DEEPSAT: AN EDA-DRIVEN LEARNING FRAMEWORK FOR SAT

Anonymous authors
Paper under double-blind review

ABSTRACT

We present *DeepSAT*, a novel end-to-end learning framework for the Boolean satisfiability (SAT) problem. Unlike existing solutions trained on random SAT instances with relatively weak supervision, we propose applying the knowledge of the well-developed electronic design automation (EDA) field for SAT solving. Specifically, we first resort to logic synthesis algorithms to pre-process SAT instances into optimized and-inverter graphs (AIGs). By doing so, the distribution diversity among various SAT instances can be dramatically reduced, which facilitates improving the generalization capability of the learned model. Next, we regard the distribution of SAT solutions being a product of conditional Bernoulli distributions. Based on this observation, we approximate the SAT solving procedure with a conditional generative model, leveraging a novel directed acyclic graph neural network (DAGNN) with two polarity prototypes for conditional SAT modeling. To effectively train the generative model, with the help of logic simulation tools, we obtain the probabilities of nodes in the AIG being logic '1' as rich supervision. We conduct comprehensive experiments on various SAT problems. Our results show that, DeepSAT achieves significant accuracy improvements over state-of-the-art learning-based SAT solutions, especially when generalized to SAT instances that are relatively large or with diverse distributions.

1 INTRODUCTION

The Boolean satisfiability (SAT) problem is one of the most fundamental combinatorial optimization problems, determining whether a combination of binary input variables exists for which a given Boolean formula returns True. It has a broad range of applications, such as planning (Büttner & Rintanen, 2005), scheduling(Horbach, 2010), and verification (Vizel et al., 2015).

SAT is NP-complete. Over the past decades, many powerful heuristic-based SAT solvers are presented in the literature (Selman et al., 1993; Sorensson & Een, 2005; Audemard & Simon, 2009; Fleury & Heisinger, 2020). Recently, a family of data-driven techniques that apply deep learning (DL) for SAT solving has emerged. Some methods try to learn optimized search policies used in existing heuristic-based SAT solvers (Kurin et al., 2020; Yolcu & Póczos, 2019; Zhang et al., 2021b; Wang et al., 2021). Though effective, the resulted model's performance is bounded by the best solutions provided by the general template of heuristics used in the underlying classical SAT solvers. As an alternative, several end-to-end DL models are proposed for SAT solving (Selsam et al., 2019; Amizadeh et al., 2019; Duan et al., 2022). These approaches try to learn SAT solutions from scratch, and they have the potential to achieve solutions beyond those of existing heuristic-based SAT solvers, despite the inferior performance as of now.

While promising, current end-to-end DL models for SAT solving share a few common weaknesses. On the one hand, there exists notable distribution diversity among different SAT distributions, which makes generalization difficult. On the other hand, the supervision cues used for model training are relatively weak. Existing models are either trained with binary classification labels (i.e., SAT or UNSAT) (Selsam et al., 2019) or in an unsupervised manner (Amizadeh et al., 2019), resulting in unsatisfactory performance.

Given the above challenges, this work re-explores the essential elements of end-to-end DL models for SAT solving. We investigate it from a fundamentally different perspective: we transfer the domain knowledge from the electronic design automation (EDA) field (Jansen et al., 2003) to SAT

solving and mimic the Boolean constraint propagation (BCP) (Wu et al., 2007) using a bidirectional propagation process.

Specifically, we first propose a pre-processing procedure that applies logic synthesis algorithms (Bjesse & Boralv, 2004; Mishchenko et al., 2006; Cortadella, 2003) to represent SAT instances as optimized and-invertor graphs (AIGs), thereby reducing the diversity between training and testing SAT distributions. Second, we formulate the SAT solving process as a generative modeling procedure of the joint Bernoulli distribution of the binary inputs. Inspired by PixelCNN (Oord et al., 2016), we factorize the joint Bernoulli distribution as a product of conditional univariate Bernoulli distribution of every variable. Next, we obtain rich supervision labels by applying efficient logic simulation on the AIG circuits to estimate the parameters in univariate Bernoulli distributions, i.e., the probability of signals in the AIG being logic '1'. The conditional generative model for approximating the conditional probability is realized as a dedicated directed acyclic graph neural network (DAGNN) with two polarity prototypes. The DAGNN learns a hidden space with good interpretability of logic values through a bidirectional propagation process conditioned on gate masks. Finally, we produce satisfying assignments for the SAT instance from the trained conditional generative model using a simple solution sampling scheme.

We refer to the proposed framework for end-to-end SAT solving as *DeepSAT*. Overall, we make the following contributions:

- To the best of our knowledge, DeepSAT is the first work that systematically transfers the knowledge from EDA to learning-based SAT solving. In particular, we apply two well-established EDA techniques, namely logic synthesis and logic simulation, for reducing the distribution diversity of SAT instances and constructing the supervision labels, respectively.
- We reformulate the SAT solving process as a conditional generative modeling procedure to sequentially determine the value of the variables based on previously resolved variables, which enables the explicit SAT assignment prediction.
- We realize the conditional generative modeling using a novel DAGNN with two polarity prototypes, and propose to train DAGNN through bidirectional propagation conditioned on gate masks to mimic the BCP in traditional SAT solving. In this way, our DAGNN learns interpretable hidden states for solution sampling compared with other GNN-based solvers.

Experimental results show that the proposed DeepSAT solver achieves superior performance on both accuracy and generalization capabilities, compared to existing end-to-end learning-based solutions.

2 Related Work

Learning-based SAT solvers. Applying deep learning techniques for combinatorial optimization (CO) has been explored in the last few years (Khalil et al., 2017; Selsam et al., 2019; Chen & Tian, 2019; Yu et al., 2021; Peng et al., 2021; Hudson et al., 2022). The problems of interest are often NP-complete and traditional methods efficient in practice usually rely on heuristics or produce approximate solutions (Yolcu & Póczos, 2019). Among them, the Boolean Satisfiability problem (SAT) is one of the most fundamental problems and becomes a popular target for learning-based solutions. There are two main lines of learning-based SAT solvers: the first is to use neural networks to learn the optimal heuristics within the conventional SAT solvers automatically (Yolcu & Póczos, 2019; Kurin et al., 2020; Zhang et al., 2021b; Wang et al., 2021). Nevertheless, the performance of these methods is bounded by the greedy strategy, which is sub-optimal in general (Amizadeh et al., 2019). The alternative is to train a deep learning model towards solving SAT from scratch (Selsam et al., 2019; Amizadeh et al., 2019; Duan et al., 2022). The representative approaches include NeuroSAT (Selsam et al., 2019) and DG-DAGRNN (Amizadeh et al., 2019). In particular, NeuroSAT processes SAT problems as a binary classification task and proposes a clustering-based post-processing analysis to find an SAT solution. DG-DAGRNN approximates logic calculation with an evaluator network and trains the model to maximize the *satisfiability value* for SAT instances.

Graph neural networks. Graph neural networks (GNNs) (Kipf & Welling, 2017) have shown their effectiveness in modeling non-Euclidean structured data and led to considerable performance improvement in various domains (Kipf & Welling, 2017; Hamilton et al., 2017; Veličković et al., 2018; Wang et al., 2019; Hu et al., 2020; Wu et al., 2020). Some recent efforts specialize GNNs to handle directed acyclic graphs (DAGs) (Zhang et al., 2019; Thost & Chen, 2021; Zhang et al.,

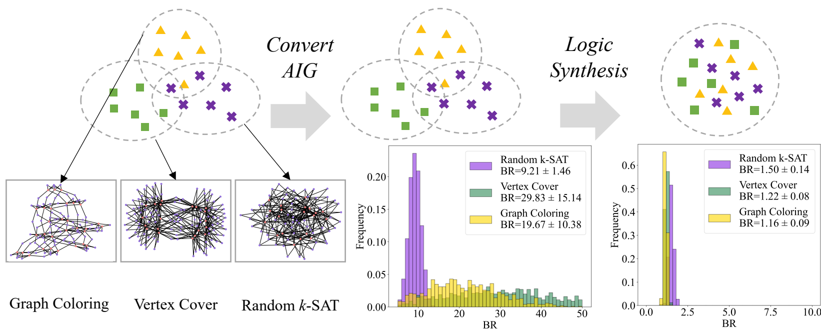


Figure 1: Pre-processing via logic synthesis.

2021c; Li et al., 2022). For example, Zhang et al. (2019) propose a variational autoencoder for DAGs and apply it to neural architecture search and Bayesian network structure learning. In this work, we leverage directed acyclic GNN (Zhang et al., 2019; Amizadeh et al., 2019; Thost & Chen, 2021) as the basic machinery to capture computational dependencies on a Boolean circuit.

3 DEEPSAT

In this section, we introduce the proposed DeepSAT framework in detail. To facilitate understanding, we first show preliminaries of SAT solving problem in Section 3.1, where three different representations of SAT instances are presented. In Section 3.2, we introduce our logic synthesis based pre-processing procedure. In Section 3.3, we present our formulation of SAT solving as a conditional generative procedure and the construction of the supervision labels. We detail the model design in Section 3.4, and show how to produce satisfying assignments with the trained conditional generative model in Section 3.5.

3.1 PRELIMINARIES

A Boolean logic/propositional formula ϕ takes a set of I variables $\{x_i\}_{i=1}^{I} \in \{True, False\}$, and combines them using Boolean operators $\{$AND–$\wedge$, OR–$\vee$, NOT–$\neg$, $\cdots\}$, returning logic '1' (*True*) or logic '0' (*False*). The Boolean satisfiability (SAT) problem asks whether there exists at least one combination of binary input $\{x_i\}_{i=1}^{I}$ for which a Boolean logic formula ϕ returns logic '1'. If so, ϕ is called *satisfiable*; otherwise *unsatisfiable*.

