# SATGL: an Open-source Graph Learning Toolkit for Boolean Satisfiability

HongTao Cheng[1,†], Jiawei Liu[1,†], Jianwang Zhai[1], Mingyu Zhao[1], Cheng Yang[1], Chuan Shi[1,✉]
[1]Beijing University of Posts and Telecommunications, Beijing, China
Email: †chenghongtao@bupt.edu.cn, †liu_jiawei@bupt.edu.cn, ✉shichuan@bupt.edu.cn

***Abstract*—As the first proven NP-complete problem, the Boolean Satisfiability (SAT) problem holds significant theoretical value and has wide-ranging practical applications. It has also led to the development of numerous SAT-related tasks, such as MaxSAT and UNSAT Core prediction. Due to the high complexity of handling these SAT-related tasks and the natural conversion of SAT formulas into graph structures, researchers have recently developed various graph learning methods to assist in prediction. However, these methods are often experimented on different datasets, with different approaches and different tasks, making it challenging to conduct unified evaluations and develop new algorithms. In this paper, we introduce the SATGL toolkit, the first open-source graph learning toolkit for the SAT problem. We expect SATGL to contribute to the advancement of artificial intelligence (AI) for SAT, facilitating SAT solving and new algorithm design.**

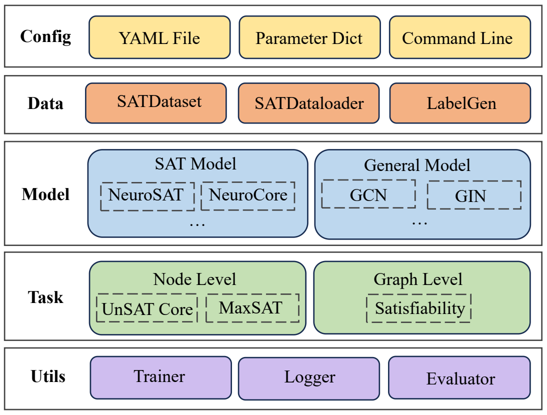

Fig. 1 Overall framework of SATGL Toolkit.

## I. Introduction

The SAT problem is a fundamental conundrum in computer science. It is the first problem to be formally proven as NP-complete. The SAT problem is highly valuable because it can represent many combinatorial problems, making it a crucial part of computational complexity theory. The relevance of SAT problem goes beyond its theoretical foundations and extends to practical applications in various domains [1]. For example, the technology's utility can be extended to various EDA scenarios, including test pattern generation, hardware verification, and scheduling [2]. The objective of the SAT problem is to determine whether a Boolean formula can be satisfied by assigning variables. Moving beyond the classical SAT, the maximum satisfiability (MaxSAT) problem is a generalization of SAT, seeking to maximize the satisfaction of clauses within propositional formulas. Moreover, the unsatisfiable core (UNSAT Core) prediction focuses on unsatisfiable formulas, aiming to identify minimal subsets of clauses that render the formula unsatisfiable.

There have been many efforts in these fields, and numerous solvers have been developed in recent years. Most SAT solvers are based on a combination of search methods and heuristic strategies to accelerate the solving process. For example, CaDiCal [3] is based on the CDCL [4] framework and applies many heuristics to search for satisfactory assignments for SAT problems. Open-WBO [5] is an open source MaxSAT solver. PicoSAT [6] is a flexible solver that seeks UNSAT Cores of SAT formulas. However, traditional solvers often require a lot of expert knowledge and complex handcrafted heuristic strategies.

Thanks to the development of AI, researchers have explored the use of deep learning in SAT problems in recent years. One of the most representative approaches is based on graph learning, in particular graph neural networks (GNNs). This is because SAT formulas can naturally be viewed as graphs, where the variables and clauses of the formulas are treated as nodes or edges in the graph [7]. As a pioneering work, NeuroSAT [2] explores GNN approaches to determine whether SAT instances are satisfiable. In addition to SAT solving, GNNs also provide opportunities for predicting other SAT-related tasks, such as MaxSAT [8] and UNSAT Core prediction [9]. These studies show that GNNs can learn from SAT problem instances and solve SAT problems to some extent, and are likely to contribute to the advancement of future SAT-solving techniques.

