HardCore Generation: Generating Hard UNSAT Problems for Data Augmentation

Joseph Cotnareanu
McGill University
Montreal, Canada
joseph.cotnareanu@mail.mcgill.ca

Zhanguang Zhang
Huawei Noah's Ark Lab
Montreal, Canada
zhanguang.zhang@huawei.com

Hui-Ling Zhen
Huawei Noah's Ark Lab
Hong Kong, China
zhenhuiling2@huawei.com

Yingxue Zhang
Huawei Noah's Ark Lab
Toronto, Canada
yingxue.zhang@huawei.com

Mark Coates
McGill University
Montreal, Canada
mark.coates@mcgill.ca

ABSTRACT

Efficiently determining the satisfiability of a boolean equation — known as the SAT problem for brevity — is crucial in various industrial problems. Recently, the advent of deep learning methods has introduced significant potential for enhancing SAT solving. However, a major barrier to the advancement of this field has been the scarcity of large, realistic datasets. The majority of current public datasets are either randomly generated or extremely limited, containing only a few examples from unrelated problem families. These datasets are inadequate for meaningful training of deep learning methods. In light of this, researchers have started exploring generative techniques to create data that more accurately reflects SAT problems encountered in practical situations. These methods have so far suffered from either the inability to produce challenging SAT problems or time-scalability obstacles. In this paper we achieve both by identifying and manipulating the key contributors to a problem's "hardness", known as cores. Where previous work has addressed cores, time costs have become very high due to the expense of traditional heuristic core detection techniques. We introduce a fast core detection procedure based on a graph neural networks. Our empirical results demonstrate that we can efficiently generate problems that remain hard to solve and retain key attributes of the original example problems. We demonstrate that the generated synthetic SAT problems can be used in a data augmentation setting to provide improved prediction of solver runtimes.

CCS CONCEPTS

• **Mathematics of computing** → **Combinatorial optimization**; • **Graph Algorithms**; • **Computing methodologies** → Generative and developmental approaches;

KEYWORDS

Boolean Satisfiability Problem, Graph Generation

IWLS '24, June 6-7, Zurich, Switzerland

ACM ISBN 978-1-4503-XXXX-X/18/06
https://doi.org/10.1145/nnnnnnn.nnnnnnn

ACM Reference Format:
Joseph Cotnareanu, Zhanguang Zhang, Hui-Ling Zhen, Yingxue Zhang, and Mark Coates. 2024. HardCore Generation: Generating Hard UNSAT Problems for Data Augmentation. In *Proceedings of 33rd International Workshop on Logic & Synthesis (IWLS '24).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The boolean satisfiability problem (the SAT problem) emerges in multiple industrial settings such as circuit design [12], cryptoanalysis [17], and scheduling [13]. While machine learning is not well suited for solving SAT problems — solvers are typically required to have perfect accuracy and return correct proofs — it does have applications in predicting wall-clock solving time for a given solver, which is important for algorithm selection [14, 15] and benchmarking [8]. SAT has also been gaining attention in Large-Language-Model reasoning, as it is a natural tool for interacting with the propositional-logical structure of many reasoning problems [18, 24].

A major challenge is a scarcity of high quality, homogeneous, real-structured data. The most commonly-used datasets have been compiled via a series of annual International SAT Competitions. The industrial origins of the compiled instances differ substantially, so the dataset is highly heterogeneous. The data is a good test for heuristic SAT solvers but for data-driven learning methods, this heterogeneous, sparse data is unsuitable. More complex models are thus forced to use randomly generated data [19]. This is problematic because the hardness-inducing dynamics in industrial data are very different from those in randomly generated problems. Training or testing on most existing randomly generated data provides little insight into how a model will perform on real industrial problems [2].

Recently, there has been interest in developing deep-learning methods for generating more realistic SAT instances. Early models [9, 23, 25] can generate instances that are structurally similar to original instances, but the problems are far too easy (a phenomenon called hardness collapse). Preserving hardness is essential. If we can only generate very easy problems, then the resultant dataset is not helpful in distinguishing the best-performing solver from the worst. It will not help a model to learn to predict real runtimes. Some recent work addresses the preservation of hardness. Li et al. [16] show impressive results and eliminate the aforementioned hardness collapse. Unfortunately, the resultant method is far too computationally expensive to be applicable for synthetic data generation

and augmentation for deep-learning. It can take over a week to generate a handful of new problem instances.

In this work, we take advantage of the connection between a problem's *core* and its hardness. The core is comprised of the identifiable minimal subsets of a boolean SAT problem that are unsatisfiable (UNSAT). At a high level, our strategy is to preserve the core of an original instance while iteratively adding random clauses to construct similar, but sufficiently diverse, problem instances that can enrich learning. To do this, we need to detect the core after each iteration of our algorithm. Unfortunately, traditional core detection algorithms are slow (often requiring a solution to the SAT problem itself) and can take hundreds of seconds [22]. Clearly it is infeasible for us to use such an algorithm if we want to build a fast generator — we need to perform core detection hundreds of times for every instance we generate.

To address this, we rephrase core detection as a binary node classification algorithm (core/not-core). We train a graph neural network to perform the task. Importantly, we can sidestep the data starvation issue, because our random data generation procedure generates hundreds of example instances that can be used for training the core detection algorithm. We can also take advantage of the fact that while it is important to identify the vast majority of clauses that belong to the core, we can tolerate a relatively high number of false-alarms by post-processing with a fast pruning algorithm.

We make the following novel research contributions:

- We propose a novel method for SAT generation that is the first that can both (i) *preserve hardness* and (ii) *generate instances in a reasonable time frame*. We can thus generate thousands of hard instances to augment a dataset in minutes or hours.
- We demonstrate experimentally that our proposed procedure preserves the key aspects of the original instances that impact solver runtimes. When performing dataset augmentation to learn to predict solver times (which lies at the core of solver benchmarking or solver selection), this is what we really need to preserve.
- We illustrate the value of our augmentation process for solver runtime prediction. For an example dataset, we show that our augmentation process can provide to a 20-50 percent improvement in mean absolute error. By contrast, other generation algorithms, achieve no statistically significant improvement.

