Generating Hard and Realistic Boolean (Un)Satisfiability Problems.

Joseph Cotnareanu

Department of Electrical & Computer Engineering

McGill University

Montréal, Québec, Canada

August, 2024

A thesis submitted to McGill University in partial fulfillment of the requirements of the degree of Master of Science.

Abstract

Efficiently determining the satisfiability of a boolean equation — the SAT problem for brevity — is crucial in industrial problems. Recently, the advent of deep learning methods has introduced potential for enhancing SAT solving. However, a 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 are exploring generative techniques to create data that more accurately reflect 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 work we address both by identifying and manipulating the key contributors to a problem's "hardness", known as cores. Although some previous work has addressed cores, the time costs are unacceptably high due to the expense of traditional heuristic core detection techniques. We introduce a fast core detection procedure that uses a graph neural network. 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 show via experiment that the generated synthetic SAT problems can be used in a data augmentation setting to provide improved prediction of solver runtimes.

Abrégé

Déterminer efficacement la satisfaisabilité d'une équation booléenne — le problème SAT en bref — est très important en industrie. Récemment, l'agrandissement du domaine de l'apprentissage profond ("deep learning") a introduit une nouvelle famille de méthodes pour adresser le problème SAT. Cependant, cette famille de méthodes requiert beaucoup de données, lesquelles sont actuellement manquantes. Les données accessibles au publique sont soit groupées en familles très petites ou sont synthétiques, générées de façon stochastique. En réponse à ce manque, les chercheurs explorent comment générer des problèmes SAT qui sont plus représentatifs de problèmes industriels en volume utilisable par les méthodes d'apprentissage profond. Ces méthodes génératives existantes sont soit incapables de générer des problèmes difficiles, ou elles sont trop coûteuses en temps et en ressource computationelle pour être applicables. Dans cette thèse, nous adressons ces deux incapacités des méthodes précédents en identifiant et manipulant le sous-ensemble responsable pour la difficulté d'une problème SAT. La méthode présentée ci-dessous n'est pas la première à adresser ces sous-ensembles, mais elle est la première à le faire à un prix informatique acceptable pour générer un haut-volume. Cet objectif est atteint en remplaçant les méthodes traditionnelles par un nouveau réseau de neurones graphique pour identifier les sous-ensembles difficiles. L'expérimentation présentée dans cette thèse démontre que nous réussissons à générer des problèmes difficiles qui

ressemblent aux problèmes industrielles. De plus, il est démontré que les problèmes générés peuvent être utilisés afin d'augmenter les données d'entraînement des réseaux de neurones qui résolvent le problème SAT, ce qui améliore leur performance.

Acknowledgements

Here I would like to express my gratitude to all those who have supported me through this Master's degree and the writing of this work. Firstly, I would like to thank my supervisor Professor Mark Coates, who's patience and careful guidance have been indispensable. Even when I thought there was little to discuss beforehand, our meetings proved to be incredibly valuable for me as I always left knowing more and feeling more sure of the path ahead. As his student, I was afforded multiple opportunities to build new connections and explore new projects. I would also like to thank my co-authors on the paper to which this thesis corresponds, Zhanguang (Vincent) Zhang and Yingxue Zhang. Vincent's guidance, suggestions and brainstorming served greatly in the development of this work. Early collaboration with Vincent on a related project also allowed me to become familiar with — and develop an intuition for — the area. Yingxue's insightful feedback and suggestions guided experimentation in a very important way, and I always looked forward to our weekly meetings where I might get some valuable criticism to act upon. I would like to thank Muberra Ozmen (PhD), who allowed me to participate in her research project early during my Master's. This was incredibly valuable to me as it allowed me to become familiar with the research process under close supervision. Finally, I would like to thank my family. It is by the strength of their support that I have completed this work.

Contents

List of Figures

List of Tables

Chapter 1

Introduction

1.1 Motivation and Context

Machine Learning and the SAT Problem The boolean satisfiability problem (the SAT problem) emerges in multiple industrial settings such as circuit design [1], cryptoanalysis [2], and scheduling [3]. While machine learning is not well suited for directly solving SAT problems — solvers are typically required to have perfect accuracy and return correct proofs — there are many ways in which machine learning is currently used in tangent with the SAT problem. Given machine-learning's significantly lower time-cost relative to classical SAT techniques and the rich literature of statistical and graphical metrics for SAT instances (useful as features for learning), the SAT problem can be a fertile area for the application of learning techniques.

One of the most prominent examples of successful application of machine learning is SATzilla, proposed by Xu et al. [4]. This algorithm trains solve-time estimators for a hand-crafted

selection of SAT solvers, and for each new SAT problem selects the estimated fastest solver. This method was insightful because it took advantage of the observation that the competitive individual SAT solvers tend to perform well on mostly non-overlapping subsets of the competition benchmark problems. By choosing the best solver per-problem — rather than averaged over all problems in the benchmark set — SATzilla significantly reduced the average solve-time. This work led the way for a diverse series of subsequent algorithm selection techniques, ranging from clustering-related approaches [5, 6], to hyper-parameter selection [7]. The SATzilla work also introduced an important set of hand-crafted, easily-computed features for machine-learning on SAT problems, which has been widely used and augmented over time.

Another fruitful application of machine learning within SAT is accelerated benchmarking. Fuchs et al. [8] present an active-learning-based method that can provide an accurate ranking of SAT solvers' benchmark times at a fraction of the cost of running the full benchmark. In short, this method selects a sample of instances from the full benchmark and evaluates the solving times for the solvers on only the subset. The active learning procedure ensures that the selected subset is suitably representative.

Data driven methods require data A major challenge for SAT-related learning is the scarcity of high quality, reasonably homogeneous, real-structured data. The most commonly-used datasets have been compiled via a series of annual International SAT Competitions

[9]. The industrial origins of the compiled instances differ substantially, so the datasets are highly heterogeneous. The datasets provide a good, diverse 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 during training [10]. 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 [11]. The clearest demonstration of the difference between real data and random data, in our view, is the notable difference in solver performance. It is generally accepted in the literature that some solvers are random-specialized and others are industry-specialized. For example, the experimental results in [12] are heavily based on this. One can also see such specialization by noting that the best solver on the SAT Competition random tracks is not the same as the best solver on the SAT Competition main track. In fact, the existence of separate tracks itself indicates an acknowledged distinction between the types of data.

Given this distinction between the behaviour of random and industrial data, training an industrial-facing model on random data leads to an out-of-distribution problem. Inference must be performed for problems that are very different from the training data. It would be equivalent to augmenting a real-world image dataset with random patterns. In most cases, the model will perform poorly. For example, SAT-solver selection is a machine-learning task

for SAT in which, given a problem, we aim to rapidly choose the solver which is likely to solve the problem the fastest [4, 13]. This is a task which greatly benefits industrial settings in which thousands of SAT problems must be solved each day and computation and time costs must be minimized. Training such a selection model on random data would result in the model learning a wholly different runtime distribution than that of the industrial data. A similar example is low-cost benchmarking, in which a model is trained to predict the performance of a solver over a benchmark [8]. This task is of interest to solver-designers, as it can be much cheaper than running a design-iteration of a new solver on the whole benchmark data. If random data is used to learn to benchmark a solver on industrial problems, the predictor would likely predict benchmarks as if the industrial data were in fact random. Thus, new methods must be designed in order to obtain more realistic (less random) SAT instances.

Recently, deep-learning methods have been introduced to generate more realistic SAT instances. Early models [14–16] can generate instances that seem structurally similar to original instances, but the problems are far easier to solve, a phenomenon called hardness collapse. Preserving hardness is essential, as generating only very easy problems renders the resultant dataset ineffective for distinguishing the best-performing solver from the worst. Additionally, such datasets fail to help the model learn to predict real runtimes. A recent study has succeeded in preserving hardness [17]. Unfortunately, the resultant method is

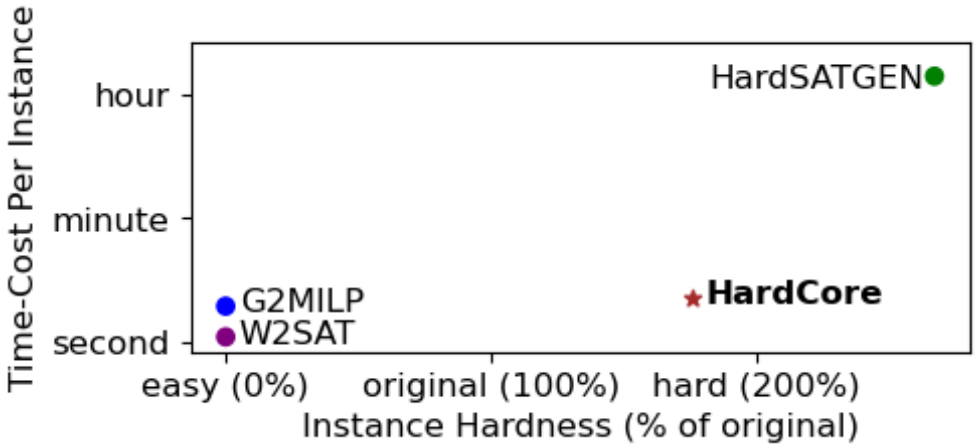

Figure 1.1: Generated Hardness (relative to original) vs. Time-Cost trade-off: previous works either provide methods that generate low-hardness problems or operate at extremely high cost. Our method generates hard problems with relatively low computational cost.

prohibitively computationally expensive for synthetic data generation and augmentation for deep-learning. It can take over a week to generate a limited number of new problem instances. We summarize the cost/hardness trade-offs in Figure 1.1. In this thesis, we aim to provide a method for the low-cost generation of hard SAT problems for the purpose of data augmentation in SAT-related applications of Deep Learning techniques.

1.2 Contributions

In this thesis, we take advantage of the connection between a problem's *core* and its hardness. The core consists of the identifiable minimal subsets of a boolean SAT problem that are unsatisfiable (UNSAT) [18]. 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 enhance learning. To do this, we need to detect the core after each iteration. Unfortunately, traditional core detection algorithms are slow and can take hundreds of seconds, as they often require solving the SAT problem [19]. Clearly, such an algorithm is impractical for building a fast generator, as core detection needs to be performed 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 circumvent 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. This hardness preservation is crucial when an augmented dataset is used to learn to predict solver times, a vital task for solver benchmarking and selection.

- We illustrate the value of our augmentation process for solver runtime prediction. On an example dataset, our augmentation process reduces mean absolute error by 20-50 percent. In contrast, all other generation algorithms achieve no statistically significant improvement.

1.3 Thesis Organization

This thesis is organized into several chapters, each describing an element of our investigation of learned data generation for SAT-related machine learning. The organization is as follows:

- *Chapter 2 - Background and Literature Review*

  In this chapter we summarize some key background theory related to the SAT problem. Specifically, we describe logical, set-based and graph-based representations of the SAT problem. We also describe the important notions of Cores and Hardness. We then outline the previous work in deep-generated SAT problems as well as Core Prediction techniques.

- *Chapter 3 - Methodology*

  In this chapter we describe our chosen problem — learned data generation for data augmentation in SAT-based learning problems. We describe our method HardCore, which combines low-cost random generation and a carefully designed learned refinement process. It can produce suitably hard problems at an acceptable computational cost.

  We then design a comprehensive set of experiments to measure the generative performance. We focus on assessing whether the generated instances exhibit similar hardness (in a distributional sense) to the original data.

  Finally, we demonstrate HardCore's ability to augment data in scarce data settings to improve predictive performance. To do this, we compare the performance of a SATzilla-based runtime estimator trained on augmented and un-augmented datasets.

- *Chapter 4 - Conclusion*

  In this chapter, we summarize the major contributions of each chapter, drawing a single clear image of our work from the contents of the thesis: our work constitutes the first SAT problem generator of its kind which can generate instances which are both hard and low-cost and which can be used as data augmentation during training to improve the performance of machine learning in SAT-related settings. We discuss the strengths

and weaknesses of the proposed approach, and identify potential directions for further exploration and improvement.

- *Author Contributions*

  The work in this thesis was conducted in collaboration with Yingxue Zhang, Vincent Zhang, and Mark Coates, who acted as supervisors, and provided significant guidance, particularly during the experimental exploration phase of the project. I conducted a thorough review of the SAT problem generation literature, identified the specific research problem, and proposed and implemented the generative method described in this thesis. I also was primarily responsible for designing and carrying out the experimentation, with the valuable guidance and feedback of my co-authors.

