HYPERGEN: A MASKED VARIATIONAL HYPERGRAPH FRAMEWORK FOR HARD UNSAT INSTANCE GENERATION

Anonymous authors
Paper under double-blind review

ABSTRACT

Realistic and challenging Boolean satisfiability (SAT) instances are essential for evaluating modern SAT solvers and training learning-based solver components, yet such instances are often scarce in application-specific domains. Existing SAT generators either rely on handcrafted structural priors or generate new formulas through predefined graph transformations, which do not explicitly model how clauses are formed. In this paper, we propose HyperGEN, a masked variational hypergraph framework for hard UNSAT instance generation. HyperGEN represents each formula in conjunctive normal form as a literal-clause hypergraph, where literals are vertices and clauses are hyperedges. It formulates instance generation as masked hyperedge reconstruction. HyperGEN employs a masked variational hypergraph autoencoder that combines a hypergraph neural network for structural encoding with latent variables, and reconstructs each masked clause through arity and membership prediction. Experiments on real-world SAT benchmarks show that HyperGEN consistently generates more challenging instances than existing generative baselines, generalizes to larger unseen instances without retraining, and provides effective data augmentation for downstream solver-runtime prediction.

1 INTRODUCTION

The Boolean satisfiability (SAT) problem is a fundamental combinatorial problem with broad applications in automated reasoning, formal verification, and artificial intelligence (Cook, 1971; Biere et al., 2009; Chen et al., 2025). Modern SAT solvers have achieved substantial practical success, yet their development and evaluation rely heavily on representative and challenging benchmark instances (Audemard & Simon, 2018; Cai & Zhang, 2021; Biere & Fleury, 2022). Moreover, learning-based components for SAT solving, such as solver selection, runtime prediction, and branching heuristics, further increase the demand for diverse training instances (Xu et al., 2008; Wang et al., 2024). However, hard real-world SAT instances are often scarce and costly to obtain, particularly for specific application domains (Wu & Ramanujan, 2019). This creates a growing need for generative methods that can produce realistic and challenging SAT instances for solver evaluation and data augmentation.

Early SAT instance generators mainly relied on random or handcrafted structural principles, including random k-SAT, Community Attachment (Giráldez-Cru & Levy, 2015), and Popularity-Similarity models (Giráldez-Cru & Levy, 2017). More recently, learning-based methods such as SAT-GEN (Wu & Ramanujan, 2019) and G2SAT (You et al., 2019) have attempted to learn structural patterns directly from existing SAT formulas. Despite these advances, generated formulas often exhibit hardness collapse: they may resemble real instances in selected graph statistics, yet become much easier to solve (Cotnareanu et al., 2024). Recent hardness-oriented methods, such as HardSATGEN (Li et al., 2023) and HardCore (Cotnareanu et al., 2024), improve the generation of challenging SAT instances, but still rely primarily on predefined graph transformations or core manipulation. Although these operations directly modify clauses and literals, they do not explicitly model how clauses are formed. This motivates a generative method that directly learns how logical constraints are formed from data, rather than obtaining them indirectly through predefined structural edits, while maintaining instance hardness.

Reviewers: please read the Reviewer Guidelines (iclr.cc/Conferences/2027/ReviewerGuidelines) and the AI Policy for Reviewers (iclr.cc/Conferences/2027/AIPolicyForReviewers). If you used AI to expand, edit, or polish your review, please provide the input text to the LLM. Better still, consider skipping the LLM and submitting your original text: we, and the authors, are much more interested in your unedited thoughts than in what an LLM has to say. AI-assisted or not, you are putting your name and reputation behind your review: LLM-generated falsehoods, hallucinations or misrepresentations are subject to disciplinary action, which may include desk-rejecting all papers you have authored.

In this work, we focus on this challenge in the setting of hard UNSAT instance generation, which is particularly relevant for evaluating SAT solvers and studying proof-search behavior. To this end, we propose HyperGEN, a generative framework that directly models clause formation while preserving instance hardness. We represent each formula in conjunctive normal form (CNF) as a hypergraph, where literals are vertices and clauses are hyperedges, since a clause naturally constitutes a variable-size set of literals and is therefore better captured as a higher-order relation than a collection of independent pairwise edges. Generating a CNF formula from scratch requires jointly determining its clause count, clause arities, and literal memberships, leading to a large combinatorial generation space. HyperGEN therefore adopts masked clause reconstruction to generate new instances by selectively rewriting existing clauses. To generate diverse SAT instances, we develop a masked variational hypergraph autoencoder (VHAE). A hypergraph neural network (HyperGNN) encoder captures the structural context of the complete formula and clause-level latent variables capture variations in how clauses are composed and enable stochastic sampling during generation. Conditioned on the masked hypergraph and sampled latent variable, the decoder reconstructs each target clause through arity prediction and literal-membership prediction, directly modeling how clauses are formed. We further employ pre-generation core scrambling and hardness refinement to increase instance diversity and mitigate newly introduced small unsatisfiable cores. Extensive experiments on real-world SAT formulas demonstrate the effectiveness of HyperGEN. HyperGEN consistently outperforms the compared generative baselines in terms of instance hardness, producing formulas that remain challenging for modern SAT solvers. It also generalizes effectively to larger unseen instances without retraining. Moreover, instances generated by HyperGEN provide effective data augmentation and improve downstream prediction of solver runtime.

Our main contributions are summarized as follows:

- **Clause-level hypergraph formulation.** We formulate hard UNSAT instance generation as a masked clause-hyperedge reconstruction problem on literal-clause hypergraphs. By treating each clause as a variable-size hyperedge, this formulation directly models clause composition from literals.
- **Masked variational instance generation.** We propose HyperGEN, a masked variational hypergraph autoencoder that generates new instances by reconstructing masked clauses rather than generating an entire CNF formula from scratch. HyperGEN combines HyperGNN-based structural encoding, clause-level latent variables, and an arity–membership decoder to support diverse instance generation while preserving hardness.
- **Comprehensive empirical evaluation.** Experiments on real-world SAT benchmarks show that HyperGEN consistently generates more challenging SAT instances than the generative baselines, achieving an average relative hardness of 110.08%. HyperGEN also retains 94.94% relative hardness on larger unseen instances without retraining and reduces the MAE of downstream solver-runtime prediction by 15.51% through data augmentation.

2 Preliminaries: Boolean Satisfiability and Unsatisfiable Cores

A Boolean Satisfiability problem instance F is typically expressed in CNF. A CNF formula consists of a set of n boolean variables $\mathcal{X} = \{x_1, x_2, \ldots, x_n\}$ and a set of m clauses $\mathcal{C} = \{C_1, C_2, \ldots, C_m\}$. Each variable x_i can take a binary value $\{0, 1\}$. For each variable x_i, there are two literals, x_i and $\neg x_i$. We denote the set of all literals by $\mathcal{L} = \{x_i, \neg x_i\}_{i=1}^{n}$. The formula F is a conjunction of all clauses and is written as $F = C_1 \wedge C_2 \wedge \cdots \wedge C_m$. Each clause C_j is a disjunction of literals: $C_j = \ell_{j1} \vee \ell_{j2} \vee \cdots \vee \ell_{jk_j}$, where $\ell_{jt} \in \mathcal{L}$ and k_j is the clause size.

A truth assignment $\alpha : \mathcal{X} \to \{0, 1\}$ assigns a Boolean value to each variable and naturally extends to literals. A clause is satisfied if at least one of its literals evaluates to true, and a CNF formula is satisfied if all of its clauses are satisfied. The SAT problem asks whether there exists an assignment α that satisfies F. If such an assignment exists, F is satisfiable; otherwise, it is unsatisfiable.

For an unsatisfiable CNF formula $F = \bigwedge_{C_j \in \mathcal{C}} C_j$, an unsatisfiable core is a subcollection of clauses $\mathcal{U} \subseteq \mathcal{C}$ such that the subformula $F_{\mathcal{U}} = \bigwedge_{C_j \in \mathcal{U}} C_j$ is still unsatisfiable (Lynce & Marques-Silva, 2004).

3 Related Work

3.1 SAT Instance Generation

SAT instance generation has long been studied for understanding solver behavior and constructing benchmarks for empirical evaluation (Balyo et al., 2023; Heule et al., 2024). Classical generators mainly focus on random k-SAT formulas, where hard instances often emerge near the satisfiability phase transition (Selman et al., 1996; Crawford & Auton, 1996). While random SAT provides a clean and controllable setting, its structural properties differ substantially from those of industrial formulas. Subsequent studies observed that real-world SAT instances exhibit rich structures, including scale-free variable interactions, modularity, and community structure (Alyahya et al., 2022). These findings motivated pseudo-industrial generators that explicitly inject such structural priors into random formulas (Giráldez-Cru & Levy, 2015; 2017). However, these approaches are largely handcrafted and do not directly learn the distribution of real SAT instances from data.