There are various forms for representing a Boolean formula. The most commonly used form is *conjunctive normal form* (CNF) (Tseitin, 1983), wherein the variables are organized as a conjunction ($\wedge$) of disjunctions ($\vee$) of variables. By convention, each disjunction inside parentheses is termed as a *clause*, and each variable (possibly negation) within a clause is termed as a *literal*. Such CNF forms are used in NeuroSAT (Selsam et al., 2019) and its following-ups (Yolcu & Póczos, 2019; Kurin et al., 2020; Zhang et al., 2021b; Duan et al., 2022) to represent Boolean formulas, resulting in bipartite graphs with two node types: literal and clause. Besides CNF, a Boolean formula can be represented in a *Boolean circuit* format (also known as Circuit-SAT), wherein the primary inputs (PIs) of the circuit denote variables of a Boolean formula, and the internal gates denote Boolean operators. In this way, more structure information can be embedded in the inputs. A usage of Circuit-SAT can be found in (Amizadeh et al., 2019), where the CNF instances constructed from Boolean formulas are further converted to circuits, and encoded as directed acyclic graphs (DAGs).

In this work, inspired by the common practice in many EDA processes, we propose to represent Boolean formulas in a unique format of circuits, i.e., *and-invertor graph* (AIG) which is more brief compared with normal circuits. An AIG contains three types of nodes: PIs, two-input AND gates, and one-input NOT gates. It is theoretically guaranteed that any Boolean circuit has an AIG counterpart. By representing SAT instances in AIGs, we can bridge SAT solving and advanced EDA algorithms, e.g., using logic synthesis to pre-process the circuit forms (See Section 3.2).

3.2 PRE-PROCESSING VIA LOGIC SYNTHESIS

Existing learning based SAT solvers (Selsam et al., 2019; Yolcu & Póczos, 2019; Kurin et al., 2020; Zhang et al., 2021b) show poor generalization ability for different SAT classes, i.e., they need to design different heuristics for different distributions of SAT problems. Figure 1 shows the distribution diversity among three SAT instances, each belonging to a different SAT class. Yolcu & Póczos

(2019) also observe that the performance of heuristic specialized to a particular SAT distribution degrades considerably on other distributions.

To this end, we leverage the *logic synthesis* techniques (Cortadella, 2003; Bjesse & Boralv, 2004; Mishchenko et al., 2006) in the EDA field to reduce the distribution diversity among different SAT distributions, thus improving the generalizability of DeepSAT. Specifically, we first represent the given SAT instances in AIG format. Then we apply two logic synthesis techniques, i.e., logic rewriting (Bjesse & Boralv, 2004; Mishchenko et al., 2006) and logic balancing (Cortadella, 2003). The former reduces the total number of AIG nodes, and latter constructs a more balanced circuit with minimal logic level. Please refer to (Brayton & Mishchenko, 2010; Eén et al., 2007) and Appendix A for more technical details about these techniques.

We quantitatively show the change of the distributions before and after logical synthesis, using a scale-independent measurement, i.e., balance ratio (BR) (Walker & Wood, 1976) - the average ratio of larger fanin region size to smaller fanin region size for each two-fanin gate, i.e., AND gate in AIG. A BR value closer to 1 indicates more balanced fanin regions of the gate. As shown in Figure 1, AIGs from different SAT sources have distinct frequency histogram of BR values. But after performing logic synthesis, these histograms become similar showing BR values close to 1, since all the AIGs are optimized to have more balanced fanin regions of the gate. This demonstrates that logic synthesis can reduce the distribution diversities among AIGs from different SAT sources, which naturally introduces a strong inductive bias (i.e., compactness and balance) for more effective SAT solving.

3.3 SAT Solving as a Conditional Generative Procedure

Problem formulation. As introduced in Section 3.1, we represent a Boolean formula as an AIG-based directed graph $\mathcal{G} = (\mathcal{V} = \{\mathcal{V}_P, \mathcal{V}_G\}, \mathcal{E})$, where the nodes $\mathcal{V}_P = \{v_i\}_{i=1}^{I}$ correspond to the primary inputs (PIs), the nodes $\mathcal{V}_\text{G} = \{v_i\}_{i=I+1}^{|\mathcal{V}|}$ denote the logic gates, and the directed edges $\mathcal{E}$ represent wires among the gates. The PI states can be viewed as a random variable following the multivariate Bernoulli distribution, $\mathbf{x} = (x_1, x_2, \cdots, x_I)^T \in \{0,1\}^I \sim \text{Bernoulli}(\boldsymbol{\theta})$, where I is the number of PIs and $\boldsymbol{\theta}$ is the parameter of the distribution. The primary output (PO) y is the state of the gate at the last logic level, which follows the univariate Bernoulli distribution, indicating whether the Boolean circuit evaluates to logic '1' or not. A feasible solution for a satisfiable Boolean circuit can be derived by:

$$\text{Solution } \mathbf{x}^* = \arg\max_{\mathbf{x}} p(\mathbf{x}|y=1; \mathcal{G}, \boldsymbol{\theta}). \tag{1}$$

Similar to PixelCNN (Oord et al., 2016), we handle the intractable joint probability distribution of $\mathbf{x}$ by factorizing it into a product of conditional univariate Bernoulli distributions:

$$p(\mathbf{x}|y=1; \mathcal{G}, \boldsymbol{\theta}) = \prod_{i=1}^{I} p(x_i|\mathbf{x}_{<i}, y=1; \mathcal{G}, \theta_i), \quad x_i|\mathbf{x}_{<i} \sim \text{Bernoulli}(\theta_i), \tag{2}$$

where $\mathbf{x}_{<i} = \{x_1, x_2, \cdots, x_{i-1}\}$ denote the PIs of which the values have been determined, and $p(x_i|\mathbf{x}_{<i}, y=1; \mathcal{G}, \theta_i) = \theta_i^{x_i}(1-\theta_i)^{x_i}$.

Supervision label construction. To obtain the joint probability in Equation 2, we need to figure out θ_i, and then calculate $p(x_i|\mathbf{x}_{<i})$. Note that different from PixelCNN (Oord et al., 2016), we do not have a natural fixed order for picking x_is. To facilitate subsequent discussions, we rewrite the condition terms $\mathbf{x}_{<i}$ in Equation 2 as $\mathbf{x}_m$. The determined PIs are designated using a mask $\mathbf{m} = (m_1, \cdots, m_{|\mathcal{V}|})^T \in \{1, 0, -1\}^{|\mathcal{V}|}$ to simulate the generative procedure, which is applied on all nodes $\mathcal{V}$ to indicate whether the state of a node is determined or not:

$$m_j = \begin{cases} 0 & \text{if} \quad v_j \in \mathcal{V}_G \text{ or } x_j \notin \mathbf{x}_m \\ 1 & \text{if} \quad x_j \in \mathbf{x}_m \text{ and } x_j = \textit{True}, \\ -1 & \text{if} \quad x_j \in \mathbf{x}_m \text{ and } x_j = \textit{False}. \end{cases} \tag{3}$$

Following a frequentest setting, we estimate a value for each θ_i by maximum likelihood estimation (MLE) (Bishop & Nasrabadi, 2006). Suppose we feed N random assignments (also known as *logic simulation* in EDA context) to the Boolean circuit $\mathcal{G}$ and observe variable x_i being logic '1' for M times. Then we can obtain the maximum likelihood estimator for θ_i of $p(x_i; \mathcal{G}, \theta_i)$:

$$\hat{\theta}_i = \frac{M}{N}. \tag{4}$$

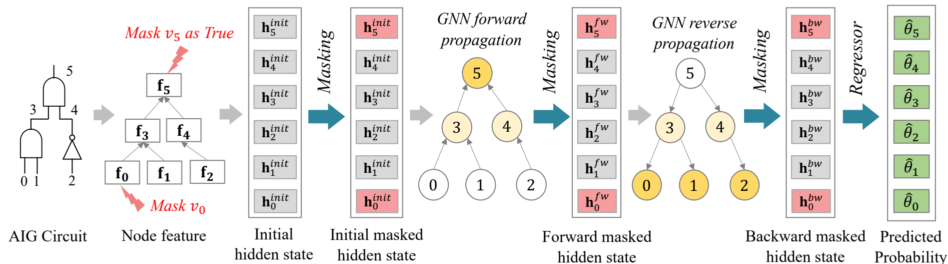


Figure 2: The overview of model architecture. We use polarity prototypes to replace the hidden states of gates with masks during forward and reverse propagation.

It should be noted that such estimation is achieved without any conditions, e.g., the circuit being satisfiable or values of some nodes are known in advance. If some conditions are imposed into the circuit, we simply filter out the random assignments that violate the conditions during logic simulation, therefore can estimate θ_i in the conditional distribution.

We refer $\hat{\theta}_i$ as the *simulated probability* of node being logic '1' in the following discussion. To avoid in-accurate estimation associated with maximum likelihood, we perform logic simulations with a large number of random patterns (e.g., 15k random patterns to each AIG in our experiments) to obtain faithfully simulated probability values. Alternatively, for larger problems, a practical way is to first use an efficient *all solutions SAT solver* (Toda & Soh, 2016) to obtain all possible satisfying solutions, and then estimate the supervision signal $\hat{\theta}_i$ from these assignments. In general, the cost of our logic simulation is on par with those of other SAT solving algorithms such as NeuroSAT (Selsam et al., 2019), where the SAT-related supervision labels are generated from a classical SAT solver. Please refer to Appendix B for more details about dataset construction.

