G2SAT: Learning to Generate SAT Formulas

Anonymous Author(s)
Affiliation
Address
email

Abstract

The Boolean Satisfiability (SAT) problem is the canonical NP-complete problem that is fundamental to Computer Science, with a wide array of applications in planning, verification, and theorem proving. Developing and evaluating practical SAT solvers relies on extensive empirical testing on a set of real-world benchmark formulas. However, the number of such real-world SAT formulas is limited, and while augmenting benchmark formulas via synthetic SAT formulas is possible, existing approaches are heavily hand-crafted and cannot simultaneously capture a wide range of characteristics of real-world SAT formulas. Here we present G2SAT, the first deep generative framework that learns to generate SAT formulas from a given set of input formulas. Our key insight is that SAT formulas can be transformed into latent bipartite graph representations which can be modeled using specialized deep generative models for graphs. The core of G2SAT is a novel deep generative model for bipartite graphs, which generates graphs via iterative node merging operations. We show that G2SAT can generate SAT formulas that closely resemble given real-world SAT formulas, as measured by both graph metrics and SAT solver behavior. Furthermore, we show that our synthetic SAT formulas can be used to improve SAT solver performance on real-world benchmarks, which opens up new opportunities for the continued development of SAT solvers and a deeper understanding of their performance.

1 Introduction

The *Boolean Satisfiability (SAT) problem* is the canonical NP-complete problem that is fundamental to Computer Science, and has many applications across Artificial Intelligence, including planning [23], verification [8], and theorem proving [14]. While SAT problems are generally hard to solve, different algorithms specialize in different subsets of SAT problems. For example, incomplete search methods such as WalkSAT [33] and survey propagation [7] are more effective at solving large, randomly generated formulas, while complete solvers leveraging *conflict-driven clause learning* (CDCL) [28] can often solve large structured SAT formulas that commonly arise in industrial settings.

Developing and evaluating modern SAT solvers heavily relies on extensive empirical testing on a suite of benchmark SAT formulas. Unfortunately, in many domains such formulas are hard to acquire and thus extremely limited. Developing expressive generators of SAT formulas is important, because it would provide for a richer set of evaluation benchmarks, which would in turn allow for developing better and faster SAT solvers. Indeed, the problem of pseudo-industrial SAT formula generation has been identified as one of the ten key challenges in propositional reasoning and search [34].

A promising direction to resolve the above challenge would be to represent SAT formulas with graph representations, and recast the problem as a graph generation task. Specifically, there exists bijective graph representations of SAT formulas, known as the literal-clause graphs (LCGs). However, generating LCGs is challenging, as they have to obey the hard bipartite constraint. Therefore, existing work in SAT generation heavily relies on hand-crafted algorithms[15, 16], focusing on

Submitted to 33rd Conference on Neural Information Processing Systems (NeurIPS 2019). Do not distribute.

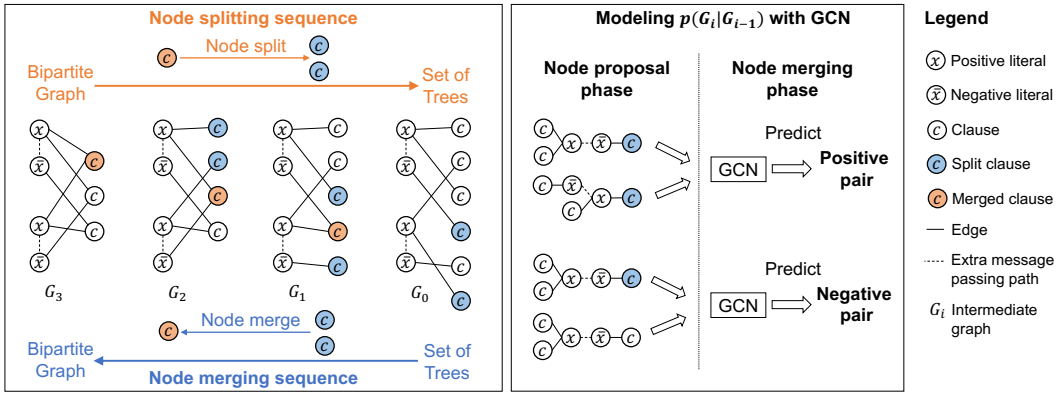

Figure 1: An overview of the proposed G2SAT model. **Top left**: A given bipartite graph can be decomposed into a set of disjoint trees by applying a sequence of node splitting operations. Orange node c in graph G_i is split into two blue c nodes in graph G_{i-1}. Every time a node is split, one more node appears in the right partition. **Right**: We use node pairs gathered from such a sequence of node splitting operations to train a GCN-based classifier that predicts whether a pair of c nodes should be merged. **Bottom left**: Given such a classifier, G2SAT generates a bipartite graph by starting with a set of trees G_0 and applying a sequence of node merging operations, where two blue nodes in graph G_{i-1} get merged in graph G_i. G2SAT uses the GCN-based classifier that captures the bipartite graph structure to sequentially decide which nodes to merge from a set of candidates. Best viewed in color.

generating formulas that fit *one* of the graph statistics exhibited by real-world SAT formulas [32, 2]. As researchers continue to uncover unique characteristics of real-world SAT formulas [2, 22, 32, 29], previous SAT generators might become invalid, and hand-crafting new models that capture all characteristics becomes increasingly difficult. On the other hand, the recent success of deep generative models for graphs [5, 27, 40, 41] has demonstrated their ability to capture *many* of the essential characteristics of real-world graphs, such as social networks and citation networks, as well as graphs arising in biology and chemistry. However, these models cannot be readily applied because they cannot enforce bipartiteness.While it might be possible to post-process the generated graphs, this compromise solution would fail to utilize unique structure of bipartite graphs and could be computationally expensive and lacking justifications.

