Learning to Generate Industrial SAT Instances*

Haoze Wu
haozewu@stanford.edu

Abstract

In this paper, we present SAT-GEN, the first implicit generative model of real-world SAT formulas. We break down the task of generating SAT formulas that resemble a real-world formula into two sub-tasks. The first is to model certain graph representation of the original formula and generated similar graphs using existing implicit graph modelling techniques. The second is to extract "reasonable" SAT formulas from the generated graphs. For the first task, instead of modelling the *Literal-Clause Graph* (LCG), a bipartite graph fully capturing a SAT formula, we choose to model the *Literal-Incidence Graph* (LIG), which is the *one-mode projection* of the LCG. Our second task, therefore is made specific to be, given a graph, generating a formula whose LIG is identical to the graph. We show that generating such formula is equivalent to finding a *minimal clique edge cover* of the given graph. We tackle this task efficiently using a *greedy hill-climbing* algorithm for the *minimum clique edge cover* (MCEC) problem. We verify experimentally that our approach generate formulas that closely resemble a given real-world formula not only in LIG-based properties, but in a wide range of important properties. To our knowledge, this is the first model that is able to do so.

1 Introduction

Conflict-driven clause learning (CDCL) SAT-solvers (Marques-Silva, Lynce, and Malik 2009) nowadays are able to solve large real-world instances of the *Propositional Boolean Satisfiability problem* (SAT) under time limits far below what theoretical estimation suggests. The further development, testing, and understanding of the performance of SAT-solvers benefit from a large amount of real-world instances. However, the number of real-world formulas is finite, and in many specific applications limited. Therefore, the design of generators of random SAT formulas that realistically capture features of real-world formulas is called for and has been identified as one of the ten challenges in propositional reasoning and search (Selman, Kautz, and McAllester 1997).

Traditionally, this problem has been formulated as one of modelling the graph representations of real-world SAT formulas. This direction is promising because past research shows that the graph representation of a industrial SAT formulas significantly differ from uniformly random formulas in features such as modularity and scale-free structures (Newsham et al. 2014; Ansótegui, Bonet, and Levy 2009). These differences are also used to explain the behavior of CDCL solvers.

All the previous work in this direction has been dedicated to developing prescribed models that can capture a subset of the desired properties (Giráldez-Cru and Levy 2015; Giráldez-Cru and Levy 2017). Despite the benefit of theoretical tractability, this approach has two disadvantages. First, it is questionable whether a hand-crafted model could capture all the essential characteristics of industrial SAT formulas. Second, there might be deep discrepancy between different families of industrial formulas (Katsirelos and Simon 2012), and a single prescribed model might not be able to account for such diversity.

As an alternative, implicit graph models present the promise that it could capture a wide range of essential (possibly yet unknown) graph-based features without specifically targeting at any one of them. Usually, an implicit graph model learns the graph topology by learning certain succinct representation of the graph (i.e., sets of random walks) (Leskovec et al. 2010; Bojchevski et al. 2018). This representation is in turn used to reconstruct graphs. Naturally, these implicit modelling techniques could be extended to the context of generating pseudo-industrial SAT formulas, though to our knowledge, no previous work has explored this direction.

In this paper, we leverage a powerful graph modelling technique (Bojchevski et al. 2018) to design the first implicit generative model of pseudo-industrial SAT formulas.

Concretely, to model certain real-world formula, our model first transforms it to its *Literal-Incidence Graph* (LIG). Then, following the method proposed by Bojchevski et al., we used a *Generative Adversarial Net* (GAN) (Goodfellow et al. 2014) to generate biased random walks that resemble the ones in the original LIG and synthesize new graphs based on the generated random walks.

Given a synthesized graph, our task then is to construct a "reasonable" formula whose LIG is identical to that graph. Using the fact that each clause in a SAT formula corresponds to a clique in the LIG of that formula, we extract a SAT formula from the generated graph by approximate a *minimum*

*The source code is available at `https://github.com/anwu1219/sat_gen/tree/map_lig`

clique edge cover using a greedy hill-climbing algorithm. Each clique in the edge cover corresponds to a clause in the resulting SAT-formula. Finally, the clique cover is expanded in a preservative manner until the number of cliques equals the number of clauses in the original formula, thus yielding a new formula with the desired number of clauses.

Our model is able to generate formulas that differ "in appearance" from the original formula, but share with it a wide range of graph-based properties, such as modularity, clustering coefficient, and scale-free structures.

Contributions

- **Contribution I:** We designed a pipeline for creating an implicit model of a real-world SAT formula ϕ: we first use learning techniques to model the LIG of ϕ and then synthesize SAT formulas based on the LIGs generated from the model.
- **Contribution II:**: We implemented the pipeline and created a pseudo-industrial SAT-formula generator, SAT-GEN, which takes as input a real formula, and generates formulas that mimics a wide range of properties of the input formula.
- **Contribution III:** We proposed an efficient method to extract a SAT formula from an arbitrary graph such that the LIG of the formula is identical to the graph. We show that this approach results formulas with desirable properties.