Chapter 2

Background Material and Literature Review

2.1 Introduction

In this chapter, we provide some background material related to the boolean satisfiability (SAT) problem and its representations, notations and measurements. Then, we discuss previous work in the sub-problems of SAT in which we operate: deep-learned generation and core detection. Finally, we discuss classification for graph problems. This will be important as our method makes use of graph classification techniques (see section 3.

2.2 Background

Definitions and Notation The Boolean Satisfiability Problem (SAT) is the problem of determining whether there exists an assignment of variable values that satisfies the given Boolean formula, rendering it true. Typically, a SAT instance is represented in Conjunctive

Normal Form (CNF), which 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 [20, 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 $c_i = \bigcup_{j=1}^{n_{l_i}} l_j$, $f = \bigcup_{i=1}^{n_c} c_i$.

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 [20, Chapter 11].

Graph Representation of CNFs There are several common CNF graph representations. In this work, we use the Literal-Clause Graph (LCG), an undirected and bipartite graph. Each node in the first set of nodes represents a clause and each node in the second represents a literal. We construct an edge for each occurrence of a literal in a clause; the set of undirected edges e is defined as $e = \bigcup_{i=1}^{n_c} \bigcup_{j=0}^{n_{l_i}} (l_{j_{c_i}}, c_i)$.

Another common graph representation of CNFs is the Literal-Incidence graph (LIG). This is a homogeneous graph representation in which all nodes represent literals, and edges are drawn when two literals share a clause. Thus the set of edges can be denoted as $e = \bigcup_{i=1}^{n_c} \bigcup_{j=0}^{n_{l_i}} \bigcup_{k=0}^{n_{l_i}} (l_{jc_i}, l_{kc_i})$.

These representations each have a major strength and weakness, and one's strength is the other's weakness. The LCG is a lossless representation of the CNF in that we are guaranteed to be able to recover the CNF that is represented by the LCG, simply by building clauses based on clause-node neighbourhoods. This is clearly a strength. However, this comes at the cost of the bipartite nature of the graph. Bipartite graphs are inherently more complex in structure given the node-type factor, and methods which leverage bipartite graphs structure require additional design considerations. To focus on methods which are designed specifically for bipartite graphs will greatly limit the selection of available off-the-shelf methods compared to homogeneous graphs. The LIG is exactly the inverse. Given only the LIG graph, there are no guarantees that the recovered CNF is the original. For example, given a triangle over literals A, B, C, one may draw a single clause $(A \vee B \vee C)$ or several small clauses $(A \vee B) \wedge (A \vee C) \wedge (B \vee C)$. This comes with the trade-off that the graph representation is simpler and more straightforward to manipulate.

Hardness Throughout this thesis, we frequently discuss the "hardness" of a problem. There is not a unique, universally-accepted mathematical definition of hardness. The hardness of a

SAT problem is some measure of the cost of solving the instance. A variety of measures of hardness have been introduced in the literature. Wu and Ramanujan [14] use the number of conflicts to measure hardness, where a conflict is an event during solving in which the current hypothetical solution is invalid. One might consider conflicts to be analogous to the number of failed attempts during solving. Xu et al. [4] implement a model which is referred to as an empirical hardness model. The goal of this model is to predict the hardness a solver will encounter when solving a problem. This model is used to select the solver which will encounter the least hardness. Practically speaking, this amounts to picking the fastest solver per instance using a per-instance runtime-prediction model, trained for some hand-selected SAT solvers. Ansótegui et al. [21] present a method for measuring hardness which involves the tree-size of the proof-tree of the problem. While interesting, this method is more complicated than necessary for most use-cases of a hardness metric. In addition, random problems are typically hard in a different way than industrial instances. This can be seen by the fact that solver solve-time ranking orderings are different when measured over random or industrial instances [9]. Thus, when hardness is of interest, we also consider solver speed rankings [15, 17, 22].

2.2.1 Graph Neural Networks (GNNs)

While not the original introduction of graph neural networks (GNNs), Hamilton et al. [23] introduce a general framework under which we might understand GNNs broadly:

$$h_v^k = \sigma(\mathbf{W} \cdot \texttt{aggregator}(h_u^{k-1}, \forall u \in \mathcal{N}(v)), \tag{2.1}$$

where σ is the sigmoid function: $\sigma(x) = \frac{1}{1+e^(-x)}$ and `aggregator` is some aggregator function. The aggregator should be designed such that its output dimension is invariant to the size of the neighborhood $\mathcal{N}(v)$. Typically, the output is also permutation invariant (for example, `MEAN, MAX, SUM`). This framework boils down to each layer of a GNN being composed of a single multi-layer perceptron (MLP) layer, executed for each node embedding on the aggregation of the node's neighborhood. This leads to each node embedding being a representation of that node's neighborhood, and therefore such GNNs can be seen as embeddors of local structure in graphs.

2.2.2 Training Binary Classifier Neural Networks

The foremost training strategy used for training binary classifier neural networks is the minimization of the classifier and target distribution's binary cross-entropy:

$$\text{BCE}(p, q) = -\sum p(x) \log q(x), \tag{2.2}$$

where p and q are the target and model prediction distributions respectively. In cases where the target distribution is not known, as is commonly the case in machine-learning where a distribution is not given but rather samples (data), we estimate the cross-entropy accordingly:

$$\mathrm{BCE}(T, q) = -\sum_{i}^{N} \frac{1}{N} \log q(x_i)\,, \tag{2.3}$$

where T is the dataset, N is the size of the dataset, and $q(x_i)$ is the probability that the model will predict the correct answer for datapoint x_i. h^0 is set as the input features for each node, and is dependent on the setting.

2.3 Related Work

In this section, we outline several categories of work relating to this thesis. The first two categories pertain directly to SAT problem generation: random generation and deep-learned generation. Random methods tend to examine hardness and industrial structure of problems according to specific statistical measurements and can contribute insight into what makes a problem hard or industrial, and how we might measure it. Deep-learned generation methods form the foundation of the generation technique that we propose in this thesis. In the approach we develop, core prediction/detection is also key, and so we also review related work addressing this task. Finally, our work is heavily motivated by the general application

of machine learning for the boolean satisfiability problem in various settings, so we present some examples of previous work in various SAT-related settings to illustrate the potential benefits machine learning methods may bring to addressing the SAT problem.

2.3.1 Random SAT generation

There exist two random SAT generation algorithms which attempt to generate instances that are similar to industrial SAT instances. Both methods target preservation of specific features that are commonly found in industrial problems.

The first method is named Community Attachment (CA), proposed by Cru and Levy [24] in 2016. The major claim of the paper is that the modularity of the problems, when represented as graphs, as described in Section 2.2, is linked to the common structure of industrial problems, and that by generating problems of similar modularity, one might generate industrially structured problems.

Community Attachment operates by initializing a set $\mathcal{N}$ of n boolean variables. Variables from the set $\mathcal{N}$ are selected to form clauses according to the following scheme. First, the set of variables is randomly partitioned into c pairwise disjoint communities of the same size. Then, with probability p a clause is formed by randomly selecting k variables from a single community, and with probability $1 - p$ a clause is formed by randomly selecting k variables

from all communities. With this method, it can be shown that the average modularity of the graphs of generated problems has a lower bound, which is dependent on the selected hyper-parameters, p and c. It is shown empirically that industrially-specialized solvers tend to perform better on the generated data than random-specialized solvers, indicating that at least an element of the structure of industrial problems, which is leveraged by specialized solvers, is also present in the generated problems. This is supported by experimental results showing that industrially-specialized solvers tend to solve the generated problems in a community-observant way: variables within a single community are usually tested by the solver consecutively. This seems to support the hypothesis that industrially-specialized solvers leverage the community structure that can be observed in graphical representations of problems. However, this logic is somewhat strained, and lacking evidence of a causal connection between community structure and "industrial-ness", it is highly likely that modularity is only a small element or side-effect of the true underlying nature of industrial instances.

The next method is named Popularity-Similarity (PS) and was proposed in 2017 by the same authors, Cru and Levy [12]. Seemingly based on the previous work's findings that industrial solvers tend to visit variables in some local fashion, the target feature in this method is called "locality". Although locality is never precisely defined in the paper, it seems to refer to clustering and modularity. As the name of the method suggests, there is in fact a second targeted factor: popularity. Popularity, or power-law, or scale-free structure, is the

notion that networks often contain several "popular nodes", which have much greater degree relative to the rest of the graph. This means that the node-degree histogram of the network follows a heavy-tailed curve. Cru and Levy [12] propose a random generation model such that the graph representation of the SAT problem demonstrates scale-free structure. The method also enforces a power law over clause length. Industrial-specialized solvers are shown to out-perform random-specialized solvers on PS-generated problems, which the authors interpret as demonstrating the industrial structure of PS-generated problems.

More recently, a method has been proposed by Zhao et al. [25] for specifically generating hard satisfiable problem which have multiple solutions. This method operates by pre-defining the solutions for the to-be-generated problem and then randomly generating a corresponding problem. Then, hardness is directly measured by solving the problem and the problem is modified if necessary to be harder. Three strategies are offered. The first is to randomly flip literals in the randomly generated clauses until the problem meets some hardness threshold. The second is to initially generate problems which satisfy the chosen hardness criteria by expressing the satisfaction of the hardness criteria as a linear programming problem and solving it. The third is to greedily generate clauses which are each as hard as possible according to the hardness criteria. While this work provides significant insight into both iterative hardness-aware generation as well as the measurement of hardness itself, it applies very specifically to satisfiable problems which have multiple solutions. Also, while the method

does generate hard problems, these hard problems are not necessarily industrially-structured. For example, there is no demonstration (or stated intent) that industrially-specialized solvers perform better on the generated instances than random-specialized solvers.

Random SAT-generation methods such as these, which aim to generate hard or industrial-structured data, demonstrate some very significant advantages and disadvantages. These methods are low-cost relative to trained methods both in that they incur no training costs and that inference cost is generally low. These methods are also straightforward and understandable. However, these advantages come at the cost that the methods are highly dependent on multiple hyper-parameters, such as cluster-size and p for CA, and the power-law constant and temperature for PS. Little information is provided in the papers regarding how to set these parameters. Additionally, it is highly likely that industrial data from a specific application is different in structure than data from another application. While these works demonstrate empirically some element of general "industrial-ness" of the generated data, there is no demonstration of an ability to emulate specific example data.

2.3.2 Deep-learned SAT generation

The problem of learned generation for SAT problems was first established by Wu and Ramanujan in 2019 with SATGEN [14], motivated by a lack of access to industrial SAT problems. SATGEN uses a graph generative adversarial network (GAN), trained using

random walks on the LIG representation. Wu and Ramanujan [14] explain that the LIG was chosen because it was found that learning the bi-partiteness of the LCG was difficult, and that the training time on the chosen bipartite methods was too expensive. To address the LIG-to-CNF decoding obstacle discussed above, a lazy hill-climbing algorithm is used to find the minimal clique edge cover of the graph, where each cover forms one clause. This choice does not lead to recovery of the original CNF, but results in a CNF with the minimal number of clauses. While this work served to introduce the problem to the generative deep-learning community, the work fell short in its evaluation. Only one hardness metric was evaluated, and for this metric the method under-performed some random generation methods while operating at much higher computational cost. Only general graph statistics were provided to demonstrate the "industrial" nature of the generated instances.

G2SAT [15] represents problems as LCGs. The graphs are progressively split into small trees, and a graph neural network (GNN) is trained to discern whether two trees should be merged to restore the original graph. While innovative, the method is slow due to its need to sample many tree pairs to form a SAT problem of sufficient size. Additionally, the method is unable to generate hard instances, with generated instances that typically show nearly 0% solve-time compared to the original problems. This work introduced the solver ranking ordering experiment, which compares the ordering of ranked solvers on original and generated problems as a method of gauging similarity.

The most recent improvement on the G2SAT framework, HardSATGEN [17], includes some domain-inspired considerations in its design, such as graph communities and cores. HardSATGEN uses the same split-merge framework that was introduced in G2SAT, with some modifications and some additions. First, the core of the original problem is sustained and remains untouched throughout the generation pipeline. Then the split-merge algorithm is applied to nodes and trees which are within the same community. This process is repeated at an inter-community level. Finally, an algorithm is applied which iteratively renders the core of the generated problem satisfiable. This iterative process gradually increases the size of the core with the goal of obtaining a core that is harder to solve. This is an insightful approach based on the intuition that the core of an UNSAT problem is the proverbial weakest link in the chain, in that by solving only the core one may determine the problem to be UNSAT. HardSATGEN is the first deep-learned SAT generation method that can generate problems which are not trivial to solve for solvers: often the generated problems take 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 cause HardSATGEN to be extremely slow, making it challenging to use in any setting that needs many new instances.