Training objective. The core of our approach is a graph-based model taking a DAG $\mathcal{G}$ and the conditions $\mathbf{m}$ as inputs to approximate the simulated probabilities at the node-level directly. Our hypothesis is that a DAG structure sufficiently defines the logic computation of the Boolean circuit and characterizes the dependence of the different gates/variables. With enough relational inductive biases, a parameterized graph-based model is able to predict the simulated probability of all gates under logic simulation. Driven by this hypothesis, we designate the training objective as the following mapping:

$$\mathcal{F} : (\mathcal{G}, \mathbf{m}) \mapsto \hat{\boldsymbol{\theta}}, \quad \text{where} \quad \hat{\boldsymbol{\theta}} = \{\hat{\theta}_i = \arg\max_{\theta_i} p(x_i | \mathbf{x_m}, y = 1; \mathcal{G}, \theta_i) | x_i \in \mathcal{G}\}, \tag{5}$$

where the condition $\mathbf{x_m}$ corresponds to the determined nodes defined by the mask $\mathbf{m}$ (Equation 3). We generate a mask $\mathbf{m}$ initialized from a zero vector by fixing the element for PO to 1 (i.e., let $y = 1$), and then assign the elements corresponds to a randomly designated condition terms $\mathbf{x}_m$ with 1 or -1. Given $\mathbf{m}$, the objective is to regress the simulated probabilities of the remaining nodes v_i being logic '1', i.e., to approximate $\hat{\theta}_i$. Each conditional probability is parameterized by a deep neural network whose architecture is chosen according to the required inductive biases for the graph-structured Boolean circuits. The function $\mathcal{F}$ is learned by minimizing the least absolute error between the prediction and the supervision label $\hat{\boldsymbol{\theta}}$.

3.4 The Proposed Model

To capture directed relational bias embedded in circuits, we leverage the dedicated directed acyclic GNNs (DAGNNs) (Zhang et al., 2019; Thost & Chen, 2021) as the model $\mathcal{F}$ in Equation 5 to learn structural information and the computational behavior of logic circuits, and embed them as vectors on every logic gate (Li et al., 2022). As shown in Figure 2, the model consists of two stages of directed acyclic GNN propagation along with the masking operations defined in Equation 3. Then an multi-layer perceptron (MLP) regressor is applied on node vectors for predicting the simulated probability.

Given a circuit graph $\mathcal{G}$, we use $\mathbf{f}_i$ and $\mathbf{h}_i$ to denote the gate type and the hidden vector, respectively. Specifically, as three types of gates, i.e., PI, AND and NOT, are present in AIGs, we encode the gate type as a 3-d one-hot embedding for each node according to its gate type. We elaborate the details of the model design in DeepSAT below.

Polarity prototypes. The training objective derived in Equation 5 requires the model to be able to condition on some given gate states. To enable such conditional modeling, we define two prototypes (Snell et al., 2017), $\mathbf{h}_{pos}$ and $\mathbf{h}_{neg}$, for two polarities of node probabilities, one for node being logic '1' (positive polar) and the other for node being logic '0' (negative polar), respectively.

Assuming every node v_i is encoded as a hidden vector $\mathbf{h}_i$, the hidden vectors for the masked nodes are replaced by their corresponding prototypes according to the value of the mask:

$$\mathbf{h}'_i = \begin{cases} \mathbf{h}_{pos} & \text{if} \quad m_i = 1, \\ \mathbf{h}_i & \text{if} \quad m_i = 0, \\ \mathbf{h}_{neg} & \text{if} \quad m_i = -1. \end{cases} \tag{6}$$

In practice, the initial hidden vectors are sampled from a normal Gaussian distribution. Therefore, we fix the hidden vectors for two polarities as two near-boundary points, i.e, all positive ones for logic 1 ($\mathbf{h}_{pos} = [1, 1, \cdots .1]$) and all negative ones for logic 0 ($\mathbf{h}_{neg} = [-1, -1, \cdots, -1]$), respectively. The two polarity prototypes can be thought of as two extreme points lying on opposite sides of a sphere in hidden space. Intuitively, during training, the hidden vectors of gates with a probability close to 1 will be pulled to all ones, and vice versa. Therefore, the two polarity prototypes facilitate learning a continuous and compact hidden space with good *interpretability* of logic values. Note that other polarity prototypes are also feasible, as long as the two polarity prototypes are far away from each other in the hidden space. Detailed explanation of the polarity prototypes are included in Appendix C

For notational simplicity, we omit the superscript prime $'{}'$', and directly use $\mathbf{h}_i$ to denote the input hidden vector after the masking operation defined in Equation 6.

Forward propagation. To aggregate the information from node v's predecessors, we implement the attention mechanism in the additive form (Thost & Chen, 2021) as:

$$\mathbf{a}_v^{fw} = \sum_{u \in \mathcal{P}(v)} \alpha_{uv}^{fw} \mathbf{h}_u^{init} \quad \text{where} \quad \alpha_{uv}^{fw} = \underset{u \in \mathcal{P}(v)}{softmax}(w_1^\top \mathbf{h}_v^{init} + w_2^\top \mathbf{h}_u^{init}), \tag{7}$$

where $\mathcal{P}(v)$ denotes the set of direct predecessors of v. Using the 'Attention' language, the inital hidden state $\mathbf{h}_v^{init}$ serves as a *query*, and the representation of predecessors $\mathbf{h}_u^{init}$ serves as a *key*. We then use the GRU to combine the aggregated information with v's information, including the encoded gate type $\mathbf{f}_v$ and the initial hidden vector $\mathbf{h}_v^{init}$ to update the hidden vector of target node v:

$$\mathbf{h}_v^{fw} = GRU([\mathbf{a}_v^{fw}, \mathbf{f}_v], \mathbf{h}_v^{init}), \tag{8}$$

wherein $\mathbf{a}_v^{fw}$, $\mathbf{f}_v$ are concatenated together and treated as input, while $\mathbf{h}_v^{init}$ is the past state of GRU. The above functions process nodes according to the logic levels defined by circuit's topological orders, starting from PIs and ending up with the single PO. After that, the node hidden vector $\mathbf{h}_v^{fw}$ is further updated by according to the same mask following Equation 6.

Reverse propagation. To model the effect of the conditional term related to satisfiability, i.e., $y = 1$ in Equation 5, we perform a reverse information propagation after the forward propagation. Specifically, we imposes the condition of satisfiability PO by masking its hidden vector as $\mathbf{h}_{pos}$, and process the graph in reversed topological order to propagate the condition to PIs. By doing so, the updated node hidden vector $\mathbf{h}_v^{bw}$ contains extra information about the satisfiability compared with $\mathbf{h}_v^{fw}$ that is obtained only using forward propagation.

The backward propagation is similar to the forward propagation regarding computation of aggregation and combination function, except that during the aggregation, we only consider the direct successors $\mathcal{S}(v)$ of target node v. The hidden vector $\mathbf{h}_v^{bw}$ is further processed using the same mask as above following Equation 6, and fed into the regressor for predicting the simulated probabilities.

Comparison with previous methods using bidirectional propagation. Adopting a reverse propagation besides the forward propagation for training DAG structures is not new in GNN-based SAT solving (Amizadeh et al., 2019; Thost & Chen, 2021), yet we highlight key differences that support the superior SAT solving performance of DeepSAT. In DAGNN (Thost & Chen, 2021) and DG-DAGRNN (Amizadeh et al., 2019), the reason to perform forward and reverse propagation is to enrich the node state vectors with information from both the ancestor and the descendant nodes. While in DeepSAT, the goal of using bidirectional propagation along with polarity prototypes is to mimic the Boolean constraint propagation (BCP) (Wu et al., 2007; Belov & Stachniak, 2010; Zhang et al., 2021a) mechanism in logic reasoning.

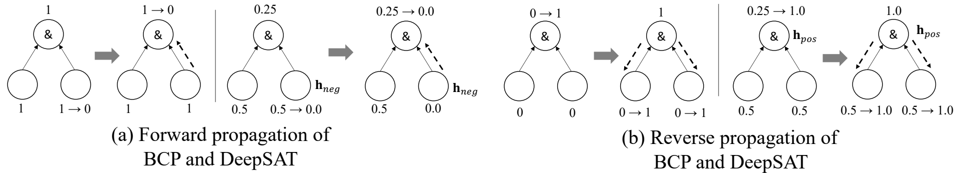


Figure 3: Example of forward/reverse propagation in BCP and DeepSAT. DeepSAT enables constraint propagation by bidirectional propagation and polarity prototypes during learning.

Specifically, BCP is initiated when a logic value is assigned to an unassigned gate v, which leads to value changing of some unassigned gates in the neighbourhood of v (i.e., fanin and fanout). There are two types of propagation considering the direction: forward propagation and reverse propagation (Figure 3). Similarly, in DeepSAT, we introduce polarity prototypes as hidden states for gates assigned with known logic states. Specially, the satisfiability is modeled by assigning logic '1' to PO. To this end, optimizing the conditional gate-level probability $p(x_i|\mathbf{x_m}, y = 1; \mathcal{G}, \theta_i)$ (Equation 2) with *bidirectional propagation and polarity prototypes* enables learning BCP in the hidden space (Figure 3). Such design encourages a continuous and compact hidden space with good *interpretability* of logic values. More importantly, with polarity-regularized hidden space, we can manipulate the states of nodes and thus perform conditional solution sampling as described below.

3.5 Solution sampling scheme

To sample solutions from the well-trained conditional model, we iteratively select the undetermined PI with the highest *confidence* values predicted by the model, i.e., the PI with probability prediction closest to 0 or 1. Specifically, given an SAT instance $\mathcal{G}$ with I variables, we conduct the following *auto-regressive* procedure: 1) We mask PO as logic '1', and generate the corresponding mask vector $\mathbf{m}_0$. 2) At t iteration, we pass $(\mathcal{G}, \mathbf{m}_t)$ to well-trained DeepSAT model to estimate the simulated probability of all un-masked PIs. The PI with the highest confidence is selected and masked as logic '0' if the prediction is smaller than 0.5, otherwise logic '1'. According to the selected PI and its masked value, a new mask $\mathbf{m}_{t+1}$ is generated. 3) Repeat the second step until all PIs have been masked (I iterations in total). Then we obtain a candidate assignments $\hat{\mathbf{x}}$ from $\mathbf{m}_I$.