Recent learning-based methods aim to generate SAT formulas by learning structural patterns from existing instances. SAT-GEN (Wu & Ramanujan, 2019) represented formulas as Literal-Incidence Graphs (LIGs) and used Graph Generative Adversarial Networks (GANs) (Goodfellow et al., 2020) to generate new formulas. G2SAT (You et al., 2019) formulated SAT generation as a graph generation problem and modified formulas through graph-level split-and-merge operations. HardSATGEN (Li et al., 2023) further introduced a fine-grained control mechanism considering community structure and unsatisfiable core to the neural split-merge paradigms. HardCore (Cotnareanu et al., 2024) proposed a core detection procedure to produce challenging UNSAT problems. MixSATGEN (Chen et al., 2024) introduced an adaptive graph interpolation approach that mixes graph structures from different instances.

3.2 Core Prediction

Unsatisfiable cores have been widely studied in SAT solving, proof checking, and formula simplification (Selsam & Bjørner, 2019; Wetzler et al., 2014). Prior work has suggested that unsatisfiable cores can be correlated with the hardness of UNSAT instances (Ansótegui et al., 2008; Cotnareanu et al., 2024). Recently, HardSATGEN (Li et al., 2023) employs the classical proof checking utility DRAT-trim (Wetzler et al., 2014) to identify small unsatisfiable cores that may be introduced during generation, and further designs a simple core-elimination procedure to better preserve challenging formula properties. However, this process is computationally expensive due to the repeated invocation of DRAT-trim. HardCore (Cotnareanu et al., 2024) proposed a Graph Neural Network (GNN)-based framework for predicting unsatisfiable cores and introduced an iterative core-refinement procedure to generate hard UNSAT instances.

4 Methodology

In this section, we present HyperGEN for hard UNSAT instance generation. As illustrated in Fig. 1, we first represent CNF formulas as literal-clause hypergraphs and formulate the generation task as masked clause-hyperedge reconstruction. HyperGEN then performs masked clause-hyperedge reconstruction using a HyperGNN encoder, clause-level variational modeling, and a HyperGNN decoder followed by arity and membership predictors. The generation pipeline further incorporates pre-generation scrambling and hardness refinement.

4.1 SAT Instance Generation

4.1.1 CNF as a Literal-Clause Hypergraph

Given a CNF formula $F = \bigwedge_{j=1}^{m} C_j$ with variables $\mathcal{X} = \{x_1, \ldots, x_n\}$, we represent it as a literal-clause hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{H})$. Different from conventional graph representations that model pairwise variable-clause relations (Li et al., 2024), we exploit the intrinsic hypergraph structure of CNF formulas. The node set $\mathcal{V} = \{v_1, \ldots, v_{2n}\}$ consists of $2n$ literal nodes induced by $\mathcal{X}$. Specifically, v_i corresponds to the positive literal x_i, and v_{n+i} corresponds to the negative literal $\neg x_i$, for $i = 1, \ldots, n$. Thus, each Boolean variable x_i is associated with two distinct literal nodes,

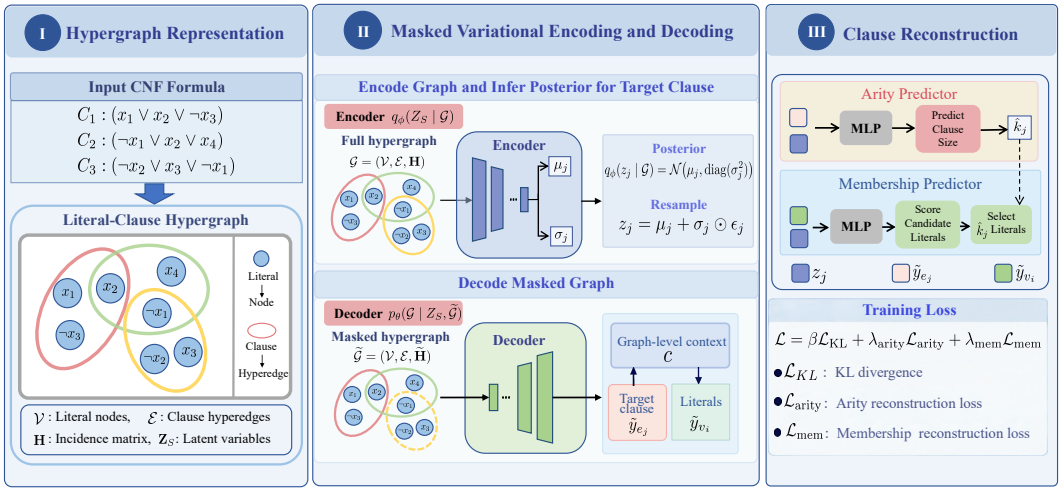


Figure 1: An overview of HyperGEN framework.

v_i and v_{n+i}. The hyperedge set $\mathcal{E} = \{e_1, \ldots, e_m\}$ corresponds to the clause set $\mathcal{C}$. Each clause $C_j \in \mathcal{C}$ induces a hyperedge $e_j \in \mathcal{E}$ connecting all literal nodes that appear in C_j. If $C_j = \ell_{j1} \vee \ell_{j2} \vee \cdots \vee \ell_{jk_j}$, then $e_j = \{v(\ell_{j1}), v(\ell_{j2}), \ldots, v(\ell_{jk_j})\}$, where $v(\ell)$ denotes the literal node corresponding to literal ℓ. Thus, each clause is represented as a variable-size hyperedge.

The incidence matrix $\mathbf{H} \in \{0,1\}^{|\mathcal{V}|\times|\mathcal{E}|}$ encodes the literal-clause incidence relation, where $\mathbf{H}_{i,j} = 1$ iff literal node v_i is incident to clause hyperedge e_j. The degree of literal node v_i is $d_i = \sum_{j=1}^{m} \mathbf{H}_{i,j}$. We refer to the size of clause hyperedge e_j as its arity and denote it by $k_j = |e_j| = \sum_{i=1}^{2n} \mathbf{H}_{i,j}$. Through the above construction, each SAT instance F is uniquely modeled as a hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{H})$, which contains all information of the original CNF formula.

4.1.2 Generation Objective

Given a literal-clause hypergraph $\mathcal{G} = (\mathcal{V}, \mathcal{E}, \mathbf{H})$, we construct a corrupted hypergraph $\widetilde{\mathcal{G}}$ by masking the incidence pattern of selected clause hyperedges. HyperGEN learns to reconstruct the original hypergraph from its corrupted version by modeling $p_\theta(\mathcal{G} \mid \widetilde{\mathcal{G}})$. Unlike standard graph reconstruction, the masked units are entire clause hyperedges, not individual nodes or pairwise edges. This objective reflects the SAT-specific structure that clauses are variable-size sets of literals. The corresponding variational formulation and training objective are introduced in Section 4.2.

4.2 The Masked Variational Hypergraph Autoencoder

4.2.1 Variational Evidence Lower Bound

We formulate SAT instance generation as a conditional probabilistic modeling problem. Given a literal-clause hypergraph $\mathcal{G}$, the masking strategy produces a corrupted hypergraph $\widetilde{\mathcal{G}} \sim \widetilde{p}(\widetilde{\mathcal{G}} \mid \mathcal{G})$. Our goal is to learn a latent variable model of the conditional distribution $p_\theta(\mathcal{G} \mid \widetilde{\mathcal{G}})$ by maximizing its log-likelihood. The unmasked structure remains unchanged, so the conditional generation task focuses on reconstructing the masked clause hyperedges.

We denote the set of masked clause hyperedges as $S \subseteq \mathcal{E}$, where each element $e_j \in S$ corresponds to a clause whose incidence information is removed in the corrupted hypergraph $\widetilde{\mathcal{G}}$. We introduce a matrix of clause-level latent variables $\mathbf{Z}_S = [\mathbf{z}_j]_{e_j \in S} \in \mathbb{R}^{|S|\times d_z}$, where each masked clause hyperedge e_j is associated with a latent variable $\mathbf{z}_j$. During training, we approximate the posterior distribution over $\mathbf{Z}_S$ by $q_\phi(\mathbf{Z}_S \mid \mathcal{G})$, which is parameterized by an encoder network. Following the variational principle, the conditional log-likelihood $\log p_\theta(\mathcal{G} \mid \widetilde{\mathcal{G}})$ can be bounded by the evidence

lower bound (ELBO) (Rezende et al., 2014; Kingma & Welling, 2013):

$$\log p_\theta(\mathcal{G} \mid \widetilde{\mathcal{G}}) \geq \mathbb{E}_{q_\phi(\mathbf{Z}_S|\mathcal{G})}\left[\log p_\theta(\mathcal{G} \mid \widetilde{\mathcal{G}}, \mathbf{Z}_S)\right] - D_{\mathrm{KL}}\left(q_\phi(\mathbf{Z}_S \mid \mathcal{G}) \| p(\mathbf{Z}_S)\right), \tag{1}$$

where $p(\mathbf{Z}_S)$ denotes the prior distribution over the latent representations, with a standard Gaussian prior.