Although these works share a similar framework and have achieved good performance in their respective experimental settings, it is challenging to re-implement them and evaluate them uniformly due to differences in the evaluation datasets and tasks. This hinders the development of GNNs for the SAT field to some extent. We believe that a toolkit is needed to better evaluate and apply these models.

To evaluate, apply, and improve existing work in a unified manner, we propose SATGL, a graph learning toolkit for boolean satisfiability. As shown in TABLE I, SATGL offers diverse support at various steps. It not only integrates over a dozen GNN models but also provides the most commonly

† Both authors contributed equally to this research.
✉ Corresponding author.

used four graph construction methods, and three SAT tasks. Additionally, as illustrated in Fig. 1, `SATGL` exhibits a high degree of modularity, which is not only user-friendly but also highly scalable. To the best of our knowledge, `SATGL` is the first graph learning toolkit for multiple SAT datasets, diverse graph construction methods, various models, and multiple SAT-related tasks. The code can be found on this website[1].

The contribution of `SATGL` can be summarized as follows:

- Firstly, we implement various GNN-based SAT models and provide users with convenient and easy-to-use interfaces. Users can easily run experiments on specific datasets and tasks with just a few lines of Python code using `SATGL`.
- Secondly, we design various GNN paradigms for different graphs and evaluate the models' performance on different graphs.
- Thirdly, we evaluate several GNN models on several datasets with different distributions to explore the impact of different model and graph construction methods on the learning ability of GNN models.

## II. Related Work

**Graph Neural Networks.** Graph Neural Networks (GNNs) have emerged as powerful tools for modeling complex interactions within graph-structured data, showcasing remarkable performance across various graph-related tasks. GCN [10] pioneers the application of convolutional operations on graph-structured data, enabling effective information propagation and feature extraction. GraphSage [11] extends the capabilities of GCNs by introducing a more scalable and flexible framework for neighborhood aggregation. GIN [12] adopts a different approach by focusing on learning permutation-invariant node representations. In our work, we focus on GNNs for learning in the context of SAT problems. Specifically, we use `SATGL` to learn and optimize logical rules represented as graphs.

**Learning-based SAT solvers.** The SAT problem, recognized as NP-complete, revolves around ascertaining the feasibility of satisfying a given Boolean formula by assigning truth values to its variables. Traditional SAT solvers rely heavily on heuristic search algorithms, which can be problematic when dealing with complex scenarios or large problem sets. With the development of deep learning, learning-based methods have been applied to solve SAT problems. Since SAT problems can naturally be modeled as graphs, many works have successfully applied GNNs to SAT solving with promising results. For example, NeuroSAT [2] trains an end-to-end GNN to predict the satisfiability of a given SAT problem. GMS [8] treats MaxSAT as a binary classification task for nodes. NeuroCore [13] learns to predict the presence of clauses in the unsatisfiable core. Our toolkit focuses on these three tasks to evaluate the performance of various models.

**Open-source EDA Tools.** Open-source electronic design automation (EDA) toolchains have drawn growing attention recently due to the promise of unleashing innovation and lowering costs in chip design. A variety of open-source EDA projects span across the design flow, from high-level synthesis to physical implementation and verification. For the entire EDA pipeline, OpenROAD [14] provides an open-source end-to-end silicon compiler, spanning from logic synthesis to routing. iEDA [15] builds the infrastructure for core EDA technologies. On the front-end design side, High-Level Synthesis (HLS) compiles high-level specifications into Register-Transfer Level (RTL) descriptions, which are further synthesized by back-end tools. Bambu [16] is an open-source HLS research framework, which takes compiler Intermediate Representations (IRs) as input. RTLLM [17] provides benchmarks for evaluating LLM-generated RTL design quality. These two works demonstrate the needs and opportunities of applying AI to front-end designs. On the back-end design side, CircuitNet [18] and CircuitNet2.0 [19] provide realistic chip design datasets for back-end machine learning tasks. The iPD [20] toolchain covers the entire physical design flow from floorplanning to routing. For formal verification, MEC [21] reduces equivalence checking time cost after technology mapping. However, there is no available toolkit for AI algorithms on the fundamental SAT problem. To bridge this gap, we develop the `SATGL` toolkit, targeting SAT solving and new AI algorithm design via graph learning. It offers the first toolkit and benchmarks focused on AI for SAT problems. By open-sourcing `SATGL`, we expect it to greatly facilitate the advancement of AI in the boolean satisfiability domain and its application in fields such as EDA.