W2SAT [22] follows an approach more similar to the original SATGEN, in that it uses LIG representations followed by a minimal clique edge cover algorithm to decode the generated LIG. A major distinction, however, is that a weighted LIG is used in which the edges of the

LIG are weighted according to the number of times the two literals share a clause. The method employs a low-cost general graph generation model, and obtains new SAT problems via graph decoding. W2SAT is extremely efficient, but like G2SAT, it is incapable of generating hard problems.

G2MILP [26] is designed to generate Mixed Integer Linear Programs (MILPs), which are the general case of SAT. A 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.

2.3.3 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 [21]. There are multiple classical, verifiable methods for Core Detection.

The most straightforward method for determining the core of a problem is by removing clauses one at a time and checking if the resulting problem remains UNSAT. If the problem does indeed remain UNSAT then the removed clause is not a part of the core. If the problem becomes SAT, then the clause is integral to the unsatisfiability of the problem and is therefore part of the core, and must be returned to the problem. When there are no more clauses which

can be removed without rendering the problem satisfiable, what is left is the core. While this method is simple to implement, it requires n_c calls to a SAT solver. Modern SAT solvers may often take minutes to solve an instance, and if a small industrial SAT problem has $n_c = 1000$ then this means that this core detection algorithm will take hours to complete [27].

A more recent advance in Core Detection is the Drat-Trim algorithm [19]. Drat-Trim requires that the problem be solved once by a SAT solver, and leverages the proof that is provided by all contemporary SAT solvers to discover the core. While significantly faster than the previously described method (Drat-Trim requires 1 call to a SAT solver whereas the previous requires thousands), SAT solvers are still considered slow (the underlying problem is NP-Complete), and so even a single call means the algorithm is slow. In response to this, Neurocore [10] 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.

2.3.4 Various Machine-Learning for SAT methods and settings

Algorithm Selection

One of the original applications of Machine-Learning for SAT was in algorithm selection: the task of selecting the fastest solver for a specific SAT problem. The motivation of this problem is that it is generally accepted that different solvers exploit different types of SAT structure. Most notably, random structured SAT problems and industrial-structured problems show very distinct performance on the same solvers. Thus, in theory (and demonstrably), selecting the best solver per-instance should significantly reduce over-all time-cost compared to selecting the best on-average solver. The first method of note to employ machine-learning for this task is the SATzilla entry to the 2007 SAT competition from authors Xu et al. [4]. The method leveraged a small machine-learning model trained on a set of hand-chosen features specific to SAT problems such as the number of variables, clauses, and clause size to predict the runtimes of a selection of competitive and complementary already-established solvers for each problem. With the help of some well-made design choices, SATzilla earned medals at the SAT competitions for several years after its introduction. Following SATzilla, several algorithm selection methods were released, most notably clustering methods ISAC and 3S [5, 6], hyper-parameter selection methods SATenstein and Hydra [7, 28] and graph-based method GraSS [13].

Low-cost SAT-Solver benchmarking

A crucial phase in the iterative design cycle of many types of algorithms is benchmarking, where the designer tests the newest version of the developing algorithm against a series of baselines and previous versions of the algorithm. In SAT solver design this phase can be costly, as each SAT problem might take minutes to solve and parallelization can be difficult due to the CPU-requiring (rather than GPU) sequential nature of SAT-solving. To address this, Fuchs et al. [8] put forward a method based on an active-learning framework for obtaining a highly accurate benchmarking of solvers with only a small fraction of the required test problems, lowering benchmarking costs considerably.

Machine Learning for Solving SAT problems

One can also find in the literature machine-learning based methods that either directly augment solvers or replace solvers entirely. Selsam et al. [29] present a graph neural network (GNN) based method for predicting the satisfiability of a problem represented as a graph. While results appear promising in the paper, the empirical results are on simple and very small random SAT problems. The authors comment that performance degrades notably as the size of the problem is increased. Practically, machine-learning seems poorly positioned to replace logical solvers entirely for SAT solving because machine-learning models are inherently approximate and can rarely be relied upon to be correct every time. The traditional SAT setting — for example at SAT competition — is not to measure accuracy of solvers, however,

but to measure their speed. Industrially, SAT solvers are employed as the final validation step because solvers will never incorrectly solve a SAT problem, and so machine-learning approaches are unlikely to ever be accepted as a total replacement of traditional solvers.

Despite this, some work has shown potential in augmenting SAT solvers with learned heuristics. SAT solvers operate by selecting a variable from the problem, assigning it a value, and checking if the assigned value satisfies the problem. Traditionally the variable is picked according to some heuristic, but Selsam and Bjørner [10] present a learned heuristic in which a GNN is trained to predict variables based on their association with the minimal unsatisfiable subset of the problem. This is motivated by the notion that finding the core of the problem is the fastest way to prove a problem as UNSAT. The design-choice of a core-based heuristic seems strange, as cores are defined as sets of clauses whereas the heuristic must select variables. Despite this, results show significant improvements in solve-time when replacing a chosen solver's heuristic with the trained GNN, even though the GNN's inference time is slower than the heuristic's computation time.

2.4 Summary

In this chapter we discussed the set-wise and graph-wise representations of SAT problems and their trade-offs. We reviewed some key attributes of SAT problems, most importantly their cores and their hardness (and measurements thereof). We then presented the foremost

work in the fields of deep-learned graph-based problem generation and core detection, noting that up until very recently both fields have been relatively unsuccessful: most generative methods are high-cost and/or generate problems that are trivial to solve. Until recently, core detection algorithms required thousands of calls to a solver.

Chapter 3

Methodology

3.1 Introduction

In this chapter, we define the problem of augmenting datasets of industrial SAT problems for Deep-Learning applications and present our approach to the problem, which is a novel multi-step generation procedure that leverages the graph structure of SAT problems. We also demonstrate empirically the method's ability to generate realistic synthetic data (which can be added to existing datasets in order to augment them). Our experiments indicate that the data augmentation for the common Empirical Hardness prediction (solve-time prediction) problem [4] results in meaningful performance improvements.

3.2 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 SAT solving 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. In this case, the appropriate loss function is the absolute error between the predicted time and the true time.

Beyond this, we are also interested in the distance between the distributions $\mathcal{D}$ and $\mathcal{D}_G$. We examine this through the lens of 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 similarity of the original and generated instances, we calculate the empirical maximum mean discrepancy (MMD) distance between these induced distributions.

3.3 HardCore

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 3.1 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.

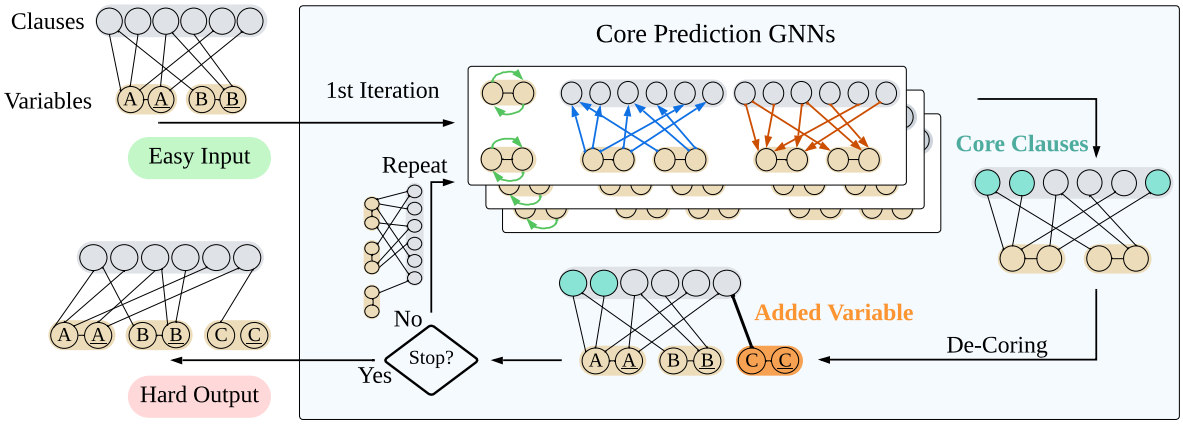

Figure 3.1: **Core Refinement**. The core refinement process comes in two steps: (1) Core Prediction, in which we use a GNN-based architecture to identify the core of the generated instance; and (2) De-Coring, in which we add a non-conflicted literal to a clause in the core, rendering the core satisfiable and giving rise to a new, larger minimal unsatisfiable subset (core). As steps (1) and (2) are repeated, the core gradually becomes harder (as will be shown in results further on.)

3.3.1 Generating Hard Instances

Trivial Cores 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 an irreducible subset of clauses of a SAT problem that is UNSAT. Small cores with few clauses are generally easier to solve due to less variable assignment combinations. An example of a trivial core is $(A \vee B) \wedge (\neg A \vee B) \wedge (A \vee \neg B) \wedge (\neg A \vee \neg B)$.

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 includes three of the 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 by a trivial one, leading to hardness collapse. Maintaining awareness of cores and potential cores in a CNF as we perform modifications is 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 3.1. The first step of the process is to identify the core of the generated instance. The addition of random new clauses during generation is very likely to create a core that is trivially easy to

solve and it may not be the same as the core of the original instance. Once we have detected this easy core, we make 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 $(A = 0, B = 0, C = 1)$ is now a satisfying solution.

As these two steps are repeated, the core of the instance gradually becomes harder. 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 [21], the refinement process can be seen as progressively raising the hardness of the problem. After the final iteration, we expect to have a hard problem which we might use for training models.

Underlying Hard Core Guarantee The Core Refinement process is designed to repeatedly eliminate easy cores, so after each iteration, the core becomes harder. Finally, after many iterations, we hope that the remaining core is as hard as the original 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 must introduce an element to our design to ensure there is a hard core. To achieve this we identify cores from the original instances and include them in the generated instances.

3.3.2 Core Prediction

We have two critical objectives for our method: low 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. Core detection fundamentally requires the solving of a subset of a SAT problem (which is itself a SAT problem), therefore making the Core Detection problem NP-Complete [19].

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, we expect that a graph neural network is a promising approach.

Graph Construction We represent each instance as a graph as outlined in Section 2.2. We make two changes: (a) we add message-passing edges to connect 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 $\mathcal{E}_{lc} = \bigcup_{i=1}^{n_c}\bigcup_{j=0}^{n_{lc_i}}(l_{j_{c_i}}, c_i)$. We denote the set of clause-to-literal directed edges by $\mathcal{E}_{cl} = \bigcup_{i=1}^{n_c}\bigcup_{j=0}^{n_{lc_i}}(c_i, l_{j_{c_i}})$.

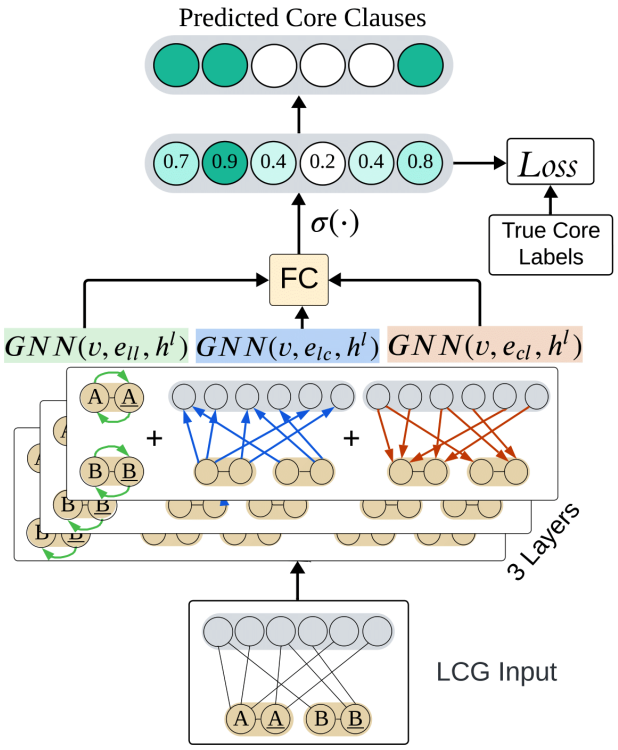

Figure 3.2: **Core Prediction GNN Architecture**. We construct our GNN using three parallel message passing neural networks (MPNN) whose calculated node embeddings are aggregated at each layer to form the layer's node embeddings. Readout is done by taking the sigmoid of a fully-connected layer on clause node embeddings and thresholding. Training is supervised by taking a binary classification loss between the true core labels and the clause nodes' core prediction probabilities.