4.2.2 Core-Preserving Masking Strategy

Standard graph masking typically selects nodes or edges uniformly at random. However, clauses in an UNSAT formula play different structural roles, particularly those belonging to its unsatisfiable core (Li et al., 2023; Cotnareanu et al., 2024). To preserve the contradictory substructure responsible for unsatisfiability, we adopt a core-preserving masking strategy. Let $\mathcal{E}_{\mathrm{core}} \subseteq \mathcal{E}$ denote the clause hyperedges in the unsatisfiable core and $\mathcal{E}_{\mathrm{rest}} = \mathcal{E} \setminus \mathcal{E}_{\mathrm{core}}$ the remaining clause hyperedges. For each input hypergraph, we randomly select a target set $S \subseteq \mathcal{E}_{\mathrm{rest}}$ and jointly mask their incidences to obtain $\widetilde{\mathcal{G}}$, while preserving all unselected incidences. Formally, the masked hypergraph is defined as $\widetilde{\mathcal{G}} = (\mathcal{V}, \mathcal{E}, \widetilde{\mathbf{H}})$, where

$$\widetilde{\mathbf{H}}_{i,t} = \begin{cases} 0, & e_t \in S, \\ \mathbf{H}_{i,t}, & e_t \notin S. \end{cases} \tag{2}$$

The initial embedding of each masked clause hyperedge $e_j \in S$ is replaced by a learnable mask embedding $\mathbf{h}_{[\mathrm{MASK}]} \in \mathbb{R}^d$.

4.2.3 Encoder

Rather than compressing a CNF formula into a single graph-level vector, we follow the variational graph autoencoder paradigm (Kipf & Welling, 2016) and infer latent variables associated with masked clause hyperedges. We approximate the posterior distribution over these latent variables by a variational distribution $q_\phi(\mathbf{Z}_S|\mathcal{G})$. Specifically, we adopt a factorized Gaussian variational posterior:

$$q_\phi(\mathbf{Z}_S \mid \mathcal{G}) = \prod_{e_j \in S} q_\phi(\mathbf{z}_j \mid \mathcal{G}), \quad q_\phi(\mathbf{z}_j \mid \mathcal{G}) = \mathcal{N}\left(\boldsymbol{\mu}_j, \mathrm{diag}\left(\boldsymbol{\sigma}_j^2\right)\right), \tag{3}$$

where $\boldsymbol{\mu}_j = f_\phi^\mu(\mathbf{y}_{e_j})$ and $\log \boldsymbol{\sigma}_j^2 = f_\phi^\sigma(\mathbf{y}_{e_j})$, with f_ϕ^μ and f_ϕ^σ denoting learnable mappings for the mean and log-variance vectors, respectively. Here, $\mathbf{y}_{e_j}$ denotes the representation of clause hyperedge e_j obtained by a HyperGNN encoder operating on the complete hypergraph $\mathcal{G}$. To enable gradient-based optimization through stochastic latent variables, we apply the reparameterization trick:

$$\mathbf{z}_j = \boldsymbol{\mu}_j + \boldsymbol{\sigma}_j \odot \boldsymbol{\epsilon}_j, \quad \text{with} \quad \boldsymbol{\epsilon}_j \overset{\text{i.i.d.}}{\sim} \mathcal{N}(\mathbf{0}, \mathbf{I}). \tag{4}$$

Here, $\odot$ denotes element-wise multiplication.

4.2.4 Decoder

The decoder reconstructs the masked clause hyperedges from the corrupted hypergraph $\widetilde{\mathcal{G}}$ and the sampled latent representations $\mathbf{Z}_S$. Specifically, a HyperGNN decoder processes $\widetilde{\mathcal{G}}$ to obtain contextual representations for clause hyperedges and literal nodes, denoted by $\widetilde{\mathbf{Y}}_{\mathcal{E}}$ and $\widetilde{\mathbf{Y}}_{\mathcal{V}}$, respectively. Detailed architectures of the encoder and decoder are provided in Appendix A.1. For each masked clause hyperedge $e_j \in S$, the decoder combines its latent representation $\mathbf{z}_j$ with the contextual representations to predict its arity k_j and literal membership. The arity specifies the number of literals contained in the clause, while the membership determines which literal nodes are incident to e_j. Accordingly, the decoding distribution is factorized as

$$p_\theta(\mathcal{G} \mid \mathbf{Z}_S, \widetilde{\mathcal{G}}) = \underbrace{\prod_{e_j \in S} p_\theta(k_j \mid \mathbf{z}_j, \widetilde{\mathcal{G}})}_{\text{Arity}} \cdot \underbrace{\prod_{e_j \in S} p_\theta(\mathbf{H}_{:,j} \mid k_j, \mathbf{z}_j, \widetilde{\mathcal{G}})}_{\text{Membership}}, \tag{5}$$

where $\mathbf{H}_{:,j}$ denotes the j-th column of the incidence matrix $\mathbf{H}$, indicating which literal nodes are incident to clause hyperedge e_j. The decoder uses two prediction modules: an **Arity Predictor** and a **Membership Predictor**.

Arity Predictor. This module estimates the arity of each masked clause. For a masked clause hyperedge $e_j \in S$, we concatenate its latent representation $\mathbf{z}_j$ and decoder hidden representation $\widetilde{\mathbf{y}}_{e_j}$ to predict its normalized arity:

$$\widehat{r}_j = \text{sigmoid}\left(f_{\text{arity}}\left([\mathbf{z}_j \parallel \widetilde{\mathbf{y}}_{e_j}]\right)\right), \tag{6}$$

where f_{arity} is an MLP. During training, $\widehat{r}_j$ is supervised by the normalized ground-truth arity $r_j = (k_j - k_{\min})/(k_{\max} - k_{\min})$, where $k_{\min}$ and $k_{\max}$ specify the dataset-specific arity range. During generation, the predicted value is mapped back to the arity range and rounded to obtain the predicted clause arity $\widehat{k}_j$.

Membership Predictor. This module predicts the literal membership of each masked clause. For each masked clause hyperedge $e_j \in S$ and candidate literal node v_i, we concatenate the clause-level latent representation $\mathbf{z}_j$ and the decoder hidden representation $\widetilde{\mathbf{y}}_{v_i}$ to compute a membership logit:

$$s_{\ell,j} = f_{\text{mem}}\left([\mathbf{z}_j \parallel \widetilde{\mathbf{y}}_{v_\ell}]\right), \qquad \ell = 1, \ldots, 2n, \tag{7}$$

where f_{mem} is an MLP. During training, these logits are supervised by the ground-truth membership labels $\mathbf{H}_{i,j}$. During generation, each variable x_i is assigned the score $a_{i,j} = \max\{s_{i,j}, s_{n+i,j}\}$, corresponding to the higher logit of its two literals. We sample $\widehat{k}_j$ distinct variables without replacement using weights proportional to $\exp(a_{i,j}/T)$, where $T > 0$ is a sampling temperature. For each sampled variable, we select the literal with the higher logit. This procedure constructs a clause with $\widehat{k}_j$ distinct variables without including both polarities of any variable.

Together, the two predictors reconstruct each clause as a variable-size unordered set of literals: the arity predictor controls its cardinality, while the membership predictor guides the selection of variables and their polarities.

4.3 Training and Generation

Training objective. Building on the variational formulation, we train the model to reconstruct both the arity and literal membership of each masked clause hyperedge. The overall training objective combines the corresponding weighted reconstruction losses with a KL divergence term:

$$\mathcal{L} = \beta\mathcal{L}_{\text{KL}} + \lambda_{\text{arity}}\mathcal{L}_{\text{arity}} + \lambda_{\text{mem}}\mathcal{L}_{\text{mem}}, \tag{8}$$

where β controls the KL regularization strength, and λ_{arity} and λ_{mem} weight the two reconstruction losses. The KL term regularizes the clause-level variational posterior toward the standard Gaussian prior. For arity reconstruction, we treat normalized arity prediction as a regression task and use mean squared error. We apply inverse-frequency weighting to balance the contributions of different arity values. For literal-membership reconstruction, we treat the membership of each candidate literal as a binary classification target. Since each clause contains only a small subset of the candidate literals, we use class-balanced binary cross-entropy over all candidate literals to prevent non-incident literals from dominating the loss. Detailed loss definitions are provided in Appendix A.2.