TABLE I The supported tasks and graphs of `SATGL`

| Model | Graph Type | Task |
|---|---|---|
| NeuroSAT | LCG | Satisfiability<br>MaxSAT<br>UNSAT Core |
| NeuroCore | LCG | Satisfiability<br>MaxSAT<br>UNSAT Core |
| NlocalSAT | LCG | Satisfiability<br>MaxSAT<br>UNSAT Core |
| QuerySAT | LCG | Satisfiability<br>MaxSAT<br>UNSAT Core |
| GMS | LCG | Satisfiability<br>MaxSAT<br>UNSAT Core |
| General GNNs | LCG<br>VCG<br>LIG<br>VIG | Satisfiability<br>MaxSAT<br>UNSAT Core |

## III. Preliminaries

**SAT Problem.** The SAT problem is represented as a Boolean expression composed of Boolean variables and logical operators. For consistency, such Boolean expressions are typically represented in Conjunctive Normal Form (CNF), which is a conjunction of clauses. Each clause is a disjunction

[1]https://github.com/BUPT-GAMMA/SATGL

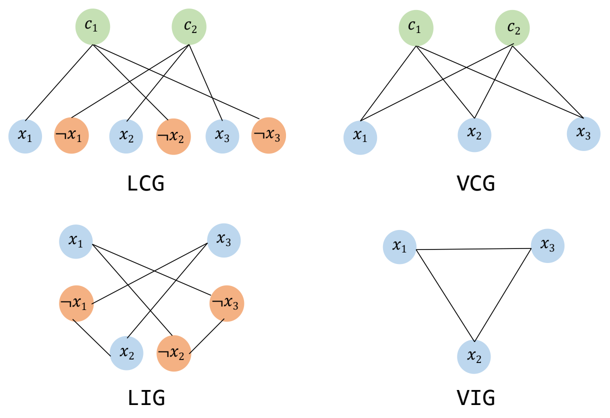

Fig. 2 The four common graph construction of formula $(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3)$.

of literals, where literals can be variables or their negations. Formally, given a CNF $F$ with $n$ variables $\{x_1, x_2, \ldots, x_n\}$ and $m$ clauses $\{C_1, C_2, \ldots, C_m\}$, $F$ can be represented as $C_1 \wedge C_2 \wedge \ldots \wedge C_m$. An SAT example expressed in CNF is:

$$(x_1 \vee \neg x_2 \vee \neg x_3) \wedge (\neg x_1 \vee x_2 \vee x_3).$$

In this example $C_1 = x_1 \vee \neg x_2 \vee \neg x_3, C_2 = \neg x_1 \vee x_2 \vee x_3$. For a CNF formula $F$, the SAT problem is to find a boolean assignment for each variable such that all clauses are satisfied (there exists an assignment such that all clauses are true). For the above formula, the assignment of all variables to 1 is one of the satisfied solutions. Typically, to model the complex interactions between variables and clauses, graph-based data structures are often employed. Commonly used graph construction methods include Literal-Clause Graph (LCG), Literal-Interaction Graph (LIG), Variable-Clause Graph (VCG) and Variable Interaction Graph (VIG). In LCG, nodes represent literals (variables or their negations), and edges represent that the literals appear in clauses. Similar to LCG, VCG nodes represent variables, and edges represent that the variables appear in clauses. Nodes in LIG are literals and edges represent their occurrences in clauses, while in VIG nodes are variables and edges represent their occurrences in clauses. Fig. 2 shows the examples of four types of graphs.