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) as illustrated in Figure 3.2. We couple these models by averaging their embeddings after each layer. 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. A single layer of the GNN is defined as follows, where σ is a non-linear activation function:

$$h^{l+1} = \sigma(\frac{1}{3}(GNN(\mathcal{V}, \mathcal{E}_{cl}, h^l) + GNN(\mathcal{V}, \mathcal{E}_{lc}, h^l) + GNN(\mathcal{V}, \mathcal{E}_{ll}, h^l))) \,, \tag{3.1}$$

$$out = \mathbb{1}_{>0.5}(\sigma(x h_c^L + b)) \,. \tag{3.2}$$

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 the model using the standard binary cross-entropy loss function.

3.4 Experiments and Results

3.4.1 Experimental Setting

Proprietary Circuit Data (LEC Internal) This LEC Internal data is a set of UNSAT instances which are created and solved during the Logic Equivalence Checking (LEC) step of circuit design. LEC needs to be performed after certain circuit optimization steps to ensure that the optimization process has not corrupted the logic of the circuit. If the logic is uncorrupted, the created SAT problem will be UNSAT. Since it is extremely rare that these optimizations in fact corrupt the circuit, more than 99% of LEC instances are UNSAT.

Synthetic Data (K-SAT Random) Acknowledging the importance of reproducibility, we also provide results on synthetic data. This data is generated by randomly sampling a CNF with m clauses of k literals over n variables. Clauses are sampled without replacement. We have previously argued that random instances differ from real instances in important ways that make them unsuitable for machine learning applied to real problems. Holding to this view, we use this data primarily to provide a surrogate to the internal data for experimental reproduction purposes, rather than to present results on a second dataset. For details concerning both the LEC Internal and K-SAT data, see Table A.1 in Appendix A.1.

Data Split and Training Details There are three **separate** groupings of the dataset: (i) Core Prediction training data, (ii) generation seeding data, and (iii) evaluation data for the runtime-prediction model trained in the data augmentation experiment. This split is chosen randomly. Core Prediction training data can be very small (we used 15 problems), because we use each problem as a seed instance 5 times and this is followed by core-refinement using a traditional core detection algorithm in which the core and problem pair are saved at each step. Each seed results in a separate input-problem to the refinement process due to the randomness in the random generation algorithm we employ. Given 200 refinement steps and the 5 generations per training problem, we obtain 15,000 problem-core pairs with which to train the core-detection GNN model. The generation seeding data are used to seed HardCore once the core predictor is trained in order to obtain generations to evaluate. These generations are then compared against the seed data for runtime similarity. Finally, these generations (and their seeds) are combined to form an augmented training dataset for a runtime-predictor model, which is evaluated on the evaluation data. The evaluation data consists of all the data which are not used for core prediction training or generation seeding. Each named dataset (LEC, k-SAT) is taken separately for each experiment.

The core prediction model is trained with a binary cross-entropy loss. We train on a single epoch with a learning-rate of 0.003 over the prepared training data, which is composed of problem-core pairs for 200 steps of the core-refinement process on 5 generations per training-

problem seed. Thus, from 15 dedicated training problems we obtain $15 \times 200 \times 5 = 15000$ training examples. Node embeddings in the model are initialized as 1's.

SAT Solvers We select 7 solvers for hardness analysis: Kissat3 [30], Bulky [31], UCB [32], ESA [32], MABGB [32], moss [32] and hywalk [33]. These solvers exhibit complementary performance characteristics: when some of these solvers perform well on certain instances, some perform very poorly. This results in diverse runtime distributions in our analysis. We run our experiments on an Intel® Xeon® Platinum 8276 CPU @ 2.20GHz cpu and 3 Nvidia Tesla V100 GPUs.

We compare to the following baselines:

- **HardSATGEN [17]**: A high-cost split-merge generator with community structure and core detection that is capable of generating hard instances at very high computational cost. Given this cost, we generate only 50 instances per dataset, using 10 seed instances and 5 generations per seed.

- **W2SAT [22]**: A low-cost generative method that utilizes a less common SAT graph representation. It tends to generate very easy problems. Given the low cost, we generate using every instance in the entire training dataset as a seed, with 5 generations per seed.

- **G2MILP** [**26**]: A low-cost VAE-based generative model designed for the general case of SAT: MILPs. Given the low cost, we generate using every instance in the entire training dataset as a seed, with 5 generations per seed.

A note on seed-generation correlation During our investigation of the method, we examined the correlation between seed hardness and generated hardness on a per-seed level. The goal of this examination was to check if the generation process constructs diverse samples by constructing an instance that is significantly different from the seed. Ideally, the generative mechanism produces diverse problem instances, but maintains a hardness distribution similar to the original data. Our finding was that the hardness of a generated instance is almost completely uncorrelated with the hardness of the associated seed instance for HardCore. In Figure 3.3, we present a scatter-plot showing a generation's runtime on the x-axis and that generation's corresponding seed runtime on the y-axis. There is no visible correlation in the form of a concentration along the diagonal (the dotted red line that can be seen in Figure 3.3). Upon attempting a least-squares linear regression to fit these points, we find a Pearson correlation score of 0.0297, supporting our conclusion that the generated and original runtimes are uncorrelated.

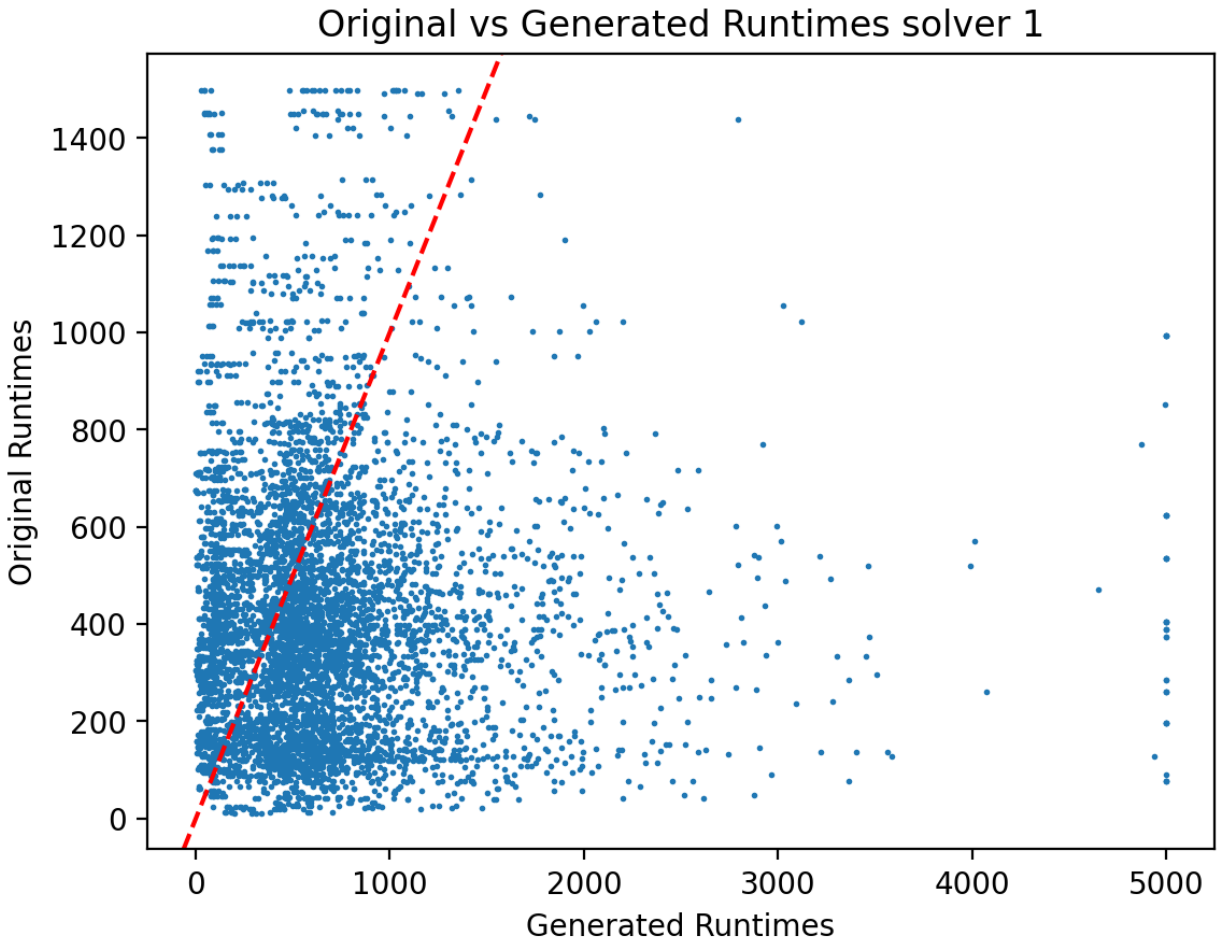

Figure 3.3: **Internal LEC** Scatter plot showing correlation of generated runtimes and corresponding original runtime. We note no visible correlation (concentration along the red-dashed diagonal).

3.4.2 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.

A note on datasets

In the following section for research questions 1-3, we focus on the Internal LEC data. This is because these experimental results are presented primarily to show that the hardness distribution can be maintained for industrial problems, making the random K-SAT data less meaningful. For research question 4, which investigates the value of data augmentation, we report results for both datasets.

Core Prediction Model Performance

Experimental validation of the Core Prediction model, reported in row "LEC" in Table 3.1, shows approximately 94% accuracy and more importantly, 96% recall (% of the core which is recovered by the predictor, also referred to as "Core Recovery" in the results below). We also report results on Core Size discrepancy, which is the difference in size of predicted and true cores. We report this because during experimentation we found that the model was prone to over-predicting cores when the training-data was made up of problems which had relatively large cores (e.g. core size is 90% of whole problem size)). Our priority is high recall because a clause that is incorrectly identified as part of the core (a false positive) can be removed during the core refinement process. On the other hand, the core refinement process cannot remove clauses that are not identified as part of the core (false negatives). Consider, for example, the

case where we have generated a problem containing an easy core that is made up of clauses from the original core (which cannot be modified for the HardCore guarantee to hold) and one non-original clause. If this single non-original clause is incorrectly classified as non-core (a false negative), the trivial core will go unrefined and the problem will remain trivial to solve. On the other hand, if we incorrectly include an additional clause, the refinement process can remove both clauses, resulting in a hard core. With this said, it is still important to maintain adequate accuracy and core-size discrepancy performance. For example, predicting the full problem as a core each time may lead to high recall if the majority of the clauses in a problem are core clauses. However, this would certainly result in the failure of our method as it would amount to randomly changing one clause each iteration.

We view 96% recall as a very promising result, and confirm this in further experiments showing our method's ability to augment datasets and generate faithfully hard problems. Table 3.1 summarizes the results of experiments that assess the GNN's core prediction performance.

Core Prediction Model Performance on Out-of-Distribution problems

In order to measure GNN generalization to new data without re-training, we create a new split of the LEC data. Each problem in the LEC data can be traced back to one of 29 circuits. By randomly splitting circuits into training circuits and test circuits (and then building

training and evaluation sets with their respective problems), we can measure generalizability. Note that we would not expect the model to generalize to problems derived from a completely different application domain (although fine-tuning a previously model in a domain adaptation strategy might be interesting to explore).

In Table 3.1 we report the GNN performance on this experiment. We discussed above that Core recovery is the priority, because if we falsely classify true-positives then we may be unable to de-core the current core (since the necessary clause may be undetected), whereas if we misclassify true-negatives then we will simply de-core a non-core clause. Given enough iterations of core-refinement, a true-positive clause will eventually be selected for de-coring (since the clause for de-coring is randomly selected from among the detected clauses). With this in mind, the threshold hyper-parameter which is used on the sigmoid outputs at model readout becomes a useful parameter in cases where classification performance is weakened: we can boost Core Recovery (recall) by lowering the threshold. Tuning this threshold is very low-cost: testing a thousand problems takes 500 seconds on a GPU. We find that by testing values $[0.1, 0.3, 0.5, 0.7, 0.9]$ — which takes 25 minutes — we can tune the threshold to provide similar recall to the in-distribution model, at the cost of accuracy and core-size difference.