2 BACKGROUND: BOOLEAN SATISFIABILITY

Definition and Notation. The Boolean Satisfiability Problem (SAT) is the problem of determining if a given Boolean formula can be made to evaluate to true. Phrased differently, its the question of whether or not there is an assignment of binary values to the variables in the formula for which the formula is true. Typically, a SAT instance is represented in Conjunctive Normal Form (CNF). In this form, the formula is written as a conjunction (logical AND) of disjunctions (logical OR), for example $f = (\neg A \vee B \vee C) \wedge (A \vee \neg C) \wedge (\neg B \vee C)$. The signed version of each variable that appears in the formula is known as a literal. For example, A and $\neg A$ are both literals of the variable A [4, Chapter 2].

Another useful representation of a CNF is as a set of sets, where each set (referred to as a clause) represents a disjunction in the CNF and contains the literals included in that disjunction. Denote the i-th clause in the formula f by c_i and the j-th literal in clause c_i as l_j. If there are n_c clauses in f and n_{l_i} literals in clause c_i, we can express the formula as:

$$c_i = \bigcup_{j=1}^{n_{l_i}} l_j\,, \qquad f = \bigcup_{i=1}^{n_c} c_i\,. \tag{1}$$

Core Definition. Given an unsatisfiable (UNSAT) instance U, there is a subset of clauses called a Minimally Unsatisfiable Subset (MUS) or a Core. This subset is the smallest possible subset of clauses from U that is UNSAT [4, Chapter 11].

Graph Representation of CNFs. There are several common graph representations for a CNF. In this work we use the Literal-Clause Graph (LCG). This is an undirected and bipartite graph. Each node in the first set of nodes in the graph represents a clause and each node in the second represents a literal. We then construct an edge for each occurrence of a literal in a clause such that the set of undirected edges e is defined as

$$e = \bigcup_{i=1}^{n_c} \bigcup_{j=0}^{n_{l_i}} (l_{j_{c_i}}, c_i)\,. \tag{2}$$

3 RELATED WORK

3.1 Deep-learned SAT generation

The problem of learned generation for SAT problems was first established in 2019 with SATGEN [23], motivated by a lack of access to industrial SAT problems. SATGEN used a graph generative adversarial network (GAN) to generate graph representations of SAT problems.

Soon after, G2SAT [25] was released. G2SAT introduced a novel generation framework in which problems are represented as graphs and the graphs are progressively split into small trees, and a graph neural network (GNN) is trained to discern which trees should be merged to restore the original graph. While innovative, the method is slow due to its need to sample tree pairs until the trained model permits sufficient merging to form a SAT problem of sufficient size. G2SAT reported impressive results on its ability to generate problems which maintained solver rankings compared with real input problems, and showed a moderate ability to improve solvers by tuning them on generated problems. G2SAT inspired several models, three of which were small modifications of graph representation or GNN model and reported slightly improved results. The most recent improvement on the G2SAT framework, HardSATGEN [16], includes many domain-inspired considerations in its design such as communities and cores. HardSATGEN was the first method for deep-learned SAT generation that demonstrated an ability to generate problems which were not trivial to solve for solvers: often the generated problems took nearly as long or even longer for a solver to solve than the corresponding seed problem. Unfortunately, however, the core awareness aspects of the design lead to HardSATGEN being extremely slow, making it challenging to use in any setting where many new instances are needed.

Aside from this family of G2SAT models, W2SAT [21] follows an approach more similar to the original SATGEN. It employs a modification of SATGEN's graph representation and a low-cost general graph generation model, and obtains new SAT problems via graph decoding. W2SAT is extremely efficient, but like G2SAT and

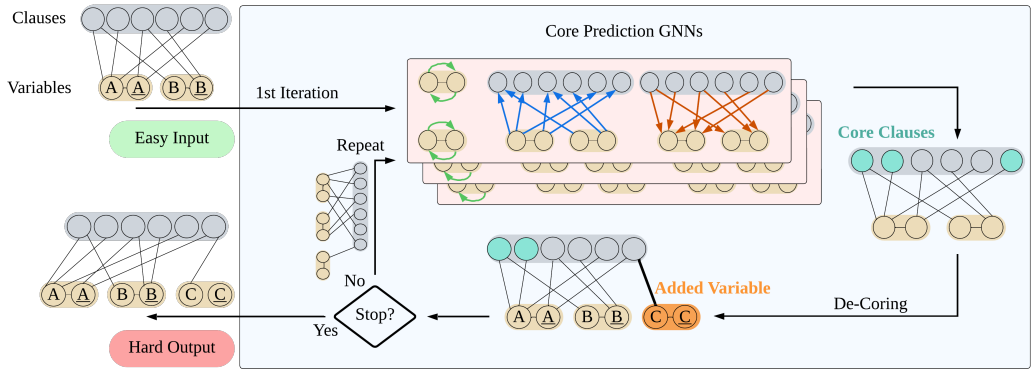

Figure 1: Core Refinement. The core refinement process comes in two steps: (1) Core Prediction in which we use a GNN-based architecture to identify the core of the generated instance and (2) De-Coring in which we add a non-conflicted literal to a clause in the core, rendering the core satisfiable and giving rise to a new, larger minimal unsatisfiable subset (core). As steps (1) and (2) are repeated the core gradually becomes larger, raising the hardness of the generated instance.

its descendants (up until HardSATGEN), it is incapable of generating problems which compare to their original counterparts in hardness.

A final method in the related work is G2MILP [10]. This method was not in fact designed for SAT problems specifically, but for Mixed Integer Linear Programs (MILPs), which are the general case of SAT. While designed with MILPs in mind, a simple but naive modification allows us to use G2MILP to generate SAT problems. The method is nearly as efficient as W2SAT but also struggles to generate hard instances.

3.2 Core Prediction

Core Detection can be a helpful tool for understanding UNSAT problems. Cores are often seen as a strong indicator of the hardness of an UNSAT problem [1]. There are multiple classical, verifiable methods for Core Detection, with the current standard being Drat-Trim[22]. Drat-trim requires that the problem be solved once by a SAT solver. This is, of course, slow. In response to this, Neurocore [19] was designed to predict the core of a SAT problem. Neurocore converts the input problem to a graph and uses a GNN to predict cores. Strangely, however, Neurocore does this on variables rather than clauses. Cores are defined to be subsets of clauses, rather than variables, and so this choice seems unnatural. Neurocore strives to be a machine-learning based variable-selection heuristic for SAT solvers, which motivates the focus on variables.

4 PROBLEM STATEMENT

Given a training set of UNSAT CNFs $S = \{f_1, f_2, ..., f_{m_S}\}$, and a corresponding set of label vectors $R = \{\mathbf{r}_1, \mathbf{r}_2, ..., \mathbf{r}_{m_S}\}$, we wish to train a generative model G that can construct new examples. The label vector $\mathbf{r} \in \mathbb{R}^d$ represents the hardness of the SAT problem and we model it as a deterministic mapping, i.e., $\mathbf{r}_1 = g(f_1)$. In our experiments, the vector is derived by recording the solution times for a pre-specified set of SAT solvers.