Pre-generation scrambling. Following HardSATGEN (Li et al., 2023), we apply core scrambling to the input CNF formula before generation. Scrambling randomizes the unsatisfiable core of the input formula while preserving its logical structure. Solver runtimes can vary across scrambled versions of the same formula (Le Berre & Simon, 2004), motivating the use of scrambling for SAT instance augmentation. Specifically, we perform variable permutation, clause permutation, and flipping of the literals. Details are provided in Appendix A.3.

Generation. Starting from the scrambled formula, we sequentially reconstruct M distinct non-core clause hyperedges sampled without replacement, where $M = \min\{\text{round}(\eta|\mathcal{E}|), |\mathcal{E}_{\text{rest}}|\}$. Here, $\eta \in [0, 1]$ denotes the generation masking ratio, which controls the requested fraction of clause hyperedges subjected to reconstruction. For each target e_j, we mask it in the current hypergraph to obtain $\widetilde{\mathcal{G}}$ and sample $\mathbf{z}_j$ from the standard Gaussian distribution. The decoder then generates a candidate clause conditioned on $\widetilde{\mathcal{G}}$ and $\mathbf{z}_j$. Duplicate candidates are resampled up to a fixed retry limit; if all attempts fail, the original clause is retained. Each accepted candidate replaces the target clause, updating the hypergraph for subsequent steps. The resulting formula then undergoes hardness refinement as described in Section 4.4. Since the scrambled unsatisfiable core remains unchanged throughout generation, the resulting formula remains unsatisfiable.

4.4 HARDNESS REFINEMENT

Generated clauses may introduce small unsatisfiable cores that make the resulting instances easier to solve. To mitigate this issue, we apply an iterative HyperGNN-based core refinement procedure to detect and weaken such newly formed cores inspired by core refinement methods (Li et al., 2023; Cotnareanu et al., 2024). A separately trained HyperGNN core detector predicts clause-level core scores $\mathbf{p} = f_\psi(\mathcal{G})$, where $p_j \in [0, 1]$ denotes the probability that clause e_j belongs to an unsatisfiable core. Clauses eligible for refinement are $\mathcal{E}^{(t)}_{\text{cand}} = \{e_j \in \mathcal{E} \setminus \mathcal{E}_{\text{core}} \mid p_j > \tau\}$, where τ is a threshold and $\mathcal{E}_{\text{core}}$ denotes the retained unsatisfiable core after scrambling. Once there is a clause that meets the condition, we introduce one new Boolean variable x_{new}:

$$C_j = (\ell_1 \vee \ell_2 \vee \cdots \vee \ell_k) \quad \longrightarrow \quad C'_j = (x_{\text{new}} \vee \ell_1 \vee \ell_2 \vee \cdots \vee \ell_k). \tag{9}$$

This operation relaxes the detected core by making the selected clauses easier to satisfy. The procedure is repeated iteratively. As refinement proceeds, newly introduced local cores are progressively disrupted, which prevents newly introduced short contradictions from dominating solver behavior.

5 EXPERIMENTS

5.1 EXPERIMENTAL SETTINGS

We conduct extensive experiments to evaluate the effectiveness of HyperGEN. Detailed experimental settings are provided in Appendix B, and additional analyses are reported in Appendix C.

Datasets. We utilize real-world SAT formulas MP1 and MDP from SAT Competition 2021 (Balyo et al., 2021) and 2022 (Balyo et al., 2022). MP1 is derived from combinational equivalence checking, while MDP is associated with the Learning Parity with Noise problem. To further evaluate generalization to larger instances, we utilize MDP-Large as an additional test set and directly apply the model trained on MDP without retraining. Dataset statistics are provided in Appendix B.1.

Benchmarking. We evaluate generated instances from two perspectives: hardness and downstream utility. For hardness evaluation, we run five SAT solvers, KISSAT-NEW, MOSS, UCB, ESA, and MAB-GB (Balyo et al., 2022), with a timeout of 5000 seconds. Using multiple solvers reduces dependence on solver-specific behavior and provides a more robust assessment of instance hardness. We measure runtime, decisions, and conflicts for each solver and report their averages across the five solvers as the hardness metrics. For downstream utility, we evaluate whether generated instances improve solver-runtime prediction when used as augmented training data. Performance is evaluated on benchmark instances using the mean absolute error (MAE) of the predicted total runtime.

Baseline Algorithms. We compare HyperGEN with four representative learning-based SAT instance generation methods: HardSATGEN (Li et al., 2023), HardCore (Cotnareanu et al., 2024), G2SAT (You et al., 2019), and MixSATGEN (Chen et al., 2024). HardSATGEN and HardCore explicitly target the generation of challenging SAT instances, while G2SAT and MixSATGEN provide representative graph-based generative baselines. For each method, we generate five instances from each test instance.

5.2 RESULTS

5.2.1 HARDNESS

We evaluate the hardness of generated instances on the MP1 and MDP datasets using solver runtime, decisions, and conflicts, averaged across the five SAT solvers described above. Runtime reflects practical solving cost, while decisions and conflicts characterize the search effort required by the solver. To facilitate comparison across datasets and metrics, we additionally report relative hardness (RH), defined as the ratio between each generated-set metric and the corresponding metric on the original instances. Avg. RH is the arithmetic mean of the relative runtime, decisions, and conflicts.

Table 1 shows that HyperGEN consistently preserves substantially more solver difficulty than the baselines. On MDP, HyperGEN achieves an Avg. RH of 94.62%, retaining nearly the full difficulty of the original instances. In comparison, HardSATGEN reaches 69.89% and HardCore only 17.47%, while G2SAT and MixSATGEN generate notably easier instances. On MP1, HyperGEN

Table 1: Hardness evaluation on MDP and MP1. We report mean runtime, decisions, and conflicts across the five SAT solvers, with relative hardness shown in parentheses as a percentage of the corresponding Original mean. Avg. relative hardness is the arithmetic mean of the three percentages. Higher values indicate greater instance hardness. Bold values denote the best results among generation methods within each dataset; Original serves as the reference. OOM indicates an out-of-memory failure.

Dataset	Method	Runtime (s) ↑	Decisions ↑	Conflicts ↑	Avg. RH ↑
MDP	Original	49.06	3,333,326.04	2,683,798.68	100.00%
	HardSATGEN	33.09 (67.45%)	2,367,340.06 (71.02%)	1,911,089.09 (71.21%)	69.89%
	HardCore	8.48 (17.29%)	578,457.71 (17.35%)	477,120.73 (17.78%)	17.47%
	G2SAT	0.00 (0.00%)	44.94 (0.00%)	4.26 (0.00%)	0.00%
	MixSATGEN	0.23 (0.47%)	21,902.64 (0.66%)	18,794.38 (0.70%)	0.61%
	HyperGEN	**46.42 (94.63%)**	**3,155,755.31 (94.67%)**	**2,537,645.94 (94.55%)**	**94.62%**
MP1	Original	45.30	1,928,937.18	1,265,807.20	100.00%
	HardSATGEN	50.04 (110.47%)	2,077,764.24 (107.72%)	1,368,570.56 (108.12%)	108.77%
	HardCore	49.70 (109.71%)	2,061,610.53 (106.88%)	1,356,818.93 (107.19%)	107.93%
	G2SAT	0.00 (0.00%)	0.82 (0.00%)	1.60 (0.00%)	0.00%
	MixSATGEN	OOM	OOM	OOM	—
	HyperGEN	**61.09 (134.86%)**	**2,317,747.19 (120.16%)**	**1,539,115.73 (121.59%)**	**125.53%**

Table 2: Hardness evaluation on MDP-Large. We report mean runtime, decisions, and conflicts across the five SAT solvers, with relative hardness shown in parentheses as a percentage of the corresponding Original mean. Avg. relative hardness is the arithmetic mean of the three percentages. Higher values indicate greater instance hardness. Bold values denote the best results among generation methods; Original serves as the reference.

Dataset	Method	Runtime (s) ↑	Decisions ↑	Conflicts ↑	Avg. RH ↑
MDP-Large	Original	330.40	14,159,141.06	10,886,502.31	100.00%
	HardSATGEN	32.62 (9.87%)	2,057,095.77 (14.53%)	1,649,741.45 (15.15%)	13.19%
	HardCore	39.47 (11.94%)	2,069,998.94 (14.62%)	1,622,879.61 (14.91%)	13.82%
	G2SAT	0.00 (0.00%)	39.54 (0.00%)	4.50 (0.00%)	0.00%
	MixSATGEN	0.53 (0.16%)	40,967.01 (0.29%)	33,787.39 (0.31%)	0.25%
	HyperGEN	**313.17 (94.78%)**	**13,440,334.25 (94.92%)**	**10,354,739.95 (95.12%)**	**94.94%**

achieves the highest values across all three hardness indicators, with 134.86% relative runtime, 120.16% relative decisions, and 121.59% relative conflicts, yielding an Avg. RH of 125.53%. This exceeds HardSATGEN and HardCore by 16.76 and 17.60 percentage points in Avg. RH, respectively. G2SAT produces substantially easier instances under these metrics, while MixSATGEN runs out of memory on MP1. Overall, these results demonstrate that HyperGEN consistently generates the hardest instances among the compared generative methods across both datasets, indicating its effectiveness in producing challenging instances for modern SAT solvers.