Table 3.1: GNN Core Prediction Performance on In and Out-of-Distribution LEC data

	↑ Core Recovery Ratio (Recall)	↓ Core Size Diff.	↑ Acc.
Circuit-Split LEC	0.97	0.05	0.65
LEC	0.960	0.009	0.940

Hardness changes over the progression of Core Refinement and Core Size

For our proposed technique to work, our key assumption is that if a problem is easy then core prediction can identify the easiest core, and then by removing that core the problem is made harder. This does not necessarily mean that the core is larger. The experimental results demonstrate that we generate challenging problem instances, providing support that our assumption is correct.

First, we evaluate the hardness of the original problems (in terms of Kissat solver time) in the LEC dataset. Note that this is a real-world dataset derived during industrial circuit design. Figure 3.4 (left) shows that there is a general trend of the hardness increasing as the core size increases, up to a threshold of 4000-5000 clauses.

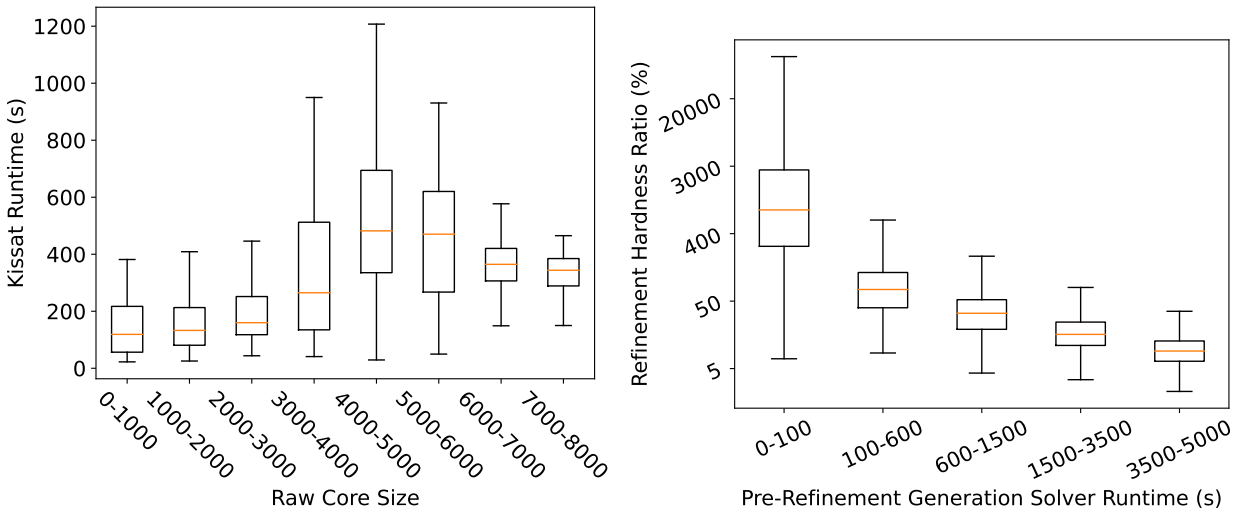

Figure 3.4: Left: Hardness for problems of varying core sizes. We note a positive correlation between core size and hardness until core size reaches approximately 5000 clauses in size. Right: hardness of core-refined problems expressed as a percentage of the same problem's hardness before refinement. We see that core refinement recovers the hardness for trivial (hardness-collapsed) problems.

As discussed above, although we observe this trend (correlation), it is not essential for our method. Much more important are the results shown in Figure 3.4 (right), where we show how the hardness changes as a result of our refinement process. The figure shows boxplots of the percentage change in hardness for different bins of initial hardness, and shows that for easy problems we increase the hardness sometimes by a factor of over 200. These large hardening factors for initially easy problems are indicative of successful recoveries of the hardness of previously hardness-collapsed problems.

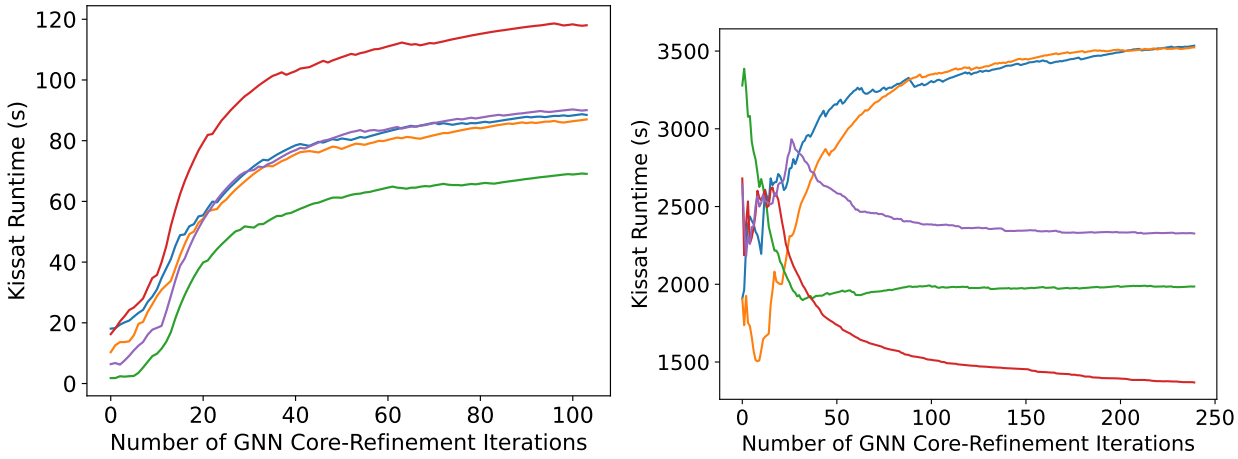

Figure 3.5: Hardness as Core Refinement Progresses. We run 5 Generations and Core-Refinements for one: Easy problem (left), Hard problem (right).

To further examine the effect of core refinement specifically on collapsed (easy) problems compared with hard problems, we present two examples in Figure 3.5. The plot on the left shows an example of 5 generations from one seed. Initially, after the addition of random clauses, all 5 generated problems are easy to solve (requiring under 20 seconds). The plot indicates a progressive hardening as refinement progresses, with a relatively rapid increase for the first 40 refinement steps, followed by a more gradual increase. After 100 iterations of refinement, the solve time has been increased by a factor of at least 4 for each generated problem.

The right plot in Figure 3.5 shows core refinement for a problem that is already difficult at the start of the process (solve time $>$ 1500 seconds). In this case, the core refinement process

increases the hardness of some of the problems and decreases the hardness of others. All of the problems remain hard after core refinement.

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 generated instance 'hard' if the average runtime of selected solvers on the instance is at minimum 80% of the average solver runtime for the original seed instance. An entire generated dataset is considered 'hard' if the average ratio (average solver time on generated instance to average solver time on seed instance) exceeds 80%. If the average ratio for the set of generated instances is below 5%, we consider that *hardness collapse* has occurred.

In Table 3.2 we compare generated with original hardness. W2SAT and G2MILP both suffer hardness collapse, whereas HardSATGEN and HardCore generate hard instances. While achieving hard problems has been a focus because of existing methods' tendency towards collapse, the primary goal is in fact to mimic the original problems' hardness rather than to achieve problems which are as hard as possible. For this reason, the desired hardness performance is 100%. HardSATGEN produces problems which are on average more than twice as hard as the original problems. While HardCore also produces problems that are too hard, it does so to a lesser degree than HardSATGEN (average ratio of 176% versus 267%).

With that being said, the primary purpose of this experiment is not to compare how close to 100% each method's hardness is, but to check which methods are even capable of generating hard instances. Therefore, the answer to the first research question is that only HardCore and HardSATGEN are able to generate hard instances.

Table 3.2: Evaluation of generated datasets on LEC data. Hardness level (%): runtimes of selected solvers averaged over all instances and solvers, as an average of the percentage of the original data's solver-wise average. Closer to 100% is better. Speed (s): average time cost to generate one instance, lower is better.

	W2SAT	HardSATGEN	G2MILP	HardCore
Hardness (%)	∼0	267	∼0	**176**
Time per instance (s)	**1.2**	6441	3.3	4.3

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. Generally, a method should be able to generate hundreds of instances per hour so that we can augment a dataset in a reasonable time frame.

In Table 3.2, 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. In contrast, HardSATGEN takes close to 2 hours to generate a single instance. To generate 1000 LEC instances at this speed we would need 75 days. HardSATGEN is so slow because it solves

an NP-Complete problem hundreds of times within its pipeline. HardSATGEN iteratively identifies the core via a traditional Core Detection technique [19] which requires that the problem be solved. Considering it would take months to generate a deep-learning-scale dataset using HardSATGEN, we determine that HardSATGEN is not fast, whereas the other methods are fast.

Question 3: Is the method able to generate datasets that are representative of the original datasets in terms of hardness distribution, while remaining diverse?

Although past work, including [15, 17, 22], has examined graph statistics such as modularity and clustering coefficients, we find little evidence that these are indicative of the hardness of generated instances. Instead, we focus on the similarity of the distributions of the hardness vectors because hardness is of primary importance when working with SAT problems.

As G2MILP and W2SAT exhibit hardness collapse, we only compare HardSATGEN and HardCore for runtime distribution analysis. Note that due to HardSATGEN's high cost, we can only generate 50 LEC instances and 50 K-SAT instances (using 10 seed-instances) within 3 days. For HardCore, we use 1445 LEC and 1321 K-SAT seed instances, again generating 5 times per seed. 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 (due to time constraints we are limited in how many HardSATGEN instances we can generate). In order to compare the original and generated SAT problems, we (1) employ a distribution distance metric, Maximum Mean Discrepancy (MMD), to perform a quantitative comparison; (2) construct visualizations of the runtime distributions; and (3) examine visualizations of the runtime orderings distributions.

Maximum Mean Discrepancy is a non-parametric measurement of the distance between two distributions. For a selected kernel k, with $k(X, Y) = \langle \phi(X), \phi(Y) \rangle$, it is defined as follows:

$$MMD(P, Q) = ||\mathbf{E}_{X \sim P}[\phi(X)] - \mathbf{E}_{Y \sim Q}[\phi(Y)]||_{\mathcal{H}} \,. \tag{3.3}$$

By applying the kernel trick to $MMD^2(P, Q)$, we get the following:

$$MMD^2(P, Q) = ||||\mathbf{E}_{X \sim P}[\phi(X)] - \mathbf{E}_{Y \sim Q}[\phi(Y)]||_{\mathcal{F}}^2 \tag{3.4}$$

$$= \mathbf{E}_P[k(X, X)] - 2\mathbf{E}_P, Q[k(X, Y)] + \mathbf{E}_Q[k(Y, Y)] \,. \tag{3.5}$$

We can form empirical approximations of the expectations as follows:

$$\mathbf{E}_P[k(X,X)] \approx \frac{1}{m(m-1)} \sum_i \sum_{j \neq i} k(x_i, x_j)\,, \tag{3.6}$$

$$2\mathbf{E}_P, Q[k(X,Y)] \approx \frac{2}{m \cdot m} \sum_i \sum_j k(x_i, y_j) \tag{3.7}$$

$$\mathbf{E}_Q[k(Y,Y)] \approx \frac{1}{m(m-1)} \sum_i \sum_{j \neq i} k(y_i, y_j)\,. \tag{3.8}$$

In practice, we adopt the radial basis function kernel, and use samples from the hardness vector distributions $X \sim P$ (original data samples) and $Y \sim P$ (generated data samples) to estimate the expectations.

Runtime distributions are compiled empirically by measuring the real-world (wall-clock) time required to solve the SAT problems in the source data. To ensure reliable measurements, we collect this data by running the SAT solvers one at a time, and all on the same computer, which is reserved only for this experiment. Thus, while there is certainly some small noise introduced due to background tasks, it is effectively negligible — especially when compared to variation in runtimes which arises from random permutation of variable and clause ordering in SAT problems [13].

As shown in Table 3.3, HardCore achieves runtime distributions far closer to the original distributions compared to HardSATGEN with respect to the MMD metric. We calculate

Table 3.3: Evaluation of generated datasets on LEC data. Maximum Mean Discrepancy (MMD) distance between distributions of generated and original datasets; lower is better.