To explore more possible assignments to satisfy the SAT instance, we also develop a simple *flipping*-based strategy to sample more solutions from the conditional model ($(I + 1)$ solutions in the worst case). Briefly, if the initial sampled assignment is not the satisfying solution, we record the order of PIs being masked and flip a specific PI in the next sampling step, following the recorded order. The implementation details, as well as the pseudo-code, are provided in Appendix D.

4 Experiments

4.1 Experimental settings

Baselines. We compare our solution with the representative end-to-end method for SAT solving in the literature[1]: NeuroSAT (Selsam et al., 2019). NeuroSAT assumes that the input problem is in CNF. There are several follow-up works of NeuroSAT (Yolcu & Póczos, 2019; Kurin et al., 2020; Zhang et al., 2021b; Duan et al., 2022) that consuming CNFs. Nevertheless, they target on different settings and NeuroSAT still stands for a strong baseline for end-to-end SAT solving. We give more discussion about the related models and implementation details in Appendix E and additional analysis in Appendix G. To enable a fair comparison, we implement DeepSAT and NeuroSAT in a unified framework[2] with PyTorch Geometric (PyG) (Fey & Lenssen, 2019) package.

Training datasets. We follow the SAT instance generation scheme proposed in NeuroSAT (Selsam et al., 2019) to generative random k-SAT CNF pairs. We use SR(n) to denote random k-SAT problems on n variables generated by this scheme. In particular, we generate a SR($3-10$) dataset of 230k SAT and UNSAT pairs. We restrict the scale of the training dataset due to limited GPU resources. However, we demonstrate that DeepSAT can generalize well to problems that are larger or with novel distributions, as shown below. For NeuroSAT, these pairs are directly used for binary classification, while DeepSAT is only trained on SAT instances. For DeepSAT, we convert CNFs

[1]We do not compare DeepSAT with DG-DAGRNN (Amizadeh et al., 2019) as we cannot reproduce its results, despite that we have rigorously followed the descriptions in the paper. Our implementation of DG-DAGRNN is also attached in the supplementary materials.

[2]The code is available at: `https://anonymous.4open.science/r/deepsat_iclr23_E513/`.

Table 1: Performance comparison of DeepSAT and NeuroSAT on in-sample instances (*Problems Solved*). All models are trained on 230k SR(3-10) training data. We consider two settings: i) Same message passing iterations, where the number of message passing iterations is determined by the number of variables. ii) Test metric converges, where models iteratively run until no instance can be solved by increasing the number of iterations. The test sets follow the same formats used in training.

		Same Iterations					Test Metric Converges				
Methods	Format	SR(10)	SR(20)	SR(40)	SR(60)	SR(80)	SR(10)	SR(20)	SR(40)	SR(60)	SR(80)
NeuroSAT	CNF	65%	58%	32%	20%	20%	92%	74%	42%	20%	20%
DeepSAT	Raw AIG	67%	60%	36%	23%	21%	94%	79%	45%	25%	23%
	Opt. AIG	72%	66%	40%	31%	23%	98%	85%	51%	37%	26%

to raw AIG[3]and Optimized AIG via logic synthesis (abbreviated as Opt. AIG). Note that in all experiments we convert the test instances to the same formats as the ones used in model training.

Evaluation metric. DeepSAT is an *incomplete* (Selman et al., 1993; Zhang et al., 2021b) solver, i.e., an instance is detected as satisfiable only after the model finds a satisfying assignment and otherwise returns *unsolved*. Therefore, we only include satisfiable instances in the testing dataset for all experiments and measure the percentage of the successfully solved SAT instances out of all instances (abbreviated as *Problems Solved*). In Appendix F, we discuss more details of the evaluation metrics for end-to-end SAT solvers.

4.2 Random k-SAT

This section shows comparisons of DeepSAT and NeuroSAT on solving random k-SAT problem. To evaluate the generalization performance, we validate our method on SAT instances generated from a much larger number of variables. In particular, we evaluate the well-trained DeepSAT on SR(10), SR(20), SR(40), SR(60) and SR(80), and compare with the results with the baseline.

Since DeepSAT and NeuroSAT use different assignment decoding schemes to yield the SAT assignments given an SAT instance, we consider two experimental settings for performance comparison: i) compare under the same computational cost, which is measured by the number of message passing iterations, and ii) compare the upper-bound of performance regardless of the computational cost. For the first setting, we fix the number of message passing iterations as I for an I-variable SAT instance. This is because DeepSAT needs to perform at least I message passing iterations to generate a solution for an I-variable SAT instance, while NeuroSAT can generate the solution for a specific problem *once* by clustering on the literal embedding after a number of bidirectional message passing iterations. Hence, for a I-variable SAT instance, we fix the number of message passing iterations as I, in which case DeepSAT can generate one and only one complete assignment and NeuroSAT are aligned with same message passing cost. For the second setting, we let both models generate as many assignments as possible until the test metric converges. In this way, NeuroSAT is queried for multiple times for assignment decoding under different iterations of message passing, in order to monitor the test metric. In other words, NeuroSAT will generate multiple assignments for one SAT instance. As for DeepSAT, we follow the proposed solution sampling scheme introduced in Section 3.5, which samples at most $I + 1$ solutions in worst case.

We present Problems Solved on different testing datasets with different formats in Table 1. From this table, we have several observations. First, compared with the baselines, DeepSAT can achieve the highest *Problems Solved* on all evaluated datasets under both experiment settings, with a significant margin. For example, on SR(20), DeepSAT trained on Opt. AIGs can solve 85% of problems, while NeuroSAT trained on CNFs only solve 74% of problems. Second, the performance of both models degrades as we increase the number of variables. Yet, DeepSAT generalizes better to bigger and harder problems than NeuroSAT.

To further validate the effectiveness of our proposed formulation, i.e., SAT solving as a generative modeling procedure, we monitor *Problems Solved* during the solution sampling procedure on SR(10). By sampling a single solution for each problem, DeepSAT can solve 72% of all problems. If we sample two more solutions for each problem from DeepSAT, the percentage of solved problems increases to 93%. Contrarily, to solve 92% of problems on SR(10), NeuroSAT requires additional tens of iterations of message propagation to make prediction coverage. DeepSAT terminates when

[3]Convert CNF to AIG using the cnf2aig tool: `http://fmv.jku.at/cnf2aig/`

Table 2: The comparison of DeepSAT and NeuroSAT on novel distributions.

Method	Format	Coloring Acc.	Domset Acc.	Clique Acc.	Vertex Acc.	Avg. Acc.
NeuroSAT	CNF	0%	44%	35%	0%	22%
DeepSAT	Raw AIG	63%	81%	77%	82%	76%
	Opt. AIG	98%	99%	92%	97%	97%

the latest sampled solutions are satisfying and samples 1.63 solutions on average for SR(10). The result on other test datasets shows the same tendency. See Appendix G.1 for more results on the proposed solution sampling scheme.

4.3 Effectiveness of Logic Synthesis

In Table 1, we see that DeepSAT maintains state-of-the-art performance even when trained on raw AIGs without logic synthesis. Specifically, DeepSAT trained on raw AIGs outperforms NeuroSAT trained on CNFs consistently across all test sets, despite that raw AIGs include less structural information for learning. For example, DeepSAT trained on raw AIGs successfully solves 94 and 79 out of 100 SR(10) and SR(20) problems, respectively, which is better NeuroSAT trained on CNFs. In summary, we believe that the performance gain of DeepSAT not only comes from EDA optimization, but also thanks to the customized bidirectional propagation with polarity prototypes that can effectively model the conditional distribution of possible assignments.

4.4 Novel Distributions

To further investigate the generalization ability of our DeepSAT model, we evaluate the model trained on SR($3-10$) on the different NP-Complete problems with novel distributions: graph k-coloring, vertex k-cover, k-clique-detection and dominating k-set. The above problems can be reduced to the SAT problems and solved by a SAT solver. We generate 100 random graphs with $6-10$ nodes and the edge percentage of 37% for each problem. For each graph in each problem type, we generate graph coloring instances ($3 \leq k \leq 5$), dominating set problems ($2 \leq k \leq 4$), clique-detection problems ($3 \leq k \leq 5$) and vertex cover problems ($4 \leq k \leq 6$). We report the results when the test metric converges. Notably, the generated problem instances are not particularly difficult for state-of-the-art SAT solvers. For our purpose, the evaluated results in this section serve as simple benchmarks to demonstrate the superiority of DeepSAT over the baseline.

As shown in Table 2, DeepSAT trained on Opt. AIGs can solve 98% of graph coloring problems, 99% of dominating set problems, 92% of clique-detection problems, and 97% of vertex cover problems. When DeepSAT is trained on raw AIGs, the performance degrades to some extent, yet is still better than NeuroSAT on these datasets. In contrast, we observe there is an apparent discrepancy in performance of NeuroSAT between novel distribution problems and the in-sample distribution used for training, even though the size of problems is similar. To sum up, DeepSAT maintains the same solving ability on novel problems, and the pre-processing of logic synthesis enables better generalization ability of novel distributions.

5 Conclusion and Future Work

This paper presents *DeepSAT*, an EDA-driven end-to-end learning framework for SAT solving. Specifically, we first propose to apply a logic synthesis-based pre-processing procedure to reduce the distribution diversity among various SAT instances. Next, we formulate SAT solving as a conditional generative procedure and construct more informative supervisions with simulated logic probabilities. A dedicated directed acyclic graph neural network with polarity prototypes is then trained to approximate the conditional distribution of SAT solutions. Our experimental results show that DeepSAT outperforms the existing end-to-end baseline.