5.2.2 Generalization to Larger Instances

We further evaluate the generalization of HyperGEN to larger SAT instances using MDP-Large. MDP-Large is an additional test set from the same MDP benchmark family, but contains larger formulas than the standard MDP test set. We directly apply the model trained on MDP to MDP-Large without retraining or fine-tuning, thereby evaluating its ability to generalize to unseen instances at larger scales.

Table 2 reports the hardness results on MDP-Large. HyperGEN achieves the highest values across all three hardness indicators, retaining 94.78% of the original runtime, 94.92% of the decisions, and 95.12% of the conflicts, with an Avg. RH of 94.94%. In contrast, HardSATGEN and HardCore achieve Avg. RH values of only 13.19% and 13.82%, respectively. Notably, HyperGEN maintains nearly the same relative hardness on MDP-Large as on the standard MDP test set, despite being applied to larger unseen instances without retraining. These results demonstrate that HyperGEN generalizes effectively to larger instances within the same benchmark family.

Table 3: Downstream solver-runtime prediction performance on MP1. MAE Reduction measures the percentage decrease in prediction error relative to training on the Original data. Higher MAE Reduction indicates greater improvement.

Training Data	MAE of Runtime Prediction ↓	MAE Reduction (%) ↑
Original	1376.53	0.00
HardSATGEN	1468.28	-6.66
HardCore	1229.72	10.67
G2SAT	1290.93	6.22
HyperGEN	**1163.02**	**15.51**

5.2.3 Downstream Task

We further evaluate whether the generated instances improve downstream solver-runtime prediction. We use the SATzilla model (Xu et al., 2008) to predict solver runtime. We report the MAE of the predicted total runtime across the evaluated solvers. We also report MAE Reduction relative to training on the original data. We compare training on the original dataset alone with training on datasets augmented by instances generated by each method.

As shown in Table 3, HyperGEN achieves the lowest prediction error, reducing the MAE from 1376.53 to 1163.02, corresponding to a 15.51% reduction. This improvement is larger than those obtained with HardCore and G2SAT, which reduce the MAE by 10.67% and 6.22%, respectively. In contrast, augmentation with HardSATGEN increases the prediction error by 6.66%. These results suggest that the generated instances of HyperGEN provide useful augmentation for downstream solver-runtime prediction. The generated instances help the downstream model better estimate solver runtimes.

5.3 Analysis

We further conduct analyses to characterize the diversity, sensitivity, and refinement behavior of HyperGEN. Specifically, we assess the diversity of generated instances relative to their corresponding original formulas, examine the sensitivity of solver hardness to the generation mask ratio, and evaluate the core prediction used in the hardness refinement module. These analyses show that HyperGEN produces diverse instances, remains stable across different mask ratios, and enables more effective core identification than the HardCore strategy. Detailed results are reported in Appendix C.

6 Conclusion

We presented HyperGEN, a generative framework for producing challenging UNSAT instances. HyperGEN represents CNF formulas as literal-clause hypergraphs and formulates instance generation as masked clause-hyperedge reconstruction, enabling direct modeling of clause composition through a masked variational hypergraph autoencoder. The framework further incorporates pre-generation scrambling and hardness refinement into the generation pipeline. Experiments on real-world SAT benchmarks show that HyperGEN consistently generates more challenging instances than the compared generative baselines, generalizes to larger unseen instances without retraining, and provides effective data augmentation for downstream solver-runtime prediction. Overall, these results demonstrate the effectiveness of HyperGEN in generating diverse and challenging SAT instances.

AI use statement

In this work, we used generative AI tools for language polishing and improving the readability of the manuscript. We have not used generative AI tools for formulating mathematical claims, proposing or refining hypotheses, designing the core methodology, or generating experimental results. Additionally, we used generative AI tools for assisting with code understanding during the research process. We have reviewed all AI-assisted work. We take responsibility for the final content of this work, including text, claims or artifacts produced with the aid of generative AI.

REFERENCES

Tasniem Nasser Alyahya, Mohamed El Bachir Menai, and Hassan Mathkour. On the structure of the boolean satisfiability problem: A survey. *ACM Comput. Surv.*, 55(3), 2022. ISSN 0360-0300.

Carlos Ansótegui, Maria Luisa Bonet, Jordi Levy, and Felip Manya. Measuring the hardness of SAT instances. In *AAAI*, volume 8, pp. 222–228, 2008.

Gilles Audemard and Laurent Simon. On the glucose SAT solver. *International Journal on Artificial Intelligence Tools*, 27(01):1840001, 2018.

Tomáš Balyo, Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (eds.). *Proceedings of SAT Competition 2021: Solver and Benchmark Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, 2021.

Tomas Balyo, Marijn J.H. Heule, Ashlin Iser, Matti Järvisalo, and Martin Suda (eds.). *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, Finland, 2022.

Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (eds.). *Proceedings of SAT Competition 2023: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Series of Publications B. Department of Computer Science, University of Helsinki, 2023.

Armin Biere and Mathias Fleury. Gimsatul, IsaSAT and Kissat entering the SAT competition 2022. In *Proceedings of SAT Competition 2022: Solver and Benchmark Descriptions*, Proceedings of SAT Competitions, pp. 10–11, 2022.

Armin Biere, Hans van Maaren, and Toby Walsh. *Handbook of satisfiability*. SAGE Publications Limited, 2009.

Armin Biere, Tobias Faller, Katalin Fazekas, Mathias Fleury, Nils Froleyks, and Florian Pollitt. CaDiCaL 2.0. In Arie Gurfinkel and Vijay Ganesh (eds.), *Computer Aided Verification - 36th International Conference, CAV 2024, Montreal, QC, Canada, July 24-27, 2024, Proceedings, Part I*, volume 14681 of *Lecture Notes in Computer Science*, pp. 133–152. Springer, 2024.

Shaowei Cai and Xindi Zhang. Deep cooperation of CDCL and local search for SAT. In *Theory and Applications of Satisfiability Testing–SAT 2021*, pp. 64–81. Springer, 2021.

Qiyue Chen, Shaolin Tan, Suixiang Gao, and Jinhu Lü. Hypersat: Unsupervised hypergraph neural networks for weighted maxsat problems. *arXiv preprint arXiv:2504.11885*, 2025.

Xinyan Chen, Yang Li, Runzhong Wang, and Junchi Yan. Mixsatgen: Learning graph mixing for sat instance generation. In *The Twelfth International Conference on Learning Representations*, 2024.

Stephen A. Cook. The complexity of theorem-proving procedures. In *Proceedings of the Annual ACM Symposium on Theory of Computing*, pp. 151–158. Association for Computing Machinery, 1971.

Joseph Cotnareanu, Zhanguang Zhang, Hui-Ling Zhen, Yingxue Zhang, and Mark Coates. Hard-core generation: Generating hard UNSAT problems for data augmentation. *Advances in Neural Information Processing Systems*, 37:62409–62431, 2024.

James M. Crawford and Larry D. Auton. Experimental results on the crossover point in random 3-sat. *Artificial Intelligence*, 81(1):31–57, 1996. ISSN 0004-3702.

Jesús Giráldez-Cru and Jordi Levy. A modularity-based random SAT instances generator. In *Proceedings of the 24th International Conference on Artificial Intelligence*, pp. 1952–1958, 2015.

Jesús Giráldez-Cru and Jordi Levy. Locality in random SAT instances. In *Proceedings of the 26th International Joint Conference on Artificial Intelligence*, pp. 638–644, 2017.

Ian Goodfellow, Jean Pouget-Abadie, Mehdi Mirza, Bing Xu, David Warde-Farley, Sherjil Ozair, Aaron Courville, and Yoshua Bengio. Generative adversarial networks. *Communications of the ACM*, 63(11):139–144, 2020.

Marijn J.H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda (eds.). *Proceedings of SAT Competition 2024: Solver, Benchmark and Proof Checker Descriptions*. Department of Computer Science Report Series B. Department of Computer Science, University of Helsinki, 2024.

Diederik P Kingma. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980*, 2014.

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

Thomas N. Kipf and Max Welling. Variational graph auto-encoders, 2016.