	W2SAT	HardSATGEN	G2MILP	HardCore
Similarity (MMD)	—	0.492	—	**0.004**

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. We note that while HardCore achieves low MMD, the solving time of individual instances is considerably different from that of their associated seeds. This implies that the low MMD of HardCore is not achieved by replicating or barely modifying seed instances. Our later experiments investigating augmentation suggest that there is sufficient diversity being injected in the generated instances.

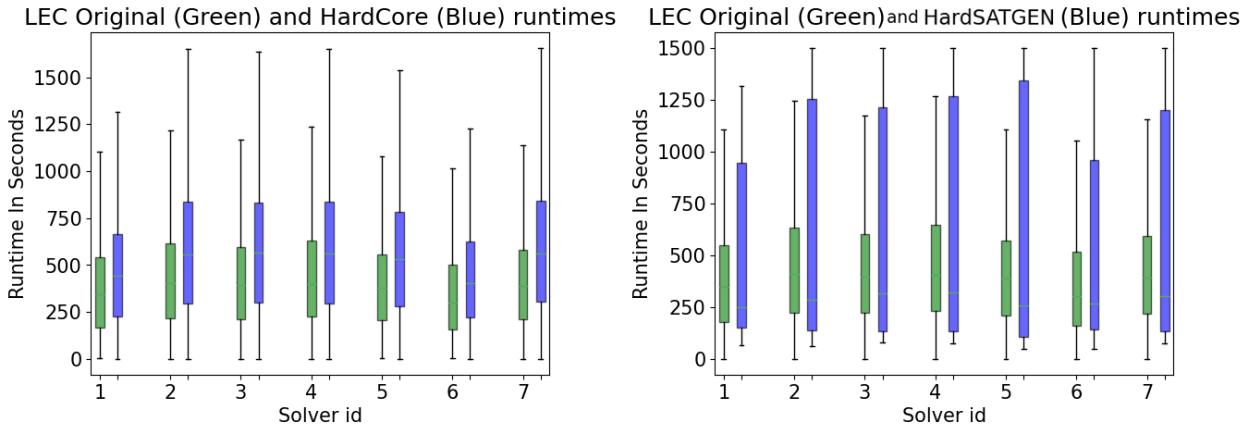

Figure 3.6: HardCore (Left) and HardSATGEN (Right). Boxplots of runtimes per solver for Original (Green) and Generated (Blue) instances on LEC data. HardCore appears to produce per-solver distributions which are much closer to the original than HardSATGEN, which tends to produce high-variance and on-average much harder problems than the original.

In Figure 3.6, we compare the visualizations of the collected samples from the original and generated single-solver runtime distributions. HardCore produces per-solver distributions whose visualizations are much more similar to the original's than HardSATGEN. In particular, we note that HardSATGEN experiences greater variation in generated hardness compared to both HardCore and the original hardnesses.

From the figure, both methods generally produce instances that are harder than the original. Given the hard core guarantees employed, problems should rarely be easier than the original's. However, if a harder potential core than the original is introduced during generation, then the potential hardness of the problem increases, and this potential can be realized via sufficient core refinement. This would explain why we both methods generate harder problems, on average, than those in the original data. From the figures and results shown above, we can conclude that our method is in fact representative of the original data. Since generations are initially random by our methodological use of random generators to produce pre-refinement problems, we know without experimentation that our generations are diverse, thus achieving the goal of representativity and diversity of our generations described in the problem statement in subsection 3.2.

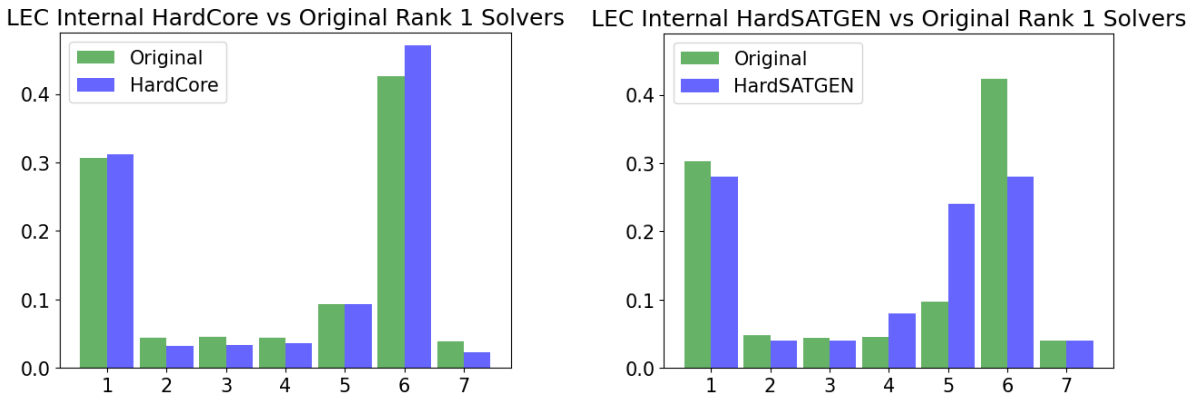

Figure 3.7: LEC Internal Rank 1 Solvers. We compare the number of times each solver is the fastest solver for original and synthetic problems. Experiments with generations by HardCore (left) and HardSATGEN (right). X axis is Solver ID and y axis is how frequently (%) the solver is ranked first by runtime.

In Figure 3.7, we see a striking similarity between the histogram for HardCore's best-performing solvers and the original's, indicating that the HardCore synthetic instances are solved most efficiently by the same solvers as the original instances. Meanwhile, a greater discrepancy can be seen between original and HardSATGEN-generated data, particularly for solvers 5 and 6. Given the notion previously discussed that solver performance rankings are heavily dependent on the underlying nature of the instances (e.g., random, industrial), a notable disparity between performance rankings of original and synthetic data — especially for first-place — might imply a difference in the underlying nature of generated and original data. Given HardSATGEN's visible disparity, we claim that while HardSATGEN may generate hard problems it fails to faithfully reproduce the nature of the problems. If the goal is not

to preserve critical structure, then random generators are a reasonable alternative, because they can successfully generate hard problems, given the appropriate hyper-parameters, and usually have much lower computational cost.

Figure 3.8 shows stacked histograms of the rankings for each solver, following up on the rank-1 histogram shown in Figure 3.7. The top row compares the ranking distribution of original LEC instances and HardCore's generations. The bottom row depicts the same comparison for HardSATGEN. Note that the original distributions are different for HardSATGEN and HardCore because the methods are provided with different quantities of seed data. Given HardSATGEN's cost, only 10 seed instances are used for generation (to generate 50 instances), whereas HardCore is given 1445 instances and generates 5780. On inspection of the figure, we note the similarity of the original and HardCore ranking distribution visualizations. For example in HardCore, solver 1's distribution of rankings shows a very similar proportion of ranks 2-6, with perhaps slightly higher rank 1 (and lower rank 7) than the original. In contrast, HardSATGEN has a very different distribution compared to the seed data. For example, rank 1 is common for solver 1 in the generated data, but solver 1 was never ranked first for the original data given to HardSATGEN. Solver 2 is never ranked first for HardSATGEN's generated data, but is first occasionally in the original data. These results suggest that there are important discrepancies between the generated and original data.

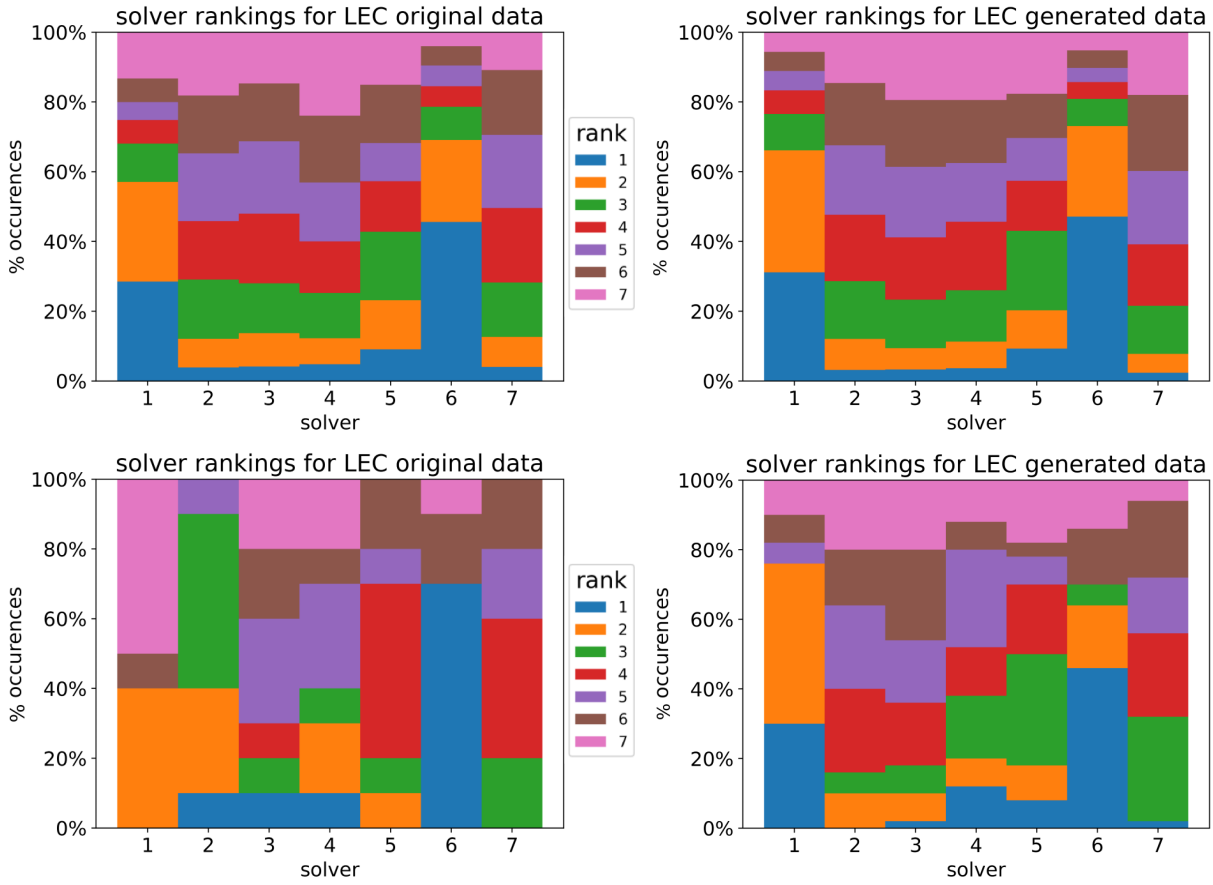

Figure 3.8: HardCore (top) and HardSATGEN (bottom) Comparison of Solver Rankings by stacked Histogram for Original and Generated LEC data. X axis indicates the solver, color represents the ranking of that solver. For example, the first column in each figure shows how frequently Solver 1 is ranked first, second, third, et cetera. By comparing the original and generated histograms (left/right) for both HardCore and HardSATGEN (top/bottom), we see that HardCore generates highly similar rankings to original data whereas HardSATGEN is not consistent with the original ranking histogram.

Question 4: Can we successfully augment training data with the method's generated data for machine learning?

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 (an architecture which uses 30 hand-picked SAT problem features, e.g., number of clauses/literals) to predict the solver runtime of one specific solver on a given instance. We repeat this for each of the 7 solvers. We calculate the Mean Absolute Error (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 as the number that could be generated in approximately 3 days of computation. The result was $N = 50$. We also compare to the un-augmented training sets and refer to it as original.

In order to observe performance over varying sizes of training data, we conduct this experiment for several quantities of original training instances, which is denoted Data Size. Three augmentation instances are provided per original instance, and augmentation is only allowed by using the original instances which are in the training set. Validation sets are selected from the original data only, with an 80/20 train/validation split. For LEC, the test-set is made

up of 10000 randomly selected problems that are not in the training or validation sets. For K-SAT, the test-set is made up of the problems from the 1351 original instances that are not included in the training or validation sets.

Table 3.4 shows that for both K-SAT random data and LEC Internal dataset, training on data augmented using HardCore leads to a 20-50 percent reduction in MAE. The gain of data augmentation increases with larger data size. In contrast, no other data generation method leads to a comparable improvement.

Table 3.4: MAE of Runtime Prediction averaged across 7 solvers and 15 trials. Asterisks are placed at the best result which passes the Wilcoxon pairwise ranking test against the second-best for $p < 0.05$. For a boxplot visualization of the trial outcomes, see Figure A.1 in Appendix A.4.