2 Preliminaries

Propositional Boolean Satisfiability problem (SAT): a SAT problem is a query over a Boolean formula, i.e., an expression that consists of Boolean variables connected by the fundamental Boolean operators "and", "or" and "not". The query asks whether there is an assignment of true/false values to the variables such that the overall formula evaluates to true.
Conjunctive Normal Form (CNF): a SAT formula in CNF is one that is in the form $C_1 \wedge C_2 \wedge \cdots \wedge C_n$. Each C_i, which shall be refered to as a *clause*, is a disjunction $(l_1 \vee l_2 \vee \cdots \vee l_k)$, where l_j is either a boolean variable or its negation. We refer to l_j as a *literal*. In short, a CNF formula is a conjunction of disjunctions. In this paper, we are only concerned with CNF formulas.
Graph Representation of SAT formulas: there are multiple ways to represent a SAT formula using graphs. In this paper we are concerned with four graphs:

- Literal-Clause Graph (LCG): variables and clauses both as nodes, occurrences of literals in clauses as edges. A LCG is bipartite and fully captures a SAT formula.
- Literal-Incidence Graph (LIG): literals as nodes, co-occurrences of two literals in a clause as edges. LIG is the one mode projection of literal nodes of the LCG;
- Variable-Clause Graph (VCG): variables and clauses both as nodes, occurrences of variables in clauses as edges;
- Variable-Incidence Graph (VIG): variables as nodes, co-occurrences of two variables in a clause as edges.

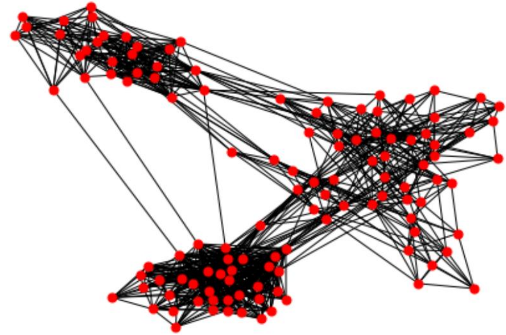

Figure 1: VIG of bmc-ibm-2

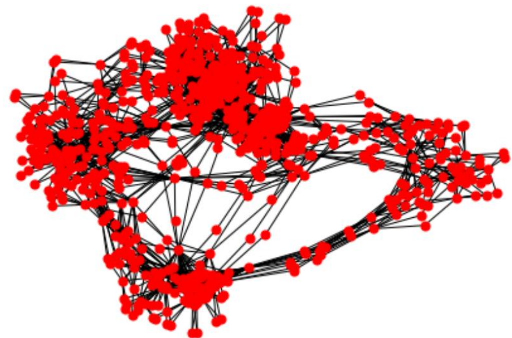

Figure 2: VCG of bmc-ibm-2

Graph-based properties of real-world SAT formulas: it has been shown that the VIGs and VCGs of real-world SAT formulas differ with those of uniformly random SAT formulas in a wide range of properties. While the VIG and VCG of random SAT formulas tend to have low modularity (around 0.3), those of real world SAT formulas tend to exhibit much stronger community structures (Newsham et al. 2014). Moreover, in the VCG of a real-world SAT formula, the degree distribution of variable nodes and that of clause nodes both tend to follow a power-law distribution (Ansótegui, Bonet, and Levy 2009).

Take a small benchmark *bmc-ibm-2* from SAT-LIB (Hoos and Stützle 2000) as an example. As demonstrated by Figure 1 and 2, community structures could be directly spotted both in the VIG and the VCG of this benchmark.

3 The SAT-GEN Model

Figure 3 provides a high-level overview of our generator. A formula is first mapped to its LIG, which is learned by an implicit graph model. The graph model produces new graphs interpreted as LIGs, from which new SAT formulas are extracted.

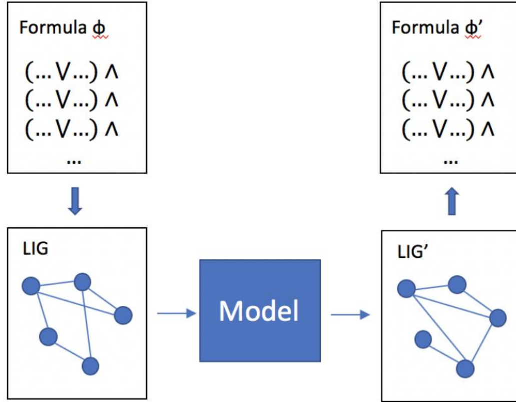

Figure 3: A high level overview of SAT-GEN

Generating Graphs via Biased Random Walks

To model the graph representation of the SAT formula, we have experimented with two implicit graph modelling techniques, the Kronfit algorithm (Leskovec et al. 2010) and the NetGAN algorithm (Bojchevski et al. 2018). We found that compared with graphs generated by Kronfit, graphs generated by NetGAN are significantly more similar to the real graph in our context.

NetGAN Bojchevski et al. formulated the problem of learning the graph topology as learning the distribution of biased random walks over the graph. In order to generate graphs that mimic some graph S with N nodes, the following four steps are performed:

1. Sample a set of biased random walks of fixed length T using a biased second-order random walk sampling strategy same as the one used in Node2Vec (Grover and Leskovec 2016).
2. Train a GAN, where the generator G is aimed to generate synthetic random walks that emulate those on S, and the discriminator D is aimed to distinguish the synthetic random walks from the real one.[1]
3. After the training finishes, sample a set of random walks with G, and construct a $N \times N$ score matrix M, where $M_{i,j}$ denotes the number of occurrences of transitions between i and j in the sampled random walks.
4. Sample edges without replacement, where the probability of an edge (i, j) being chosen is $\frac{M_{i,j}}{\sum_{u,v} M_{u,v}}$, until the desired amount of edges (i.e., as many as in S) is reached.

[1]We use the same architecture as in the original work, where both the generator and the discriminator use the Long Short-Term Memory architecture (Hochreiter and Schmidhuber 1997), and the training is conducted based on the Wasserstein GAN framework (Arjovsky, Chintala, and Bottou 2017). Instead of constructing our own model, We used the source code of NetGAN.

Stopping Criterion During the training, a graph is generated using the strategy described in step 3 and 4 periodically. The training is terminated if the edge-overlap between the generated graph and the original graph reached certain threshold e.