Daniel Le Berre and Laurent Simon. The essentials of the sat 2003 competition. In Enrico Giunchiglia and Armando Tacchella (eds.), *Theory and Applications of Satisfiability Testing*, pp. 452–467, Berlin, Heidelberg, 2004. Springer Berlin Heidelberg.

Yang Li, Xinyan Chen, Wenxuan Guo, Xijun Li, Wanqian Luo, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, and Junchi Yan. HardSATGEN: Understanding the difficulty of hard SAT formula generation and a strong structure-hardness-aware baseline. In *Proceedings of the 29th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, 2023.

Zhaoyu Li, Jinpei Guo, and Xujie Si. G4SATBench: Benchmarking and advancing SAT solving with graph neural networks. *Transactions on Machine Learning Research*, 2024. ISSN 2835-8856.

Inês Lynce and João Marques-Silva. On computing minimum unsatisfiable cores. In *SAT 2004 - The Seventh International Conference on Theory and Applications of Satisfiability Testing, 10-13 May 2004, Vancouver, BC, Canada, Online Proceedings*, 2004.

Danilo Jimenez Rezende, Shakir Mohamed, and Daan Wierstra. Stochastic backpropagation and approximate inference in deep generative models. In *International conference on machine learning*, pp. 1278–1286. PMLR, 2014.

Bart Selman, David G. Mitchell, and Hector J. Levesque. Generating hard satisfiability problems. *Artificial Intelligence*, 81(1):17–29, 1996. ISSN 0004-3702.

Daniel Selsam and Nikolaj Bjørner. Guiding high-performance SAT solvers with unsat-core predictions. In *Theory and Applications of Satisfiability Testing–SAT 2019*, pp. 336–353. Springer, 2019.

Wenxi Wang, Yang Hu, Mohit Tiwari, Sarfraz Khurshid, Kenneth McMillan, and Risto Miikkulainen. NeuroBack: Improving CDCL SAT solving using graph neural networks. In *International Conference on Learning Representations*, 2024.

Nathan Wetzler, Marijn J. H. Heule, and Warren A. Hunt. Drat-trim: Efficient checking and trimming using expressive clausal proofs. In *Theory and Applications of Satisfiability Testing – SAT 2014*, pp. 422–429. Springer International Publishing, 2014.

Haoze Wu and Raghuram Ramanujan. Learning to generate industrial sat instances. In *Proceedings of the International Symposium on Combinatorial Search*, volume 10, pp. 206–207, 2019.

Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. Satzilla: portfolio-based algorithm selection for sat. *Journal of artificial intelligence research*, 32:565–606, 2008.

Jiaxuan You, Haoze Wu, Clark Barrett, Raghuram Ramanujan, and Jure Leskovec. G2sat: Learning to generate sat formulas. *Advances in neural information processing systems*, 32, 2019.

A Implementation Details

A.1 HyperGNN backbone

We provide detailed specifications of both the encoder and decoder networks. The backbone operates on two types of representations: literal-node embeddings and clause-hyperedge embeddings.

Input projection. Let $\mathbf{X}_{\mathcal{E}} \in \mathbb{R}^{|\mathcal{E}| \times d_{\mathcal{E}}^{\text{in}}}$ and $\mathbf{X}_{\mathcal{V}} \in \mathbb{R}^{|\mathcal{V}| \times d_{\mathcal{V}}^{\text{in}}}$ denote the input feature matrices of the clause hyperedges and literal nodes, respectively. Two type-specific MLPs project them into a shared d-dimensional representation space:

$$\mathbf{U}_{\mathcal{E}}^{(0)} = f_{\mathcal{E}}^{\text{in}}(\mathbf{X}_{\mathcal{E}}), \qquad \mathbf{U}_{\mathcal{V}}^{(0)} = f_{\mathcal{V}}^{\text{in}}(\mathbf{X}_{\mathcal{V}}), \tag{10}$$

where $\mathbf{U}_{\mathcal{E}}^{(0)} \in \mathbb{R}^{|\mathcal{E}| \times d}$ and $\mathbf{U}_{\mathcal{V}}^{(0)} \in \mathbb{R}^{|\mathcal{V}| \times d}$.

For the corrupted hypergraph $\widetilde{\mathcal{G}}$, the projected representation of every masked clause hyperedge is replaced by the same learnable mask vector:

$$\widetilde{\mathbf{u}}_{e_j}^{(0)} = \begin{cases} \mathbf{h}_{[\text{MASK}]}, & e_j \in S, \\ f_{\mathcal{E}}^{\text{in}}(\mathbf{x}_{e_j}), & e_j \notin S, \end{cases} \tag{11}$$

where $\mathbf{h}_{[\text{MASK}]} \in \mathbb{R}^d$. The literal representations are initialized as $\widetilde{\mathbf{U}}_{\mathcal{V}}^{(0)} = f_{\mathcal{V}}^{\text{in}}(\mathbf{X}_{\mathcal{V}})$. The input projection modules are shared between the complete and corrupted views, whereas the subsequent encoder and decoder HyperGNNs have separate parameters.

Bidirectional incidence message passing. Let $\mathbf{B} \in \{0,1\}^{|\mathcal{V}| \times |\mathcal{E}|}$ denote the incidence matrix used by a HyperGNN: $\mathbf{B} = \mathbf{H}$ for the encoder and $\mathbf{B} = \widetilde{\mathbf{H}}$ for the decoder. We define the clause-arity and literal-degree matrices as

$$\mathbf{D}_{\mathcal{E}} = \operatorname{diag}\left(\mathbf{B}^{\top}\mathbf{1}_{|\mathcal{V}|}\right), \qquad \mathbf{D}_{\mathcal{V}} = \operatorname{diag}\left(\mathbf{B}\mathbf{1}_{|\mathcal{E}|}\right). \tag{12}$$

We use $\mathbf{D}_{\mathcal{E}}^{\dagger}$ and $\mathbf{D}_{\mathcal{V}}^{\dagger}$ to denote their diagonal pseudoinverses, where the reciprocal of a zero-degree entry is defined as zero.

At layer ℓ, literal representations are first propagated to clause hyperedges:

$$\mathbf{M}_{\mathcal{E}}^{(\ell)} = \mathbf{D}_{\mathcal{E}}^{\dagger}\mathbf{B}^{\top}\mathbf{U}_{\mathcal{V}}^{(\ell)}\mathbf{W}_{\mathcal{V}\to\mathcal{E}}^{(\ell)}. \tag{13}$$

The aggregated messages are concatenated with the previous clause representations and passed through an MLP:

$$\widehat{\mathbf{U}}_{\mathcal{E}}^{(\ell+1)} = f_{\mathcal{E}}^{(\ell)}\left([\mathbf{M}_{\mathcal{E}}^{(\ell)} \,\|\, \mathbf{U}_{\mathcal{E}}^{(\ell)}]\right). \tag{14}$$

The updated clause representations are then propagated back to the literal nodes:

$$\mathbf{M}_{\mathcal{V}}^{(\ell)} = \mathbf{D}_{\mathcal{V}}^{\dagger}\mathbf{B}\widehat{\mathbf{U}}_{\mathcal{E}}^{(\ell+1)}\mathbf{W}_{\mathcal{E}\to\mathcal{V}}^{(\ell)}, \tag{15}$$

followed by

$$\widehat{\mathbf{U}}_{\mathcal{V}}^{(\ell+1)} = f_{\mathcal{V}}^{(\ell)}\left([\mathbf{M}_{\mathcal{V}}^{(\ell)} \,\|\, \mathbf{U}_{\mathcal{V}}^{(\ell)}]\right). \tag{16}$$

Here, $\mathbf{W}_{\mathcal{V}\to\mathcal{E}}^{(\ell)}$ and $\mathbf{W}_{\mathcal{E}\to\mathcal{V}}^{(\ell)}$ are learnable linear transformations, while $f_{\mathcal{E}}^{(\ell)}$ and $f_{\mathcal{V}}^{(\ell)}$ are type-specific update MLPs.

For the encoder, the provisional representations are normalized separately for clause hyperedges and literal nodes:

$$\mathbf{U}_{\mathcal{E}}^{(\ell+1)} = \text{GraphNorm}_{\mathcal{E}}^{(\ell)}\left(\widehat{\mathbf{U}}_{\mathcal{E}}^{(\ell+1)}\right), \qquad \mathbf{U}_{\mathcal{V}}^{(\ell+1)} = \text{GraphNorm}_{\mathcal{V}}^{(\ell)}\left(\widehat{\mathbf{U}}_{\mathcal{V}}^{(\ell+1)}\right). \tag{17}$$

For the decoder, the provisional representations first undergo the masked-clause contextualization described below.

