GraSS: Combining Graph Neural Networks with Expert Knowledge for SAT Solver Selection

Zhanguang Zhang
Huawei Noah's Ark Lab
Montreal, Canada
zhanguang.zhang@huawei.com

Didier Chetelat
Huawei Noah's Ark Lab
Montreal, Canada

Joseph Cotnareanu
McGill University
Montreal, Canada

Amur Ghose
Huawei Noah's Ark Lab
Montreal, Canada

Wenyi Xiao
Huawei Noah's Ark Lab
Hong Kong, China

Hui-Ling Zhen
Huawei Noah's Ark Lab
Hong Kong, China

Yingxue Zhang
Huawei Noah's Ark Lab
Montreal, Canada

Jianye Hao
Huawei Noah's Ark Lab
Beijing, China

Mark Coates
McGill University
Montreal, Canada

Mingxuan Yuan
Huawei Noah's Ark Lab
Hong Kong, China

ABSTRACT

Boolean satisfiability (SAT) problems are routinely solved by SAT solvers in real-life applications, yet solving time can vary drastically between solvers for the same instance. This has motivated research into machine learning models that can predict, for a given SAT instance, which solver to select among several options. Existing SAT solver selection methods all rely on some hand-picked instance features, which are costly to compute and ignore the structural information in SAT graphs. In this paper we present GraSS, a novel approach for automatic SAT solver selection based on tripartite graph representations of instances and a heterogeneous graph neural network (GNN) model. While GNNs have been previously adopted in other SAT-related tasks, they do not incorporate any domain-specific knowledge and ignore the runtime variation introduced by different clause orders. We enrich the graph representation with domain-specific decisions, such as novel node feature design, positional encodings for clauses in the graph, a GNN architecture tailored to our tripartite graphs and a runtime-sensitive loss function. Through extensive experiments, we demonstrate that this combination of raw representations and domain-specific choices leads to improvements in runtime for a pool of seven state-of-the-art solvers on both an industrial circuit design benchmark, and on instances from the 20-year Anniversary Track of the 2022 SAT Competition.

IWLS '24, June 6-7, Zurich, Switzerland

ACM ISBN 978-x-xxxx-xxxx-x/YY/MM…$15.00
https://doi.org/10.1145/nnnnnnn.nnnnnnn

KEYWORDS

GNN, SAT, Algorithm Selection

ACM Reference Format:
Zhanguang Zhang, Didier Chetelat, Joseph Cotnareanu, Amur Ghose, Wenyi Xiao, Hui-Ling Zhen, Yingxue Zhang, Jianye Hao, Mark Coates, and Mingxuan Yuan. 2024. GraSS: Combining Graph Neural Networks with Expert Knowledge for SAT Solver Selection. In *Proceedings of 33rd International Workshop on Logic & Synthesis (IWLS '24).* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/nnnnnnn.nnnnnnn

1 INTRODUCTION

The Boolean satisfiability (SAT) problem is one of the most fundamental computer science problems, with numerous applications in planning and scheduling [9, 24], formal software verification [16, 22] and electronic circuit design [17, 21]. A SAT instance consists of a formula with Boolean variables, such as $(x_1 \vee \overline{x}_2) \wedge (\overline{x}_1 \vee x_2 \vee x_3) \wedge \overline{x}_1$, and the problem involves finding an assignment of values for each variable x_i which makes the whole formula true, or proving that no such assignment exists. Although the problem is NP-complete [8], many SAT solvers have been designed over the years and modern CDCL-based solvers are routinely able to solve industrial problems within minutes [33].

Structural differences between different SAT problems mean that the choice of solver can have a dramatic impact on the solving time. This has motivated the use of machine-learning based methods for selecting the optimal solver to use for a given instance, with the hope that data-driven models can see patterns where humans have been unsuccessful. The most influential of those has probably been SATzilla [46], which has won several times the annual SAT Competition [7]. This model relies on machine learning algorithms that require a fixed-dimensional vector of features as input, irrespective of the actual instance size (number of clauses and variables). This necessarily implies that some aspects of a SAT problem are not taken into account when performing solver selection.

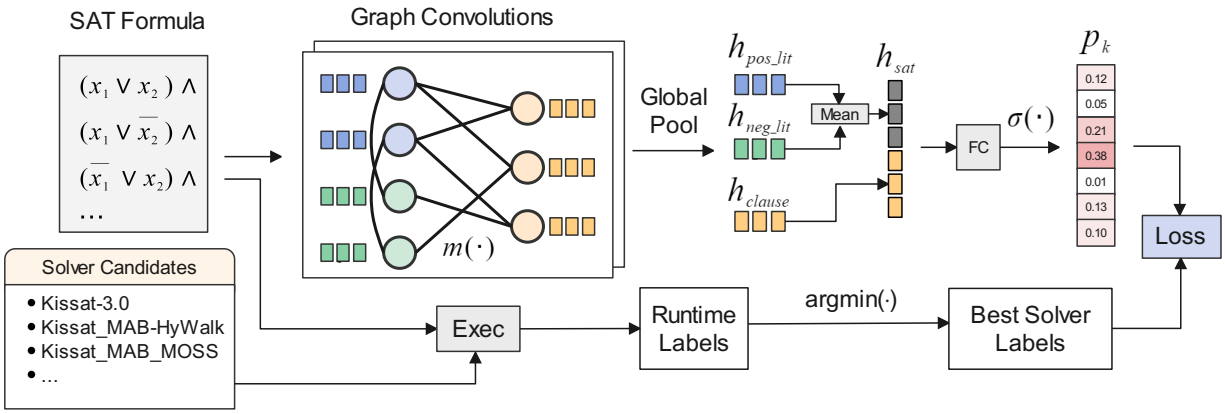


Figure 1: The workflow of our method. SAT instances are represented as literal-clause graphs with hand-designed attributes. Rounds of heterogeneous graph convolutions are applied, which modify the attributes. The attributes of the clause and variable nodes are then averaged, before being fed to a linear layer followed by a softmax over the various solvers. The convolutions and the linear layer are trained to minimize by gradient descent a runtime-sensitive classification loss computed from runtimes collected on training SAT instances.