**GNN Paradigm.** GNN is a branch of deep learning that has been developing fast in recent years, with an excellent ability to solve tasks related to abstract graph structures. The paradigm can be described as follows: given a graph $G = (V, E)$, where $V$ denotes the set of nodes and $E \subseteq V \times V$ represents the set of edges, GNN models take $G$ as input and encode each node $v$ as an initial embedding vector $h_v^{(0)}$. The core of GNN lies in its message-passing mechanism, which iteratively updates node embeddings by aggregating information from neighboring nodes. Formally, the operation

TABLE II The paradigm of four graphs of general GNNs

| Graph | Paradigm |
|---|---|
| LCG | $m_l^{(k)} = \text{AGG}^{(k)}\left(MLP(\{h_c^{(k-1)})\vert c \in N(l)\}\right)$ |
| | $m_c^{(k)} = \text{AGG}^{(k)}\left(MLP(\{h_l^{(k-1)})\vert l \in N(c)\}\right)$ |
| | $h_l^{(k)} = \text{UPD}_l^{(k)}\left(h_l^{(k-1)}, FLIP(h_l^{(k-1)}), m_l^{(k)}\right)$ |
| | $h_c^{(k)} = \text{UPD}_c^{(k)}\left(h_c^{(k-1)}, m_c^{(k)}\right)$ |
| VCG | $m_v^{(k)} = \text{AGG}^{(k)}\left(MLP(\{h_c^{(k-1)})\vert c \in N(v)\}\right)$ |
| | $m_c^{(k)} = \text{AGG}^{(k)}\left(MLP(\{h_v^{(k-1)})\vert v \in N(c)\}\right)$ |
| | $h_v^{(k)} = \text{UPD}_v^{(k)}\left(h_v^{(k-1)}, m_v^{(k)}\right)$ |
| | $h_c^{(k)} = \text{UPD}_c^{(k)}\left(h_c^{(k-1)}, m_c^{(k)}\right)$ |
| LIG | $m_l^{(k)} = \text{AGG}^{(k)}\left(MLP(\{h_{l'}^{(k-1)})\vert l' \in N(l)\}\right)$ |
| | $h_l^{(k)} = \text{UPD}_l^{(k)}\left(h_l^{(k-1)}, FLIP(h_l^{(k-1)}), m_l^{(k)}\right)$ |
| VIG | $m_v^{(k)} = \text{AGG}^{(k)}\left(MLP(\{h_{v'}^{(k-1)})\vert v' \in N(v)\}\right)$ |
| | $h_v^{(k)} = \text{UPD}_v^{(k)}\left(h_v^{(k-1)}, m_v^{(k)}\right)$ |

* $l$, $c$, $v$ represents literal, clause and variable nodes. FLIP($h_l^k$) returns the embedding of corresponding negation $h_{\neg l}^k$.

at the $k$-th iteration (layer) of GNNs can be defined as follows:

$$m_v^{(k)} = \text{AGG}^{(k)}\left(\{h_u^{(k-1)} \vert u \in N(v)\}\right), \tag{1}$$

$$h_v^{(k)} = \text{UPD}_v^{(k)}\left(h_v^{(k-1)}, m_v^{(k)}\right), \tag{2}$$

where $h_v^{(k)}$ represents the embedding vector of node $v$ after the $k$-th iteration. In the aggregation step, the message $m_v^{(k)}$ is computed for each node by aggregating embeddings from its neighbors $N(v)$. Subsequently, in the updating step, the embedding of each node is updated by incorporating the aggregated message. After $K$ iterations, the final embedding $h_v^{(T)}$ for each node is obtained. As a pioneering work, NeuroSAT first applied the GNNs paradigm to LCG graphs, and many subsequent works followed a similar paradigm, and inspired by them, different GNN paradigms were designed for our different graphs. In SATGL, we extend general GNNs to use the four graphs mentioned above. The message-passing and update paradigm follow TABLE II. Then, the node embeddings are used for various downstream tasks. Depending on the specific downstream task, GNNs will apply different READOUT functions:

$$h_o = \text{READOUT}\left(\{h_u^{(k)} \vert u \in V\}\right). \tag{3}$$

For example, in graph-level tasks, a common approach is to average all node embeddings to obtain the graph embedding.