While DeepSAT demonstrates considerable improvements in end-to-end SAT solving, there is still a significant performance gap compared to state-of-the-art heuristic-based SAT solvers. In our future work, on the one hand, we plan to improve its performance further; on the other hand, we would also explore novel joint solutions that combine them, e.g., using constraint propagation mechanism learned in DeepSAT to guide better heuristic in classical Circuit-SAT solvers.

References

Saeed Amizadeh, Sergiy Matusevych, and Markus Weimer. Learning to solve circuit-SAT: An unsupervised differentiable approach. In *International Conference on Learning Representations*, 2019.

Gilles Audemard and Laurent Simon. Predicting learnt clauses quality in modern sat solvers. In *Twenty-first international joint conference on artificial intelligence*. Citeseer, 2009.

Francesco Barbato, Marco Toldo, Umberto Michieli, and Pietro Zanuttigh. Latent space regularization for unsupervised domain adaptation in semantic segmentation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 2835–2845, 2021.

Anton Belov and Zbigniew Stachniak. Improved local search for circuit satisfiability. In *International Conference on Theory and Applications of Satisfiability Testing*, pp. 293–299. Springer, 2010.

Christopher M Bishop and Nasser M Nasrabadi. *Pattern recognition and machine learning*, volume 4. Springer, 2006.

Per Bjesse and Arne Boralv. Dag-aware circuit compression for formal verification. In *IEEE/ACM International Conference on Computer Aided Design, 2004. ICCAD-2004.*, pp. 42–49. IEEE, 2004.

Robert Brayton and Alan Mishchenko. Abc: An academic industrial-strength verification tool. In *International Conference on Computer Aided Verification*, pp. 24–40. Springer, 2010.

Markus Büttner and Jussi Rintanen. Satisfiability planning with constraints on the number of actions. In *International Conference on Automated Planning and Scheduling*, pp. 292–299, 2005.

Xinyun Chen and Yuandong Tian. Learning to perform local rewriting for combinatorial optimization. *Advances in Neural Information Processing Systems*, 32, 2019.

Jordi Cortadella. Timing-driven logic bi-decomposition. *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, 22(6):675–685, 2003.

Jacob Devlin, Ming-Wei Chang, Kenton Lee, and Kristina Toutanova. Bert: Pre-training of deep bidirectional transformers for language understanding. *arXiv preprint arXiv:1810.04805*, 2018.

Alexey Dosovitskiy, Lucas Beyer, Alexander Kolesnikov, Dirk Weissenborn, Xiaohua Zhai, Thomas Unterthiner, Mostafa Dehghani, Matthias Minderer, Georg Heigold, Sylvain Gelly, Jakob Uszkoreit, and Neil Houlsby. An image is worth 16x16 words: Transformers for image recognition at scale. In *International Conference on Learning Representations*, 2021.

Haonan Duan, Pashootan Vaezipoor, Max B Paulus, Yangjun Ruan, and Chris Maddison. Augment with care: Contrastive learning for combinatorial problems. In *International Conference on Machine Learning*, pp. 5627–5642. PMLR, 2022.

Niklas Eén, Alan Mishchenko, and Niklas Sörensson. Applying logic synthesis for speeding up sat. In *International Conference on Theory and Applications of Satisfiability Testing*, pp. 272–286. Springer, 2007.

Matthias Fey and Jan E. Lenssen. Fast graph representation learning with PyTorch Geometric. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*, 2019.

Armin Biere Katalin Fazekas Mathias Fleury and Maximilian Heisinger. Cadical, kissat, paracooba, plingeling and treengeling entering the sat competition 2020. *SAT COMPETITION*, 2020:50, 2020.

Gaëtan Hadjeres, Frank Nielsen, and François Pachet. Glsr-vae: Geodesic latent space regularization for variational autoencoder architectures. In *2017 IEEE Symposium Series on Computational Intelligence (SSCI)*, pp. 1–7. IEEE, 2017.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017.

Kaiming He, Xinlei Chen, Saining Xie, Yanghao Li, Piotr Dollár, and Ross Girshick. Masked autoencoders are scalable vision learners. *arXiv preprint arXiv:2111.06377*, 2021.

Marijn JH Heule, Oliver Kullmann, Siert Wieringa, and Armin Biere. Cube and conquer: Guiding cdcl sat solvers by lookaheads. In *Haifa Verification Conference*, pp. 50–65. Springer, 2011.

Andrei Horbach. A boolean satisfiability approach to the resource-constrained project scheduling problem. *Annals of Operations Research*, 181(1):89–107, 2010.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020.

Benjamin Hudson, Qingbiao Li, Matthew Malencia, and Amanda Prorok. Graph neural network guided local search for the traveling salesperson problem. In *International Conference on Learning Representations*, 2022.

Dirk Jansen et al. *The electronic design automation handbook*. Springer, 2003.

Guoliang Kang, Lu Jiang, Yi Yang, and Alexander G Hauptmann. Contrastive adaptation network for unsupervised domain adaptation. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 4893–4902, 2019.

Elias Khalil, Hanjun Dai, Yuyu Zhang, Bistra Dilkina, and Le Song. Learning combinatorial optimization algorithms over graphs. *Advances in neural information processing systems*, 30, 2017.

Diederik P Kingma and Jimmy Ba. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2015.

Diederik P Kingma and Max Welling. Auto-encoding variational bayes. *arXiv preprint arXiv:1312.6114*, 2013.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2017.

Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. Can q-learning with graph networks learn a generalizable branching heuristic for a sat solver? *Advances in Neural Information Processing Systems*, 33:9608–9621, 2020.

Min Li, Sadaf Khan, Zhengyuan Shi, Naixing Wang, Huang Yu, and Qiang Xu. Deepgate: learning neural representations of logic gates. In *Proceedings of the 59th ACM/IEEE Design Automation Conference*, pp. 667–672, 2022.

Alan Mishchenko, Satrajit Chatterjee, and Robert Brayton. Dag-aware aig rewriting: A fresh look at combinational logic synthesis. In *2006 43rd ACM/IEEE Design Automation Conference*, pp. 532–535. IEEE, 2006.

Aäron van den Oord, Nal Kalchbrenner, Oriol Vinyals, Lasse Espeholt, Alex Graves, and Koray Kavukcuoglu. Conditional image generation with pixelcnn decoders. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*, pp. 4797–4805, 2016.

Yun Peng, Byron Choi, and Jianliang Xu. Graph learning for combinatorial optimization: a survey of state-of-the-art. *Data Science and Engineering*, 6(2):119–141, 2021.

Bart Selman, Henry A Kautz, Bram Cohen, et al. Local search strategies for satisfiability testing. *Cliques, coloring, and satisfiability*, 26:521–532, 1993.

Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L. Dill. Learning a SAT solver from single-bit supervision. In *International Conference on Learning Representations*, 2019.

Yujun Shen, Jinjin Gu, Xiaoou Tang, and Bolei Zhou. Interpreting the latent space of gans for semantic face editing. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 9243–9252, 2020.

Jake Snell, Kevin Swersky, and Richard Zemel. Prototypical networks for few-shot learning. In *Proceedings of the 31st International Conference on Neural Information Processing Systems*, pp. 4080–4090, 2017.

Niklas Sorensson and Niklas Een. Minisat v1. 13-a sat solver with conflict-clause minimization. *SAT*, 2005(53):1–2, 2005.

Veronika Thost and Jie Chen. Directed acyclic graph neural networks. In *International Conference on Learning Representations*, 2021.

Takahisa Toda and Takehide Soh. Implementing efficient all solutions sat solvers. *Journal of Experimental Algorithmics (JEA)*, 21:1–44, 2016.

Grigori S Tseitin. On the complexity of derivation in propositional calculus. In *Automation of reasoning*, pp. 466–483. Springer, 1983.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph attention networks. In *International Conference on Learning Representations*, 2018.

Yakir Vizel, Georg Weissenbacher, and Sharad Malik. Boolean satisfiability solvers and their applications in model checking. *Proceedings of the IEEE*, 103(11):2021–2035, 2015.

A Walker and Derick Wood. Locally balanced binary trees. *The Computer Journal*, 19(4):322–325, 1976.

Wenxi Wang, Yang Hu, Mohit Tiwari, Sarfraz Khurshid, Kenneth McMillan, and Risto Miikkulainen. Neurocomb: Improving sat solving with graph neural networks. *arXiv e-prints*, pp. arXiv–2110, 2021.

Zhongdao Wang, Liang Zheng, Yali Li, and Shengjin Wang. Linkage based face clustering via graph convolution network. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*, pp. 1117–1125, 2019.

Chi-An Wu, Ting-Hao Lin, Chih-Chun Lee, and Chung-Yang Huang. Qutesat: a robust circuit-based sat solver for complex circuit structure. In *2007 Design, Automation & Test in Europe Conference & Exhibition*, pp. 1–6. IEEE, 2007.

Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and S Yu Philip. A comprehensive survey on graph neural networks. *IEEE transactions on neural networks and learning systems*, 32(1):4–24, 2020.

Emre Yolcu and Barnabás Póczos. Learning local search heuristics for boolean satisfiability. *Advances in Neural Information Processing Systems*, 32, 2019.

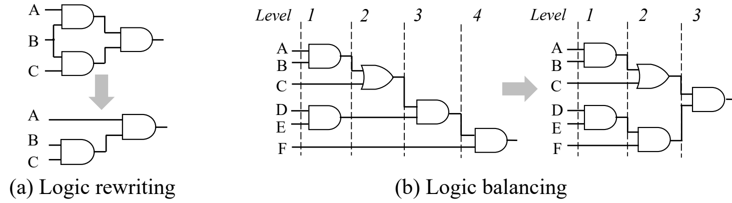
Tianshu Yu, Runzhong Wang, Junchi Yan, and Baoxin Li. Deep latent graph matching. In *International Conference on Machine Learning*, pp. 12187–12197. PMLR, 2021.