In recent years, machine learning has been revolutionized by deep learning models trained on raw descriptions of data points such as image pixels or text strings [18]. In particular, surprising success has been found in a variety of combinatorial optimization tasks by representing optimization problems as graphs, and feeding them as inputs to graph neural networks [5]. Such models are able to take the complete representation of a problems as input, in a size-independent way, and see patterns where humans have been unable to distinguish any.

In this work, we propose **GraSS** (**Gra**ph Neural Network **S**AT Solver **S**elector), the first graph neural network (GNN) based method for automatic SAT solver selection. We represent instances as literal-clause graphs [19], thus encoding the entirety of the information pertaining to an instance. To improve performance further, we also endow the graph with hand-designed features representing domain-knowledge about which aspects of the graph should be particularly useful for solver selection, as well as positional encodings for the clauses to allow for order-specific effects. Our GNN model consists of learned graph convolutions operating over each type of edge, with a node-specific pooling operation prior to a linear classifier. We train our model in a supervised manner with a runtime-sensitive classification loss. The training data consists of a collection of instances for which the runtimes of multiple solvers have been collected.

On both a large-scale industrial circuit design benchmark, and on instances from the Anniversary Track of the 2022 SAT Competition [2], we report improvements in performance compared to seven competitive solvers, as well as state-of-the-art machine learning approaches. We also perform a complete ablation study to rigorously test the importance of each component of our proposed pipeline.

In summary, our contributions are as follows.

- We propose the first approach for SAT solver selection that makes complete use of the SAT instance data, by representing each instance as an attributed graph and using a GNN model.
- We propose a model architecture that is tailored to the tripartite graph representations.
- We design novel node-level features to incorporate domain-specific knowledge.
- We report for the first time the value of including positional encodings for clauses in the graphical representation of an instance for a SAT-related machine learning task.
- We introduce a novel runtime-sensitive classification loss, which could be of value for general algorithm selection tasks.
- We report state-of-the-art empirical performance on two hard SAT benchmarks and conduct extensive ablation studies to confirm the value of our architectural choices.

Collectively, these elements strongly suggest our approach should be regarded as a new standard in the field of SAT solver selection.

2 RELATED WORK

2.1 SAT Solver Selection

There exists a rich literature describing machine learning models for the selection of the optimal SAT solver for a given instance. This approach is sometimes referred to as portfolio-based SAT solving. A detailed summary is provided by Holden et al. [20, Section 6].

SATzilla [46] and its successors [45, 47] are a family of classification models that have won multiple prizes in the SAT Competition (2007 and 2009) and the SAT Challenge (2012). SATzilla uses hand-selected features to characterize each SAT instance for best solver selection. The latest version of SATzilla [45] consists of a feature cost classifier to predict if the entire set of features can be computed within a time threshold and an algorithm selector for instances with

feature cost less or equal to this threshold. Whereas the first version [46] applies ridge regression, the latest version [45] uses a cost-sensitive decision tree for the algorithm selection model.

In addition to the SATzilla family, hand-picked features have also been used with k-nearest neighbor or clustering methods to select the best algorithm [23, 32, 34]. CSHC [32] uses a cost-sensitive hierarchical clustering algorithm to iteratively partition the feature space into clusters in a supervised top-down fashion. In combination with the static algorithm schedule of 3S [23], it achieved better performance than SATzilla in the 2011 SAT Competition. ArgoSmArT [34] uses a combination of a k-Nearest Neighbors (KNN) model and a multi-armed bandit algorithm. The KNN performs solver selection based on the nearest instance to the queried instance in a metric space.

Finally, Loreggia et al. [31] proposed a deep learning approach based on convolutional neural networks (CNNs). They take textual representations of SAT instances in a standard format, and convert them into grayscale images by replacing each character with its corresponding ASCII code. These images are then rescaled to 128x128 pixels, and fed to a CNN, which is trained to predict the fastest solver in the pool. Although employing deep learning, this approach is not lossless, as the rescaling required by the CNN architecture leads to information loss, unlike our approach.

2.2 Algorithm Selection

More generally, SAT solver selection is a special case of algorithm selection, which aims to select, for a given input, the most efficient algorithm from a set of candidate algorithms. This is particularly important for computationally hard problems, where there is typically no single algorithm that outperforms all others for all inputs. Besides SAT solving, algorithm selection techniques have achieved remarkable success in various applications such as Answer Set Programming (ASP) [15] and the Traveling Salesperson Problem (TSP) [1]. A thorough literature review is provided in Kerschke et al. [25].

Algorithm selection methods can be roughly divided between offline and online methods. Offline methods, such as this work, rely on training ahead of time on a labeled dataset, whereas online algorithms attempt to improve selection performance as more and more cases are run. Although providing worse initial performance, online methods avoid the computational cost of the initial training phase and the problem of distribution shift between training and test data, and methods based on based on reinforcement learning or multi-armed bandits have been proposed for this purpose [10, 15].

The many design choices in algorithm selection systems pose new challenges for efficient system development. For example, AutoFolio [30] automatically configures the entire framework, including budget allocation for pre-solving schedules, pre-processing procedures (such as transformations and filtering) and algorithm component selection. It achieved competitive performance in multiple scenarios from the ASLib [4] benchmark.

2.3 Graph Neural Networks in SAT Solving

Several works have explored applications of GNNs to various aspects of SAT solving in the past, even if not specifically to the problem of solver selection. In all these works, some kind of graphical representation of SAT instances is used. Multiple suggestions have appeared in the literature: these include lossless representations like literal-clause graphs (LCGs) and variable-clause graphs (VCGs), and lossy representations like literal-incidence graphs (LIGs) and variable-incidence graphs (VIGs) [19]. These different representations strike a balance between graph size and information content, and have found success in various SAT-related tasks.