Post-processing the Score Matrix Since we chose to model the LIG, we require that in the generated graphs by NetGAN, no edge exists between nodes denoting conjugate literals: if an edge exists between literal l and $\bar{l}$ in the LIG, the formulas that has such LIG must contain a clause $(l \vee \bar{l} \vee ...)$, which is vacuously true. We must exclude clauses like these as our goal is to generate non-trivial formulas. Therefore, we post-process the score matrix produced in step 3 by setting scores between conjugate literals to 0.

As we shall discuss later, it is possible to conduct more extensive post-processing of the score matrix in order to enforce stronger properties of the generated SAT formulas. However, this is beyond the scope of this paper.

Why Learning LIG?

A natural choice of graph representation to learn is the LCG, as it fully captures the SAT formula. However, we argue that modelling LIG is a wiser choice.

A trade-off exists between the difficulty of modelling a graph representation, and the complexity of extracting a formula from the graph representation.

While it is relatively easy to map a LCG to a SAT formula (the neighbors of each clause node form a clause), learning the topology of a LCG is hard. We found that not only the graph modelling techniques that we tried fail to fully capture the bipartiteness of the LCG, the training time is also unaffordable for large formulas.

On the other hand, while it is easier for NetGAN to model LIG, it is not initially obvious how to extract a SAT formula from a LIG. However, by leveraging the prior knowledge about the structures of the generated LIG, we designed an efficient method to extract from a generated LIG formulas that have several desirable properties.

Extracting SAT Formulas from LIGs

Given a graph generated by NetGAN, our goal is to generate formulas whose LIGs are identical to it. Moreover, the generated formulas must contain the same number of clauses as the original formula.

The difficulty of achieving this goal is that a LIG, as a *one-mode projection* from LCG, only contains information about which literals occur in a same clause, but does not tell us exactly what are the clauses in the formula. For instance, both of the following two formulas have Figure 4 as their LIG: $\phi_1 = (v_1 \vee v_2 \vee \bar{v_3})$ and $\phi_2 = (v_1 \vee v_2) \wedge (v_2 \vee \bar{v_3}) \wedge (v_1 \vee \bar{v_3})$. Moreover, $\phi_3 = \phi_1 \wedge \phi_2$, and $\phi_4 = \phi 1 \wedge (v1)$ also share the same LIG as ϕ_1 and ϕ_2.

Despite this "curse of freedom", we could still design a principled way to generate reasonable SAT formulas from a LIG because we know a priori that any generated formulas of interests must have certain properties. In particular, the generated formula cannot have duplicated clauses, unit

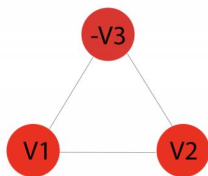

Figure 4: A simple LIG

clauses, or subsumable clauses. A clause C is subsumable if there is a shorter clause, C', in the formula, such that each literal in C' is in C. If C is subsumable, then removing C does not have any impact on the satisfiability of the formula. These three properties are reasonable to enforce in the generated formulas because the formulas that we train on have these properties. Fortunately, the problem of extracting from the generated graphs formulas with those properties is equivalent to finding minimal clique edge covers of the generated graphs.

Lemma 1. *The clauses of a SAT formula form a clique edge cover of its LIG.*

Proof. By the definition of LIG, there is an edge between any two literals in the same clause. Therefore, each clause corresponds to a clique in its LIG. The clique consists of nodes corresponding to the literals in the clause. Similarly, any edge (l_1, l_2) in the LIG must be covered by some clause that contains the two edges. □

Lemma 2. *A formula does not have duplicated clauses, unit clauses, or subsumable clauses if and only if its clauses form a minimal clique edge cover*[2] *of its LIG.*

Proof. Suppose the clauses form a minimal clique edge cover. Then the formula cannot have duplicated clauses or unit clauses, because removing those clauses do not reduce the number of covered edges. Neither can the formula have subsumable clauses, because removing the clauses that subsumes the subsumable clauses also does not redue the number of covered edges.

In the other direction. Suppose a formulas does not have those three kinds of clauses but is not a minimal clique edge cover. In other words, we could remove some clause without changing the number of edges in the LIG. This only possible if the removed clause is a unit clause (which does not correspond to any edges in the LIG), or a duplicated clauses, or a clause that subsumes some other clause, a contradiction. □

What we have seen so far is that the question of extracting a reasonable SAT formula from a LIG is equivalent to finding a minimal clique edge cover of the graph. However, not all minimal clique edge covers can be accepted as reasonable formulas. There are two further constraints. First, the number of clauses in the generated formulas must be equal to the original formula; second, the clause length distribution of the generated formulas should mimic that of the original formulas (recall that the clause degree often follow a power-law distribution), which suggests that instead of only having short clauses, the generated formulas should contain long clauses.

To find a minimal edge cover of a fixed size, it is easy to undershoot than to overshoot: if we have a smaller minimal clique edge cover than what is required, it is easy to expand it to a larger one of desired size. On the other hand, reducing a larger minimal edge cover to a smaller one is more computationally expensive.

Lemma 3. *In a SAT formula ϕ, for any clause C of length K ($K > 2$), there exists 3 clauses C_1, C_2, and C_3, each of length $K-1$, such that if we replace C with the conjunction of C_1, C_2, and C_3 in ϕ, the LIG of ϕ remains unchanged.*

Proof. We replace C with three of its sub-clauses of length $K-1$. Any two sub-clauses have $K-2$ literals in common. Without loss of generality, suppose literal l_1 is in C_1 and not in C_2, and literal l_2 is in C_2 and not in C_1. Let the set of literals shared by C_1 and C_2 be L. In other words, $C = L \cup \{l_1\} \cup \{l_2\}$. L itself forms a clique. Both l_1 and l_2 is connected to each node in L. Thus, to construct the clique corresponding to C, the only edge missing is (l_1, l_2).