## IV. SATGL Framework

The SATGL toolkit framework is based on PyTorch and DGL [22], where PyTorch is a widely-used open-source deep learning library that has gained prominence in the machine learning community and DGL is one of the most popular graph learning frameworks, providing efficient low-level operators for graph data processing and graph neural network

```
...
task: satisfiability
dataset_path: ./dataset/3-sat
...
model_settings:
  model: neurosat
  emb_size: 128
...
epochs: 100
lr: 1e-4
weight_decay: 1e-10
...
```

Fig. 3 An example of SATGL config file.

implementations. Leveraging the strengths of these frameworks, SATGL provides a versatile and powerful environment for graph-based machine learning. Following a modular and decoupled architecture, SATGL allows for combinations of diverse models, datasets, and training configurations. Researchers and practitioners can seamlessly build experiments following the instructions of SATGL. As shown in Fig. 1, a complete training process in SATGL involves the following important sections: *Config*, *Task*, *Data*, *Model*, *Utils*.

## A. *Config*

The *Config* module aims to assist users in customizing various parameters required for experiments. It reads both the default configuration file provided and the user-defined configuration file in *YAML* format. These configurations are then used to generate a *Config* instance for subsequent training processes. Additionally, to facilitate direct modification of configurations in the code or via command-line arguments, the *Config* module also reads Python Dictionary and command-line parameters. The priority order of configuration from highest to lowest, is command-line parameters, Python Dictionary, and YAML files. The higher-priority configurations will override lower-priority configurations. Fig. 3 shows an example of SATGL config file.

## B. *Task*

SATGL supports three mainstream SAT tasks: satisfiability, MaxSAT, and UNSAT Core prediction.

- **Satisfiability** [2]: As a fundamental task in computer science and logic, satisfiability is treated as a graph-level task. Given a SAT problem, the label is true if the problem can be satisfied, otherwise false. Satisfiability can be regarded as a graph classification problem.
- **MaxSAT** [8]: As an extension of the classical SAT problem, MaxSAT introduces the objective of maximizing the number of satisfied clauses in a Boolean formula. Unlike satisfiability, MaxSAT is treated as a node-level task, the label of the MaxSAT task is the assignment of the variables when the number of satisfied clauses is maximized. MaxSAT can be regarded as a variable classification problem.
- **UNSAT Core** [13]: Similar to MaxSAT, UNSAT Core is another node-level task within the SATGL framework. The goal of UNSAT Core prediction is to identify the minimal subset of clauses within an unsatisfiable problem. The label of the UNSAT Core task is whether each clause is in the minimum unsatisfiable subset. UNSAT Core can be regarded as a clause classification problem.

## C. *Data*

The *Data* module is designed to load datasets and construct data loaders. Users are required to specify the task type and the path to the data in the configuration file. This allows the *Data* module to automatically generate the corresponding dataset and data loader based on the provided configurations. In the following, we will provide a detailed introduction to the *Data* module.

**Dataset**. The *Dataset* module is responsible for preprocessing the data. Specifically, the dataset must perform the following steps:

- Load Data: To read the data, the dataset module needs to access the data from the *dataset_path* specified by the user. The data consists of conjunctive normal form (CNF) files and a label file. CNF is a standard format of SAT problems, which is equivalent to the original SAT problem. The CNF files must adhere to the DIMACS format, a widely used standard file format for representing CNF formulas. The label file is in CSV format, containing the labels corresponding to each CNF file.
- Graph Construction: The SAT problem is defined as a Boolean formula containing Boolean variables and logical connectives. The *Dataset* module supports four different types of graphs, and all construction methods are supported, including: LCG, LIG, VCG, and VIG.
- Auxiliary information generation: The *Dataset* module stores essential information about the SAT problem, e.g., the number of variables, the number of clauses, etc.

**DataLoader**. After creating the dataset, the dataloader will collate and batch the dataset. In SATGL, we utilize DGL's batch method.

**LabelGen**. We also provide APIs for generating labels. For the satisfiability task, we use the PySAT toolkit [23] to determine whether SAT problems have solutions. For the MaxSAT task, we use the Open-WBO solver to obtain variable assignments for MaxSAT problems. For the UNSAT core task, we use PicoSAT-965 to determine if each clause is part of the UNSAT core.