Some prior works have explored the use of GNNs to learn local search heuristics in SAT solvers [28, 48]. Yolcu and Póczos [48] represent SAT formulas as variable-clause graphs (VCGs) and a GNN model is trained to select variables whose sign to flip at every step through a Markov decision process (MDP). The learned heuristic is shown to reduce the number of steps required to solve the problem. Graph-Q-SAT [28] uses a deep Q-network (DQN) with GNN architecture to learn branching heuristics in conflict driven clause learning (CDCL) solvers. Each SAT formula is converted into a VCG, and GNN layers are used to predict the Q-value of each variable. The variable with the highest Q-value for the specific assignment is selected for branching. The learned heuristic is shown to significantly reduce the number of iterations required for SAT solving.

Instead of relying on existing SAT solvers, NeuroSAT [38] uses a GNN-based model to predict satisfiability of an instance in an end-to-end manner. Each SAT formula is represented by a literal-clause graph (LCG), as in our work. After several steps of message-passing, the updated embedding of each literal is projected to a scalar "vote" to indicate the confidence that the formula is satisfiable. The votes are averaged together and passed through a sigmoid function to produce the model's probability that the instance is satisfiable. On randomly generated instances from a SR(40) distribution, NeuroSAT solved 70% of SAT problems with an accuracy of 85%. A subsequent work, NeuroCore [37], uses a lighter NeuroSAT model to predict the "core" of instance, which is the smallest unsatisfiable subset of clauses. This prediction is then used to guide variable selection in SAT solver algorithms.

Finally, graph neural networks have also been widely used for SAT instance generation. For example, G2SAT [49] and HardSATGEN [29] represent instances as LCGs, and generate new variants from an iterative splitting and merging process driven by a GNN. Furthermore, W2SAT [44] extends this approach by representing instances as weighted graphs encoding literal co-occurence among clauses, while using a similar generation mechanism.

3 APPROACH

We now describe our proposed approach. The workflow of our method is shown in Figure 1.

3.1 Problem

Our SAT solver selection problem can be formally described as follows. We have a fixed pool of SAT solvers $S_1, \ldots, S_K$ and a SAT instance a, and we must select a solver so as to solve the instance in the smallest runtime possible.

To do so, at train-time, we are given a collection of SAT instances and for each instance a_i, we are given sampled solving times $t_i^1, ..., t_i^K$ from each solver. This dataset can be used for training a machine learning model offline in a supervised manner. At test-time, a separate collection of SAT instances are given to the

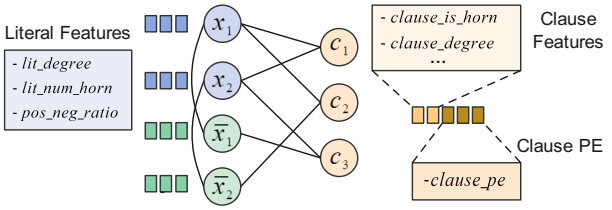


Figure 2: Literal-clause graph representation of a SAT instance used in this work. The instance is converted to CNF form, and nodes represent positive literals, negative literals and clauses. Edges are drawn between clauses and literal nodes if the literal participate in the clause, and edges are also drawn between positive and negative nodes of the same variable. The nodes are endowed with feature vectors described in Appendix A.

model, which selects a solver for each. The selected solver for each instance is run, and the average runtime over all testing instances is used to benchmark performance of the selector model.

3.2 Representation and features

The inputs to the model are individual SAT instances. We assume that they are formulated in conjunctive-normal form (CNF), that is as a formula $c_1 \wedge \cdots \wedge c_m$ where each c_i is a *clause*, which in turn is in the form $c_i = l_1 \vee l_2 \vee \ldots$, where each *literal* l_i stands for a variable x_j in the problem, or its negation $\bar{x}_j$. Any SAT problem can be converted into an equivalent CNF problem in linear time [40].

Given CNF SAT instances, we input them to the machine learning model as literal-clause graphs (LCGs) [19] endowed with extra information in the form of node features. A representation is provided in Figure 2. The literal-clause graph of a SAT instance is an undirected graph with three types of node: one node c_j per clause in the graph, and two nodes x_i and $\bar{x}_i$ per variable in the graph, representing itself and its negation respectively. An edge is drawn between a clause node c_j and a variable node $x_i/\bar{x}_i$ if the variable (respectively, its negation) appears as literal in the clause. Finally, an edge is drawn between every positive and negative variable node.

In addition, to every clause node and variable node, we attach feature vectors. Most features are hand-designed and are inspired by those used by SATzilla [45, 47]. They represent expert knowledge that is known to be critical for SAT solving process, such as the presence of Horn clauses, which are clauses containing at most one positive literal. These are especially important for the solving process as the collection of Horn clauses can be proved within linear time [11]. Besides these hand-designed features, clause node features are also enriched with a positional encoding, described in the next subsection. A complete list of the features used is provided in Appendix A.

3.2.1 Clause positional embeddings. In principle, satisfiability of a SAT formula is not affected by permuting the variables or clauses, and literal-clause graphs are permutation-invariant as well. In practice, however, we found solver runtimes can be sensitive to the order in which clauses are provided as input. We conducted a study with the popular Kissat 3.0 [3] solver on the industrial LEC dataset described further in Section 4. As can be seen in Figure 3, shuffling clauses sometimes led to very large variations in runtime. In contrast, shuffling variables showed limited impact.

This is in line with previously reported remarks on other SAT solvers [12, 39]. A possible explanation could be the algorithmic design of modern solvers, for which the storage architecture of variables relies on a doubly linked list and the initial storage order follows the parsing order of the clauses. This results in variations in cache miss rates depending on the provided clause ordering. In contrast, variable ordering usually only impacts the variable labels used by the solvers.