This edge is created by C_3, since l_1 and l_2 must both be present in C_3. Otherwise C_3 would be identical to one of $C1$ and C_2. □

In order to find a minimal clique edge cover that contains both a targeted number of cliques and long clauses, it is sensible to start with a minimal clique edge cover as small as possible and expand it if necessary. While deciding the minimum clique edge cover of a graph is NP-complete, we could use an efficient technique to find relatively small clique edge covers. This technique is greedy hill-climbing.

Admittedly, at this point we could only justify the usage of this approach with intuitions and experimental results. Theoretical analysis about the clique size (a.k.a. clause length) distribution extracted using this approach is crucial but is left as future work.

Algorithm 1 describes the method to generate a formula with size n from a given graph G such that the formula's LIG is identical to G. Since the greedy hill-climbing algorithm operates over a set of cliques in G, we first enumerate the set of cliques in the original graph. While complete clique enumeration is again an NP-Complete problem, we found that real-world formulas rarely contain clauses larger than 15. In practice, a clique enumeration does not appear to be a runtime bottleneck. After all cliques of size below 15 in G are enumerated, greedy hill-climbing is conducted to approximate a minimum clique edge cover. Finally, this edge cover is expanded by repeatedly breaking down a clique chosen at uniform random in the way described in lemma 3, until the desired number of clauses is reached.

In the next subsection, we take a closer look at the greedy hill-climbing algorithm in our context.

[2]A clique edge cover S is minimal if and only if by removing any clique in S, S would not be an edge cover.

Algorithm 1 LIG to SAT Formula

```
1: procedure LIG2SAT(G, n)
2:     C ← enumerate_all_cliques(G)
3:     cover ← GHC(C, num_edges(G))
4:     return expand_to_n_clauses(cover, n_clauses)
5: end procedure
```

A Greedy Hill-Climbing Algorithm for Minimum Clique Edge Cover (MCEC)

Recall that the MCEC problem is the task of finding the smallest set of cliques in a given graph G, such that the union of the set of cliques is identical to G. A greedy hill-climbing algorithm takes in a set of cliques in the graph of interest, and repeatedly finds the clique that results in the largest marginal gain of edges, until all edges are covered. To expedite this process, we conduct lazy hill-climbing, where a dictionary mapping a clique to its marginal gain from previous iterations is kept updated and used to prevent redundant computation of marginal gains. Algorithm 2 is a sketch of the implementation of the lazy hill-climbing for MCEC.

Algorithm 2 Lazy Hill-Climbing for MCEC

```
 1: procedure LHC(C, n_edges)
 2:     cover ← ∅                      ▹ The set of chosen cliques
 3:     E ← ∅                          ▹ The set of covered edges
 4:     G ← {}         ▹ A dictionary of previous marginal gains
 5:     m ← ∞                        ▹ The previous marginal gain
 6:     while size(E) < n_edges do
 7:         clique, C, G, m ← LARGEST_GAIN(C, E, G, m)
 8:         E ← E ∪ edges(clique)
 9:         cover ← cover ∪ {clique}
10:     end while
11:     return cover
12: end procedure
13:
14: procedure LARGEST_GAIN(C, E, G, m)
15:     gain ← 0                   ▹ The maximal gain seen so far
16:     cur ← C.next         ▹ Iterating over the set of cliques
17:     best ← cur
18:     while !has_key(cur, G) or gain < G[cur] do
19:         new_gain = gain(cur, E)
20:         G[cur] ← new_gain
21:         if new_gain == 0 then
22:             remove(C, cur)
23:             continue
24:         end if
25:         if new_gain > gain then
26:             gain ← new_gain
27:             best_clique ← cur_clique
28:             if new_gain == m then break
29:             end if
30:         end if
31:         cur = C.next
32:     end while
33:     reorder(G, C)            ▹ Reordering C based on G
34:     return best, C, G, gain
35: end procedure
```

4 Experiment

In this section, we discuss in details the experiments we performed to evaluate SAT-GEN.

Dataset

We used the industrial and academic SAT benchmarks from the SAT-LIB (Hoos and Stützle 2000) and the past SAT competitions [3]. The two data sources contain thousands of SAT formulas generated for various purposes (e.g., bounded model checking, planning, cryptography).

We ran SAT-GEN on benchmarks of different applications and sizes. We used the SatElite preprocesser (Eén and Biere 2005) to remove subsumable, unit, and duplicates clauses. After pre-processing, we transformed a formula into its LIG and apply NetGAN to it. The number of nodes in the LIGs that we trained on ranges from 182 to 2244. The number of edges ranges from 919 to 12582.

Hyper-parameters tuning

As a rather complex artifact, SAT-GEN has multiple hyper-parameters. We found that most of them do not have significant impacts on the quality of generated formulas. The ones that matter the most are the stopping criterion and the random-walk strategy.

We set the stopping threshold e to be 75%. That is, the training is terminated when the generated graph and the original graph has 75% edge-overlap. One might question whether such a high edge-overlap threshold would yield any positive results trivial as they might simply be explained by the edge-overlap. As a sanity check, we measured the modularity of graphs generated in the following way: we first took the intersection between the original graph and a graph generated by SAT-GEN, and then added edges at uniform random to the intersection graph until it has the same number of edges as the original graph. We found that graphs generated in this way have much lower modularity than the original graph. This suggests that the GAN was not simply remembering edges in the original graph but actually learned deeper structures of it.