He-Teng Zhang, Jie-Hong R Jiang, and Alan Mishchenko. A circuit-based sat solver for logic synthesis. In *2021 IEEE/ACM International Conference On Computer Aided Design (ICCAD)*, pp. 1–6. IEEE, 2021a.

Muhan Zhang, Shali Jiang, Zhicheng Cui, Roman Garnett, and Yixin Chen. D-vae: A variational autoencoder for directed acyclic graphs. *Advances in neural information processing systems*, 32, 2019.

Wenjie Zhang, Zeyu Sun, Qihao Zhu, Ge Li, Shaowei Cai, Yingfei Xiong, and Lu Zhang. Nlocalsat: boosting local search with solution prediction. In *Proceedings of the Twenty-Ninth International Conference on International Joint Conferences on Artificial Intelligence*, pp. 1177–1183, 2021b.

Xitong Zhang, Yixuan He, Nathan Brugnone, Michael Perlmutter, and Matthew Hirn. Magnet: A neural network for directed graphs. *Advances in Neural Information Processing Systems*, 34, 2021c.



(a) Logic rewriting (b) Logic balancing

Figure 4: Examples of graph optimization based on logic synthesis.

Table 3: The balance ratio and logic level before and after EDA opt. (Avg. $\pm$ Std.)

	Graph Coloring		Random k-SAT		k-Clique	
	BR	Level	BR	Level	BR	Level
Before EDA opt.	19.67 ± 10.38	45.71 ± 7.04	9.21 ± 1.46	67.58 ± 34.34	29.83 ± 15.14	106.75 ± 44.00
After EDA opt.	1.16 ± 0.09	14.15 ± 0.99	1.50 ± 0.14	12.40 ± 0.96	1.21 ± 0.08	13.29 ± 0.81

A Logic Synthesis

We apply two techniques of logic synthesis in Section 3.2, namely logic rewriting and logic balancing. In this section, we give more details about these two techniques. Examples of graph optimization based on logic synthesis are included in Figure 4.

A.1 Logic Rewriting

Rewriting is a greedy circuit optimization technique for minimizing the circuit structure (Bjesse & Boralv, 2004; Mishchenko et al., 2006). It enumerates sub-graphs and replaces them with the equivalent smaller sub-graphs iteratively. To facilitate subsequent discussion, we first define *cut* as a set of nodes on the circuits with only one root node. A cut is *K-feasible* if the number of variables does not exceed K. In AIG rewriting (Mishchenko et al., 2006), all non-redundant 4-feasible cuts are pre-computed and stored in advance. With the above prerequisite, the rewriting algorithm can be summarized as the following four steps: (1) select a certain node as the root node in the topological order from PI to PO; (2) perform the 4-feasible cut enumeration; (3) in each iteration, get a cut and corresponding Boolean function; (4) if there is an implementation of the function with a smaller number of gates, choose the implementation; otherwise, leave the AIG unchanged. Figure 4(a) shows an example where the sub-graph can be replaced by a simplified one without logic modification.

A.2 Logic Balancing

Logic balance creates an equivalent circuit with minimum delay, i.e., the minimum number of logic levels. As suggested by its name, the optimized circuits is derived by balancing the various fan-in regions of multi-input gates. The main algorithm constructs a Boolean function trees and decomposes the tree from root to leaves (Cortadella, 2003). In our setting, since all AND gates only have two fan-in wires, the logic balance refers to logic bi-decomposition problem. In each balancing iteration: (1) the logic tree is decomposed into several sub-trees; (2) balance the sub-tree recursively; (3) insert the balanced sub-trees back to the main stem. Figure 4(b) shows a logic balancing instance, where the balancing achieves one logic level reduction.

The above logic synthesis techniques are implemented in the open-source tool ABC (Brayton & Mishchenko, 2010) with commands *rewrite / rw* and *balance / b*, respectively. It should be noted that both commands achieve the locally optimal effect based on the current topological structure. In other words, the circuit will not be modified when executing the same command for multiple times. Therefore, including some perturbation into the circuit can expose new minimizing opportunities. For example, (Eén et al., 2007) perturbs the AIG by decomposing the local structure and (Cortadella, 2003) transforms the circuit based on different laws alternately. The *rewrite* and *balance* operations are considered as perturbation factors mutually. In this work, we perform both operations iteratively for three times (*rw; b; rw; b; rw; b.*) to obtain optimized AIGs.

Table 4: The Error between Logic Simulation and Enumeration.

# Samples	Difference	Error Rate
100k	0.0007	1.75%
10K	0.0019	4.75%
1K	0.0144	36.00%
0.1K	0.1037	259.25%

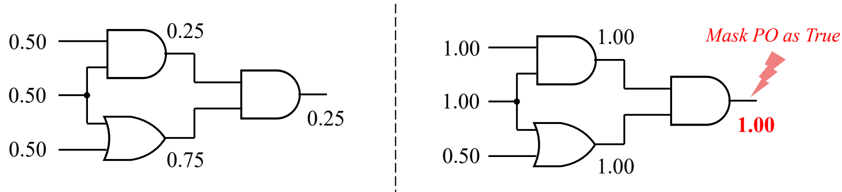


Figure 5: An example of logic simulation w/ and w/o mask.

A.3 Other Quantitative Analysis

Beside the balance ratio (BR) we present in the main context, we also evaluate the number of logic level ((Level), which indicates the depth of the circuits. A smaller number of logical levels means a more evenly distributed arrangement of the logical units. As can be observed from the Tab. 3, the optimized AIGs typically have BR values that are closer to 1 and a smaller number of logical levels compared with those of the original AIGs, showing that the distribution diversity of AIGs after optimization is reduced.

B Dataset Construction by Logic Simulation

The constructed training dataset $\mathcal{D}$ consists of tuples of $(\mathcal{G}^{(i)}, \mathbf{m}^{(i)}, \hat{\boldsymbol{\theta}}^{(i)})$, where $\mathcal{G}^{(i)}$ is a Boolean circuit, $\mathbf{m}^{(i)}$ is a conditional mask and $\hat{\boldsymbol{\theta}}^{(i)}$ is the simulated probabilities for every node in the Boolean circuit.

In the logic simulation, we aim at estimating the probability of a node being logic '1', which is used as a supervision signal in the following experiments. Specifically, the probability is estimated by counting the number of times the value of a gate becomes logic '1' when feeding N random assignments to the circuit. For example, if a gate becomes logic 1 for 1,000 times under 10,000 random patterns, then the supervision of this gate will be a real value 1,000/10,000=0.1, which is the estimated probability of this gate being logic '1'.

Empirically, we apply 15k random patterns to each AIG to construct the training data. Given a conditional mask $\mathbf{m}$, we only count the random patterns that fulfill the condition. A tiny example is shown in Figure 5. When we mask the PO as logic '1', the simulated probability of nodes will be changed accordingly.

In the previous end-to-end solutions (Selsam et al., 2019; Amizadeh et al., 2019), a classical SAT solver is used to offer single-bit of supervision (SAT/UNSAT) (Selsam et al., 2019), or exclude unsatisfiable circuits from datasets (Amizadeh et al., 2019). Since the classical SAT solver is based on some hand-crafted searching heuristic, tens of thousands of backtracking / branching are required to obtain the final satisfying solutions or give the unsatisfying certificates. On the other hands, the random simulation does not require any specialized designs, but a random input generator. Moreover, the logic simulation can be accelerated using parallelization algorithms and hardware (e.g., GPUs). Consequently, the cost of our logic simulation is on par with those of other learning-based SAT solving algorithms. For larger problems, an alternative is to first use an efficient "all solutions SAT solvers" to obtain all possible satisfying solutions, and then estimate the supervision signal $\hat{\theta}_i$ from these assignments.

Similarly to (Li et al., 2022), we observe that by applying 15k random patterns to each AIG, we can achieve a balance between training efficiency and the quality of learning. We experiment with the numbers of samples ranging from 100 to 100k on the SR(20) training dataset (Table 4). The results show that the performance stopped increasing after the number of samples reaching 10k. Note that for smaller SAT problems, we randomly generate samples without considering the repetitions. For exampple, we enumerate all possible assignments for an SR(20) instance and get the overall logic-1 probability as the ground truth. Then, we perform the logic simulation with 100K, 10K, 1K, and 0.1K samples. The table below shows the average difference between simulated logic-1 probability and ground truth. Since the final training loss of our model is about 0.04, the error caused by logic simulation with 10K samples is only 4.75%. Therefore, we believe that simulating with 10K samples offers a good trade-off between simulation runtime and error. For the feasibility of covering enough satisfiable solutions, in our case, since the training dataset SR(3)-SR(10) is rather small, it is very likely that logic simulation with 10k patterns can cover enough satisfiable solutions for estimating.

C Polarity Prototypes

C.1 Polarity Prototypes as Latent Space Regularization

We interpret the polarity prototypes as one of latent space regularization techniques. Incorporating regularized structure into latent space is preferred in many scenarios, such as data generation (Hadjeres et al., 2017; Kingma & Welling, 2013; Shen et al., 2020), unsupervised domain adoption (Kang et al., 2019), and few-show learning (Snell et al., 2017). In general, strategies belonging to this class make use of additional constraints imposed on the latent vectors, effectively reducing the extent of space each of them can occupy (Barbato et al., 2021). In this paper, we introduce two polarity prototypes with specific physical meanings (logic '1' or logic '0') into latent space, and facilitate learning a continuous and compact hidden space with good *interpretability* of logic values. More importantly, with polarity-regularized embedding space, we can manipulate the states of nodes and thus do the conditional solution sampling.