We assume that the m_S CNFs in the training set are i.i.d. examples from an underlying distribution $\mathcal{D}$. We denote the generative model distribution by $\mathcal{D}_G(S)$, highlighting that it is dependent on the random training set S. We can obtain a new dataset of m_G i.i.d. samples S_G using the generative model. The total number of samples in the augmented set $\tilde{S}$ is then $m_S + m_G$.

Our primary goal is to derive a generative procedure that produces sufficiently representative but also diverse samples such that the error obtained by training a model on the augmented dataset $\tilde{S}$ is less than that obtained by training on the original dataset S.

As an example task, we consider the prediction of runtime for a candidate solver. This is an important component of algorithm selection where we wish to choose which solver(s) to run for a given problem instance in order to minimize solution time. In this case, the appropriate loss function is the absolute error between the predicted time and the true time.

We are also interested in evaluating the distance between the distributions $\mathcal{D}$ and $\mathcal{D}_G$. We examine this through the lens of the hardness label vectors. The application of g to the CNF descriptors generated according to $\mathcal{D}$ or $\mathcal{D}_G$ induces distributions in $\mathbb{R}^d$. To evaluate the simlarity of the original and generated instances, we calculate the empirical maximum mean discrepancy distance between these induced distributions.

5 METHODOLOGY

Our generation strategy can be broken into three steps: (1) extraction of the core from a seed instance; (2) addition of random new clauses, generated with low cost; and (3) iterative core refinement.

Figure 1 provides an overview of the key core refinement procedure. It consists of a two-step cycle of (a) high-speed core extraction using our novel GNN-based method; and (b) unconflicted literal addition to break any undesirably easy core. The iterative cycle ends when either the limit of iterations is reached.

5.1 Generating Hard Instances

Trivial Cores. We have already established that cores are the primary underlying hardness providers in UNSAT instances, because

a solver must only determine that a subset of a CNF is UNSAT for the whole CNF to be UNSAT, and a core is the smallest subset of clauses of a CNF that is UNSAT. A smaller core is generally easier to solve because it is likely to involve fewer variables than the CNF, meaning that the total number of variable assignment combinations to try is reduced. Therefore small cores — that is, cores made up of few clauses — are likely to make the CNF trivially easy. An example of a trivial core is $(A \vee B) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee \neg B)$. Here, only 4 combinations of variable assignments must be attempted before a naive solver knows that the core (and therefore the CNF which includes this snippet) is UNSAT: $(A = 0, B = 0), (A = 0, B = 1), (A = 1, B = 0), (A = 1, B = 1)$.

Whenever we add a new random clause to an UNSAT instance, there is the danger of creating a trivial core. For example, consider an UNSAT instance which is hard and includes three of the four clauses from the example above: $(A \vee B) \wedge (\neg A \vee B) \wedge (A \vee \neg B)$. If during generation we unknowingly add the clause $(\neg A \vee \neg B)$, the UNSAT instance's large (hard) core will be replaced with a trivial one, leading to hardness collapse. Maintaining an awareness of the cores and potential cores in a CNF as we perform modifications is very challenging. We take a different approach which we refer to as *Core Refinement*.

Core Refinement. The Core Refinement process is made up of two steps that are repeated n times, where n is the number of generated clauses. The procedure is depicted in Figure 1. The first step of the process is to identify the core of the generated instance. The addition of random new clauses in step (2) is very likely to create core that is trivially easy to locate. It is very unlikely to be the same as the core of the original instance. Once we have detected this easy core, we render it satisfiable by adding a new literal to a clause in the core. The addition of a single, flexible literal eliminates the constraints of the core and makes it possible to satisfy.

Returning to the previous example, the UNSAT CNF $(A \vee B) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee \neg B)$ can be made satisfiable by modifying any of the clauses in this fashion: $(A \vee B \vee C) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee \neg B)$. The introduction of literal C in the first clause means that $(A = 0, B = 0, C = 0)$ is now a satisfying solution.

As these two steps are repeated, the core of the instance gradually becomes larger and is likely to be much more difficult. The process ends after a fixed number of iterations. In our experiments we choose this to be the number of generated clauses. Since the hardness of the core is the hardness of the instance [1], the refinement process can be seen as progressively raising the hardness of the problem.

Underlying Hard Core Guarantee. The Core Refinement process is designed on repeatedly eliminate easy cores, so after each iteration, the core becomes harder. Finally, after many iterations, we hope that the remaining core is hard (similar to the original example instance). This process can only be guaranteed to lead to a hard core if an underlying hard core exists in the instance at the start of the refinement process. Refinement then whittles away easy cores until only the hard one remains.

There is a possibility of creating a hard core through the random generation of clauses, but we cannot rely on this. We therefore must introduce an element to our design so that we can rely on there being a hard core. To do this we identify cores from the original instances and mix them into the generated instances.

5.2 Core Prediction

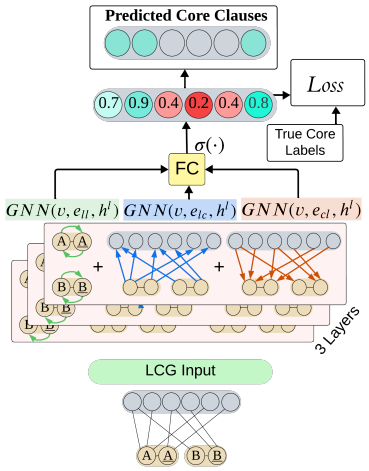

Figure 2: Core Prediction GNN Architecture. We construct our GNN using three parallel message passing neural networks (MPNN) whose calculated node embeddings are aggregated at each layer to form the layer's node embeddings. Readout is done by taking the sigmoid of a fully-connected layer on clause node embeddings and thresholding the values to identify positive and negative predictions. Training is supervised by taking a binary classification loss between the true core labels and the clause nodes' core prediction probabilities.

We have two critical objectives for our method: low time cost and hard outputs. While the Core Refinement process serves us well in generating hard instances, a naive implementation using existing core detection algorithms is unacceptably expensive in terms of computation requirements. Current core detection algorithms require that the instance be solved by a solver [22]. This means that the Core Refinement process must address an NP-Complete problem, since solving a SAT instance is NP-Complete.