To address the sensitivity to clause ordering, we include positional encodings among the clause features. These encode the position of a clause within the CNF formula. We follow the classical encodings from Vaswani et al. [41] and endow the k^{th} clause with a 10-dimensional embedding

$$PE(k, 2i) = \sin\left(\frac{k}{10000^{2i/10}}\right),$$
$$PE(k, 2i{+}1) = \cos\left(\frac{k}{10000^{2i/10}}\right),$$

where $i = 0, \ldots, 4$. This vector is concatenated with the rest of the clause node features.

3.3 Model

We use a graph neural network (GNN) model to predict which solver to use for a given instance. These models operate on graphs by repeatedly modifying node embeddings through graph convolution operations, and have emerged as a standard paradigm for dealing with graph-structured data, both in SAT solving [19] and more widely for combinatorial optimization in general [5]. However, we deviate from standard graph convolution frameworks by interpreting our literal-clause graph as a graph with three types of edges: (i) from clause to literal nodes; (ii) from literal to clauses nodes; and (iii) between positive and negative literal nodes. These graphs can then be interpreted as "heterogeneous graphs", and we can apply heterogenous GNN methodologies [50].

In this framework, the graph convolution steps take the following edge-type-dependent form. Let $x_j \in \mathbb{R}^d$ stand for the feature vector at node j. For every edge $(j, i) \in \mathcal{E}_k$ of type $k \in \{1, \ldots, K\}$ from node j to node i, we compute a message $m_{i,j,k} = \phi_k(x_i, x_j)$ where ϕ_k is a learnable "message function". We then update the feature vector at node i by the formula

$$\overline{m}_{i,k} = \rho_k\left(\left\{m_{i,j,k} \,\middle|\, (j,i) \in \mathcal{E}_k\right\}\right),$$
$$x_i \leftarrow \delta\left(\left\{\psi_k(x_i, \overline{m}_{i,k}) \,\middle|\, k = 1 \ldots, K\right\}\right),$$

where ρ_k is the "message aggregation" function for edge type k, ψ_k is the learnable "update" function, and δ is the "edge-type aggregation" function.

We apply this framework to our neural network as follows. For clarify, a diagram of the architecture is provided as Figure 1. The clause, positive and negative literal node embeddings are initialized

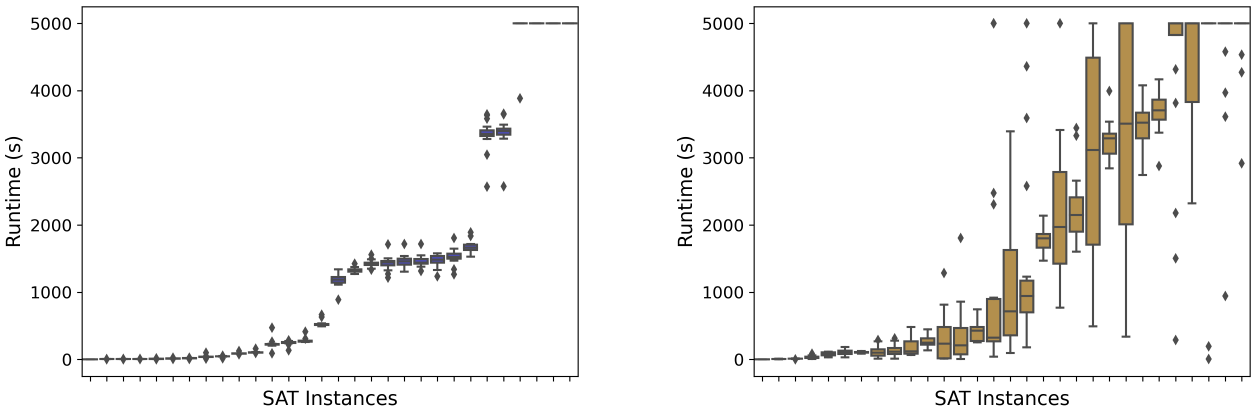


Figure 3: Runtime variation introduced by permutation. Thirty SAT instances are randomly sampled from SAT Competition data, and the order of variables (left) and clauses (right) are shuffled twenty times. The instances are then solved by Kissat 3.0 with a 5,000s cutoff, and the collected runtimes are plotted in ascending order of mean runtime.

with the 17-, 3-, and 3-dimensional feature vectors described in Table 7,

$$\begin{aligned} x_{\text{clause},i} &\leftarrow \text{features}_{\text{clause},i} \quad \forall \text{ clause node } i \\ x_{\text{poslit},i} &\leftarrow \text{features}_{\text{poslit},i} \quad \forall \text{ positive literal node } i \\ x_{\text{neglit},i} &\leftarrow \text{features}_{\text{neglit},i} \quad \forall \text{ negative literal node } i \end{aligned}$$

These are then transformed by two rounds of convolutions. We set ϕ_k and ψ_k as in the classical graph convolutions of Kipf and Welling [27], with a relu nonlinearity and mean aggregation functions for ρ_k and δ. That is, we compute

$$x_{\text{clause},i} \leftarrow \text{relu}\Big(b + \sum_{j \in N_{\text{poslit}}(i)} \frac{W_{\text{poslit}} x_{\text{poslit},j}}{\sqrt{\deg(i)\deg(j)}} + \sum_{j \in N_{\text{poslit}}(i)} \frac{W_{\text{neglit}} x_{\text{neglit},j}}{\sqrt{\deg(i)\deg(j)}}\Big)$$

3.4 Training

We train in a supervised fashion on a dataset of training SAT instances, for which runtimes have been collected ahead of time on each solver of interest. Since our model produces a distribution over the K possible solvers, we could treat the problem as simple classification with a cross-entropy loss. However, this would not be well-aligned with our objective of minimizing solving runtime, since it would equally penalize incorrect predictions, irrespective of the amount of additional runtime induced by the selection of a suboptimal solver. Instead, we propose the regret-like loss

$$\mathcal{L} = \frac{1}{N} \sum_{i=1}^{N} \left(\sum_{k=1}^{K} p_i^k t_i^k - t_i^* \right)^2, \tag{1}$$

where p_i^k and t_i^k are the model probability and runtime for instance $i = 1, \ldots, N$ and solver $k = 1, \ldots, K$, respectively, and $t_i^* = \min_k t_i^k$ is the best time achieved by any solver on the instance. This has the advantage of more directly optimizing final runtime, taking into account that not all mistakes are equally impactful on solving time. We minimize this loss using the Adam [26] algorithm with early stopping.

