauses. We show that augmenting a SAT instance with clauses learnt by the solver during its execution does not always mean to make the instance easier, especially in the case of satisfiable instances. However, we also show that augmenting the formula with a set of clauses based on the community structure of the formula improves the performance of the solver in many cases. Interestingly, this improvement is especially relevant in satisfiable instances. In particular, we use the set of clauses learnt from solving all subformulas consisting in pairs of connected communities.

We implement this approach as a preprocessor, and we show that it works experimentally on some representative sets of industrial instances, especially in satisfiable formulas. Interestingly, the SAT solver MiniSAT-blbd, which was the winner of the satisfiable track of the last SAT Competition 2014, enhanced with our technique improves its performance. It is also the case of Glucose, which improves its performance when it is enhanced with our technique in both satisfiable and unsatisfiable instances. To the best of our knowledge, this is the first time that community structure has been used to improve the performance of a CDCL SAT solver.

An important development of our work could be the design of a parallel solver. Each core could work only on a subset of the initial clauses, without communications. This could also allow us to extend our approach to tuples of communities instead of pairs of communities.

Our approach can also be improved by trying to guess which pairs of communities are important to work on. We are currently investigating this. At last, it is also important to link the community structure of formulas with their initial problem and generation. Linking the original problem with the detected communities is also an ongoing work.

Bibliography

[1] Ansótegui, C., Giráldez-Cru, J., Levy, J.: The community structure of SAT formulas. In: Proc. of SAT'12. pp. 410–423 (2012)

[2] Audemard, G., Simon, L.: Predicting learnt clauses quality in modern SAT solvers. In: Proc. of IJCAI'09. pp. 399–404 (2009)

[3] Biere, A.: Lingeling essentials, A tutorial on design and implementation aspects of the the SAT solver Lingeling. In: Proc. of POS'14 (2014)

[4] Blondel, V.D., Guillaume, J.L., Lambiotte, R., Lefebvre, E.: Fast unfolding of communities in large networks. Journal of Statistical Mechanics: Theory and Experiment 2008(10), P10008 (2008)

[5] Brandes, U., Delling, D., Gaertler, M., Görke, R., Hoefer, M., Nikoloski, Z., Wagner, D.: On modularity clustering. IEEE Trans. on Knowledge and Data Engineering 20(2), 172–188 (2008)

[6] Chen, J.: A bit-encoding phase selection strategy for satisfiability solvers. In: Proc. of TAMC'14. pp. 158–167 (2014)

[7] Eén, N., Sörensson, N.: An extensible SAT-solver. In: Proc. of SAT'03. pp. 502–518 (2003)

[8] Gomes, C.P., Selman, B., Kautz, H.A.: Boosting combinatorial search through randomization. In: Proc. of the Fifteenth National Conf. on Artificial Intelligence, AAAI'98. pp. 431–437 (1998)

[9] Järvisalo, M., Heule, M., Biere, A.: Inprocessing rules. In: Proc. of IJCAR'12. pp. 355–370 (2012)

[10] Katebi, H., Sakallah, K.A., Marques-Silva, J.P.: Empirical study of the anatomy of modern SAT solvers. In: Proc. of SAT'11. pp. 343–356 (2011)

[11] Katsirelos, G., Simon, L.: Eigenvector centrality in industrial SAT instances. In: Proc. of CP'12. pp. 348–356 (2012)

[12] Martins, R., Manquinho, V.M., Lynce, I.: Community-based partitioning for MaxSAT solving. In: Proc. of SAT'13. pp. 182–191 (2013)

[13] Moskewicz, M.W., Madigan, C.F., Zhao, Y., Zhang, L., Malik, S.: Chaff: Engineering an efficient SAT solver. In: Proc. of DAC'01. pp. 530–535 (2001)

[14] Newman, M.E.J., Girvan, M.: Finding and evaluating community structure in networks. Phys. Rev. E 69(2), 026113 (2004)

[15] Newsham, Z., Ganesh, V., Fischmeister, S., Audemard, G., Simon, L.: Impact of community structure on SAT solver performance. In: Proc. of SAT'14. pp. 252–268 (2014)

[16] Pipatsrisawat, K., Darwiche, A.: A lightweight component caching scheme for satisfiability solvers. In: Proc. of SAT'07. pp. 294–299 (2007)

[17] Silva, J.P.M., Sakallah, K.A.: GRASP: A search algorithm for propositional satisfiability. IEEE Trans. Computers 48(5), 506–521 (1999)

[18] Simon, L.: Post mortem analysis of SAT solver proofs. In: Proc. of POS'14. pp. 26–40 (2014)