We adopt the strategy of approximating the Core Detection algorithm. Since an instance can be naturally represented using a bipartite graph, and the goal of core detection is binary classification of each clause (node) as part of the core, we expect that a graph neural network is a promising approach.

Graph Construction. We represent each instance as a graph as outlined in Section 2. We make two changes: (a) we add message-passing edges to connecting matching positive and negative literals (e.g, $\neg A$ and A); (b) we replace each undirected edge with two directed edges. These changes are designed to facilitate the diffusion of information in the GNN. We denote the set of literal-literal message passing edges by $\mathcal{E}_{ll} = \bigcup_{i=1}^{n_v} (l_{i+}, l_{i-})$, where n_v is the number of variables in the instance. We denote the set of literal-to-clause directed edges by $\mathcal{E}_{lc} = \bigcup_{i=1}^{n_c} \bigcup_{j=0}^{n_{l_{c_i}}} (l_{j_{c_i}}, c_i)$. We denote the set of clause-to-literal directed edges by $\mathcal{E}_{cl} = \bigcup_{i=1}^{n_c} \bigcup_{j=0}^{n_{l_{c_i}}} (c_i, l_{j_{c_i}})$.

GNN Architecture. Given the heterogeneous nature of our graph, arising from different node and edge types, we use three Graph Message Passing models (one for each edge type). We couple these models by averaging their embeddings after each layer. We define a single layer as:

$$h^{l+1} = \sigma(\frac{1}{3}(GNN(\mathcal{V}, \mathcal{E}_{cl}, h^l) + GNN(\mathcal{V}, \mathcal{E}_{lc}, h^l) + GNN(\mathcal{V}, \mathcal{E}_{ll}, h^l))), \quad (3)$$

where σ is a non-linear activation function. Finally, we obtain a core membership probability for each clause node by passing the embeddings through a fully connected linear readout layer followed by a sigmoid function to the clause node embeddings. We threshold the values to obtain positive and negative classifications of core membership:

$$out = \mathbb{1}_{>0.5}(\sigma(xh_c^L + b)). \quad (4)$$

Training. Our augmentation process is motivated by a scarcity of data. We must therefore address this when training the core detection GNN. We achieve augmentation of the available data by executing the generation pipeline described above for a small number of instances, using a slow, traditional but proof-providing tool for Core Detection in the Core Refinement process. By saving the instance-core pair after each iteration of the core refinement process, we can construct sufficient supervision data for training the Core Prediction GNN model. Although the instance-core pairs we construct this way are correlated, there is sufficient variability for the GNN model to generalize well to other instances. We train using the standard binary cross-entropy loss function.

6 EXPERIMENTS AND RESULTS

6.1 Experimental Setting

Data. We use two datasets for the following experiments. One is a proprietary dataset of industrial Logical Equivalence Checking (LEC) problems. LEC is a step in the automated circuit design process in which a digital circuit is modified for spacial and cost savings by various algorithms. The output of the modification process must be tested against the input to ensure that the final circuit is logically equivalent to the input, which is done by forming a SAT problem out of the two circuits. If the problem is found to be UNSAT then the two are equivalent. In practice, the extreme majority (more than 99%) of LEC problems are UNSAT. The second dataset is one that was generated randomly by generating random graphs and transforming them to CNFs using the Tseitin Transformation [20]. This method is chosen because it is typically used to transform logic circuit graphs to CNFs and will give the random dataset a more realistic view than uniformly and randomly generating CNFs directly. For data statistics such as problem size, hardness and dataset size, see Table 1 in the Appendix.

SAT Solvers. We select 7 solvers for hardness analysis: Kissat3 [3], Bulky [7], UCB [5], ESA [5], MABGB [5], moss [5] and hywalk [6]. These solvers are fairly complementary in their performance in that when some of these solvers perform well on an instance, some perform very poorly. This way, we cover a good spread of runtime distributions in our analysis. We run our experiments on a Intel(R) Xeon(R) Platinum 8276 CPU @ 2.20GHz cpu and 3 Nvidia Tesla V100 GPUs.

We compare to the following baselines:

- **HardSATGEN [16]**: A high-cost split-merge generator with community structure and core detection that is capable of generating hard instances.
- **W2SAT [21]**: A low-cost generative method that utilizes a less common SAT graph representation which was reported to generate very easy problems.
- **G2MILP [10]**: A low-cost VAE-based generative model designed for the general case of SAT: MILPs.

6.2 Research Questions

Our work is motivated by the goal of **fast** generation of **hard** and **realistic** UNSAT datasets for **data augmentation**. Given these goals, we now establish our strategy for evaluating our model, identifying the key research questions that our experiments explore.

Question 1: Is the method able to generate hard instances? In order to quantify 'hardness', we choose the wall-clock solving time for each solver as a metric. We deem a set of generated instances 'hard' if the average solver runtime is at minimum 80% of the original dataset's average hardness. If average solver time for the set of generated instances is below 5%, we consider that *hardness collapse* has occurred.

Question 2: Is the method fast? We measure generation speed by the time required to generate an instance (in seconds). We evaluate this by measuring the wall-clock time of each model during inference and dividing by the number of generated instances. While we do not require extremely fast generation, a method should be able to generate hundreds of instances per hour so that we can augment a dataset in a reasonable time frame.

Question 3: Is the method able to generate datasets that are similar to the original datasets in terms of hardness distribution? Previous work such as [16, 21, 25] has elected to measure how close their generations are to real data (the training data) by comparing the similarity between each seed instance and its resulting model output. Given that our goal is to provide a set of instances to augment a dataset, similarity between a specific seed and a generated instance is not necessary (and perhaps not even desirable). We focus on distribution-level experimentation and analysis. Although past work has examined graph statistics such as modularity and clustering coefficients, we find little evidence that these are indicative of the hardness of generated instances. We focus on the similarity of the distributions of the hardness vectors (vectors of solver runtimes) because hardness is of primary importance when working with SAT problems. SAT algorithm researchers focus on solve-times when designing and comparing methods; machine learning researchers target solve-time prediction and solver ranking/selection problems. We conduct three analyses to answer this question: (i) a visual inspection of datasets' runtime distributions across solvers; (ii) a visual inspection of datasets' solver ranking distributions; and (iii) an evaluation of the distance between runtime distributions using the Maximum Mean Discrepancy (MMD) metric.