C.2 Learnable Polarity Prototypes

Similar to learnable tokens in the Transformer-based models Dosovitskiy et al. (2021); He et al. (2021); Devlin et al. (2018), we can randomly initialize two polarity prototypes and make them learnable along with the model parameters by back-propagation. By this way, the polarity prototypes can be discovered by the model automatically. In this subsection, we train another same DeepSAT model besides learnable polarity prototype and compare with the default setting (non-learnable polarity prototypes) under same training iterations. As a result, the model with learnable setting has PE = 0.1107, which under-performs than the original DeepSAT model with fixed polarity prototypes (PE = 0.0648). Therefore, we use the fixed polarity prototypes in our experiments.

D Solution Sampling Scheme

In this section, we provide more details of our proposed solution sampling scheme. The pseudo-code is provided in Algorithm 1.

To sample solutions from the well-trained conditional model, we iteratively select the undetermined PI with the highest *confidence* values predicted by the model, i.e., the PI with probability prediction closest to 0 or 1. Specifically, given an SAT instance $\mathcal{G}$ with I variables, we conduct the following *auto-regressive* procedure:

1. We mask PO as logic '1', and generate the corresponding mask vector $\mathbf{m}_0$.
2. At t iteration, we pass $(\mathcal{G}, \mathbf{m}_t)$ to well-trained DeepSAT model to estimate the simulated probability of all un-masked PIs. The PI with the highest confidence is selected and masked as logic '0' if the prediction is smaller than 0.5, otherwise logic '1'. According to the selected PI and its masked value, a new mask $\mathbf{m}_{t+1}$ is generated.
3. Repeat the second step until all PIs have been masked (I iterations in total).

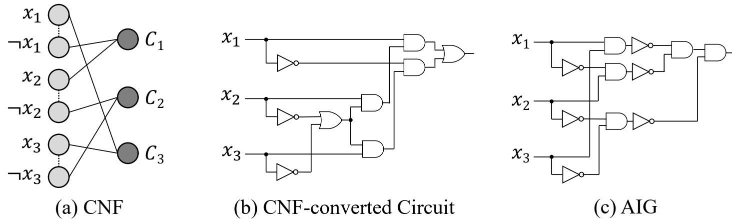
Algorithm 1: Conditional Inference with Backtracking

```
Input: The problem instance in AIG format G. The number of variables #var. The trained
       model DeepSAT.
Output: Return the satisfiable assignment (for predicted SAT instance) or empty ∅ (for
        predicted unknown instance)
   /* Iterative solving function                                                    */
 1 Function IterativeSolve(G, DeepSAT, agn, #var):
 2   | NoAgned = #var − (agn == 0) ;
 3   | O = ∅ ;
 4   | for it ← NoAgned to #var do
 5   |   | m = obtainMask(agn) ; // Generate mask
 6   |   | V = DeepSAT(G, m);
 7   |   | idx = findMostConfidentDecision(V, agn) ;
 8   |   | agn[idx] = (V < 0.5) ? −1 : 1 ;
 9   |   | O.append(idx) ; // Save decision order
10   | end
11   | return (agn, O) ;
   /* Initial solving assumption                                                    */
12 x* = [0] * #var ;
13 x*, O = IterativeSolve(G, DeepSAT, x*, #var) ;
14 if VerifySolution(G, x*) then
15   | return x* ;
16 end
   /* Flipping-based strategy with O                                                */
17 for i ← 0 to len(agn) do
18   | agn = [0] * #var ;
19   | for j ← 0 to i do
20   |   | agn[O[j]] = x*[j] ; // Pre-assign the variables
21   | end
22   | agn[O[i]] = (agn[O[i]] == 1) ? −1 : 1 ; // Flip one assigned variables
23   | agn, _ = IterativeSolve(G, DeepSAT, agn, #var) ; // Try another assumption
24   | if VerifySolution(G, agn) then
25   |   | return agn ;
26   | end
27 end
28 return ∅ ; // Fail to find assignment, mark the instance as unknown
```

To explore more possible assignments to satisfy the SAT instance, we also develop a simple *flipping*-based strategy (line 17 - line 27 in Alg. 1) to sample more solutions from the conditional model ($(I+1)$ solutions in the worst case). To be specific, during the initial solution sampling step, we record the masking order in the iteratively solving procedure as $\mathbf{O} = \{o_1, o_2, \ldots, o_n\}$. For the round r, where $r = 1, 2, \ldots, n$, we modify the initial mask $\mathbf{m}_0^r$ as Eq. 9, where flip the rth approximated assignment and keep the former assignment as the initial mask $\mathbf{m}_0^r$ to estimate another possible solution.

$$m_{O_i} = \begin{cases} x^*_{O_i} & i < r \\ \neg x^*_{O_i} & i = r \\ 0 & i > r \end{cases} \tag{9}$$

For example, if we get the initial solution $\mathbf{x}^* = \{0, 0, 0\}$ for a three variable SAT instance and the iterative order is $\mathbf{O} = \{v_3, v_2, v_1\}$. For the round $r = 1$, the initial mask should be $\mathbf{m}_0^1 = \{m_{PO} = 1, m_{O_r} = \neg x^*_{O_r}\}$, i.e., $\mathbf{m}_0^1 = \{m_{PO} = 1, m_{v_3} = 1\}$. For the round $r = 2$, the initial mask should be $\mathbf{m}_0^2 = \{m_{PO} = 1, m_{v_3} = 0, m_{v_2} = 1\}$. The above re-sampling procedure is executed until finding a satisfying solution or exceeding round n.



(a) CNF (b) CNF-converted Circuit (c) AIG

Figure 6: Three graph representations of the Boolean formula in Equation 10.

Table 5: The training and model setting of default DeepSAT

	Configuration	Value
Training	Optimizer	Adam
	Learning rate	1e-4
	Weight decay	1e-10
	Training epochs	20
	Batch size	64
Model	Hidden vector dimension	64
	Aggregator	Attention
	Update function	GRU
	Regressor	3-layer MLP with ReLU

E Implementation Details and Comparison to Related Models

E.1 Different Circuit formats

NeuroSAT consumes CNF as a bipartite graph, while DG-DAGRNN and DeepSAT can process various circuit formats. Here we give an example of SAT instances, and visualize it under different circuit formats in Figure 6. The example we use is Equation 10 in the CNF format, which consists of 3 variables.

$$\phi := (\neg x_1 \vee x_2) \wedge (\neg x_2 \vee \neg x_3) \wedge (x_1 \vee x_3) \tag{10}$$

E.2 Implementation Details of DeepSAT

DeepSAT consists of one forward propagation layer and one reverse propagation layer. In particular, we use a single-layer GRU and a attention-based aggregation function to instantiate each propagation layer. The forward propagation layer and reverse propagation layer do not share the parameters. The dimension of node hidden states is set as 64. The regressor is a 3-layer MLP with hidden dimension 64. We train DeepSAT for 60 epochs with batch-size 128 on 4 Nvidia V100 GPUs. The Adam optimizer (Kingma & Ba, 2015) is adopted with the learning rate 10^{-4} and weight decay 10^{-10}. We use the topological batching technique introduced in (Thost & Chen, 2021) to accelerate the training. The details of the model setting and training setting are listed in Tab. 5

E.3 Implementation Details of NeuroSAT

NeuroSAT supports the instance in CNF format. We reproduce the NeuroSAT based on the original configurations. The message-passing is performed for 10 iterations between literals and clauses during training. The dimension of node hidden states is 128.

E.4 Implementation Details of DG-DAGRNN

DG-DAGRNN is a GNN model specialized for Circuit-SAT problems. Following the the original paper, we set the number of node hidden states is 100. We also adjust temperature τ in evaluator

network dynamically as $\tau = t^{-0.4}$, where t is the number of training epochs. The k in loss function is set 10 in the following experiment. Besides, we set the number of GNN iterations as 10.

Besides the model configuration in Amizadeh et al. (2019), we also provide the modified configuration to further improve the model performance. 1) Since the CNF is converted into a circuit based on Cube and Conquer paradigm Heule et al. (2011), the number of fan-in wires for different gates is not fixed. The original aggregation function in GNN sums up all messages from predecessors without considering the various number of fan-in wires. We replace the original aggregation with an attention-based aggregation function Li et al. (2022) to calculate the weighted sum. 2) The original DG-DAGRNN includes a linear projection to project the hidden state into lower dimension vectors and feedback as the next layer inputs. The information about gate type loses after multiple message-passing rounds. We remove the linear projection and directly feedback the concatenation of hidden state and one-hot encoded gate type into next layer propagation. 3) The SAT instance in the original training dataset consists of a large number of clauses. Usually, there are only 1 or 2 out of 2^n satisfiable assignment for a SR(n) instance. Find an assignment to satisfy such instance is extremely difficult. We augment the training dataset with many simple instances by removing the clauses from the difficulty instances.

We try to reproduce DG-DAGRNN (including the original configuration and our modified configuration described above) and also apply the same EDA optimizations to it, as the official code of DG-DAGRNN remains publicly unavailable up to now. In order to test the correctness of our implementation, we try our re-implementation of DG-DAGRNN on SR(3) problems and train for 200 epochs. As the instances in the training dataset are easily satisfied by random guessed assignment, the initial testing accuracy is 33%. After training for the first 86 epochs, the testing accuracy improves only 1%. The model converges after 100 epochs and only achieves 44% testing accuracy. Although we explore the other configurations with different k values in the loss function ($k = 0.1, 1, 5, 10$), the model still converges slowly and only improves a little testing accuracy. Unfortunately, when we train the model on the same dataset in DeepSAT (Section 4), with all the efforts, we still fail to train DG-DAGRNN to converge.

We attribute the difficulty of training DG-DAGRNN to its vulnerability to training crash. On the one hand, DG-DARGNN employs a soft-evaluator as the reward network and is a variant of Policy Gradient method during training. As we observe empirically, the training is sensitive to several hyper-parameters and is extremely difficult to optimize even we follow the same setting described in the original paper. On the other hand, increasing the number of logical levels would result in a large stack of smoothed max and min functions in DG-DAGRNN. Consequently, the gradient tends to vanish and thus the training tends to fail. Moreover, even though we cannot reproduce DG-DAGRNN, the results DeepSAT obtains is still better than the ones reported in the original paper.