Here we propose G2SAT, the first deep generative model that *learns* to generate SAT formulas from a given set of input formulas. We use literal-clause graphs (LCGs) to represent SAT formulas, and formulate the task of SAT formula generation as a bipartite graph generation problem. Our key insight is that any bipartite graph can be generated by starting with a given set of trees, and then applying a sequence of *node merging* operations over the nodes coming from one of the two partitions. As we merge nodes, trees also get merged, and complex bipartite structures will appear (Figure 1, left). In this way, a set of input bipartite graphs (SAT formulas) can be characterized by a distribution over the sequence of node merging operations. Assuming we can capture/learn the distribution over which pairs of nodes to merge, we can start with a set of trees and then keep merging nodes in order to generate realistic bipartite graphs (i.e., realistic SAT formulas). G2SAT models this iterative node merging process in an auto-regressive manner, where a node merging operation is viewed as a sample from the underlying conditional distribution that is parameterized by a Graph Convolutional Network (GCN) [18, 25], and the same GCN is shared across all the steps of the generation process.

A challenge we have to resolve is how to generate training data for our generator, which corresponds to inferring the sequential generative process from the static input SAT formulas. To resolve the challenge, we proceed as follows (Figure 1). At training time, we define the reverse operation of node merging as *node splitting*, and apply this node splitting operation to a given input bipartite graph (a real-world SAT formula) to decompose it into a set of trees. We then reverse the splitting sequence where our goal is to start with a set of trees (which describe SAT clause sizes) and learn to apply a set of node merging operations in order to arrive back to a realistic SAT formula. Our idea is to train a GCN-based classifier that decides which two nodes to merge next, based on the structure of the graph generated so far.

At graph generation time, we initialize G2SAT with a set of trees and iteratively sample random node pairs from the conditional distribution parametrized by the trained GCN model until user-specified

stopping criteria are met. We develop an efficient two-phase generation procedure, where in the node proposal phase, candidate node pairs are randomly drawn, and in the node merging phase, the learned GCN model is applied to select the most likely node pair to merge.

Our experiments demonstrate that G2SAT is able to generate formulas that highly resemble the input real-world SAT instances in many graph-theoretic properties such as modularity and the presence of scale-free structures, with 24% lower relative error compared with state-of-the-art methods. More importantly, G2SAT generates realistic formulas that capture hardness of real-world SAT formulas. In particular, we apply different SAT solvers to SAT formulas generated by G2SAT. We find that SAT solvers that specialize in real-world SAT instances consistently outperform solvers that specialize in random SAT instances on graphs generated by G2SAT. This suggests that G2SAT learns to generate SAT instances that are more similar to real-world instances than to random SAT instances. Moreover, results also suggest that we can use synthetic formulas to more reliably tune the hyper-parameters of SAT solvers, achieving 18% faster SAT solver run time on unseen formulas compared with tuning the model only on the available training SAT formulas.

2 Related Work

SAT Generators. Existing synthetic SAT generators are hand-crafted models that are typically designed to generate formulas that fit a particular graph statistic. The mainstream generators for realistic SAT instances include the Community Attachment (CA) model [15], which generates formulas with a given Variable-Incidence Graph modularity, and the Popularity-Similarity (PS) model [16], which generates formulas with a specific Variable-Clause Graph degree distribution. In addition, there are also generators for random k-SAT instances [6] and crafted instances that come from translations of structured combinatorial problems, such as graph coloring, graph isomorphism, and ramsey numbers [26]. Currently, all SAT generators are hand-crafted and machine learning provides an exciting alternative.

Deep Graph Generators. Existing deep generative models of graphs fall into two categories. In the first class are models that focus on generating perturbations of a given graph, by direct decoding from computed node embeddings [24] or latent variables [17], or learning the random walk distribution of a graph [5]. The second class comprises generative models that can learn to generate a graph by sequentially adding nodes and edges [27, 41, 40]. Domain specific generators for molecular graphs [21, 10] or 3D point cloud graphs [38] have been developed as well. However, current deep generative models of graphs do not readily apply to SAT instance generation. Thus, we develop a novel bipartite graph generator that respects all the constraints of graphical representations of SAT formulas and generates the formula graph as a sequence of node merging operations.

Deep learning over SAT instances. NeuroSAT also represents SAT formulas as graphs and computes node embeddings using GCNs [36]. However, NeuroSAT focuses on using the embeddings to solve SAT formulas, while we aim at generating SAT formulas.

3 Preliminaries

Goal of generating SAT formulas. Our goal is to design a SAT generator that, given a suite of SAT formulas, generates new SAT formulas with similar properties. Our aim is to capture not only graph theoretic properties, but also realistic SAT solver behavior. For example, if we train our G2SAT model on formulas from application domain X, then solvers that traditionally excel in solving problems in domain X, should also excel in solving G2SAT formulas (rather than, for example, solvers that specialize in solving random SAT formulas).

SAT formulas and their graph representations. A SAT formula ϕ is constructed by Boolean variables x_i and logical operators $\wedge$, $\vee$, and $\neg$. A SAT formula is satisfiable if there exists an assignment of Boolean values to the variables such that the overall formula evaluates to true. In this paper, we are concerned with formulas in Conjunctive Normal Form (CNF)[1], which have the form of conjunctions of disjunctions. Each disjunction is called a *clause* c_i, while a boolean variable x_i or its negation $\neg x_i$ is called a *literal* l_j. For example, $(x_1 \vee x_2 \vee \neg x_3) \wedge (\neg x_1 \vee \neg x_2)$ is a CNF formula with 2 clauses that can be satisfied by assigning true to x_1 and false to x_2.