On the other hand, we observed that when the random walks are biased towards exploring local structures, NetGAN yields the optimal results. To enforce such bias, we set the return parameter p of the biased random walks to be 1 and the in-out parameter q to be 16 [4].

Evaluation

To evaluate the adequacy of SAT-GEN in a comprehensive manner, three kinds of experiments were conducted on the generated formulas. First, we measured the closeness between the graph-based properties of the generated formulas and the original formulas. Second, we measured the clause-overlap between the generated formulas and the original for-

[3] http://www.satcompetition.org/

[4] For details, see Grover and Leskovec (2016).

mulas. Finally, we evaluated the SAT-solver performance on the generated formulas.

Graph-based properties We mainly focused on the graph-based properties mentioned in previous literature as described in section 2. In particular, we measured modularity (of VIG, LIG, VCG, and LCG), scale-free structures (in VCG) and clustering coefficient (of LIG and VIG).

We used an implementation of the Louvain Algorithm [5] to measure the modularities (Blondel et al. 2008).

To measure whether a formula has scale-free structures, we must check whether the clause degrees and the variable degrees in the VCG respectively follow a power-law distribution. In other words, we must check whether there exists α_v and α_c, such that the expected number of variables with degree k in a VCG, $f_v^{real}(k)$, is approximately $ck^{-\alpha_v}$ and the expected number of clauses with length k in a VCG, $f_c^{real}(k)$, is approximately $ck^{-\alpha_c}$ (c is some normalizing factor). We used an implementation of the maximum likelihood method [6] for computing an estimation of α_v and α_c (Clauset, Shalizi, and Newman 2009). To evaluate the fit, we computed the distance d_{pow} between the cumulative function of f^{real} and the cumulative function of $ck^{-\alpha}$. In addition, we also measured whether the variable degree and the clause degree of a generated formula respectively follows a exponential distribution, by approximating two rate parameters, λ_v and λ_c, and computing the distances between the experimental and theoretical cumulative distribution functions, d_{exp}.

Following the metric in the previous work (Ansótegui, Bonet, and Levy 2009), we consider a formula to have scale-free structures if d^c_{pow} is less than both d^c_{exp} and 0.1, or d^v_{pow} is less than both d^v_{exp} and 0.1.

We also measured a wide range of other graph-based properties using the python *NetworkX* module (Schult 2008). However, for simplicity, in this paper, we only report the clustering coefficient of the LIG and the VIG in addition to modularity and scale-free structures. Since VCG and LCG are bi-partite, their clustering coefficients must be zero. Therefore, we omit those two metrics from the statistics.

Clause-overlap We measured the percentage of overlapping clauses, o_{gan}, between the formulas generated by SAT-GAN and their corresponding real-world formulas. This is to demonstrate that despite sharing deeper properties with the original formulas, the generated formulas are "apparently" different from the original ones.

We also measured the clause-overlap, o_{direct}, between formulas generated by directly applying greedy hill-climbing and cover-expansion on the LIG of the real-world formulas. We took a high o_{direct} as a sign that using greedy hill-climbing to extract SAT formulas is an adequate method. Moreover, for the same formula ϕ, if $o^{\phi}_{gan} << o^{\phi}_{direct}$, we could justified the usage of GAN by arguing that it contributes to the diversity of generated formulas.

SAT-Solver Performance Past experiments have shown that *local-search* SAT-solvers specialize in solving uniformly random SAT formulas, while CDCL SAT-solvers are better at solving industrial SAT formulas (Järvisalo et al. 2012). To examine whether this trend holds for the formulas generated by SAT-GEN, We compared the performance of the latest version of a local-search SAT-solver, *walksat* (Selman, Kautz, and Cohen 1999) and a CDCL SAT-solver *Minisat* (Eén and Sörensson 2004), both on the generated formulas and on uniformly random formulas of the same size. If walksat performs better on the random formulas and worse on the generated formulas, we would take this as an indicator that the generated formulas are realistic.

[5]https://github.com/taynaud/python-louvain

[6]http://www.iiia.csic.es/ levy/software/scalefree.cpp

Baselines

We compared SAT-GEN with two state-of-the-art pseudo-industrial SAT-formula generators. Both generators are prescribed models designed to match a specific property.

Community Attachment The Community Attachment (CA) model generates formula with a given VIG modularity (Giráldez-Cru and Levy 2015). The model takes in five inputs n, m, k, c, Q, where n is the desired number of variables, m the desired number of clauses, k the desired length of each clause, c the size of a partition of the VIG, and Q is the desired VIG modularity. The output of the algorithm is a SAT formula that has n variables, m clauses each of length k. The optimal modularity for any c-partition of the formula is approximately Q.

In order to use CA to generate formulas mimicking a real-world formula ϕ, we need to compute the statistics of the five values above in ϕ and use them as input to CA.

The Popularity-Similarity Model The Popularity-Similarity model (PS) generates formula with a given α_v and α_c (Giráldez-Cru and Levy 2017). In addition, the formulas generated by PS are guaranteed to have high modularity. The model takes in seven inputs $n, m, k, K, \alpha_v, \alpha_c, T$, where n is the desired number of variables, m the desired number of clauses, k the minimum clause length, K the average clause length, and T a hyper-parameter that reduces modularity at the cost of drifting away from scale-free structures. While the optimal value of T is different for different formulas, we found that $T = 0.5$ is adequate in most cases.

Remark Since SAT-GEN has a different flavor from any of the previous SAT-formula generators, we must take comparison between SAT-GEN, CA, and PS with a grain of salt. SAT-GEN is designed to mimic one specific formula, while CA and PS are designed to match one specific property.