Note that we will add DG-DAGRNN as another baseline once the authors release the code.

E.5 Comparison to Related Works

We compare DeepSAT with NeuroSAT in the main text, and describe the reason why we do not consider DG-DAGRNN as the baseline above. Besides NeuroSAT and DG-DAGRNN, there are other existing end-to-end SAT solutions. We enumerate the published works here and elaborate the differences between these models and DeepSAT. In addition, we give the explanation why we consider NeuroSAT as the strong baseline in our experiments.

ContrastiveSAT (Duan et al., 2022) studies contrastive learning on end-to-end SAT solvers. They demonstrate that using label-preserving augmentations of CNFs can learn a meaningful representation, which can achieve achieve comparable test accuracy to fully-supervised learning while using only 1% of the labels for finetuning. The baseline is also NeuroSAT. The code of this work is not open-sourced and they only argue a similar performance of proposed method compared with NeuroSAT. In addition, they follow a unsupervised setting, which is differs from ours. Thus, we do not consider ContrastiveSAT as another baseline.

(Yolcu & Póczos, 2019; Kurin et al., 2020) use Reinforcement Learning to train branching heuristic in stochastic local search (SLS) solvers and conflict driven clause learning (CDCL) solvers. The input representation in these two works all follows the one used in NeuroSAT: a bipartite graph

Table 6: The number of problems solved v.s. the number of sampled solutions for SR(10)

# Sample Solutions	1	2	3	4	5	6	7	8	9	10	11
# Problem Solved	72	8	11	1	2	2	1	0	1	0	0

Table 7: The number of problems solved v.s. the number of sampled solutions for SR(20)

# Sample Solutions	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15	16	17	18	19	20	21
# Problem Solved	66	5	4	2	0	2	1	0	1	2	0	0	0	0	1	1	0	0	0	0	0

representation defined by CNF. More importantly, the design principle of graph neural networks is the same with NeuroSAT, where the message passing is iteratively conducted between literals and clauses. These two works are learning-aided SAT solvers, and in the original works the baselines are classical SAT solvers. Therefore, we do not consider them as the baselines for comparison. Note that the proposed DeepSAT has the potential to aid the conventional Circuit-SAT solvers, and we plan to explore this direction in the future work.

NLocalSAT (Zhang et al., 2021b) is a learning-aided SAT solver, which boosts the stochastic local search (SLS) solvers by initializing the assignments, which is compared with classical SLS solvers such as CCAnr and YalSAT. Though NLocalSAT can also be evaluated as an end-to-end SAT solver, the model structure of NLocalSAT is almost the same as NeuroSAT, expect the authors modify the learning objective. Hence, we compare our model with NeuroSAT directly.

F Discussion on evaluation metric for end-to-end SAT solvers

Since the proposed DeepSAT is an incomplete solver, it predicts a problem as SAT if and only if it finds a satisfying assignment, otherwise returns UNKNOWN. To this end, if we evaluate DeepSAT on both SAT and UNSAT instances, the false positive rate (FP) will be 0 because the UNSAT ones would never be predicted as SAT by DeepSAT. Moreover, we focus on generating a solution for SAT instead of the binary SAT/UNSAT prediction problem, since the latter is less attractive unless a satisfying assignment can be given for every instance that is predicted as SAT. Therefore, we only consider testing on satisfiable instances. Note that testing only on satisfiable instances is also adopted in DG-DAGNN.

G Additional Analysis

In this section, we conduct ablation studies to investigate the effectiveness of our proposed methods. To enable a fair comparison, all models are trained for 10 epochs. Beside the evaluation metric *Problem Solved* defined in Section 4.1, we also report *Prediction Error* (PE), the least absolute error between the prediction and the supervision label. PE can be treated as the loss of models during training and validation, reflecting how well the model fits into the datasets. In general, PE has a strong correlation with the metric *Problem Solved*. The smaller PE is, the higher *Problem Solved* model can achieve. Besides, we construct a validation SAT dataset with 100 SAT instances.

G.1 Solution Sampling Scheme

The number of re-sampling varies for different problems. We show the statistics of the number of sampling solutions for solving 100 SR(10) problems (Table 6) and 100 SR(20) problems (Table 7). As can be observed, DeepSAT samples 1.6 solutions on average for SR(10) and terminates when the latest sampled solutions are satisfying. In terms of the number of iterations, the results are still better than the baselines. The result on 100 SR(20) shows the same tendency.

Please note that even though the DeepSAT produces I+1 solutions in the worst case (‘I=number of vars‘), in most cases, the proposed method can generate satisfying assignments in the first few sampling rounds.

Table 8: The comparison of directed and undirected model

Model	Aggregator	Prediction Error	Problem Solved
Undirected	Conv. Sum	0.4296	1%
	DeepSet	0.4298	0%
	GatedSum	0.4299	0%
	Attention	0.4297	1%
Directed	Conv. Sum	0.0669	33%
	DeepSet	0.0651	57%
	GatedSum	0.0675	35%
	Attention	0.0648	64%

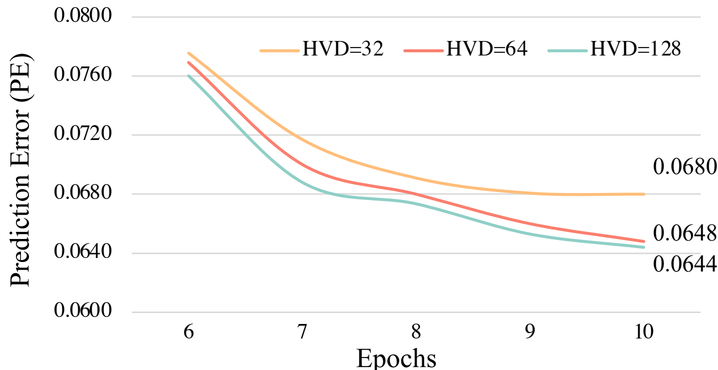


Figure 7: The comparison of models with different hidden vector dimension for prediction errors.

Furthermore, to explore the effectiveness of our proposed solution sampling scheme in Section 3.5, we design other three simplified sampling schemes and use 100 SR(10) problems to evaluate them. In the first sampling scheme, we only mask the PO and use DeepSAT to get the probabilities of sat assignments on PIs. By setting the threshold at 0.5, we obtained 4% accuracy under this setting. In the second inference setting, we again mask the PO, but use DeepSAT iteratively to achieve the SAT assignment on each PIs one by one. During each iteration, we choose the PI with most uncertain probability (i.e., PI with probability around 0.5) to be masked. We obtained only 1% accuracy under this setting. In the third inference setting, we again mask the PO and use DeepSAT iteratively to achieve the SAT assignment on each PIs one by one. But during each iteration, we choose the PI with most certain probability i.e., either close to 1 or close to 0, to be masked. We obtained 70% accuracy under this setting.

To sum up, the proposed inference scheme outperforms other simplified sampling schemes. Note that in this work, we keep the solution sampling scheme as simple as possible. We expect designing more complex sampling designs to be able to further improve DeepSAT's performance. For example, we can use Reinforcement Learning to train an agent to decide which PI to be masked during sampling. We leave it for future work.

G.2 Effectiveness of Directed Acyclic GNNs

To verify the effectiveness of our DAG-based design in Section 3.3, we compare the performance of *Directed* GNN models and *Undirected* GNN models. For the undirected GNN, we replace the forward and reverse propagation as two undirected propagation layers, where the propagation is conducted bidirectionally and simultaneously. Both models are equipped with the attention-based aggregator Thost & Chen (2021); Veličković et al. (2018) as Equation 7. The experimental results are shown in Table 8. The directed model with Attention aggregator shows PE = 0.0648 and solves 64% SAT problems. Not surprisingly, since the undirected model does not capture the relational bias in circuits, such a model performs poorly in representing circuits. In addition, the undirected model with the same aggregator almost solves no SAT problems because the model predicts the masked simulation probability quite inaccurate (PE = 0.4297).

G.3 DIFFERENT AGGREGATION FUNCTION

In order to demonstrate the effectiveness of aggregation function in Equation 7, we compare Attention-based aggregation with other 3 different aggregator designs, which include representative works for DAG learning, i.e., Convolutional Sum (abbr. Conv. Sum) Selsam et al. (2019), Gated Sum Zhang et al. (2019) and DeepSet Amizadeh et al. (2019). The attention-based aggregator has the lowest PE (PE = 0.0648) than the other three aggregators, including the Conv. Sum (PE = 0.0669), Gated Sum (PE = 0.0675) and DeepSet (PE = 0.0651) (see Tab. 8). We suspect that the superiority of the attention mechanism comes from its modeling capacity for logic computation. Specifically, when we do the logic computation in digital circuits, the controlling value of a logic gate determines the output of that gate. Therefore, controlling values are far more important than non-controlling values. To mimic this behaviour, the attention mechanism can learn to assign high weights for controlling inputs of gates and give less importance to the rest of the inputs. allocates a higher attention weight to these dominant node neighbors. In summary, the DAG-based models with an attention-based aggregator achieves the best results, compared to other kinds of aggregation functions.

G.4 HIDDEN VECTOR DIMENSION

We investigate the influence of the hidden vector dimension (HVD) by training three models with three different setting: HVD = 32, 64 and 128, respectively. The prediction errors of various configurations in the last 5 epochs are shown in Figure 7. With the increasing hidden vector dimension, the prediction error decreases. Yet, we observe that with more training iterations, the performance gap between models with latent vector dimension as 64 and 128 becomes less significant. At training epochs 10, the PE gap is only 0.0004, about 0.62% PE of the model with 64 hidden vector dimension. Therefore, we set the latent vector dimension as 64 in the experiments.