3.5 Inference

At test-time, to predict which SAT solver to use, we convert the SAT instance into our graph representation, compute its node features, and feed the graph to our trained GNN model, which outputs a probability distribution over the solvers of interest. The solver with highest probability is chosen for solving the instance, with the runtime being reported.

4 EXPERIMENTS

We now compare the performance of our proposed approach against competitors in the literature. We implement our model using the `pytorch` [35] and `dgl` [43] libraries. We train the model on a single Nvidia Tesla V100 GPU with a learning rate of $1e{-}3$ for up to 100 epochs, and use the same GPU at test-time.

4.1 Base solvers

We train and evaluate on a portfolio of seven top-performing solvers from recent SAT Competitions [3, 14]: (a) `Kissat-3.0`, (b) `bulky`, (c) `HyWalk`, (d) `MOSS`, (e) `mabgb`, (f) `ESA` and (g) `UCB`. Among them, `Kissat-3.0` and `bulky` are based on `Kissat`, which is the winner of the 2020 SAT Competition and is known for its efficient data structure design [14]. The other five solvers are based on `UCB` and differ in their bandit-based scoring mechanism for branching. It should be noted that our method works for any choice of candidate solvers.

4.2 Datasets

We train and evaluate on two datasets.

Table 1: Dataset statistics. We report the average number of instances, variables and clauses, for different groups of instances.

Name	# instances	# variables	# clauses	Name	# instances	# variables	# clauses
Circuit 1	788	10,404	39,502	Circuit 16	22,050	1,002	3,969
Circuit 2	49	6,199	26,108	Circuit 17	10	1,140	3,914
Circuit 3	36	3,612	13,079	Circuit 18	798	1,058	3,829
Circuit 4	2,014	1,783	7,074	Circuit 19	38	1,067	3,770
Circuit 5	21	2,421	6,830	Circuit 20	14,840	952	3,663
Circuit 6	3,823	1,559	6,281	Circuit 21	2,024	929	3,593
Circuit 7	2	401	6,172	Circuit 22	23	1,040	3,256
Circuit 8	25,591	1,563	6,171	Circuit 23	401	932	3,085
Circuit 9	615	1,639	6,034	Circuit 24	105	909	2,297
Circuit 10	582	1,709	5,940	Circuit 25	1,420	851	2,109
Circuit 11	736	1,412	5,308	Circuit 26	69	627	1,658
Circuit 12	99	1,341	5,246	Circuit 27	16	497	1,418
Circuit 13	56	1,540	5,230	Circuit 28	298	454	1,392
Circuit 14	978	1,224	4,757	Circuit 29	564	411	1,389
Circuit 15	662	1,051	4,109	Circuit 30	21	491	1,165

(a) LEC data. The instances are regrouped by the circuit optimization sequence from which they were generated.

Runtime Range (s)	# instances	# variables	# clauses
(0, 1]	747	2,505	48,163
(1, 10]	416	4,212	74,203
(10, 100]	427	4,415	98,151
(100, 1500]	498	4,250	135,969

(b) SAT Competition data. The instances are regrouped by their best runtime among the base solvers of Subsection 4.1.

Logic Equivalence Checking (LEC). This is a proprietary dataset generated from logic equivalence checking steps in electronic circuit design. Circuits undergo a large number of optimization steps during logic synthesis, and at each one of the steps, it is necessary to verify that the circuits before and after optimization are functionally equivalent. This is done by verifying that the two circuits produce the same outputs for all possible inputs, which is equivalent to solving a SAT problem [17, 21]. We collected logic equivalence checks from the optimization of 30 industrial circuits, yielding a total of 78,727 SAT instances. A summary of dataset statistics is provided as Table 1a.

SAT Competition (SC). This is a subset of the Anniversary Track Benchmark of the 2022 SAT Competition [2], which itself was created by collecting all instances from the Main, Crafted and Application tracks of the previous SAT competitions up to that year. We ran each instance of the Anniversary benchmark through each of the seven solvers in the portfolio, and excluded those that could not be solved within 1,500 seconds by any solver, as well as those with more than 20,000 variables, yielding 2,088 SAT instances. A summary of dataset statistics is provided as Table 1b.

4.3 Baselines

We compare our approach with the following baselines.

Best Base Solver. The individual solver among the portfolio of seven that had the best performance on the training data, measured in average runtime over all instances. In practice, this was the `bulky` solver for both datasets.

SATzilla07 [46]. We adapt this landmark SAT solving machine learning model, based on a linear ridge regression model trained to predict runtimes based on global handcrafted features that summarize SAT instance characteristics. Since our work focuses on SAT solver selection, we omit the presolving process in the original SATzilla pipeline. We also remove features in the original model that require probing. This leaves 33 global features (#1-33 in the original article). The model is trained from the `Ridge` class in the `scikit-learn` [36] library with default settings. We convert the approach into a SAT solver selection model by selecting the solver with shortest predicted runtime.

SATzilla12 [45]. We also adapt the updated SATzilla model from 2012, which was based on random forest classification. Again, we remove the presolving process, and only use the features that do not require probing (features #1-55 in the original article). We train a random forest model between each pair of solvers, weighting each training instance by the absolute difference in runtime between the two solvers, for $7(7-1)/2 = 21$ models in total. Each model is trained from the `RandomForestClassifier` class in `scikit-learn` with 99 trees and $\lceil log_2(55) \rceil + 1 = 7$ sampled features in each tree. At test-time, each model is used to vote which solver it prefers in its pair for solving an instance, and the final solver choice is made from the solver that has received the most votes.