[1] Any SAT formula can be converted to an equisatisfiable CNF formula in linear time [37].

Traditionally, researchers have explored 4 different graph representations for SAT formulas [4]: (1) *Literal-Clause Graph (LCG)*: all the literals and clauses are treated as nodes, the occurrence of a literal in a clause indicates an edge. An LCG is bipartite and there exists a bijection between CNF formulas and LCGs. (2) *Literal-Incidence Graph (LIG)*: only the literals are viewed as nodes, two literals have an edge if they co-occur in a clause. (3) *Variable-Clause Graph (VCG)*: obtained by merging the positive and negative literals of the same variables in an LCG. (4) *Variable-Incidence Graphs (VIG)*: obtained by performing the same literal merging operation on LIGs. In this paper, we use LCGs to represent SAT formulas.

LCGs as bipartite graphs. We represent a bipartite graph $G = (\mathcal{V}^G, \mathcal{E}^G)$ by its node set $\mathcal{V}^G = \{v_1^G, ..., v_n^G\}$ and edge set $\mathcal{E}^G = \{(v_i^G, v_j^G) | v_i^G, v_j^G \in \mathcal{V}^G\}$. In the rest of paper, we omit the superscript G whenever it is possible to do so. Nodes in a bipartite graph can be split into two disjoint partitions $\mathcal{V}_1$ and $\mathcal{V}_2$, $\mathcal{V} = \mathcal{V}_1 \cup \mathcal{V}_2$, where edges only exist between nodes in different partitions, i.e., $\mathcal{E} = \{(v_i, v_j) | v_i \in \mathcal{V}_1, v_i \in \mathcal{V}_2\}$. An LCG with n literals and m clauses has $\mathcal{V}_1 = \{l_1, ..., l_n\}$ and $\mathcal{V}_2 = \{c_1, ..., c_m\}$, where $\mathcal{V}_1$ and $\mathcal{V}_2$ are referred to as the literal partition and the clause partition, respectively. We may also write out l_i as x_j or $\neg x_j$ when explaining the literal sign is necessary.

Benefits of using LCGs to generate SAT formulas. While we choose to use LCGs because they are bijective to SAT formulas, one might argue that the LIG is also a viable alternative. Unlike LCGs, there are no explicit constraints over LIGs, thus previously developed general deep graph generators could in principle be applied. However, the ease of generating LIGs is compromised by the loss of key information in SAT formulas. In particular, given a pair of literals, an LIG only encodes whether they co-occur but fails to encode how many times and in which clauses they co-occur. It can be further shown that an LIG corresponds to a number of SAT formulas that is at least exponential to the number of 3-cliques in the LIG. This ambiguity severely restricts LIG's use in SAT generation.

4 G2SAT Framework

4.1 G2SAT: Generating Bipartite Graphs by Iterative Node Merging Operations

As discussed in Section 3, a SAT formula is uniquely represented by its LCG which is a bipartite graph. From the perspective of generative models, our primary objective is to learn a distribution $p_{model}(G)$ over bipartite graphs, based on a set of observed bipartite graphs $\mathbb{G}$ sampled from data distribution $p(G)$, where each bipartite graph $G \in \mathbb{G}$ may have a different number of nodes and edges. Due to the complex dependency between nodes and edges, directly learning $p(G)$ is very challenging. Therefore, we generate a graph via an n-step iterative process, $p(G) = \prod_{i=1}^{n} p(G_i | G_1, ..., G_{i-1})$, where G_i refers to an intermediate graph at step i in the iterative generation process. Since we focus on generating static graphs, we assume that the order of the generation trajectory does not matter, as long as the same graph is generated. This assumption implies the following Markov property over the conditional distribution, $p(G_i | G_1, ..., G_{i-1}) = p(G_i | G_{i-1})$.

The key to a successful iterative graph generative model is a proper instantiation of the conditional distribution $p(G_i | G_{i-1})$. Existing approaches [27, 40, 41] often model $p(G_i | G_{i-1})$ as the distribution over random addition of nodes or edges to G_{i-1}. While in theory this formulation enables generation of any kind of graphs, it cannot satisfy the hard partition constraint for bipartite graphs. In contrast, our proposed G2SAT has a simple generation process that guarantees to preserve the bipartite partition constraint, without hand-crafted generation rules or post-processing procedures. The G2SAT framework relies on node splitting and merging operations, which are defined as follows.

Definition 1. *The node splitting operation applied to node v is defined as removing some edges between node v and its neighboring nodes, and then connecting those edges to a new node u. The node merging operation over two nodes u, v is defined as removing all the edges between node v and its neighboring nodes, and then connecting those edges to node u. Formally, we have* $\mathrm{NodeSplit}(u, G)$ *that returns a tuple* (u, v, G')*, and* $\mathrm{NodeMerge}(u, v, G)$ *that returns a tuple* (u, G').

Note that by this definition, a node merging operation can always be reversed by a node splitting operation. The idea of G2SAT is then motivated by the following observation.