	K-SAT Random				LEC Internal				
Data Size	10	20	30	40	100	200	300	400	500
HardSATGEN-*N*	2416	2306	2172	2182	666	797	605	617	463
HardSATGEN-Strict	2179	2578	2488	2456	627	742	565	638	513
W2SAT	2606	2046	1807	1377	724	704	634	611	535
Original	2750	2743	2109	1449	707	795	557	606	526
HardCore	**2156**	**1796***	**1615**	**930***	**514**	**481***	**369***	**282***	**338***

Table 3.5: We repeat the data augmentation experiment on two sub-families of SAT Competition data (FDMUS and Tseitin) on our method of augmentation.

	FDMUS				Tseitin					
Data Size	100	200	300	400	500	10	20	30	40	50
HardCore	**0.220**	**0.197**	**0.162**	**0.142**	**0.142**	3618.9	**3410.0**	**3311.4**	**3417.7**	**3419.4**
Original	0.246	0.213	0.184	0.173	0.145	**3369.6**	3581.9	3576.1	3544.7	3608.5

A note on SAT Competition Data SAT Competition data is an aggregation of thousands of families of problems and is therefore highly heterogeneous. As discussed briefly in the introduction, this heterogeneity is highly unfavourable for machine learning algorithms and tasks. Thus, machine-learning papers are often required to find creative ways to provide large-scale experimental data (and this may not conform to real-world data).

In order to demonstrate that our method can generalize to other datasets, we now provide results on the data augmentation experiment conducted above, on all UNSAT problems from the SAT Competition data coming from the "Tseitin" family, and separately on all UNSAT problems from the SAT Competition "FDMUS" family, in table 3.5. We see that while the improvement is reduced compared to that achieved on the k-SAT and LEC data, our method still provides improvement.

3.5 Summary

In this chapter we presented our method, HardCore, which generates realistic SAT problems by randomly generating clauses and subsequently refining the core to enhance hardness via a

low-cost (relative to classical methods) graph neural network. We demonstrate the low-cost nature of our method via wall-clock runtime comparisons with existing methods which either perform competitively with our method or significantly underperform our method. We evaluate the degree to which our method generates realistic problems by comparing (visually or numerically) various measurements of Hardness distributions, including per-solver runtime distributions, ranking distributions and over-all runtime distributions. On comparisons between our method and the only other method which could produce problems with non-zero solve times (HardSATGEN), our method produced more faithful generations while operating at a thousandth of the cost. Finally, we found that our method — in particular on the real internal data — outperformed existing augmentation techniques. In fact, our method was the only method to consistently offer performance improvements over the non-augmented training-set.

Chapter 4

Conclusion

4.1 Conclusion

We have presented a method for generating UNSAT problems that preserves hardness without imposing an unreasonable computational cost. Most existing deep-learned SAT generation algorithms consistently suffer from hardness collapse, generating instances that require tenths of a second to solve, while the original problems used to seed the algorithms require several minutes. Recently a method was presented by Li et al. [17] which produces problems that are considerably harder than the original seed instances. This method is not practical, however, as even with powerful computational resources, it can take months to generate the amount of data needed for training. The algorithm relies on traditional, computationally-demanding SAT-solving techniques for post-hoc refinement of generated instances.

Our proposed method leverages some innate characteristics of UNSAT problems as well as their graphical structure. It targets the core of the SAT problem, and iteratively performs

refinement using a GNN-based core detection procedure. This has dramatically lower cost than traditional SAT-solving based core detection methods. Our method also leverages low-cost random generation, relying on the post-hoc refinement process to convert the problems' hardness profiles from random to faithful. Experimentally, we demonstrate that the method produces instances that are similar to the original instances in hardness profile, in terms of (1) individual solver hardness distributions; (2) multi-solver hardness distributions; and (3) multi-solver ranking distributions. We also show experimentally that we achieve these results at much lower cost than the only previous work capable of producing hard problems. Finally, we confirm that the data we generate is indeed useful for augmentation by training a well-known hardness-estimator model. We show that the predictive performance improves considerably when we augment the training set using data generated using our method.

4.2 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 proposed approach, this limitation is unavoidable because cores are only present in UNSAT problems. While previous algorithms such as G2SAT [15] and W2SAT [22] do not share this limitation, these models do not produce hard problems and so suffer from far more significant limitations. HardSATGEN [17], which is able to produce hard problems, is also limited to UNSAT problems because of its dependence on core structure.

Another limitation is that our work relies solely upon empirical results to demonstrate its efficacy, and these results are only presented on two datasets, one of which is synthetic. The primary motivation of this work was to address the data scarcity problem which makes developing deep-learning-based methods for SAT difficult. Designing a deep-learning-based method for SAT ourselves, we of course suffered from this problem as well. To partially address this concern, we conducted several trials and statistical significance testing to ensure the reliability of our empirical analysis.

Scalability to very large problems must be considered in the limitations section. Given the graph-structure of our method, memory and computation requirements of the method scale quadratically with the size of the problem. In the public SAT database [11], problems can be found with millions of variables. These problems are out of reach for our method at the moment. For our experimental hardware (32GB GPU) and our implementation of the graph building/storage ($O(nm)$ for a problem with n clauses and m variables), a problem with 256,000 variables and 1,000,000 clauses would require $256,000 \times 1,000,000 \times 1 = 256 \times 10^9$ bits, or 32 gigabytes. Given a GPU with 32GB of memory, this would be a breaking point for the method. Of course this is the worst case, which only occurs for a completely dense graph representation. In practice, clauses in the LEC data, for example, tend to have on average 3 or 4 variables. Since in the LCG clause nodes are only connected to the variables of which they consist, each clause node would then only have degree of 3 or 4, meaning the

graph is very sparse. Thus, in practice the primary memory cost of our model scales more so according to $O(dn)$, assuming average number of variables in a clause is d, and assuming the implementation is adapted to leverage the sparse structure (using an edge-list instead of a dense adjacency matrix, for example).

A final limitation of the method presented in this work is its dependence on a random generation module. This choice was made primarily in the interest of keeping inference cost low, and the focus of this work was in developing a fast refinement strategy. However, random pseudo-industrial methods have been criticized as being too narrow in their generative diversity due to the focus on one or two specific statistics in each method [15, 22]. While our empirical results did show that the refinement strategy successfully produced hardness profiles similar to industrial problems, perhaps performance might have been improved with a learned generation procedure. Seeing as the primary contribution of the method is the refinement process, which can be interpreted as a post-processing procedure, any generation process could in principle be used to provide the refinement process with inputs.

4.3 Future Work

As discussed, a primary limitation of this work is that it can only be used to generate UNSAT problems because of its focus on the cores of problems, which are only found in UNSAT problems (cores are minimal UNSAT subsets of a problem, and if a subset is UNSAT then the

whole problem is UNSAT). There is, however, a concept analogous to cores for SAT problems. The *backbone* of a SAT problem is the set of literals which are found in **all** satisfying solutions to the problem [34]. Literals in the backbone can be interpreted as the fixed variables of the problem, constrained to the point that they must be set but not over-constrained such that the problem becomes UNSAT. For example, given the problem $(\neg A) \wedge (A \vee B) \wedge (\neg A \vee C)$, $\neg A$ must be part of the backbone as it is alone in the first clause and therefore must always be set. As a result, B is a part of the backbone as in the second clause A is always false. C, however, is not a member of the backbone because C only appears in a clause with $\neg A$, which must always be true, meaning that whether C or $\neg C$ is true, the third clause will be satisfied. Thus, one might solve a SAT problem only by determining the assignments of the backbone (similarly to how one might solve an UNSAT problem only by detecting the core). Given this, there is likely the potential for a similar algorithm to HardCore which uses backbone-refinement (literals can be added to the backbone by introducing a clause with only that literal, and can be removed by introducing new satisfied literals to that literal's clauses) and backbone detection using a GNN node-classifier on literal nodes.

Early in the development of the method presented in this thesis, there was discussion of publishing an anonymized version of the internal dataset used. While this would be significant for the field (no cohesive large-scale public industrial dataset exists currently), it proved to be challenging. Given the core guarantee used in our method, a fraction of the original core

persists, tainting the anonymity of the generated dataset. Anonymizing the data, whether by developing the method such that the core guarantee is no longer required, or showing that no meaningful information can be extracted from the generated data, could be a promising and impactful avenue for future work.

Another limitation discussed previously is the random component to our method. While off-the-shelf bi-partite graph-generation methods (e.g. the split-merge framework presented by You et al. [15], the bipartite graph diffusion model presented by Chopin et al. [35]) are scarcer than their homogeneous counterparts, they exist and perhaps could be used to replace the random generation element of the method. At minimum, a more in-depth study of the impact of the use of different random methods within HardCore and the setting of their hyper-parameters would be valuable.

Two connected and very interesting directions for future work are a focus on specific, highly structured problem families (e.g. graph-coloring problems, planning problems) and more sophisticated de-coring mechanisms. For highly structured problems, the generation mechanism would have to be designed such that a formula adheres to the required structure. Structured problems such as the graph-coloring are often easy to generate. For example, a graph coloring problem can be generated by generating a graph in any random way, and then applying the coloring problem. A pigeonhole problem can be generated simply by

selecting some parameters for the problem. Core Prediction itself would likely remain un-changed. In fact, if the training data (and test-data) are focused on one structured class of formulas, we would expect high performance from the prediction model. Special care, however, would be required to ensure that the de-coring operation does not corrupt the required structure of the problem. Under the current framework of adding a new variable to de-coring target clauses, the structure of the problem may be changed in such a way that it no longer conforms to the strict structure of the family. De-coring operations would therefore have to be specially designed for each family type. This brings us to the second of the pair of directions mentioned, which is more sophisticated core-refinement methods. In this work, we chose to use the established de-coring paradigm from the HardSATGEN method, considering it to be highly interpretable due to its simplicity. More complicated strategies, which might add or remove existing variables from the clauses (instead of adding new ones), have the potential to introduce unforeseen conflicts, and possibly a new core that is easier than the current de-coring target. The current strategy guarantees the removal of the current core, without constructing a new one, which is desirable. Since the approach proved experimentally effective, we did not explore other strategies. However, a major weakness of the current mechanism is that it effectively removes the target clause from the problem by making it trivially satisfiable to the solver.

Appendix A

Further Experimental Details

A.1 Data

Table A.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.

	var.	clause	runtimes (s)	count
LEC	1328	5167	388	78730
K-SAT	398	1751	2700	1351

A.2 Hyper-parameters

In our design process, given the cost of running experiments — in particular measuring runtime of generated instances — we did not conduct exhaustive hyper-parameter searches. Hyper-parameters were set following design considerations and rationales, which will be discussed here.

- The random generation method we use is Popularity-Similarity. This has several hyper-parameters: average clause size, β_c, β_v and T. Average clause size determines the average number of literals per generated clause, β_c and β_v are constants in the probability distribution for clause and variable selection, respectively, and T is a constant in the exponent of the probability of an edge existing between clause and variable. Conducting an exhaustive search over these hyper-parameters is expensive because the evaluation of each configuration is via runtime-measurement, which requires the solving of a large number of SAT problems by multiple solvers. We communicated with the authors of the paper which presented HardSATGEN, and were able to obtain their hyper-parameter configuration for Popularity-Similarity (PS), which was included among their reported baselines. For continuity with previous work and in the interest of reducing the computational budget, we used the provided configuration.

- The GCN backbone within our core prediction module has two hyper-parameters, namely the number of hidden dimensions and the number of layers. Three potential values were chosen for initial exploration of layer size: [3, 4, 15]. In many applications, GCN networks are configured to have only 3 or 4 layers. This is because GNN networks in general are prone to over-smoothing as the number of layers increases. 15 layers was added to validate this behaviour within our context. For hidden dimension size we chose two potential values: [32, 64]. Our findings were that as the model size increased via additional layers and

hidden feature size, there was minimal improvement in performance. Thus, we selected the smallest defined configuration of 3 layers and hidden dimension of 32.

- Finally, there is the Core-Refinement hyper-parameter that specifies the number of iterations. This value can be set in terms of the number of generated clauses, since one clause is modified at each iteration. The safest setting is to set the number of iterations to be equal to the number of generated clauses, such that, if necessary, the method is allowed to modify every generated clause. In practice, this was the setting we used.

A.3 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.

Algorithm 1 Algorithm for generating 1 K-SAT Random instance.