Multi-target masked-clause contextualization. For notational simplicity, all representations in this paragraph refer to decoder states computed from the corrupted hypergraph $\widetilde{\mathcal{G}}$, and we omit the tilde from their notation. Because the incidence edges of every target clause $e_j \in S$ are removed, these masked clause nodes cannot directly aggregate messages from their original incident literals. We therefore introduce a multi-target contextualization module after bidirectional message passing at each decoder layer.

Given the provisional representations $\widehat{\mathbf{U}}_{\mathcal{E}}^{(\ell+1)}$ and $\widehat{\mathbf{U}}_{\mathcal{V}}^{(\ell+1)}$, we first compute the mean representation of the masked clause nodes and a graph-level summary of the literal nodes:

$$\overline{\mathbf{u}}_S^{(\ell+1)} = \frac{1}{|S|} \sum_{e_j \in S} \widehat{\mathbf{u}}_{e_j}^{(\ell+1)}, \qquad \overline{\mathbf{u}}_{\mathcal{V}}^{(\ell+1)} = \frac{1}{|\mathcal{V}|} \sum_{v_i \in \mathcal{V}} \widehat{\mathbf{u}}_{v_i}^{(\ell+1)}. \tag{18}$$

These two summaries are concatenated and transformed by a layer-specific context MLP:

$$\mathbf{c}^{(\ell+1)} = f_{\mathrm{ctx}}^{(\ell)} \left(\left[\overline{\mathbf{u}}_{\mathcal{V}}^{(\ell+1)} \parallel \overline{\mathbf{u}}_S^{(\ell+1)} \right] \right). \tag{19}$$

The provisional representation of every masked clause is then replaced by the shared contextual representation:

$$\widehat{\mathbf{u}}_{e_j}^{(\ell+1)} \leftarrow \mathbf{c}^{(\ell+1)}, \qquad e_j \in S. \tag{20}$$

The same context is broadcast to all literal nodes through another layer-specific MLP:

$$\widehat{\mathbf{u}}_{v_i}^{(\ell+1)} \leftarrow f_{\mathrm{bcast}}^{(\ell)} \left(\left[\widehat{\mathbf{u}}_{v_i}^{(\ell+1)} \parallel \mathbf{c}^{(\ell+1)} \right] \right), \qquad v_i \in \mathcal{V}. \tag{21}$$

The contextualized representations are subsequently normalized:

$$\mathbf{U}_{\mathcal{E}}^{(\ell+1)} = \mathrm{GraphNorm}_{\mathcal{E}}^{(\ell)} \left(\widehat{\mathbf{U}}_{\mathcal{E}}^{(\ell+1)} \right), \qquad \mathbf{U}_{\mathcal{V}}^{(\ell+1)} = \mathrm{GraphNorm}_{\mathcal{V}}^{(\ell)} \left(\widehat{\mathbf{U}}_{\mathcal{V}}^{(\ell+1)} \right). \tag{22}$$

This mechanism allows the masked clause to interact with the global literal context despite its removed incidence links. For a minibatch containing multiple formulas, the mean pooling, context construction, and context broadcasting are performed independently for each formula. Consequently, all target clauses in the same formula share the decoder context at each layer, while their clause-specific latent variables are introduced later by the arity and membership predictors.

Layer aggregation and output representations. To retain information from different message-passing depths, both the encoder and decoder use concatenation-based layer aggregation. For each node type $\mathcal{T} \in \{\mathcal{E}, \mathcal{V}\}$, the input representation $\mathbf{U}_{\mathcal{T}}^{(0)}$ and the normalized representations $\mathbf{U}_{\mathcal{T}}^{(1)}, \ldots, \mathbf{U}_{\mathcal{T}}^{(L)}$ from all message-passing layers are concatenated along the feature dimension. The resulting $(L+1)d$-dimensional representation is then mapped back to dimension d by a type-specific output MLP.

When applied to the complete hypergraph $\mathcal{G}$, the encoder produces the clause-hyperedge representations $\mathbf{Y}_{\mathcal{E}}$ and literal-node representations $\mathbf{Y}_{\mathcal{V}}$. For each target clause $e_j \in S$, the corresponding row $\mathbf{y}_{e_j}$ of $\mathbf{Y}_{\mathcal{E}}$ is used to parameterize the mean and log-variance of its variational posterior.

When applied to the corrupted hypergraph $\widetilde{\mathcal{G}}$, the decoder produces $\widetilde{\mathbf{Y}}_{\mathcal{E}}$ and $\widetilde{\mathbf{Y}}_{\mathcal{V}}$. For each target clause $e_j \in S$, $\widetilde{\mathbf{y}}_{e_j}$ is combined with its latent variable $\mathbf{z}_j$ by the arity predictor, while each literal representation $\widetilde{\mathbf{y}}_{v_i}$ is combined with $\mathbf{z}_j$ by the membership predictor.

The encoder and decoder use the same layer-aggregation scheme but have separate HyperGNN and output-MLP parameters.

A.2 Training Loss

Let $\mathcal{B}$ denote a batch of formulas. For each formula $\mathcal{G}_g \in \mathcal{B}$, let $S_g \subseteq \mathcal{E}_{\mathrm{rest}}^{(g)}$ denote its set of masked target clauses, with $m_g = |S_g|$. The model computes the KL, arity, and membership losses separately for each target clause.

Arity reconstruction loss. Using the normalized target r_j and prediction $\widehat{r}_j$ defined in Section 4.2.4, we optimize an inverse-frequency-weighted mean squared error:

$$\mathcal{L}_{\text{arity}}^{(g,j)} = w_{k_j} \left(\widehat{r}_j - r_j\right)^2, \tag{23}$$

where k_j is the ground-truth arity of target clause e_j.

Let $\mathcal{D}$ denote the training set and let $\mathcal{K}$ be the set of arities occurring among maskable non-core clauses.

The sampling probability of arity k is estimated by assigning equal weight to every training formula:

$$p_k = \frac{1}{|\mathcal{D}|} \sum_{\mathcal{G}_g \in \mathcal{D}} \frac{\left|\left\{e_j \in \mathcal{E}_{\text{rest}}^{(g)} : k_j = k\right\}\right|}{|\mathcal{E}_{\text{rest}}^{(g)}|}. \tag{24}$$

We set the corresponding arity weight to

$$w_k = \frac{1}{|\mathcal{K}| p_k}. \tag{25}$$

This weighting balances the expected contributions of different arity values during training.

Membership reconstruction loss. For a target clause $e_j \in S_g$, the incidence entry $H_{i,j}^{(g)} \in \{0,1\}$ indicates whether literal node $v_i \in \mathcal{V}_g$ belongs to the clause. We define the incident and non-incident literal sets as

$$\mathcal{P}_{g,j} = \left\{v_i \in \mathcal{V}_g : H_{i,j}^{(g)} = 1\right\}, \qquad \mathcal{N}_{g,j} = \left\{v_i \in \mathcal{V}_g : H_{i,j}^{(g)} = 0\right\}. \tag{26}$$

Because non-incident literals substantially outnumber incident literals, we average their binary cross-entropy losses separately:

$$\mathcal{L}_{\text{mem}}^{(g,j)} = -\frac{1}{|\mathcal{P}_{g,j}|} \sum_{v_i \in \mathcal{P}_{g,j}} \log \sigma(s_{i,j}) - \frac{1}{|\mathcal{N}_{g,j}|} \sum_{v_i \in \mathcal{N}_{g,j}} \log\left(1 - \sigma(s_{i,j})\right). \tag{27}$$

Here, $\sigma(s) = (1 + \exp(-s))^{-1}$ denotes the sigmoid function.

Multi-target loss aggregation. For each target clause $e_j \in S_g$, let $\mathcal{L}_{\text{KL}}^{(g,j)}$ denote the clause-level KL divergence defined by the variational formulation in the main text. It is computed only for the latent variable of the target clause; no literal-level KL term is used.

The loss associated with target clause $e_j \in S_g$ is

$$\mathcal{L}^{(g,j)} = \beta \mathcal{L}_{\text{KL}}^{(g,j)} + \lambda_{\text{arity}} \mathcal{L}_{\text{arity}}^{(g,j)} + \lambda_{\text{mem}} \mathcal{L}_{\text{mem}}^{(g,j)}. \tag{28}$$

The loss is averaged hierarchically: first over masked targets within each formula and then over formulas in the batch:

$$\mathcal{L}^{(g)} = \frac{1}{|S_g|} \sum_{e_j \in S_g} \mathcal{L}^{(g,j)}, \quad \mathcal{L} = \frac{1}{|\mathcal{B}|} \sum_{g \in \mathcal{B}} \mathcal{L}^{(g)}. \tag{29}$$

This two-stage reduction gives every formula equal weight regardless of the number of clauses masked within that formula.