Observation 1. *A bipartite graph can always be transformed into a set of trees by a sequence of node splitting operations over nodes in one of the partitions.*

The proof of this claim is that the node splitting operation strictly reduces the node degree; therefore, iteratively applying node splitting over the nodes in a partition can make all the nodes in this partition to be of degree 1, which results in a set of trees (Figure 1, Left). This observation implies that a bipartite graph can always be generated via *a sequence of node merging operations*. In G2SAT, we always merge clause nodes in the clause partition $\mathcal{V}_2^{G_{i-1}}$ for a given graph G_{i-1}. We then design the following instantiation of $p(G_i|G_{i-1})$,

$$p(G_i|G_{i-1}) = p(\text{NodeMerge}(u, v, G_{i-1})|G_{i-1}) = \text{Multimonial}(\mathbf{h}_u^T \mathbf{h}_v / Z | \forall u, v \in \mathcal{V}_2^{G_{i-1}}) \quad (1)$$

where h_u, h_v are the embeddings for nodes u, v, Z is the normalizing constant to ensure the distribution $\text{Multimonial}(\cdot)$ is valid. We aim for embeddings $\mathbf{h}_u$ that would capture the multi-hop neighborhood structural information of a node u, while being computed from a single trainable model that can generalize across different generation stages and different graphs. Therefore, we use the GraphSAGE model [18] to compute node embeddings, which is a variant of GCNs that has been shown to have strong inductive learning capabilities across different graphs. Specifically, the l-th layer of GraphSAGE can be written as

$$\begin{aligned} \mathbf{n}_u^{(l)} &= \text{AGG}(\text{RELU}(\mathbf{Q}^{(l)} \mathbf{h}_v^{(l)} + \mathbf{q}^{(l)} | v \in N(u))) \\ \mathbf{h}_u^{(l+1)} &= \text{RELU}(\mathbf{W}^{(l)} \text{CONCAT}(\mathbf{h}_u^{(l)}, \mathbf{n}_u^{(l)})) \end{aligned} \quad (2)$$

where $\mathbf{h}_u^{(l)}$ is the l-th layer node embedding for node u, $N(u)$ is the local neighborhood of u, $\text{AGG}(\cdot)$ is an aggregation function such as mean pooling, and $\mathbf{Q}^{(l)}, \mathbf{q}^{(l)}, \mathbf{W}^{(l)}$ are trainable parameters. The input node features are length-3 one-hot vectors, which is used to represent the three node types in LCGs, i.e., positive literals, negative literals and clauses. In addition, since each literal and its negation are tightly related, we add an additional message passing path between them.

4.2 Scalable G2SAT with Two-phase Generation Scheme

LCGs can easily have tens of thousands of nodes, thus there are millions of candidate node pairs that could be merged. This makes the computation of normalizing constant Z in Equation 1 infeasible. To avoid this issue, we design a two-phase scheme to instantiate Equation 1, which includes a node proposal phase and a node merging phase (Figure 1, right). Intuitively, the idea is to first have a fixed oracle to propose random candidate node pairs, and then a model only needs to decide if the proposed node pair should be merged or not, which is much easier to learn compared with making decision from millions of candidate options. Instead of directly learning and sampling from $p(G_i|G_{i-1})$, we introduce additional random variables U and V to represent random nodes, and then learn the joint distribution $p(G_i, u, v|G_{i-1}) = p(u, v|G_{i-1}) p(G_i|G_{i-1}, u, v)$. Here, $p(u, v|G_{i-1})$ corresponds to the node proposal phase and $p(G_i|G_{i-1}, u, v)$ models the node merging phase.

In theory, $p(u, v|G_{i-1})$ can be any distribution as long as it has non-empty support. Since LCGs are inherently static graphs, there is little prior knowledge or side information on how this iterative generation process should proceed. Therefore, we implement the node proposal phase such that a random node pair is sampled from all candidate clause nodes uniformly at random. Then in the node merging phase, instead of computing dot product between all possible node pairs, the model only needs to compute dot product between the sampled node pairs. Specifically we have

$$\begin{aligned} p(G_i, u, v|G_{i-1}) =& p(u, v|G_{i-1}) p(\text{NodeMerge}(u, v, G_{i-1})|G_{i-1}, u, v) \\ =& \text{Uniform}(\{(u, v)|\forall u, v \in \mathcal{V}_2^{G_{i-1}}\}) \, \text{Bernoulli}(\sigma(\mathbf{h}_u^T \mathbf{h}_v)|u, v) \end{aligned} \quad (3)$$

where Uniform is the discrete uniform distribution, $\sigma(\cdot)$ is the sigmoid function.

4.3 G2SAT at Training Time

Using the proposed two-phase generation scheme described in Section 4.2, we essentially transformed the bipartite graph generation task into a binary classification task. We train the model to minimize the following binary cross entropy loss:

$$\mathcal{L} = -\mathbb{E}_{u,v \sim p_{pos}}[\log(\sigma(\mathbf{h}_u^T \mathbf{h}_v))] - \mathbb{E}_{u,v \sim p_{neg}}[\log(1 - \sigma(\mathbf{h}_u^T \mathbf{h}_v))] \quad (4)$$

where p_{pos}, p_{neg} are the distributions over positive and negative node pairs. We say a node pair is positive, if it should be merged as is observed in the training data, and vice versa. To acquire these

necessary training data from input bipartite graphs, we develop an algorithm that is summarized in Algorithm 1. Given an input bipartite graph G, we apply the node splitting operation to the graph for $n = |\mathcal{E}^G| - |\mathcal{V}_2^G|$ steps, which guarantees the input graph to be decomposed into a set of trees. Within each step, a random node s in partition $\mathcal{V}_2^{G_i}$ with degree greater than 1 is chosen to be split, where a random subset of edges that connect to s is chosen to connect to a new node. After the split operation, we get an updated graph G_{i-1}, as well as the split nodes u^{pos}, v^{pos}, which are treated as a positive node pair. Then, another node v^{neg} is randomly chosen from nodes in $\mathcal{V}_2^{G_{i-1}}$ other than u^{pos}, v^{pos}, and u^{pos}, v^{neg} are viewed as a negative node pair. The data tuple $(u^{pos}, v^{pos}, v^{neg}, G_{i-1})$ is saved into dataset $\mathcal{D}$. We also save step count n and graph G_0 as "graph templates", which are later used to initialize G2SAT at inference time. The procedure is repeated r times until desired number of data points are gathered. Finally, G2SAT is trained with the dataset $\mathcal{D}$ to minimize the objective listed in Equation 4.