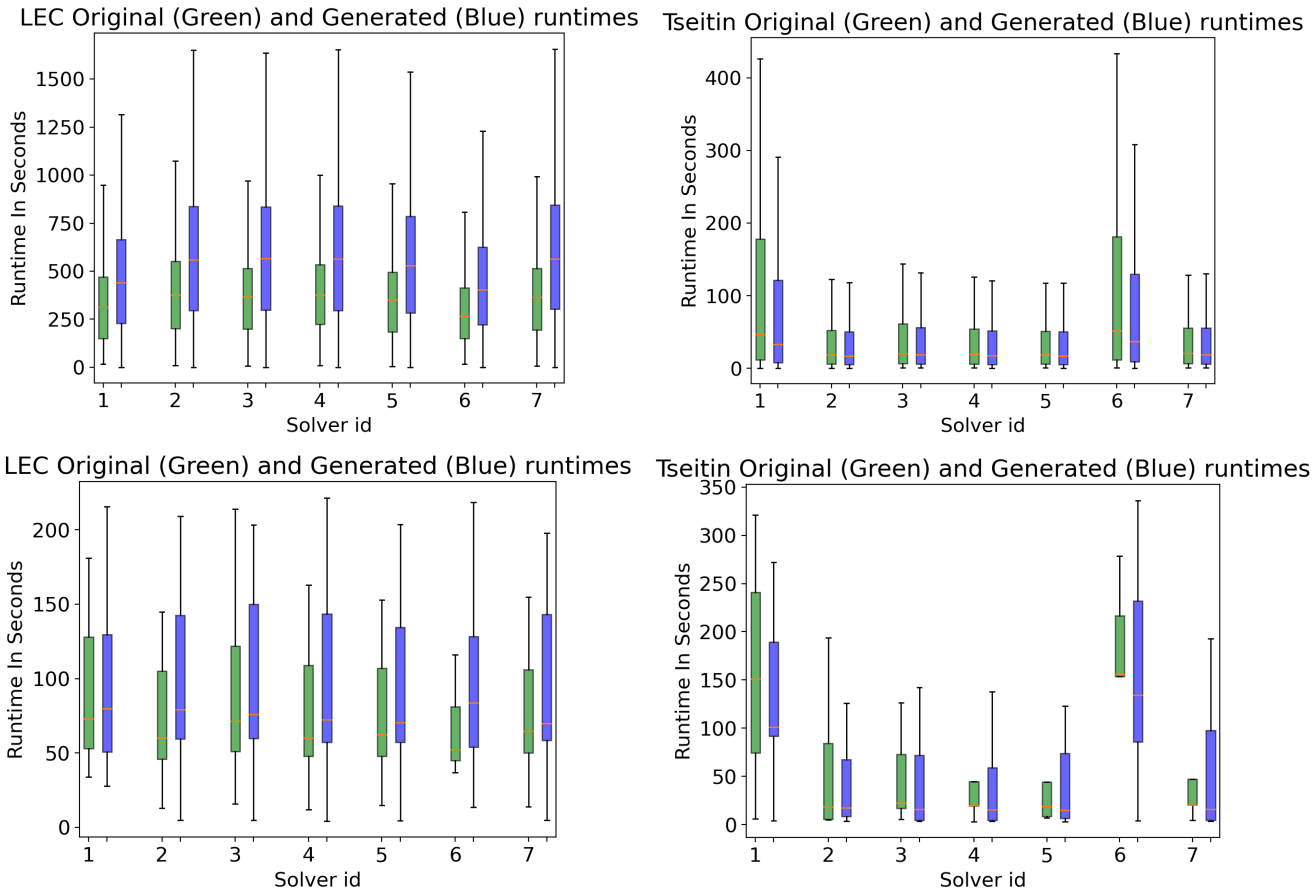

Figure 3: HardCore (top) and HardSATGEN (bottom); LEC (left) and Tseitin (right). Boxplots of runtimes per solver for Original (Green) and Generated (Blue) instances.

Question 4: Can we successfully augment training data with the method's generated data for machine learning? This question can be answered directly — we evaluate whether a model trained on an augmented dataset can perform better solver runtime prediction than the same model trained on the original dataset.

6.3 Q1: Do we generate hard instances?

In Table **??** we compare generated with original hardness. W2SAT and G2MILP both suffer hardness collapse, whereas HardSATGEN and HardCore generate hard instances.

6.4 Q2: Is the method fast?

In Table **??** the division between fast and slow procedures is very clear: W2SAT, G2MILP and HardCore all exhibit similar instance generation times, with W2SAT being the fastest. By contrast, HardSATGEN takes close to 2 hours to generate a single instance.

6.5 Q3: Are generated data similar to original data in terms of hardness distribution?

G2MILP and W2SAT exhibit hardness collapse, rendering distributional analysis of the solver runtime distributions pointless. Therefore, we compare HardSATGEN and HardCore. Note that due to HardSATGEN's high cost, we can only generate 50 LEC instances and 25 Tseitin instances in about 3 days. In the following experiments, we compare "original" and "generated" data. Here, "original" refers to only those instances used as seeds during inference for each model; "generated" refers to the outputs. Hence, the "original" sets for HardSATGEN and HardCore are different because the number of seed instances is different (because due to time constraints we are limited in how many HardSATGEN instances we can generate).

MMD. Table **??** shows that HardCore achieves runtime distributions far closer to the original distributions compared to HardSATGEN with respect to the MMD metric. We calculate these values by taking the MMD between the set of instances used as seeds during generation (a subset of the training set) and the corresponding set of generated instances. Although HardSATGEN preserves structural aspects of the seed instance, the solver time vectors are