A.3 Pre-generation Scrambling

Before generation, we scramble the input formula using three transformations. Variable permutation randomly exchanges the variables appearing in the retained unsatisfiable core, consistently across both polarities and all their occurrences. Clause permutation randomly exchanges the positions of core clauses. Phase flipping exchanges the positive and negative literals of selected core variables throughout the formula. The numbers of variable permutations, clause permutations, and phase flips are controlled by ratios ρ_{var}, ρ_{clause}, and ρ_{phase}, respectively.

A.4 HARDNESS REFINEMENT DETAILS

The core detector is trained independently from the generative model. Its input is the literal–clause hypergraph of a CNF formula. For each Boolean variable x_i, the positive and negative literal features are

$$\mathbf{x}_{v_i} = [1, 0, 1], \qquad \mathbf{x}_{v_{n+i}} = [0, 1, 1]. \tag{30}$$

For a clause hyperedge e_j with arity k_j, the input feature is

$$\mathbf{x}_{e_j} = [1, k_j, k_j/\overline{k}], \qquad \overline{k} = \frac{1}{|\mathcal{E}|} \sum_{e_j \in \mathcal{E}} k_j. \tag{31}$$

Separate linear layers followed by ReLU map the literal and clause features into a shared hidden space. The detector then applies L_{core} HyperGNN layers with literal-to-clause and clause-to-literal propagation. At layer ℓ, the updates are

$$\mathbf{M}_{\mathcal{E}}^{(\ell)} = \text{ReLU}\left(\mathbf{D}_{\mathcal{E}}^{\dagger}\mathbf{H}^{\top}\mathbf{U}_{\mathcal{V}}^{(\ell)}\mathbf{W}_{\mathcal{V}\to\mathcal{E}}^{(\ell)}\right), \quad \mathbf{M}_{\mathcal{V}}^{(\ell)} = \text{ReLU}\left(\mathbf{D}_{\mathcal{V}}^{\dagger}\mathbf{H}\mathbf{M}_{\mathcal{E}}^{(\ell)}\mathbf{W}_{\mathcal{E}\to\mathcal{V}}^{(\ell)}\right), \tag{32}$$

$$\mathbf{U}_{\mathcal{E}}^{(\ell+1)} = \mathbf{U}_{\mathcal{E}}^{(\ell)} + \mathbf{M}_{\mathcal{E}}^{(\ell)}, \quad \mathbf{U}_{\mathcal{V}}^{(\ell+1)} = \mathbf{U}_{\mathcal{V}}^{(\ell)} + \mathbf{M}_{\mathcal{V}}^{(\ell)}. \tag{33}$$

Finally, a linear classification head followed by a sigmoid function maps each final clause representation to a core-membership score $p_j \in [0, 1]$.

Following HardCore (Cotnareanu et al., 2024), we construct the detector dataset from proof-guided refinement trajectories of the training UNSAT instances. At each step, CaDiCaL (Biere et al., 2024) generates a DRAT proof and DRAT-trim (Wetzler et al., 2014) extracts a core from the current formula. The current formula and extracted core form one training example. We then relax one extracted-core clause outside the retained original core to obtain the formula for the next step. The training data consist of CNF formulas paired with their extracted core files. A separate core detector is trained for each dataset. A clause is labeled as positive if it occurs in the associated extracted core and as negative otherwise. We train the detector using binary cross-entropy with square-root inverse-frequency weights for the core and non-core classes, reducing the effect of class imbalance.

B EXPERIMENTAL DETAILS

B.1 DATASET

Table 4 summarizes the statistics of the datasets.

Table 4: Statistics of the datasets: N_v – number of variables; N_c – number of clauses.

Dataset	$min.N_v$	$max.N_v$	$min.N_c$	$max.N_c$
MP1	4,759	6,368	14,203	19,030
MDP	552	573	3,078	3,230
MDP-Large	620	757	3,478	4,274

B.2 HYPERPARAMETERS

The raw clause and literal feature dimensions are 1 and 9, respectively. The hidden and latent representation dimensions are both set to 16. Both the encoder and decoder contain six HyperGNN layers. We train the generator for 6,000 steps using the Adam optimizer (Kingma, 2014) with an initial learning rate of 10^{-3} and weight decay 10^{-4}. The learning rate is multiplied by 0.99 every 30 steps, and the gradient norm is clipped to 10. The batch size is 15, and each instance is repeated twice with different random masks. The numbers of simultaneously masked clauses are $\{1, 2, 4, 8\}$, with batch proportions $\{0.4, 0.2, 0.2, 0.2\}$. We set $\lambda_{\text{arity}} = \lambda_{\text{mem}} = 1$. The KL weight follows a sigmoid schedule with $\beta_{\min} = 4.5 \times 10^{-4}$ and $\beta_{\max} = 10^{-3}$. The generation ratio is $\eta = 0.01$. Candidate variables are sampled with temperature $T = 0.8$. If the resulting clause duplicates an existing clause, discrete clause sampling is repeated up to 10 times; the original target clause is

retained if all attempts fail. The core detector uses six HyperGNN layers and a hidden dimension of 64. The learning rate is 7×10^{-4}. For the downstream task, each instance is encoded by a 33-dimensional SATzilla feature vector. The model is trained using mean squared error loss and Adam optimizer (Kingma, 2014) with learning rate 5×10^{-3}. The batch size is 32. We use a random 80/20 training/validation split. We report results averaged over 15 splits.

B.3 Evaluation Configuration

All training experiments are conducted on 3 NVIDIA A100-SXM4-40GB GPUs, while generation and evaluation are conducted on an NVIDIA A6000 GPU.

C Additional Results

C.1 Diversity of generated instances.

We further evaluate the diversity of generated instances relative to their corresponding original formulas. The clause change rate measures the proportion of clauses that differ from their corresponding clauses in the original formula, while the clause Jaccard similarity measures the overlap of clause contents between the generated and original formulas, disregarding clause order and literal order within each clause. As shown in Table 5, HyperGEN achieves clause change rates of 96.41% on MDP and 90.41% on MP1, whereas the corresponding Jaccard similarities are only 0.0370 and 0.0963. These results indicate very limited overlap between the generated and original instances, showing that the generated formulas are not merely obtained through a small number of local clause modifications. These results further demonstrate that HyperGEN generates instances with substantial clause-level diversity.

Table 5: Diversity of instances generated by HyperGEN relative to their corresponding original test instances as measured by clause change rate and clause Jaccard similarity. Results are reported as mean $\pm$ standard deviation over 50 pairs of original and generated instances.

Dataset	Clause Change Rate (%)	Clause Jaccard Similarity
MDP	96.41 ± 0.30	0.0370 ± 0.0093
MP1	90.41 ± 0.21	0.0963 ± 0.0086

C.2 Sensitivity to the mask ratio

We further investigate the sensitivity of HyperGEN to the mask ratio, which controls the proportion of clauses masked and reconstructed during generation. We vary the mask ratio from 0.01 to 0.05 while keeping all other settings unchanged, and evaluate the generated instances on MDP using runtime, decisions, and conflicts. As shown in Table 6, HyperGEN exhibits stable solver hardness across the tested mask ratios.

Table 6: Sensitivity analysis of the mask ratio on MDP. We report the mean solver runtime, decisions, and conflicts under different mask ratios.

Mask Ratio η	Runtime (s) $\uparrow$	Decisions $\uparrow$	Conflicts $\uparrow$
0.01	46.42	3,155,755.31	2,537,645.94
0.02	45.29	3,101,218.74	2,495,840.58
0.03	46.03	3,147,193.06	2,531,790.63
0.04	46.53	3,172,451.94	2,551,401.30
0.05	45.00	3,074,540.20	2,479,201.16

C.3 Core Prediction for Hardness Refinement

We evaluate the effectiveness of the HyperGNN-based core detector used in the hardness refinement procedure. We compare it with the HardCore selection strategy. We evaluate Balanced Clause

Recall (BCR), defined as the average of core recall and non-core recall. Core recall measures how many core clauses are recovered, while non-core recall focuses on clauses outside the core. If all clauses are predicted to be 1, then recall will be 1. Therefore, non-core recall is an important metric, which means how many of the true non-core are correctly identified. As shown in Table 7, HyperGEN outperforms HardCore on both datasets, improving BCR from 0.7771 to 0.7973 on MP1 and from 0.6884 to 0.7198 on MDP. These results indicate more effective core identification than the HardCore strategy.

Table 7: Core prediction performance for hardness refinement. Balanced Clause Recall is computed as the average of core recall and non-core recall.

Method	MP1	MDP
HardCore	0.7771	0.6884
HyperGEN	**0.7973**	**0.7198**