Algorithm 1 G2SAT at training time

Input: Bipartite graphs $\mathcal{G}$, repeat time r
Output: Graph templates $\mathcal{T}$
$\mathcal{D} \leftarrow \varnothing, \mathcal{T} \leftarrow \varnothing$
for $k = 1, \ldots, r$ **do**
  $G \sim \mathcal{G}, n \leftarrow |\mathcal{E}^G| - |\mathcal{V}_2^G|, G_n \leftarrow G$
  for $i = n, \ldots, 1$ **do**
    $s \sim \{u | u \in \mathcal{V}_2^{G_i}, \text{Degree}(u) > 1\}$
    $(u^{pos}, v^{pos}, G_{i-1}) \leftarrow \text{NodeSplit}(s, G_i)$
    $v^{neg} \sim \mathcal{V}_2^{G_{i-1}} \setminus \{u^{pos}, v^{pos}\}$
    $\mathcal{D} \leftarrow \mathcal{D} \cup \{(u^{pos}, v^{pos}, v^{neg}, G_{i-1})\}$
  $\mathcal{T} \leftarrow \mathcal{T} \cup \{(G_0, n)\}$
Train G2SAT with $\mathcal{D}$ to minimize Eq. 4

Algorithm 2 G2SAT at inference time

Input: Graph templates $\mathcal{T}$, number of output graphs r, number of proposed node pairs o
Output: Generated bipartite graphs $\mathcal{G}$
for $k = 1, \ldots, r$ **do**
  $(G_0, n) \sim \mathcal{T}$
  for $i = 0, \ldots, n-1$ **do**
    $\mathcal{P} \leftarrow \varnothing$
    for $j = 1, \ldots, o$ **do**
      $u \sim \mathcal{V}_2^{G_i}$, $v \sim \{s | s \in \mathcal{V}_2^{G_i}, (s, x) \notin \mathcal{E}^{G_i}, (s, \neg x) \notin \mathcal{E}^{G_i}, \forall x \in N(u)\}$
      $\mathcal{P} = \mathcal{P} \cup \{(u, v)\}$
    $(u^{pos}, v^{pos}) \leftarrow \text{argmax}\{\mathbf{h}_u^T \mathbf{h}_v | (u, v) \in \mathcal{P}, \mathbf{h}_u = \text{GCN}(u), \mathbf{h}_v = \text{GCN}(v)\}$
    $G_{i+1} \leftarrow \text{NodeMerge}(u^{pos}, v^{pos}, G_i)$
  $\mathcal{G} = \mathcal{G} \cup \{G_n\}$

4.4 G2SAT at Inference Time

A trained G2SAT model can be used to generate graphs. We summarize the algorithm in Algorithm 2. At graph generation time, we first initialize G2SAT with a graph template sampled from $\mathcal{T}$ gathered at training time, which specifies the initial graph G_0 and the number of generation steps n. Note that G2SAT can in fact take bipartite graphs with arbitrary size as input and iterate for a variable number of steps. The reason we specify the initial state and the number of steps is to control the behavior of G2SAT and simplify the experiment setting.

At each generation step, we use the two-phase generation scheme mentioned in Section 4.2. At the node proposal phase, we additionally make sure the sampled node pair does not correspond to a vacuous clause, i.e., if u, v is a valid node pair, then $(v, x) \notin \mathcal{E}^{G_i}, (v, \neg x) \notin \mathcal{E}^{G_i}, \forall x \in N(u)$. We parallelize the algorithm by sampling o random node pair proposals at once and feeding them to the node merging phase. In the node merging phase, although following Equation 3 would allow us to sample from the true distribution, we find in practice that it usually requires sampling a large number of candidate nodes pairs until a positive node pair is predicted by GCN model. Therefore, we propose a greedy algorithm that selects the most likely node pair to be merged among the o proposed node pairs and merges the node. Admittedly, the generated graphs are biased from the true data distribution, however, we do find the generated graphs to be reasonable as revealed by our comprehensive experiments. After n steps, the generated graph G_n is saved as an output.

5 Experiments

5.1 Dataset and Evaluation

Dataset. We use 10 smallest real-world SAT formulas from the SATLIB [20] and past SAT competitions [1]. The two data sources contain SAT formulas generated from a variety of application domains, e.g., bounded model checking, planning, and cryptography. We use the standard SatElite preprocesser [11] to remove duplicate clauses and perform possible polynomial propagation. The preprocessed formulas contain 82 to 1122 variables and 327 to 4555 clauses.