5 Results

We first conduct a comprehensive case study on a relatively small benchmark from the SATLIB, namely *ssa2670-141*. Then we outline and discuss the experimental results for four other benchmarks.

Case study: A Circuit Fault Analysis Benchmark

Benchmark ssa2670-141 originally contains 986 variables and 2315 clauses. After pre-processing using SatElite, the benchmark contains 91 variables and 377 clauses. The clause length ranges from 2 to 8 and the average clause

	num clauses	VIG clust.	LIG clust.	VIG mod.	LIG mod.	VCG mod.	LCG mod.
ssa2670-141	377	0.582	0.351	0.520	0.559	0.647	0.584
SAT-GEN	**377.5**	0.513	**0.340**	0.477	**0.542**	**0.632**	0.537
PS (T=0)	273.75	0.818	0.616	0.678	0.732	0.832	**0.588**
PS (T=1.5)	352	**0.572**	0.464	**0.482**	0.584	0.731	0.552
CA	375	0.368	0.247	0.412	0.499	0.629	0.502

Table 1: Median Clustering coefficients and modularities of formulas generated to mimic benchmark 2670-141. For each property, the model with the closest statistics to the original formula is bolded.

	α_v	d^v_{pow}	λ_v	d^v_{exp}	α_c	d^c_{pow}	λ_c	d^c_{exp}
ssa2670-141	4.84	0.052	0.265	0.094	3.56	0.054	0.783	0.024
SAT-GEN	4.64	0.070	0.183	0.079	5.28	0.053	1.012	0.024
PS (T=0)	4.14	0.086	0.190	0.060	3.61	0.042	0.525	0.038
PS (T=1.5)	4.39	0.083	0.210	0.060	4.07	0.042	0.693	0.023
CA	7.06	0.078	0.394	0.068	-	-	-	-

Table 2: Most likelihood values of α and λ for a power-low and an exponential distribution for the formulas generated to mimic benchmark 2670-141.

length is about 3. The LIG of the benchmark contains 182 nodes and 1062 edges.

Graph-based properties We consider 4 models, SAT-GEN, PS with T = 0 and 1.5, and CA. For each model, we generated 100 formulas and computed their median clustering coefficient, modularity, and scale-free structures.

Table 1 summarizes the median clustering coefficient and modularity of the generated formulas.

Overall, SAT-GEN is the only generator that can capture all of the modularity and cluster coefficient statistics. Since SAT-GEN learns the LIG of the original formulas, it is not surprising that the formulas generated by SAT-GEN has similar LIG statistics as the original formula. What is interesting is that those formulas also have similar VIG, LCG, and VCG statistics as the original formula. Another noteworthy observation is that the original LIG has low clustering coefficient and high modularity, which is only captured by SAT-GEN and CA.

Table 2 summarizes the statistics for the scale-free structures of the generated formulas. The last four metrics are not applicable to CA because it can only generate formulas with uniform clause length.

In the original formula, while the variable degree follows a power-law distribution, the clause degree does not (as d^c_{pow} is smaller than d^c_{exp}). SAT-GEN is the only generator able to capture this property, while the variable degree in the formulas generated by PS fits a exponential distribution better.

In short, in terms of capturing the graph topology of a given SAT-formula, SAT-GEN significantly outperforms the other two generators on this benchmark.

Clause-overlap In this subsection, we show that while SAT-GEN is able to capture a wide range of graph-based properties, it also generates a diverse set of formulas.

The average clause-overlap between a formula generated by SAT-GEN and the original formula is only 14%. The average clause-overlap between two generated formulas is only about 12%. By contrast, if we generate formulas by applying greedy-hill climbing and cover-expansion directly on the LIG of the original graph, the average clause-overlap between a generated formula and the original formula is 40%. And the average clause-overlap between two generated formulas is 58%.

This suggests that a direct clique edge cover extraction on the original LIG fails to generate diverse formulas, which justifies the usage of GAN.

	SAT	Conflicts	Flips
ssa2670-141	0	166	-
SAT-GEN	0.3	6.72	268.5
PS (T=0)	0	2.27	-
PS (T=1.5)	0	2.15	-
CA	0.57	53.73	708

Table 3: Average proportion of Satisfiable instances, average number of conflicts, and average number of flips.

SAT-solver performance In this section we report the performance of a CDCL-solver, Minisat, and a local-search solver, walksat, on the generated formulas by SAT-GEN. Since the size of the formula is relatively small, instead measuring the runtime of the solver, we measured the number of conflicts generated by the Minisat, and the number of flips generated by walksat, which are both linearly correlated with the runtime.

Table 3 shows the SAT-solver performance on the formulas generated by the four models as well as the original formula. SAT-GEN and CA are able to generate both satisfiable and unsatisfiable formulas while the formulas generated by PS are always trivially unsatisfiable. In general, the generated formulas appear to be much easier than the original formula. Since comparing the number of conflicts and flips is itself meaningless, we compared their relative growth when the solvers are tasked to solve uniformly random formulas of the same size.