## D. *Model*

In the *Model* module, we implement two different types of models: general models and specific models. General models include eight commonly used GNN models such as GCN [10], GIN [12] and more. Specific models are tailored for SAT-related tasks, including NeuroSAT [2] and its variants - NeuroCore [13], NLocalSAT [24], QuerySAT [25] and GMS [8].

To enable support for different tasks within the framework, we implement a wrapper class to provide post-processes for different tasks. SATGL makes it easy for users to adapt the

```
# load config file
config = Config(
    config_file_list=['./example.yaml']
)

# get dataset and dataloader
data = get_data(config)

# initialize model and trainer
model = get_model(config)
Trainer = get(trainer)

# start train
Trainer.train(
    model,
    data.train_dataloader,
    data.valid_dataloader,
    data.test_dataloader
)
```

Fig. 4 An example of `SATGL` training process.

framework to their needs. Users can switch between different methods by simply modifying the *task* and *model_settings*. Fig. 4 illustrates an example of usage.

## E. Utils

The `Utils` contains other important modules, The `Trainer` primarily supportthe s training and evaluation process, while the `Logger` records the experiment information, and the `Evaluator` provides evaluation metrics for tasks.

## V. In Practice

In this section, we will give a comprehensive overview of how to use `SATGL`, and Fig. 4 gives an example. First, the user needs to specify a config file. Here we only read YAML files, but it is also possible to use Python dictionaries and command line options. Users can customize the config to suit their needs for different datasets, models, and other parameters. Next, the *get_data* function is called to automatically generate the dataset and dataloader. Then, the *get_model* function is used to get the desired model. Then, the config is passed to the trainer, which initializes the components in the Trainer based on the config. Finally, *train* is run to train and evaluate the model.

## VI. Evaluation

We evaluate the performance of all baselines across three distinct tasks: Satisfiability, MaxSAT, and UNSAT Core.

## A. Datasets

Datasets from three different distributions are used: SR, 3-SAT, and K-Clique. The SR dataset follows the paradigm proposed by NeuroSAT to generate random SAT problems. The 3-SAT dataset is a classic distribution of SAT problems in which each clause consists of three literals. K-Clique is a classical problem in graph theory, which involves finding a complete subgraph with k vertices in an undirected graph (i.e., every pair of vertices in the subgraph is connected by an edge), we restrict $3 \leq k \leq 5$. As NeuroSAT [2], for all distributions, we generate pairwise instances that differed by only one clause, but for the UnSAT Core task, we use only unsatisfied instances as UNSAT Core only exists in unsatisfied instances. All datasets are split into training, validation, and testing sets in an 8:1:1 ratio. Details of each dataset are shown in TABLE III.

TABLE III The overview of datasets

| Distribution | #Variable | #Clause | #Benchmark |
|---|---|---|---|
| SR | 10-40 | 47-308 | 2000 |
| 3-SAT | 10-40 | 55-175 | 2000 |
| K-Clique | 30-100 | 387-4465 | 2000 |

TABLE IV Results on Satisfiability task

| Model | SR | 3-SAT | K-Clique |
|---|---|---|---|
| NeuroSAT | 0.815 | 0.825 | **0.780** |
| NeuroCore | 0.765 | 0.825 | 0.610 |
| NlocalSAT | 0.830 | 0.845 | 0.565 |
| QuerySAT | 0.730 | 0.825 | 0.635 |
| GMS | **0.845** | **0.860** | 0.740 |

## B. Experiment Settings

- **Model Settings**: The hidden dimension of each model is 128. The number of MLP layers is set to 3. All GNNs perform 32 iterations of message passing.
- **Training Settings**: For all models, we use Adam [26] with a learning rate selected from $\{10^{-5}, 5 \times 10^{-5}\}$ and weight decay is $10^{-10}$. Due to the limitations of the GPU memory, we set the batch size to 32 or 16.
- **Training Method for Satisfiability Tasks**: During training, pairwise satisfiable (SAT) and unsatisfiable (UNSAT) problems are included in batches. This approach is crucial for improving the model's performance and speeding up the training process. Since each pair differs by only one literal, not using this training method may cause the model's performance to deteriorate.
- **Loss Function and Evaluation Metric**: As all three tasks are classification tasks, we use binary cross-entropy as the loss function for training and evaluate the accuracy of classification.