Table 1: Graph statistics of generated formulas (mean ± std. (relative error to training formulas))

Method	VIG		VCG			LCG
	Clustering	Modularity	Variable α_v	Clause α_c	Modularity	Modularity
Training	0.50±0.07	0.58±0.09	3.57±1.08	4.53±1.09	0.74±0.06	0.63±0.05
CA	0.33±0.08(34%)	0.48±0.10(17%)	6.30±1.53(76%)	N/A	0.65±0.08(12%)	0.53±0.05(16%)
PS(T=0)	0.82±0.04(64%)	0.72±0.13(24%)	3.25±0.89(9%)	**4.70±1.59(4%)**	0.86±0.05(16%)	**0.64±0.05(2%)**
PS(T=1.5)	0.30±0.10(40%)	0.14±0.03(76%)	4.19±1.10(17%)	6.86±1.65(51%)	0.40±0.05(46%)	0.41±0.05(35%)
G2SAT	**0.41±0.09(18%)**	**0.54±0.11(7%)**	**3.57±1.08(0%)**	4.79±2.80(6%)	**0.68±0.07(8%)**	0.67±0.03(6%)

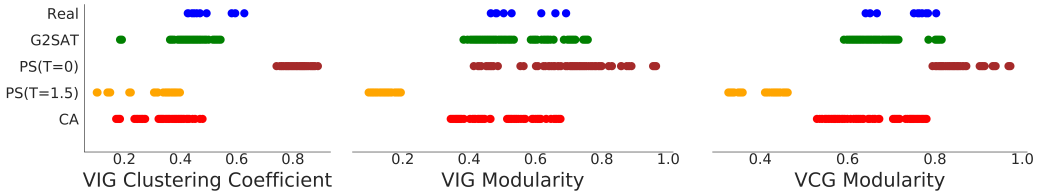

Figure 2: Scatter plots of distributions of selected properties of the generated formulas

We aim to evaluate if the generated SAT formulas preserve the properties of input training SAT formulas, in terms of graph statistics and SAT solver performance. We then evaluate if the generated SAT formulas can indeed help designing better domain-specific SAT solvers.

Graph statistics. We focus on the graph statistics studied previously in the SAT literature [32, 2]. In particular, we consider VIG, VCG and LCG representations of SAT formulas. We measure the modularity [31] (in VIG, VCG, LCG), average clustering coefficient [30] (in VIG) and the scale-free structure parameters as measured by variable α_v and clause α_c [2, 9] (in VCG).

SAT solver performance. We report the relative SAT solver performance, i.e., given k SAT solvers, we rank them based on both the performance over training SAT formulas and generated SAT formulas, and evaluate how well the two rankings align. Previous research have shown that SAT instances could be made hard in various post-processing approaches [35, 39, 13]. Therefore, the absolute hardness of the formulas is not a good indicator of how realistic the formulas are. On the other hand, it is not trivial for a post-processing procedure to precisely manipulate the relative performance over a set of SAT solvers. Therefore, we report the relative solver performance for fairer comparison. We took top-3 winning solvers from both the application track and the random track in the 2018 SAT competition [19], which are named as I_1, I_2, I_3, and R_1, R_2, R_3 respectively. As confirmed by our experimental results, solvers that focus on real-world SAT formulas (I_1, I_2, I_3) indeed outperform solvers that focus on random SAT formulas (R_1, R_2, R_3), over the training formulas; therefore, we measure if on the generated formulas, the application-focused solvers I can also correctly outperform random-focused solvers R, as measured by accuracy. All the run time performances are measured by wall clock time under careful experiment settings (detailed in Appendix).

Application: Developing better SAT solvers. Finally, we simulate the real application scenario, where people wish to use the synthetic formulas for developing better SAT solvers. Specifically, we use either the 10 training SAT formulas or the generated SAT formulas to guide the hyper-parameter selection of a popular SAT solver called Glucose [3, 12]. We conduct grid search over 2 of its hyper-parameters, i.e., the variable decay that influences the variable ordering of the search tree, and the clause decay that influences which learned clauses are to be removed [12], from range $[0.75, 0.85, 0.95]$ and $[0.7, 0.8, 0.9, 0.99, 0.999]$ respectively. We measure the run time of the SAT solvers using the optimal hyper-parameters found by grid search, over 22 real-world SAT formulas unobserved to any of the models. Since training SAT formulas are sparse, we expect using the abundant generated SAT formulas can lead to more robust hyper-parameters.

5.2 Models

We compare G2SAT with two state-of-the-art generators for real-world SAT formulas. Both generators are prescribed models designed to match a specific graph property. To properly generate formulas using these baselines, we set their arguments to match the corresponding statistics in the training set. We generate 200 formulas using G2SAT and the baseline models respectively.

Table 2: Relative SAT Solver Performance on training as well as synthetic SAT formulas

Method	Solver ranking	Accuracy
Training	$I_2, I_3, I_1, R_2, R_3, R_1$	100%
CA	$I_2, I_3, I_1, R_2, R_3, R_1$	**100%**
PS(T=0)	$R_3, I_3, R_2, I_2, I_1, R_1$	33%
PS(T=1.5)	$R_3, R_2, I_3, I_1, I_2, R_1$	33%
G2SAT	$I_1, I_2, I_3, R_2, R_3, R_1$	**100%**

Table 3: Performance gain when using generated SAT formulas to tune SAT solvers

Method	Best parameters	Runtime(s)	Gain
Training	(0.95, 0.9)	2679	N/A
CA	(0.75, 0.99)	2617	2.31%
PS(T=0)	(0.75, 0.999)	2668	0.41%
PS(T=1.5)	(0.95, 0.9)	2677	0.07%
G2SAT	(0.95, 0.99)	**2190**	**18.25%**

G2SAT. We implement G2SAT with a 3-layer GraphSAGE model using mean pooling and ReLU activation [18] with hidden and output embedding size of 32. We use Adam optimizer with learning rate 0.001 to train the model until validation accuracy plateaus.