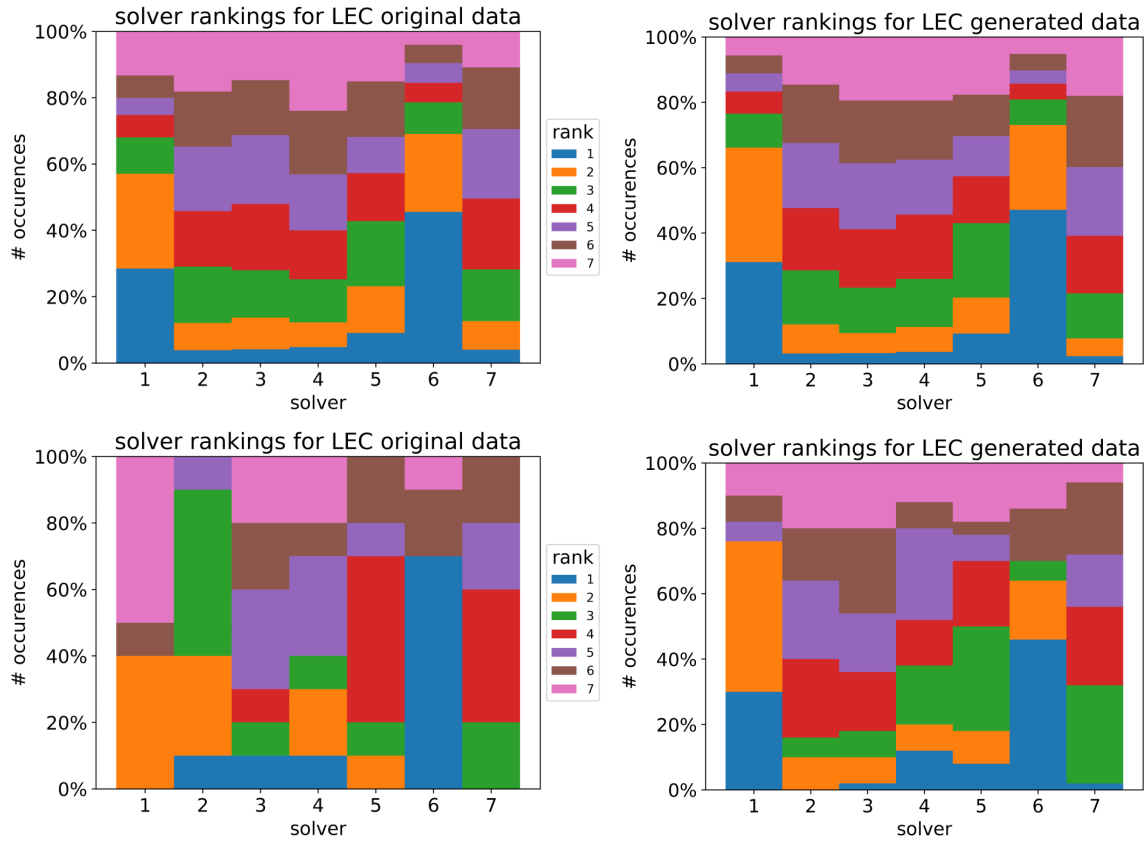

Figure 4: HardCore (top) and HardSATGEN (bottom) Comparison of Solver Ranking Histograms for Original and Generated LEC data. For the analogous results on Tsietin data, see Figure 6 in the Appendix

considerably different from those of the seeds. These results imply that, in the view of the SAT solvers, HardCore's generated dataset is closer to the original dataset than HardSATGEN.

Runtime Distribution Boxplots. In Figure 3 we visually compare the per-solver runtime distribution of HardCore's generated datasets to the corresponding original datasets. For the LEC generated data, the instances take longer to solve on average than the originals, but the ordering of the solver performance is preserved, with solvers 1 and 6 being slightly superior. For the Tseitin data, the solver time box plots are very similar for the generated and original instances. The original dataset has longer tails (harder instances) for solvers 1 and 6. Figure 3 shows that HardSATGEN achieves similar results in terms of the box plots of solver times. For the Tseitin dataset, the boxplots differ more, particularly for solvers 6 and 7.

Solver Ranking Histograms. In Figure 4 we use stacked bar plots to visually compare the solver-ranking distribution for LEC and HardCore's LEC generation. These plots show how often each solver ranks first, second, etc. with respect to completion time. We see that solvers 1 and 6 are most commonly the fastest. We observe a striking resemblance between the plots for the original and generated instances. For example, note the near identical shape of rank 1's histogram (blue) across solvers for both the LEC original and HardCore generated data.

Figure 4 indicates that HardSATGEN does not preserve the solver ranking distribution. The stacked bar plots for the generated instances are very different from those of the original instances. In particular, for the LEC data, solver 2 is never the best solver in the seed instances and only the second best 40% of the time. For the generated instances, it is the fastest 30% of the time.

6.6 Q4: Can we successfully augment data?

We address the task of runtime prediction and compare the performance of two models: one trained on only original data and the other trained on a dataset augmented with generated instances. We train the SATzilla model to predict solver runtime of one specific solver on a given instance. We repeat this for each of the 7 solvers. The task in the experiment is to predict total runtime of a solver over a benchmarking set of instances. We calculate the MAE of the predicted total runtime for each solver and average over the solvers. We compare HardCore, W2SAT and two versions of HardSATGEN: (i) HardSATGEN-Strict and (ii) HardSATGEN-N. For HardSATGEN-Strict, we only generate as many instances as possible in the time it takes HardCore to generate the desired number of instances. For HardSATGEN-N, we generated N instances, where N was selected

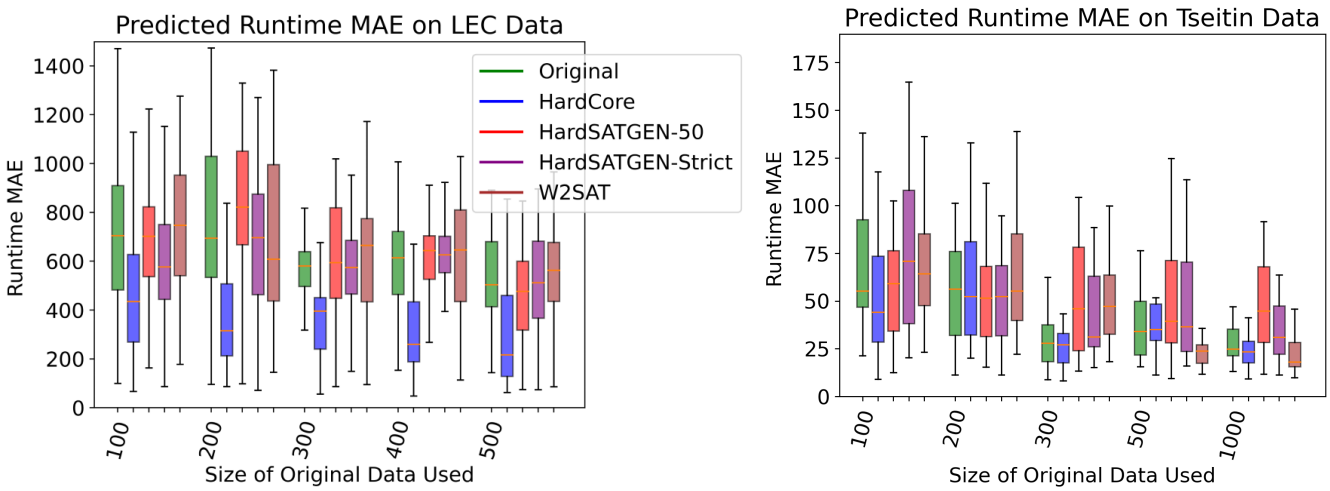

Figure 5: LEC Benchmark-Runtime Prediction Error.

as the number that could be generated in approximately 3 days of computation. We also compare to the un-augmented training sets and refer to it as Original.