$m \sim N(\mu_m, \sigma_m)$
$c \sim N(\mu_c, \sigma_c)$
$n \leftarrow$ `int`(mc)
`cnf` $\leftarrow$ `randkcnf(3, m, n)`
▷ where randkcnf(k, m, n) returns cnf m with k-var clauses from n variables.

A.3.1 K-SAT Random Generation

Algorithm 1 shows the process by which we generated K-SAT Random instances as discussed in section 3.4.1. We randomly sample hyper-parameters (number of clauses, number of variables) from a small window in order to introduce some additional variety into the dataset, and generate by randomly sampling sets of 3 variables without replacement. In our work we chose $m \sim N(400, 100)$, $c \sim N(4.4, 0.05)$.

A.4 Supplementary Results

A.4.1 Fine-Grained results on Data Augmentation Experiment

In section 3.4.2 we compare the performance of two runtime prediction models: one trained on only original data and the other trained on a dataset augmented with generated instances.

To observe performance over differing levels of data availability, we conduct this experiment

for several quantities of original training instances — denoted Data Size. 3 Augmentation instances are allowed per original, and augmentation is only allowed by using the original instances in the training set. Validation sets are selected from the original data only, with an 80/20 split train/validation split.

In Figure A.1 we can that while there is considerable overlap with whiskers of the other methods, HardCore outperforms all other methods on all data sizes by at least one quartile of results. In addition to increased prediction accuracy (lower MAE), HardCore demonstrates a tendency to reduce variance in performance, which we note by the lower whisker-to-whisker spread of the boxplots. This effect is especially notable in data-size 200, but can also be seen relative to other augmentation methods for data size 300.

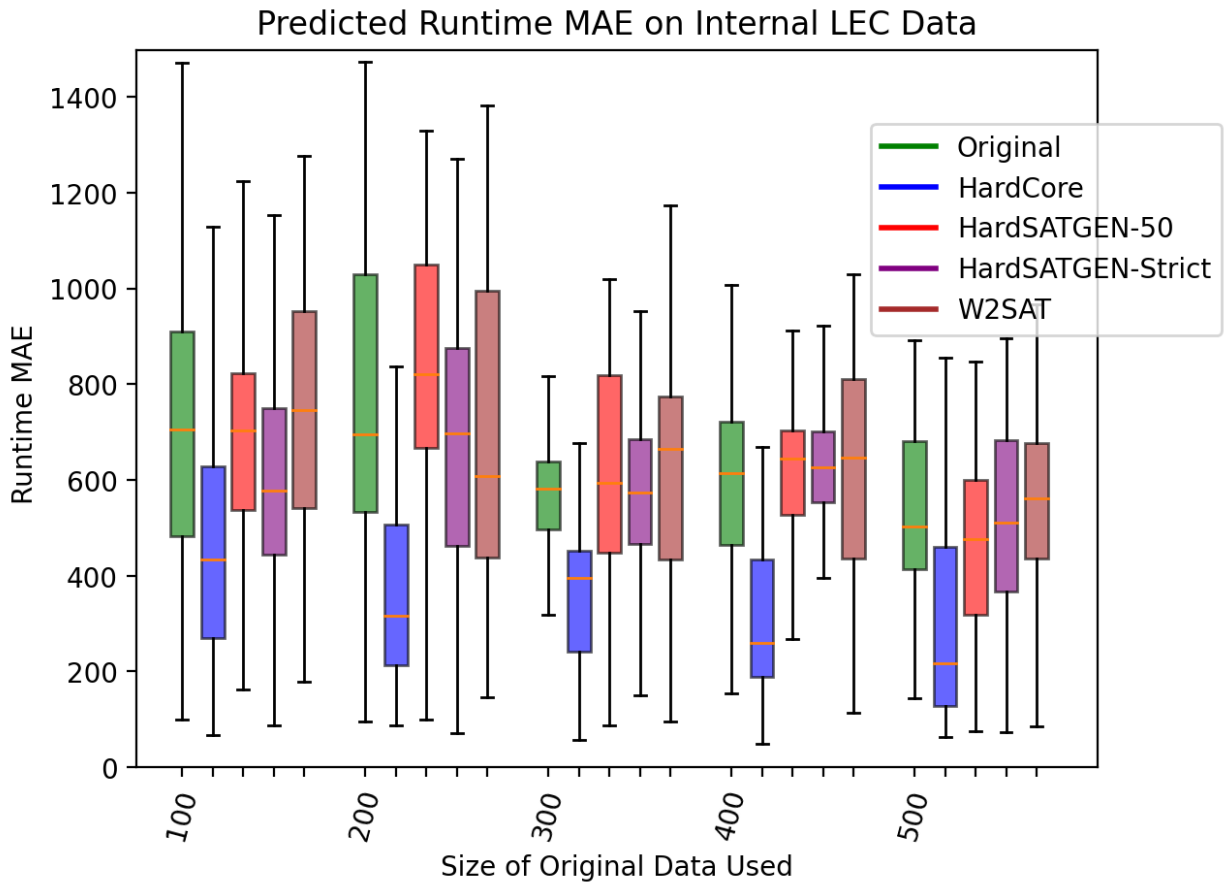

Figure A.1: Mean MAE on Runtime Prediction. Boxplot-view of results presented in Table 3.4 for LEC data.

Bibliography

[1] E. Goldberg, M. R. Prasad, and R. K. Brayton, "Using SAT for combinational equivalence checking," in *Proc. Conf. Design, Automation and Test in Europe*, 2001.

[2] A. Ramamoorthy and P. Jayagowri, "The state-of-the-art boolean satisfiability based cryptanalysis," in *Proc. Materials Today*, vol. 80, 2023, pp. 2539–2545.

[3] P. Habiby, S. Huhn, and R. Drechsler, "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)*, 2021, pp. 1–4.

[4] L. Xu, F. Hutter, H. H. Hoos, and K. Leyton-Brown, "SATzilla: Portfolio-based algorithm selection for SAT," in *Journal Artificial Intelligence Research*, June 2008, pp. 565–606.

[5] S. Kadioglu, Y. Malitsky, M. Sellmann, and K. Tierney, "ISAC – instance-specific algorithm configuration," in *Proc. European Conf. Artificial Intell.*, 2010, p. 751–756.

[6] Y. Malitsky, A. Sabharwal, H. Samulowitz, and M. Sellman, "Satisfiability solver selector (3S)," in *Proc. SAT Challenge*, 2012, pp. 565–606.

[7] A. R. KhudaBukhsh, L. Xu, H. H. Hoos, and K. Leyton-Brown, "SATenstein: Automatically building local search SAT solvers from components," in *Proc. Int. Joint Conf. Artificial Intell.*, 2009, pp. 517–524.

[8] T. Fuchs, J. Bach, and M. Iser, "Active learning for SAT solver benchmarking," in *Proc. Int. Conf. Tools and Algorithms for the Construction and Analysis of Systems.* Springer, 2023, pp. 407–425.

[9] M. Heule, M. Järvisalo, and M. Suda, "SAT Competition 2018," in *Journal Satisfiability, Boolean Modeling and Computation*, vol. 11, 09 2019, pp. 133–154.

[10] D. Selsam and N. Bjørner, "Guiding high-performance SAT solvers with UNSAT-core predictions," in *Proc. Int Conf. Theory and Applications of Satisfiability Testing*, 2019, pp. 336–353.

[11] T. Balyo, M. J. Heule, M. Iser, M. Järvisalo, and M. S. (Eds.), "Solver and benchmark descriptions," in *Proc. SAT Competition*, 2022.

[12] J. G. Cru and J. Levy, "Locality in random SAT instances," in *Proc Int. Joint Conf. on Artificial Intelligence*, 2017, pp. 638–644.

[13] Z. Zhang, D. Chetelat, J. Cotnareanu, A. Ghose, W. Xiao, H.-L. Zhen, Y. Zhang, J. Hao,

M. Coates, and M. Yuan, "GraSS: Combining graph neural networks with expert knowledge for SAT solver selection," in *Proc. ACM SIGKDD Conf. Knowledge Discovery and Data Mining*, 2024.

[14] H. Wu and R. Ramanujan, "Learning to generate industrial SAT instances," in *Proc. Int. Symp. Combinatorial Search*, vol. 10, no. 1, 2019, pp. 206–207.

[15] J. You, H. Wu, C. Barrett, R. Ramanujan, and J. Leskovec, "G2SAT: Learning to generate SAT formulas," in *Proc. Adv. Neural Inf. Process. Syst.*, 2019.

[16] I. Garzón, P. Mesejo, and J. Giráldez-Cru, "On the performance of deep generative models of realistic SAT instances," in *Proc. Int. Conf. Theory and Applications of Satisfiability Testing*, 2022, pp. 3:1–3:19.

[17] Y. Li, X. Chen, W. Guo, X. Li, J. Huang, H.-L. Zhen, M. Yuan, and J. Yan, "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*, 2023.

[18] N. Dershowitz, Z. Hanna, and A. Nadel, "A scalable algorithm for minimal unsatisfiable core extraction," in *Proc. Int. Conf. Theory and Applications of Satisfiability Testing*, 2006, pp. 36–41.

[19] N. Wetzler, M. J. Heule, and W. A. Hunt, "DRAT-trim: Efficient checking and trimming using expressive clausal proofs," in *Proc. Int. Conf. Theory and Applications of Satisfiability Testing*, 2014, pp. 422–429.

[20] A. Biere, M. Heule, H. von Maaren, and T. Walsh, *Handbook of Satisfiability.* IOS Press, 2009.

[21] C. Ansótegui, M. L. Bonet, J. Levy, and F. Manya, "Meauring the hardness of SAT instances," in *Proc. AAAI Int. Conf. Artificial Intell.*, 2008.

[22] W. Wen and T. Yu, "W2SAT: Learning to generate SAT instances from weighted literal incidence graphs," 2023, arXiv preprint arXiv:2302.00272.

[23] W. Hamilton, Z. Ying, and J. Leskovec, "Inductive representation learning on large graphs," in *Proc. Adv. Neural Inf. Process. Syst.*, I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, and R. Garnett, Eds., vol. 30, 2017.

[24] J. G. Cru and J. Levy, "Generating SAT instances with community structure," in *Artificial Intelligence*, 2016, p. 119–134.

[25] D. Zhao, L. Liao, W. Luo, J. Xiang, H. Jiang, and X. Hu, "Generating random SAT instances: multiple solutions could be predefined and deeply hidden," *Journal of Artificial Intelligence Research*, vol. 76, pp. 435–470, 2023.

[26] Z. Geng, X. Li, J. Wang, X. Li, Y. Zhang, and F. Wu, "A deep instance generative framework for MILP solvers under limited data availability," in *Proc. Adv. Neural Inf. Process. Syst.*, Dec 2023.

[27] A. Belov, I. Lynce, and J. Marques-Silva, "Towards efficient MUS extraction," *AI Commun.*, vol. 25, no. 2, p. 97–116, apr 2012.

[28] L. Xu, H. H. Hoos, and K. Leyton-Brown, "Hydra: Automatically configuring algorithms for portfolio-based selection," in *Proc. AAAI Conf. Artificial Intell.*, 2010, pp. 210–216.

[29] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, "Learning a SAT solver from single-bit supervision," in *Proc. Int. Conf. Learning Representations*, 2019.

[30] A. Biere, K. Fazekas, M. Fleury, and M. Hessinger, "Cadical, kissat, paracooba, plingeling and treengeling entering the SAT competition," in *Proc. SAT Competition*, 2020, p. 50.

[31] M. Fleury and A. Biere, "GIMSATUL, ISASAT, KISSAT," in *Proc. SAT Competition*, 2022, p. 10.

[32] M. S. Cherif, D. Habet, and C. Terrioux, "Kissat MAB: Upper confidence bound strategies to combine VSIDS and CHB," in *Proc. SAT Competition*, 2022, p. 14.

[33] M. S. Chowdhury, "kissat-hywalk-gb, kissat-hywalk-exp, kissat-hywalk-exp-gb, and malloblin entering the SAT competition," in *Proc. SAT Competition*, 2023, p. 28.

[34] P. Kilby, J. Slaney, S. Thiébaux, T. Walsh *et al.*, "Backbones and backdoors in satisfiability," in *Proc. AAAI Conf. Artificial Intelligence*, 2005, pp. 1368–1373.

[35] B. Chopin, H. Tang, and M. Daoudi, "Bipartite graph diffusion model for human interaction generation," in *Proc. IEEE/CVF Winter Conf. Applications of Computer Vision (WACV)*, January 2024, pp. 5333–5342.