Community Attachment (CA). The CA model generates formulas to fit a desired VIG modularity [15]. The output of the algorithm is a SAT formula with n variables and m clauses, each of length k, such that the optimal modularity for any c-partition of the VIG of the formula is approximately Q.

Popularity-Similarity (PS). The PS model generates formulas to fit desired α_v and α_c [16]. T is a hyper-parameter that decides the trade-off between modularity and α_v, α_c of generated formulas. We use two versions of PS, with $T = 0$ and $T = 1.5$.

5.3 Results

Graph statistics. The graph statistics of generated SAT formulas are shown in Table 1. We observe that G2SAT is the only model that is able to closely fit *all* the graph properties that we measure, whereas the baseline models can only fit some of the statistics and fail to perform well in the remaining statistics. Surprisingly, G2SAT can even fit the modularity better than CA, which is tailored at fitting the modularity. We compute the relative error over generated graph statistics with respect to the ground-truth statistics, and G2SAT can reduce the relative error by 24% on average compared with baseline methods. To further illustrate this performance gain, we plot the distribution of selected properties over the generated formulas in Figure 2, where each dot corresponds to a graph. It is shown that G2SAT can do nice interpolation and extrapolation on all the statistics of the input graphs, while the baselines can only do well on some of the statistics.

SAT solver performance. As is shown in Table 2, the ranking of solver performance over formulas generated by G2SAT and CA, as well as the ranking over training graphs, are able to nicely align. Notably, both models are able to correctly generate formulas on which application-focused solvers (I_1, I_2, I_3) outperform random-focused solvers (R_1, R_2, R_3), achieving 100% accuracy. By contrast, PS models almost completely fail in the task. Even more interestingly, our solver ranking aligns better with the 2018 SAT competition results compared with using the training SAT formulas.

Application: Developing better SAT solvers. The run time gain of tuning with synthetic formulas compared with tuning with a small set of real-world formulas is demonstrated in Table 3. While all the generators are able to improve the SAT solver's performance by suggesting different hyper-parameter configurations, astonishingly, G2SAT is the only method that can find one that achieves huge performance gain (18% faster run time) on unobserved SAT formulas. Although our experiment is rather small scale, the promising results indicate that G2SAT could have opened up the opportunities for developing better SAT solvers, even in application domains where benchmarks are sparse.

6 Conclusion

In this paper, we have introduced G2SAT, the first deep generative model for SAT formulas. In contrast to existing SAT formula generators, G2SAT does not rely on heavily hand-crafted algorithms and is able to generate diverse SAT formulas similar to input formulas, as measured by many graph statistics and SAT solver performance. We believe that G2SAT has great potential for helping the design of SAT solvers when SAT benchmarks are sparse, which is supported by our experiments.

References

[1] The international sat competitions web page. `http://www.satcompetition.org/`.

[2] C. Ansótegui, M. L. Bonet, and J. Levy. On the structure of industrial sat instances. In I. P. Gent, editor, *Principles and Practice of Constraint Programming - CP 2009*, pages 127–141, Berlin, Heidelberg, 2009. Springer Berlin Heidelberg.

[3] G. Audemard and L. Simon. Predicting learnt clauses quality in modern sat solvers. In *Proceedings of the 21st International Jont Conference on Artifical Intelligence*, IJCAI'09, pages 399–404, San Francisco, CA, USA, 2009. Morgan Kaufmann Publishers Inc.

[4] A. Biere, A. Biere, M. Heule, H. van Maaren, and T. Walsh. *Handbook of Satisfiability: Volume 185 Frontiers in Artificial Intelligence and Applications*. IOS Press, Amsterdam, The Netherlands, The Netherlands, 2009.

[5] A. Bojchevski, O. Shchur, D. Zügner, and S. Günnemann. NetGAN: Generating graphs via random walks. In J. Dy and A. Krause, editors, *Proceedings of the 35th International Conference on Machine Learning*, volume 80 of *Proceedings of Machine Learning Research*, pages 609–618, Stockholmsmässan, Stockholm Sweden, 10–15 Jul 2018. PMLR.

[6] Y. Boufkhad, O. Dubois, Y. Interian, and B. Selman. Regular random k-sat: Properties of balanced formulas. *J. Autom. Reason.*, 35(1-3):181–200, Oct. 2005.

[7] A. Braunstein, M. Mézard, and R. Zecchina. Survey propagation: An algorithm for satisfiability. *Random Struct. Algorithms*, 27:201–226, 2005.

[8] E. Clarke, A. Biere, R. Raimi, and Y. Zhu. Bounded model checking using satisfiability solving. *Form. Methods Syst. Des.*, 19(1):7–34, July 2001.

[9] A. Clauset, C. R. Shalizi, and M. E. J. Newman. Power-law distributions in empirical data. *SIAM Review*, 51:661–703, 2009.

[10] N. De Cao and T. Kipf. Molgan: An implicit generative model for small molecular graphs. *arXiv preprint arXiv:1805.11973*, 2018.

[11] N. Eén and A. Biere. Effective preprocessing in sat through variable and clause elimination. In F. Bacchus and T. Walsh, editors, *Theory and Applications of Satisfiability Testing*, pages 61–75, Berlin, Heidelberg, 2005. Springer Berlin Heidelberg.

[12] N. Eén and N. Sörensson. An extensible sat-solver. In E. Giunchiglia and A. Tacchella, editors, *Theory and Applications of Satisfiability Testing*, pages 502–518, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[13] G. Escamocher, B. O'Sullivan, and S. D. Prestwich. Generating difficult sat instances by preventing triangles. *CoRR*, abs/1903.03592, 2019.