ArgoSmArT [34]. This is an approach based on a k-nearest neighbors model trained for classification. We used the same 29 features as in the original paper, which form a subset of the 33 features used in our adaptation of SATzilla07. We use the `KNeighborsClassifier` class in `scikit-learn` with $k = 9$ neighbors.

CNN [31]. We reimplement the approach of Loreggia et al. [31], where the CNF formula is interpreted as text, converted to its ASCII values and then to a grayscale image, before being resized to 128x128 pixels and fed to a Convolutional Neural Network (CNN). We use the same architecture as in the paper, which we implement in the `pytorch` [35] library, and train it over a cross-entropy loss with the Adam [26] algorithm, a learning rate of 1e-3 and early stopping.

4.4 Metrics

We report the following metrics for performance evaluation. Results are averaged over five train-test folds over the data, and the average and standard deviation over those five folds are reported.

Average Runtime (Avg Runtime). For each instance, the method selects a solver, and this solver is used to solve the instance, reporting a runtime. These runtimes are then averaged over all instances. In other words, this is the average runtime that would be observed if this method were used for all instances, as a "portfolio" solver. Lower is better.

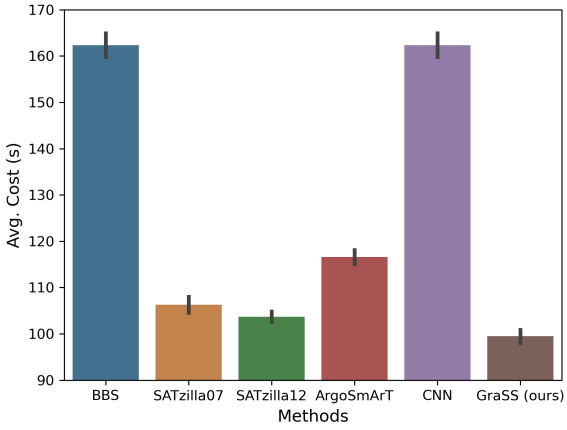
Percentage of Solved Instances (Solved). The percentage of instances that can be solved by the selected solver within a cutoff time of 500s. Higher is better.

Table 2: Main results on the LEC and SC benchmarks. We report the average and standard deviation over 5 train-test folds. For every best result, we bold the number and conduct a Wilcoxon signed-rank test to test whether the distribution of the differences in results between this method and the next best method for every instance and fold is equally distributed around zero. An asterisk (*) next to the number denotes a p-value lower than 0.05. †The CNN on the LEC dataset predicted bulky for every instance, giving identical results to "Best base solver".

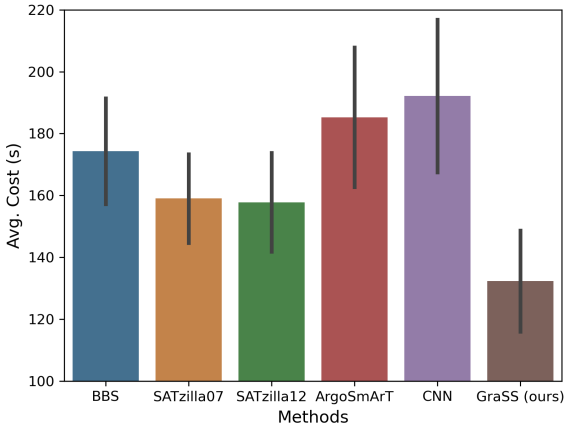
	Avg Runtime (s) ↓	Solved (%) ↑	ACC ↑
	LEC		
Best base solver	382.783±1.877	74.7±0.2	0.424±0.001
SATzilla07	346.492±1.516	76.8±0.2	0.467±0.001
SATzilla12	344.290±1.115	77.2±0.1	*__0.487__±0.004
ArgoSmArT	353.241±1.093	76.3±0.2	0.457±0.002
CNN†	382.783±1.877	74.7±0.2	0.424±0.001
GraSS (ours)	*__341.549__±1.440	*__77.7__±0.2	0.480±0.006
	SC		
Best base solver	250.902±21.669	82.9±1.7	0.210±0.013
SATzilla07	227.643±11.955	83.5±2.2	0.330±0.018
SATzilla12	222.146±18.311	84.0±2.0	0.366±0.009
ArgoSmArT	227.121±14.312	83.9±1.7	*__0.449__±0.023
CNN	253.832±11.985	82.4±0.8	0.296±0.016
GraSS (ours)	*__220.251__±16.360	__84.6__±1.8	0.259±0.026

Table 3: Detailed comparison of the performance of GraSS and the next best method, SATzilla12, over best-runtime quantiles. Results are averaged over 5 folds, and measured in seconds.

Best-Runtime Quantile	LEC		SC	
	SATzilla12	GraSS	SATzilla12	GraSS
[0, 0.25]	__103.387__	104.575	__0.440__	0.673
(0.25, 0.50]	231.500	__229.448__	__14.541__	18.691
(0.50, 0.75]	390.999	__386.567__	143.847	__142.970__
(0.75, 1]	653.293	__647.625__	736.697	__721.190__



(a) LEC benchmark.



(b) SC benchmark.

Figure 4: Cost of wrong prediction: the runtime difference between predicted solver and the optimal solver, when the selector has made a mistake. Average across five folds are shown, with standard deviation as error bar. Lower is better.

Classification Accuracy (ACC). The accuracy of selecting the optimal solver for a given instance. This measures how efficient a method is in selecting the optimal solver. Higher is better.

4.5 Main Results

We report in Table 2 the main results of our experiments on the Logic Equivalence Checking (LEC) and SAT Competition (SAT) datasets, respectively. As can be seen, our proposed GraSS method outperform competing approaches in average runtime, as well as in percentage of problems solved within our 500s cutoff time. Interestingly, this is true despite the method not being as accurate in selecting the optimal solver for every instance, as measured by the accuracy metric. This suggests that an important component of its success lies in its improved robustness to error: when the method makes mistakes, they impact runtime less than competing methods.