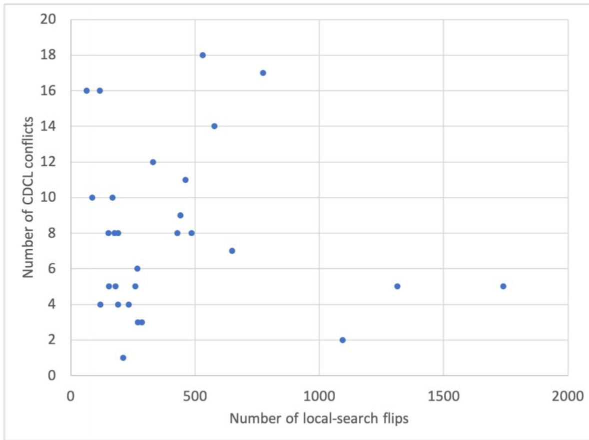

Figure 5: SAT-solver performance on generated formulas

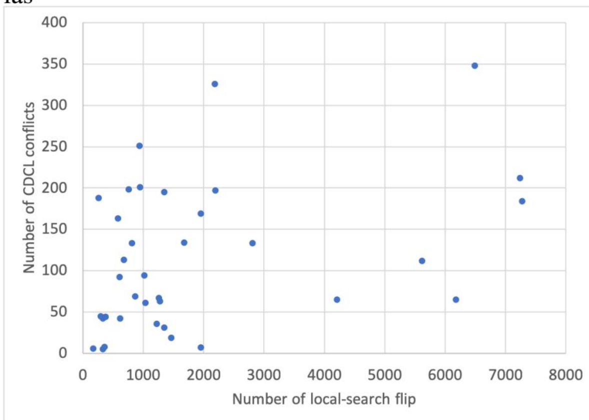

Figure 6: SAT-solver performance on uniformly random formulas

Figure 5 is the scatter plot of the number of CDCL conflicts and local-serarch flips for solving each formula generated by SAT-GEN. Figure 6 is the same plot for solving uniformly random 3-SAT formulas with the same number of variables and clauses as the original benchmark.

As demonstrated by the two plots, although for both solvers, random instances took longer to solve, the CDCL SAT-solver appears to struggle more. This is consistent with the observation that CDCL SAT-solvers are better at solving real-world formulas, and local-search SAT-solvers are better at solving uniformly random formulas. However, from the experimental results obtained so far, we could not conclude that the CDCL solver outperformed the local-search solver on the formulas generated by SAT-GEN.

Other benchmarks and Further Discussion

Graph-based properties Table 4 demonstrates the graph-based properties of formulas generated to mimic 4 other formulas. Here we only consider PS with T set to 0.5. We found that overall, SAT-GEN is able to closely capture a significantly wider range of graph-based properties than the other two methods.

The only property that SAT-GEN sometimes fails to capture is the power-law distribution of clause degrees. We found that in those cases the clause length distribution of a SAT-GEN formula often better fits an exponential distribution. This suggests that the way we extracted a SAT formula from a LIG has limitations. In particular, during the cover-expansion process, currently we are breaking down cliques selected at uniform random. This might be replaced by more sophisticated methods to enforce a power-law distribution of clique size in the resulting clique edge cover.

Clause-overlap We found that with the current setting, most formulas generated by SAT-GEN have low clause-overlap (below 20%) with the original formulas.

SAT-solver performance One difficulty yet to be tackled is that the formulas generated by SAT-GEN, though not trivially UNSAT as the ones generated by PS, are significantly easier than the original formulas. This makes comparison of Minisat and walksat difficult, as most instances can be solved within 0.01 seconds by both solvers, regardless of the size of the formula.

We have identified the cause of this problem: many generated formulas can be directly solved or reduced to much smaller problems by simply olving their binary constraints, which can be done in polynomial times. Therefore, to generate harder instances, we believe it might be necessary to enforce stronger properties of the 2-SAT sub-structure in the generated formulas. This might be achieved either during the cover-expansion process, or earlier, when a graph is synthesized from the score matrix.

6 Conclusions

In this paper, we introduced SAT-GEN, the first implicit generative model of real-world SAT formulas. In contrast to the previous pseudo-industrial SAT-formula generators such as the Community Attachment (CA) and the Popularity-Similarity model (PS), SAT-GEN is aimed to capture all the graph-based properties of a given SAT formula without targeting at any specific one.

At the highest level, SAT-GEN reads a real-world SAT-formula and generates random formulas that mimic it. Internally, SAT-GEN first transforms the formula to its LIG, then uses an implicit graph modelling technique, NetGAN, to generate realistic biased random walks on the LIG. Next, using the generated random walks, the model constructs new graphs that are interpreted as LIGs. Finally, running a greedy hill-climbing algorithm for minimum clique edge cover, followed by an cover-expansion process, SAT-GEN extracts SAT-formulas from the generated graphs.

We have shown that SAT-GEN captures a wider range of properties of real-world SAT formulas than any of the previous generators. In particular, properties that have been shown to be important, such as modularity and scale-free structures are captured.

We have encountered two difficulties. First, our way of extracting SAT formulas from an LIG sometimes fails to capture the power-law distribution of the clause degree. Second, the generated formulas are often much easier than the

Table 4: Graph-based properties of 4 other benchmarks from different families. The first one is a Multi-robot Path Planning problem, the second one Circuit Fault Analysis, the third one Bounded Model Checking, and the last one Bit Verification.

	num. vars	**num. clauses**	**VIG clust.**	**LIG clust.**	**VIG mod.**	**LIG mod.**	**VCG mod.**	**LCG mod.**
mrpp4_4#4_5	309	2517.0	0.428	0.357	0.468	0.520	0.783	0.716
SAT-GEN	309	2517.45	0.397	0.331	0.433	0.524	0.695	0.607
PS	309	2361.45	0.658	0.579	0.439	0.526	0.755	0.592
CA	309	2515.50	0.234	0.120	0.401	0.454	0.601	0.525