## C. Satisfiability

TABLE IV presents the results of all SAT models on the Satisfiability task. According to the results, GMS achieves the best performance on SR and 3-SAT datasets, and NeuroSAT achieves the best performance on K-Clique datasets. The reason for the inferior performance of NeuroCore and QuerySAT compared to the other baselines is that they replace RNN (e.g. LSTM) with MLP in their model architectures for efficiency reasons. Also, due to the larger size of the graphs in the K-Clique dataset, all models do not perform as well on the K-Clique dataset as they do on the other two datasets.

## D. MaxSAT

TABLE V presents the experimental results on the MaxSAT task. The results show little variation across baselines, NlocalSAT achieves the best performance on SR and K-Clique datasets, and GMS achieves the best performance on 3-SAT datasets. All models performed well on the K-Clique dataset, this may be due to the solutions of the K-Clique distribution have some regularity.

TABLE V Results on MaxSAT task

| Model | SR | 3-SAT | K-Clique |
|---|---|---|---|
| NeuroSAT | 0.786 | 0.761 | 0.943 |
| NeuroCore | 0.779 | 0.745 | 0.938 |
| NlocalSAT | **0.787** | 0.767 | **0.944** |
| QuerySAT | 0.781 | 0.762 | 0.942 |
| GMS | 0.781 | **0.773** | 0.943 |

### *E. UNSAT Core*

TABLE VI presents the experimental results on the UNSAT core task. Similar to the MaxSAT task, the UNSAT Core is also a node-level task. The performance of the different baselines is very close, but it can be observed that NlocalSAT slightly outperforms the other baselines.

TABLE VI Results on UNSAT Core task

| Model | SR | 3-SAT | K-Clique |
|---|---|---|---|
| NeuroSAT | 0.883 | 0.693 | 0.743 |
| NeuroCore | 0.883 | 0.693 | 0.733 |
| NlocalSAT | **0.885** | **0.698** | 0.734 |
| QuerySAT | 0.877 | 0.690 | 0.733 |
| GMS | 0.882 | 0.693 | **0.747** |

### *F. Graph Paradigm Evaluation*

TABLE VII gives the results of different graphs using two general GNNs. We only perform experiments on the MaxSAT task. For the satisfiability task, the pairwise benchmarks differ only in the value of one literal, so their VCG and VIG graphs remain identical, but with different labels. As for the UNSAT Core prediction task, LIG and VIG do not construct clause nodes and are not suitable for this task, therefore.

Upon observing the experimental results, all baselines perform significantly better on the LCG compared to the other three graphs. This is due to the fact that the LCG is equivalent to the original SAT problem, while the other three graphs lack some information. The LIG loses information of clauses, but still distinguishes positive and negative literals, resulting in slightly higher accuracy of variable assignment than VIG and VCG, which merge positive and negative literals.

## VII. Conclusions and Future Work

This work proposes `SATGL`, an open-source graph learning toolkit designed for the SAT field, which is useful for comparing the performance of different GNN models on SAT-related tasks. `SATGL` allows users to experiment with diverse tasks and datasets. In the future, `SATGL` aims to expand its capabilities by including additional tasks and more datasets. Additionally, `SATGL` will further support more popular GNN models in the SAT field.

## Acknowledgment

This work is supported in part by the National Natural Science Foundation of China (No. U20B2045, U1936220, 62192784, 62172052, 62002029, 61772082) and Beijing Natural Science Foundation (No. 4244107).

TABLE VII Results using different paradigms on MaxSAT

| Model | Graph | SR | 3-SAT | K-Clique |
|---|---|---|---|---|
| GCN | LCG | 0.769 | 0.750 | 0.939 |
|  | VCG | 0.516 | 0.518 | 0.938 |
|  | LIG | 0.564 | 0.625 | 0.938 |
|  | VIG | 0.516 | 0.518 | 0.938 |
| GIN | LCG | **0.783** | **0.756** | **0.947** |
|  | VCG | 0.516 | 0.518 | 0.938 |
|  | LIG | 0.560 | 0.619 | 0.938 |
|  | VIG | 0.517 | 0.518 | 0.938 |