To understand this phenomenon further, we looked at the runtime difference between the predicted solver and the optimal solver, whenever a mistake is made. As can be seen in Figure 4, the average cost of wrong prediction is substantially lower for our method than for competitors, especially for the SC dataset.

We further analyze the performance difference between our approach and the next best method in average runtime, SATzilla12. Table 3 regroups the instances by rough measure of difficulty, namely by grouping them by which quartile (0-25%, 25-50%, 50-75% or 75-100%) their best runtime achieved on any solver falls into, and

Table 4: Comparison of the time taken to compute the features required for each method from `.cnf` files. We report the mean and standard deviation in seconds over the instances for each dataset.

	LEC	SC
Best base solver	–	–
SATzilla07	6.767	35.932
SATzilla12	8.643	194.155
ArgoSmArT	6.472	41.266
CNN	1.353	**2.401**
GraSS (ours)	**1.232**	33.862

compares the performance of SATzilla12 and GraSS on each subgroup. As can be seen, SATzilla12 performs better on easy instances, while GraSS performs better on hard instances.

Finally, as described in Subsection 4.4, our timing results only report the time taken by the solvers in optimizing the instances. In particular, this means we exclude from the numbers the time taken to compute the features necessary to take the decision, which is dependent on the storage format used to save the instances. For our experiments, we chose to save them on a hard drive in the standard `.cnf` text file format [6], and we report for completeness the time taken to compute the features for each method in Table 4. Other storage formats would lead to different timings.

4.6 Ablation Study

We extend our analysis by considering the impact of various methodological choices on performance over the LEC benchmark.

We first evaluate the impact of the graph neural network architecture, by comparing our approach with a variant that uses the same convolution weights for every edge, effectively treating it as a homogeneous graph (*Homogeneous*). We also compare with a NeuroSAT-style architecture (*NeuroSAT variant*) inspired by Selsam et al. [38], which was originally designed for satisfiability prediction (sat/unsat). Their model also uses a literal-clause graph to encode instances, although with learned initial node embeddings, and uses a very deep LSTM-GNN hybrid architecture with 26 layers and custom graph convolution operations. We implement the same, but replace the final layer which computes a scalar "vote" for every literal, and takes the average vote before a sigmoid activation, by an averaging of the literal embeddings, followed by a linear layer and a softmax activation. We also use 4 layers instead of 26 for tractability on our dataset, whose instances are substantially larger than those in the original paper. As can be seen in Table 5, our approach improves over these alternatives in every metric.

We next evaluate our choice of node features. We compare it against random normal values (*Random*), as in Selsam et al. [38]); a one-hot vector indicating whether the node represents a clause, positive or negative literal (*Node-type*), as used in Li et al. [29], Yolcu and Póczos [48] and You et al. [49]; and Laplacian Positional Encodings (*Laplacian PE*), as introduced in Dwivedi et al. [13]. We also compare against a variant of our approach consisting of the same hand-designed features, but without the clause positional embeddings (*Custom*). As can be seen in Table 6, our choices outperform these alternative approaches in every metric.

Table 5: Exploration of alternative architectures on the LEC benchmark. We report the average and standard deviation over 5 train-test folds.

	Avg Runtime (s) ↓	Solved (%) ↑	ACC ↑
Homogeneous	343.339±0.987	77.4±0.2	0.464±0.009
NeuroSAT variant	383.132±5.284	74.3±1.0	0.423±0.008
GraSS (ours)	**341.549**±1.440	**77.7**±0.2	**0.480**±0.006

Table 6: Exploration of alternative node features on the LEC benchmark. We report the average and standard deviation over 5 train-test folds.

	Avg Runtime (s) ↓	Solved (%) ↑	ACC ↑
Random	352.258±2.114	76.2±0.5	0.444±0.008
Node-type	344.088±1.603	77.1±0.3	0.474±0.002
Laplacian PE	347.632±1.274	76.9±0.3	0.454±0.003
Custom	343.143±1.621	77.3±0.3	0.476±0.003
Custom + PE (ours)	**341.549**±1.440	**77.7**±0.2	**0.480**±0.006

5 LIMITATIONS

Although our experiments strongly establish the superiority of our approach in the presented scenario, several limitations can be noted. Deep learning methods are well-known to be data hungry, and perform best in regimes where training sets are large. It is plausible that in scenarios where a limited number of timed instances are available, performance would not be competitive against simpler models. In addition, in many scenarios it might be desirable to learn online, updating models as examples stream in: our method cannot be readily adapted to this situation, as training requires runtime labels on every solver for each instance, and adapting graph neural networks to online learning is challenging [42].

6 CONCLUSION

This work proposed a novel supervised approach to SAT solver selection, based on representing instances as literal-clause graphs and training a graph neural network to select, from this representation, a SAT solver among a fixed portfolio so as to minimize solving runtime. The graph representations are endowed with node features that encode domain knowledge, and in the case of clause nodes, also position within the SAT formula. The resulting scheme is shown to outperform competing approaches on two benchmarks, one from an industrial circuit design application and one from the annual SAT solver competitions.

REFERENCES

[1] David L. Applegate, Robert E. Bixby, Vašek Chvátal, and William J. Cook. 2007. *The Traveling Salesman Problem: A Computational Study.* Princeton University Press.
[2] Tomas Balyo, Marijn J. H. Heule, Markus Iser, Matti Järvisalo, and Martin Suda. 2022. SAT Competition. https://satcompetition.github.io/2022

[3] Armin Biere and Mathias Fleury. 2022. Gimsatul, IsaSAT and Kissat entering the SAT Competition 2022. In *Proc. of SAT Competition 2022 – Solver and Benchmark Descriptions (Department of Computer Science Series of Publications B, Vol. B-2022-1)*, Tomas Balyo, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda (Eds.). University of Helsinki, 10–11.