	α_v	d^v_{pow}	λ_v	d^v_{exp}	α_c	d^c_{pow}	λ_c	d^c_{exp}
mrpp4_4#4_5	2.591	0.092	0.056	0.050	5.796	0.092	1.369	0.059
SAT-GEN	2.374	0.179	0.048	0.082	8.292	0.058	1.549	0.027
PS	2.489	0.052	0.038	0.125	3.971	0.023	0.466	0.029
CA	6.072	0.132	0.198	0.107				

	num. vars	**num. clauses**	**VIG clust.**	**LIG clust.**	**VIG mod.**	**LIG mod.**	**VCG mod.**	**LCG mod.**
bf0432-007	473	2038.0	0.493	0.327	0.666	0.784	0.766	0.763
SAT-GEN	473	2038.6	0.417	0.342	0.604	0.772	0.758	0.674
PS	473	1887.5	0.562	0.467	0.718	0.776	0.860	0.624
CA	473	2036.5	0.249	0.177	0.614	0.649	0.746	0.561

	α_v	d^v_{pow}	λ_v	d^v_{exp}	α_c	d^c_{pow}	λ_c	d^c_{exp}
bf0432-007	4.061	0.067	0.178	0.070	3.616	0.088	0.725	0.096
SAT-GEN	3.989	0.051	0.156	0.050	6.265	0.050	1.007	0.035
PS	3.602	0.069	0.154	0.045	4.210	0.030	0.704	0.024
CA	6.367	0.090	0.350	0.067	-	-	-	-

	num. vars	**num. clauses**	**VIG clust.**	**LIG clust.**	**VIG mod.**	**LIG mod.**	**VCG mod.**	**LCG mod.**
bmc-ibm-7	860	4797.0	0.609	0.341	0.715	0.719	0.782	0.715
SAT-GEN	860	4797.4	0.470	0.321	0.671	0.707	0.754	0.649
PS	860	4324.5	0.636	0.535	0.646	0.718	0.858	0.633
CA	860	4796.6	0.185	0.119	0.700	0.711	0.762	0.610

	α_v	d^v_{pow}	λ_v	d^v_{exp}	α_c	d^c_{pow}	λ_c	d^c_{exp}
bmc-ibm-7	2.796	0.064	0.066	0.092	3.216	0.035	0.669	0.042
SAT-GEN	2.734	0.029	0.066	0.118	9.440	0.036	1.128	0.019
PS	2.705	0.050	0.075	0.095	3.781	0.025	0.693	0.034
CA	5.530	0.126	0.252	0.098				

	num. vars	**num. clauses**	**VIG clust.**	**LIG clust.**	**VIG mod.**	**LIG mod.**	**VCG mod.**	**LCG mod.**
countbitsrotate016	1122	4555	0.469	0.421	0.688	0.673	0.796	0.690
SAT-GEN	1122	4555.564	0.302	0.272	0.563	0.618	0.736	0.607
PS	1122	4276.5	0.504	0.426	0.832	0.869	0.912	0.663
CA	1122	4554.5	0.167	0.154	0.668	0.683	0.768	0.543

	α_v	d^v_{pow}	λ_v	d^v_{exp}	α_c	d^c_{pow}	λ_c	d^c_{exp}
countbitsrotate016	5.158	0.103	0.334	0.124	5.168	0.092	1.737	0.029
SAT-GEN	4.518	0.070	0.204	0.114	6.692	0.069	1.846	0.007
PS	4.126	0.076	0.210	0.027	6.021	0.033	0.867	0.010
CA	5.684	0.102	0.336	0.075	-	-	-	-

real-world formulas. We have discussed potential ways to tackle these two problems, which will be left as future work.

Nevertheless, we believe that our preliminary exploration of a graph-based implicit SAT-formula generator shows that it is a promising direction to explore.

7 Acknowledgements

This project extends from a summer research project under the supervision of Dr. Raghuram Ramanujan. In that project, we used NetGAN to directly learn the LCG of a real formula.

All the work related to learning LIG and extracting formulas from it, were conducted for the purpose of this course. We would like to thank Dr. Raghuram Ramanujan for providing helpful advice when this project was being conducted.

References

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

[Arjovsky, Chintala, and Bottou 2017] Arjovsky, M.; Chintala, S.; and Bottou, L. 2017. Wasserstein generative adversarial networks. In *Proceedings of the 34th International Conference on Machine Learning, ICML 2017, Sydney, NSW, Australia, 6-11 August 2017*, 214–223.

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

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

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

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

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

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

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

[Goodfellow et al. 2014] Goodfellow, I. J.; Pouget-Abadie, J.; Mirza, M.; Xu, B.; Warde-Farley, D.; Ozair, S.; Courville, A.; and Bengio, Y. 2014. Generative adversarial nets. In *Proceedings of the 27th International Conference on Neural Information Processing Systems - Volume 2*, NIPS'14, 2672–2680. Cambridge, MA, USA: MIT Press.

[Grover and Leskovec 2016] Grover, A., and Leskovec, J. 2016. Node2vec: Scalable feature learning for networks. In *Proceedings of the 22Nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD '16, 855–864. New York, NY, USA: ACM.

[Hochreiter and Schmidhuber 1997] Hochreiter, S., and Schmidhuber, J. 1997. Long short-term memory. *Neural Comput.* 9(8):1735–1780.

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

[Järvisalo et al. 2012] Järvisalo, M.; Berre, D. L.; Roussel, O.; and Simon, L. 2012. The international sat solver competitions. *AI Magazine* 33.

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

[Leskovec et al. 2010] Leskovec, J.; Chakrabarti, D.; Kleinberg, J.; Faloutsos, C.; and Ghahramani, Z. 2010. Kronecker graphs: An approach to modeling networks. *J. Mach. Learn. Res.* 11:985–1042.

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

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

[Schult 2008] Schult, D. A. 2008. Exploring network structure, dynamics, and function using networkx. In *In Proceedings of the 7th Python in Science Conference (SciPy*, 11–15.

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

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