[14] H. Ganzinger, G. Hagen, R. Nieuwenhuis, A. Oliveras, and C. Tinelli. Dpll(t): Fast decision procedures. In R. Alur and D. A. Peled, editors, *Computer Aided Verification*, pages 175–188, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

[15] J. Giráldez-Cru and J. Levy. A modularity-based random sat instances generator. In *Proceedings of the 24th International Conference on Artificial Intelligence*, IJCAI'15, pages 1952–1958. AAAI Press, 2015.

[16] J. Giráldez-Cru and J. Levy. Locality in random sat instances. In *Proceedings of the Twenty-Sixth International Joint Conference on Artificial Intelligence, IJCAI-17*, pages 638–644, 2017.

[17] A. Grover, A. Zweig, and S. Ermon. Graphite: Iterative generative modeling of graphs. *arXiv preprint arXiv:1803.10459*, 2018.

[18] W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. In *Advances in Neural Information Processing Systems*, 2017.

[19] J. M. J. . S. Heule, M J H. Proceedings of sat competition 2018 : Solver and benchmark descriptions.

[20] H. H. Hoos and T. Stützle. Satlib: An online resource for research on sat. pages 283–292. IOS Press, 2000.

[21] W. Jin, R. Barzilay, and T. Jaakkola. Junction tree variational autoencoder for molecular graph generation. *International Conference on Machine Learning*, 2018.

[22] G. Katsirelos and L. Simon. Eigenvector centrality in industrial sat instances. In M. Milano, editor, *Principles and Practice of Constraint Programming*, pages 348–356, Berlin, Heidelberg, 2012. Springer Berlin Heidelberg.

[23] H. Kautz and B. Selman. Planning as satisfiability. In *Proceedings of the 10th European Conference on Artificial Intelligence*, ECAI '92, pages 359–363, New York, NY, USA, 1992. John Wiley & Sons, Inc.

[24] T. N. Kipf and M. Welling. Variational graph auto-encoders. *arXiv preprint arXiv:1611.07308*, 2016.

[25] T. N. Kipf and M. Welling. Semi-supervised classification with graph convolutional networks. *International Conference on Learning Representations*, 2017.

[26] M. Lauria, J. Elffers, J. Nordström, and M. Vinyals. Cnfgen: A generator of crafted benchmarks. In S. Gaspers and T. Walsh, editors, *Theory and Applications of Satisfiability Testing – SAT 2017*, pages 464–473, Cham, 2017. Springer International Publishing.

[27] Y. Li, O. Vinyals, C. Dyer, R. Pascanu, and P. Battaglia. Learning deep generative models of graphs. *arXiv preprint arXiv:1803.03324*, 2018.

[28] J. Marques-Silva, I. Lynce, and S. Malik. Conflict-driven clause learning sat solvers. In *Handbook of Satisfiability*, 2009.

[29] N. Mull, D. J. Fremont, and S. A. Seshia. On the hardness of sat with community structure. In N. Creignou and D. Le Berre, editors, *Theory and Applications of Satisfiability Testing – SAT 2016*, pages 141–159, Cham, 2016. Springer International Publishing.

[30] M. E. Newman. Clustering and preferential attachment in growing networks. *Physical review E*, 64(2):025102, 2001.

[31] M. E. Newman. Modularity and community structure in networks. *Proceedings of the national academy of sciences*, 103(23):8577–8582, 2006.

[32] Z. Newsham, V. Ganesh, S. Fischmeister, G. Audemard, and L. Simon. Impact of community structure on sat solver performance. In C. Sinz and U. Egly, editors, *Theory and Applications of Satisfiability Testing – SAT 2014*, pages 252–268, Cham, 2014. Springer International Publishing.

[33] B. Selman, H. Kautz, and B. Cohen. Local search strategies for satisfiability testing. *Cliques, Coloring, and Satisfiability: Second DIMACS Implementation Challenge*, 26, 09 1999.

[34] B. Selman, H. Kautz, and D. McAllester. Ten challenges in propositional reasoning and search. In *Proceedings of the 15th International Joint Conference on Artificial Intelligence - Volume 1*, IJCAI'97, pages 50–54, San Francisco, CA, USA, 1997. Morgan Kaufmann Publishers Inc.

[35] B. Selman, D. G. Mitchell, and H. J. Levesque. Generating hard satisfiability problems. *Artif. Intell.*, 81:17–29, 1996.

[36] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill. Learning a sat solver from single-bit supervision. *arXiv preprint arXiv:1802.03685*, 2018.

[37] G. S. TSEITIN. On the complexity of derivation in propositional calculus. *Structures in Constructive Mathematics and Mathematical Logic*, pages 115–125, 1968.

[38] D. Valsesia, G. Fracastoro, and E. Magli. Learning localized generative models for 3d point clouds via graph convolution. *International Conference on Learning Representations*, 2019.

[39] K. Xu, F. Boussemart, F. Hemery, and C. Lecoutre. A simple model to generate hard satisfiable instances. In *Proceedings of the 19th International Joint Conference on Artificial Intelligence*, IJCAI'05, pages 337–342, San Francisco, CA, USA, 2005. Morgan Kaufmann Publishers Inc.

[40] J. You, B. Liu, R. Ying, V. Pande, and J. Leskovec. Graph convolutional policy network for goal-directed molecular graph generation. *Advances in Neural Information Processing Systems*, 2018.

[41] J. You, R. Ying, X. Ren, W. Hamilton, and J. Leskovec. GraphRNN: Generating realistic graphs with deep auto-regressive models. In *International Conference on Machine Learning*, 2018.