Figure 5 shows that for the LEC dataset, training after augmenting using HardCore leads to a significant reduction in MAE. No other data generation method leads to a consistent improvement.

In Figure 5 we also observe that for the Tseitin dataset, none of the generation methods achieves a consistent improvement. HardCore and HardSATGEN-25 provide an improvement when the original dataset is very small. The Tseitin dataset is much more homogeneous than the LEC dataset, so the learning task is easier, as evidenced by the much smaller MAE values. It is thus more challenging for augmentation to provide a sizeable benefit. Surprisingly, W2SAT augmentation leads to an improvement when there is a larger amount of original data, despite W2SAT exhibiting hardness collapse. This may result from an unintended regularization effect as the injection of easy examples reduces a tendency to overestimate the runtime.

7 LIMITATIONS

The primary limitation of our work is that it is restricted to UNSAT problems. While some SAT applications are almost entirely UNSAT (e.g., circuit design), many are not. With our adopted approach, this limitation is unavoidable because cores are only present in UNSAT problems. However, there is a concept for SAT problems analogous to the core, known as a *backbone*. Focusing on preserving the backbone is a potential avenue for an analogous method for satisfiable SAT problems.

Another limitation is that, like many graph models, our method struggles to scale to large problems. As the size of the SAT problem increases, memory and computation costs scale in polynomial complexity meaning that the SAT problems which have millions of clauses are currently out of reach for this method.

8 CONCLUSION

We present a fast method for generating UNSAT problems that preserves hardness. Existing deep-learned SAT generation algorithms either (1) are incapable of generating problems that are even 5% as hard as the example input problems; or (2) can generate hard problems but take many hours for each instance. Our proposed method targets the core of a SAT problem and iteratively performs refinement using a GNN-based core detection procedure. Our experiments demonstrate that the method generates instances with a similar solver runtime distribution as the original instances. For a more challenging industrial dataset, we show that data augmentation using our proposed technique leads to a significant reduction in runtime prediction error.

REFERENCES

[1] Carlos Ansótegui, Maria Luisa Bonet, Jordi Levy, and Felip Manya. 2008. Meauring the Hardness of SAT Instances. In *Proc. AAAI Int. Conf. Artificial Intell.*
[2] Tomas Balyo, Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda (Eds.). 2022. Proc. of SAT Competition: Solver and Benchmark Descriptions.
[3] Armin Biere, Katalin Fazekas, Mathias Fleury, and Maximillian Hessinger. 2020. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition. *SAT COMPETITION*, 50.
[4] Armin Biere, Marjin Heule, Hans von Maaren, and Toby Walsh. 2009. *Handbook of Satisfiability.* IOS Press.
[5] Mohamed Sami Cherif, Djamal Habet, and Cyril Terrioux. 2022. Kissat MAB: Upper Confidence Bound Strategies to Combine VSIDS and CHB. *SAT COMPETITION*, 14.
[6] Md Solimul Chowdhury. 2023. kissat-hywalk-gb, kissat-hywalk-exp, kissat-hywalk-exp-gb, and malloblin Entering the SAT Competition. *SAT COMPETITION* 1428, 28.
[7] Mathias Fleury and Armin Biere. 2022. GIMSATUL, ISASAT, KISSAT. *SAT COMPETITION*, 10.
[8] Tobias Fuchs, Jakob Bach, and Markus Iser. 2023. Active Learning for SAT Solver Benchmarking. In *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 407–425.
[9] Iván Garzón, Pablo Mesejo, and Jesús Giráldez-Cru. 2022. On the Performance of Deep Generative Models of Realistic SAT Instances. In *Proc. Int. Conf. Theory and Applications of Satisfiability Testing (SAT 2022)*, Vol. 236. 3:1–3:19.
[10] Zijie Geng, Xijun Li, Jie Wang, Xiao Li, Yongdong Zhang, and Feng Wu. 2023. A Deep Instance Generative Framework for MILP Solvers Under Limited Data Availability. In *Proc. Adv. Neural Inf. Process. Syst.*
[11] Jesus Giraldez-Cru and Jordi Levy. 2017. Locality in Random SAT Instances. In *Proc Int. Joint Conf. on Artificial Intelligence (IJCAI).* 638–644.

[12] Evguenii Goldberg, Mukul R. Prasad, and Robert King Brayton. 2001. Using SAT for combinational equivalence checking. In *Proc. Conf. Design, Automation and Test in Europe*.
[13] Payam Habiby, Sebastian Huhn, and Rolf Drechsler. 2021. Optimization-based Test Scheduling for IEEE 1687 Multi-Power Domain Networks Using Boolean Satisfiability. In *Proc. Int. Conf. Design & Technology of Integrated Systems in Nanoscale Era (DTIS)*. 1–4.
[14] Serdar Kadioglu, Yuri Malitsky, Meinolf Sellmann, and Kevin Tierney. 2010. ISAC –Instance-Specific Algorithm Configuration. In *Proc European Conf. Artificial Intell.* 751–756.
[15] Ashiqur R. KhudaBukhsh, Lin Xu, Holger H. Hoos, and Kevin Leyton-Brown. 2009. SATenstein: Automatically Building Local Search SAT Solvers From Components. In *Proc. Int. Joint Conf. Artificial Intell.* 517–524.
[16] Yang Li, Xinyan Chen, Wenxuan Guo, Xijun Li, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, and Junchi Yan. 2023. HardSATGEN: Understanding the Difficulty of Hard SAT Formula Generation and A Strong Structure-Hardness-Aware Baseline. In *Proc. of ACM SIGKDD Conf. Knowledge Discovery and Data Mining*.
[17] Athilinagam Ramamoorthy and P. Jayagowri. 2023. The state-of-the-art Boolean Satisfiability based cryptanalysis. *Materials Today: Proceedings* 80 (2023), 2539–2545.
[18] Kyle Richardson and Ashish Sabharwal. 2021. Pushing the Limits of Rule Reasoning in Transformers through Natural Language Satisfiability. In *Proc. AAAI Conf. Artificial Intelligence*.
[19] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding High-Performance SAT Solvers with Unsat-Core Predictions. In *Proc. Int Conf. Theory and Applications of Satisfiability Testing*. 336–353.
[20] Grigori Tseitin. 1966. On the complexity of derivation in propositional calculus. In *Leningrad Seminar Mathematical Logic*.
[21] Weihuang Wen and Tianshu Yu. 2023. W2SAT: Learning to generate SAT instances from Weighted Literal Incidence Graphs. arXiv:2302.00272 [cs.LG]
[22] Nathan Wetzler, Marijn J.H. Heule, and Warren A. Hunt. 2014. DRAT-trim: Efficient Checking and Trimming Using Expressive Clausal Proofs. In *Theory and Applications of Satisfiability Testing*. Springer, 422–429.
[23] Haoze Wu and Raghuram Ramanujan. 2019. Learning to generate industrial sat instances. In *Proc. Int. Symp. Combinatorial Search*, Vol. 10. 206–207.
[24] Xi Ye, Qiaochu Chen, Isil Dillig, and Greg Durrett. 2023. SatLM: Satisfiability-Aided Language Models Using Declarative Prompting. arXiv:2305.09656 [cs.CL]
[25] Jiaxuan You, Haoze Wu, Clark Barrett, Raghuram Ramanujan, and Jure Leskovec. 2019. G2SAT: Learning to Generate SAT Formulas. In *Proc. Adv. Neural Inf. Process. Syst.*