## References

[1] K. Iwama, "SAT-variable complexity of hard combinatorial problems," in *Proc. IFIP 13th World Computer Congress*, 1994, pp. 253–258.
[2] D. Selsam *et al.*, "Learning a SAT Solver from Single-Bit Supervision," in *Proc. ICLR*, 2018.
[3] S. D. QUEUE, "CaDiCaL at the SAT Race 2019," *SAT RACE*, vol. 2019, p. 8, 2019.
[4] M. W. Moskewicz *et al.*, "Chaff: Engineering an efficient SAT solver," in *Proc. DAC*, 2001, pp. 530–535.
[5] R. Martins *et al.*, "Open-WBO: A modular MaxSAT solver," in *Proc. SAT*, 2014, pp. 438–445.
[6] A. Biere *et al.*, "Consistency checking of all different constraints over bit-vectors within a SAT solver," in *Proc. FMCAD*, 2008, pp. 1–4.
[7] W. Guo *et al.*, "Machine learning methods in solving the boolean satisfiability problem," *Machine Intelligence Research*, pp. 1–16, 2023.
[8] M. Liu *et al.*, "Can graph neural networks learn to solve the MaxSAT problem?" in *Proc. AAAI*, vol. 37, 2023, pp. 16 264–16 265.
[9] Z. Shi *et al.*, "Satformer: Transformers for SAT solving," *arXiv preprint arXiv:2209.00953*, 2022.
[10] T. N. Kipf *et al.*, "Semi-Supervised Classification with Graph Convolutional Networks," in *Proc. ICLR*, 2016.
[11] W. Hamilton *et al.*, "Inductive representation learning on large graphs," *Proc. NeurIPS*, 2017.
[12] K. Xu *et al.*, "How Powerful are Graph Neural Networks?" in *Proc. ICLR*, 2018.
[13] D. Selsam *et al.*, "Neurocore: Guiding high-performance SAT solvers with unsat-core predictions," *CoRR, abs/1903.04671*, 2019.
[14] T. Ajayi *et al.*, "INVITED: Toward an Open-Source Digital Flow: First Learnings from the OpenROAD Project," in *Proc. DAC*, 2019, pp. 1–4.
[15] X. Li *et al.*, "iEDA: An Open-Source Intelligent Physical Implementation Toolkit and Library," in *Proc. ASPDAC*, 2023.
[16] F. Ferrandi *et al.*, "Invited: Bambu: an Open-Source Research Framework for the High-Level Synthesis of Complex Applications," in *Proc. DAC*, 2021, pp. 1327–1330.
[17] Y. Lu, S. Liu, Q. Zhang, and Z. Xie, "RTLLM: An Open-Source Benchmark for Design RTL Generation with Large Language Model," *ArXiv*, vol. abs/2308.05345, 2023.
[18] Z. Chai *et al.*, "CircuitNet: an open-source dataset for machine learning applications in electronic design automation (EDA)," *Science China Information Sciences*, vol. 65, 2022.
[19] X. Jiang *et al.*, "CircuitNet 2.0: An Advanced Dataset for Promoting Machine Learning Innovations in Realistic Chip Design Environment," in *Proc. ICLR*, 2024.
[20] X. Li *et al.*, "iPD: An Open-source intelligent Physical Design Toolchain," in *Proc. ASPDAC*, 2024.
[21] L. Ni *et al.*, "MEC: An Open-source Fine-grained Mapping Equivalence Checking Tool for FPGA," *Proc. ISEDA*, pp. 131–136, 2023.
[22] M. Y. Wang, "Deep graph library: Towards efficient and scalable deep learning on graphs," in *Proc. ICLR*, 2019.
[23] A. Ignatiev *et al.*, "PySAT: A Python toolkit for prototyping with SAT oracles," in *Proc. SAT*, 2018.
[24] W. Zhang *et al.*, "NLocalSAT: boosting local search with solution prediction," in *Proc. IJCAI*, 2021.
[25] E. Ozolins *et al.*, "Goal-aware neural sat solver," in *Proc. IJCNN*, 2022.
[26] D. P. Kingma and J. Ba, "Adam: A method for stochastic optimization," *arXiv preprint arXiv:1412.6980*, 2014.