A APPENDIX

A.1 Data

	var.	clause	runtimes	count
LEC	1328	5167	388	78730
Tseitin	86	81	603	9982

Table 1: Data Statistics. Note that LEC is a much larger dataset than Tseitin in every regard: average variable and clause counts, average hardness on Kissat solver and dataset size.

Tseitin Generations. We generate the Tseitin dataset by generating random graphs with between 40 and 60 nodes. We draw edges between nodes with probability

$$(-0.004(n-40)+0.11)(Uniform(0.95,1.05),$$

where n is the number of nodes. These values and this relationship was found by trial and error with the goal of producing mostly trivially easy or moderately difficult problems. The reason for this was that we generate approximately 10-20 times more data than we need since most are filtered for either being trivially easy or too hard. Generating too-hard instances is high cost because it takes the full timeout duration to discover that they are too hard, whereas trivially easy problems are nearly free to discard. We filter problems where the sum of runtimes across the 7 solvers is less than 10s and where all solvers time out, where timeout is 1500s. We then use the Tseitin transformation on the random graph to obtain an UNSAT instance.

A.2 HardCore GNN Core Prediction Implementation Details

We implement HardCore in `DGL` using 3 Graph Convolutional Network layers combined into a hetero-GNN, where outputs of each layer are aggregated with a mean using the `hererograph` package in `DGL`. We train using 15 problems from the dataset, and we obtain training cnf-core pairs using Drat-Trim in the Core Refinement step for 200 iterations per instance. We train for 1 epoch using Binary Cross Entropy loss.

For random generation in our method, we use Popularity-Similarity [11] with hyper parameters `-k 3 -b 0.5 -B 1 -T 1.3`.

B SUPPLEMENTARY RESULTS

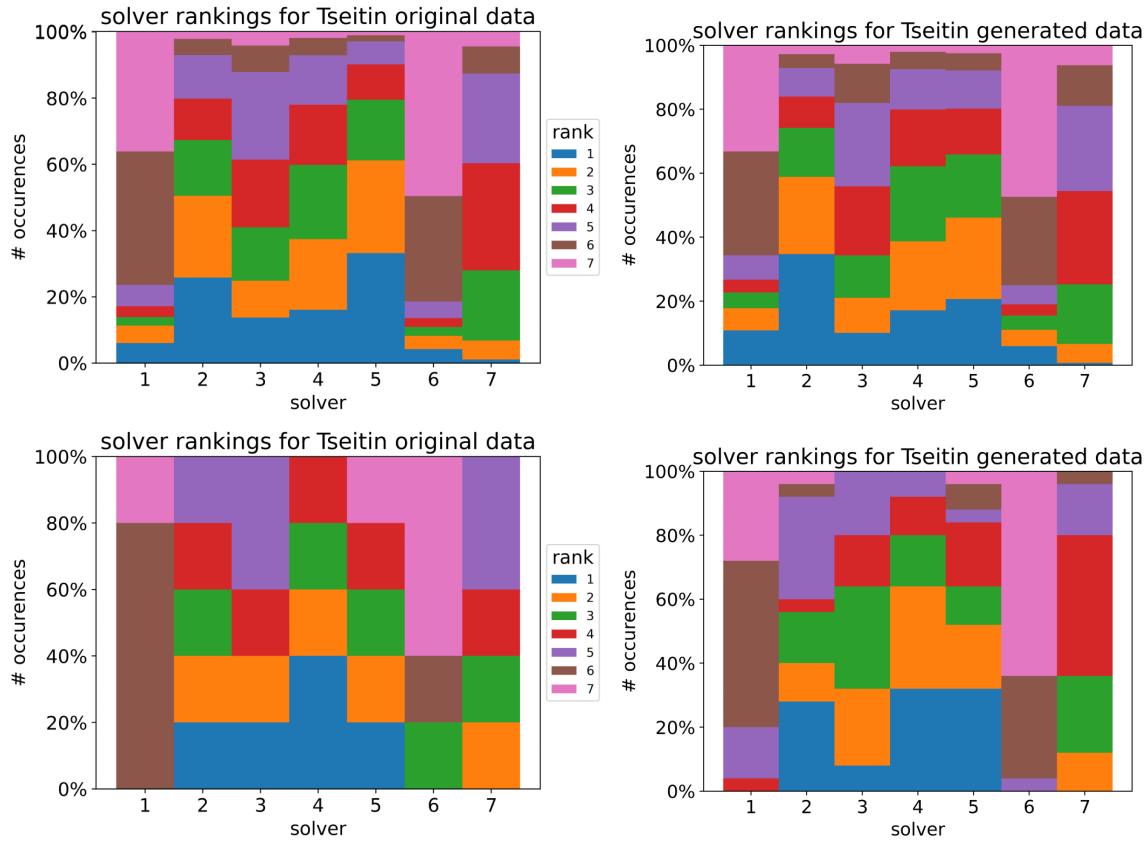

Figure 6: HardCore (top) and HardSATGEN (bottom) Comparison of Solver Ranking Histograms for Original and Generated Tseitin data.