[4] Bernd Bischl, Pascal Kerschke, Lars Kotthoff, Marius Lindauer, Yuri Malitsky, Alexandre Fréchette, Holger Hoos, Frank Hutter, Kevin Leyton-Brown, Kevin Tierney, and Joaquin Vanschoren. 2016. ASlib: A benchmark library for algorithm selection. *Artificial Intelligence* (2016).

[5] Quentin Cappart, Didier Chételat, Elias B Khalil, Andrea Lodi, Christopher Morris, and Petar Veličković. 2023. Combinatorial optimization and reasoning with graph neural networks. *Journal of Machine Learning Research* 24 (2023), 1–61.

[6] SAT Competition Committee. 2009. Benchmark Submission Guidelines. http://www.satcompetition.org/2009/format-benchmarks2009.html. Accessed: February 8th, 2024.

[7] SAT Competition Committee. 2023. The International SAT Competition Web Page. http://www.satcompetition.org. Accessed: February 8th, 2024.

[8] Stephen A Cook. 2023. The complexity of theorem-proving procedures. In *Logic, Automata, and Computational Complexity: The Works of Stephen A. Cook*. 143–152.

[9] James M Crawford and Andrew B Baker. 1994. Experimental results on the application of satisfiability algorithms to scheduling problems. In *Proceedings of the 2nd AAAI Conference on Artificial Intelligence*. 1092–1097.

[10] Hans Degroote, Bernd Bischl, Lars Kotthoff, and Patrick De Causmaecker. 2016. Reinforcement learning for automatic online algorithm selection-an empirical study. In *Proceedings of the 16th ITAT Conference Information Technologies - Applications and Theory*.

[11] William F Dowling and Jean H Gallier. 1984. Linear-time algorithms for testing the satisfiability of propositional Horn formulae. *The Journal of Logic Programming* (1984).

[12] Vijay Durairaj and Priyank Kalla. 2005. Variable ordering for efficient SAT search by analyzing constraint-variable dependencies. In *Proceedings of the 8th International Conference on the Theory and Applications of Satisfiability Testing*. Springer, 415–422.

[13] Vijay Prakash Dwivedi, Chaitanya K Joshi, Anh Tuan Luu, Thomas Laurent, Yoshua Bengio, and Xavier Bresson. 2023. Benchmarking graph neural networks. *Journal of Machine Learning Research* 24, 43 (2023), 1–48.

[14] Nils Froleyks, Marijn Heule, Markus Iser, Matti Järvisalo, and Martin Suda. 2021. SAT Competition 2020. *Artificial Intelligence* (2021).

[15] Matteo Gagliolo and Jürgen Schmidhuber. 2011. Algorithm portfolio selection as a bandit problem with unbounded losses. *Annals of Mathematics and Artificial Intelligence* (2011).

[16] Malay Ganai and Aarti Gupta. 2007. *SAT-based Scalable Formal Verification Solutions*. Springer.

[17] Evguenii I Goldberg, Mukul R Prasad, and Robert K Brayton. 2001. Using SAT for combinational equivalence checking. In *Proceedings Design, Automation and Test in Europe. Conference and Exhibition 2001*. IEEE, 114–121.

[18] Ian J. Goodfellow, Yoshua Bengio, and Aaron Courville. 2016. *Deep Learning*. MIT Press, Cambridge, MA, USA.

[19] Wenxuan Guo, Hui-Ling Zhen, Xijun Li, Wanqian Luo, Mingxuan Yuan, Yaohui Jin, and Junchi Yan. 2023. Machine learning methods in solving the Boolean satisfiability problem. *Machine Intelligence Research* (2023), 1–16.

[20] Sean B Holden et al. 2021. Machine learning for automated theorem proving: Learning to solve SAT and QSAT. *Foundations and Trends in Machine Learning* 14 (2021), 807–989.

[21] Shi-Yu Huang and Kwang-Ting Tim Cheng. 2012. *Formal Equivalence Checking and Design Debugging*. Vol. 12. Springer Science & Business Media.

[22] Franjo Ivančić, Zijiang Yang, Malay K Ganai, Aarti Gupta, and Pranav Ashar. 2008. Efficient SAT-based bounded model checking for software verification. *Theoretical Computer Science* 404 (2008), 256–274.

[23] Serdar Kadioglu, Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. 2011. Algorithm selection and scheduling. In *Proceedings of the 17th International Conference on the Principles and Practice of Constraint Programming*.

[24] Henry Kautz, David McAllester, Bart Selman, et al. 1996. Encoding plans in propositional logic. In *Proceedings of the Fifth International Conference on Principles of Knowledge Representation and Reasoning*. 374–384.

[25] Pascal Kerschke, Holger H Hoos, Frank Neumann, and Heike Trautmann. 2019. Automated algorithm selection: Survey and perspectives. *Evolutionary computation* (2019).

[26] Diederik P Kingma and Jimmy Ba. 2014. Adam: A method for stochastic optimization. *arXiv preprint arXiv:1412.6980* (2014).

[27] Thomas N. Kipf and Max Welling. 2017. Semi-Supervised Classification with Graph Convolutional Networks. In *Proceedings of the 5th International Conference on Learning Representations*.

[28] Vitaly Kurin, Saad Godil, Shimon Whiteson, and Bryan Catanzaro. 2020. Can Q-Learning with Graph Networks Learn a Generalizable Branching Heuristic for a SAT Solver?. In *Proceedings of the 34th International Conference on Neural Information Processing Systems*.

[29] Yang Li, Xinyan Chen, Wenxuan Guo, Xijun Li, Wanqian Luo, Junhua Huang, Hui-Ling Zhen, Mingxuan Yuan, and Junchi Yan. 2023. 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*.

[30] Marius Lindauer, Holger H Hoos, Frank Hutter, and Torsten Schaub. 2015. Autofolio: An automatically configured algorithm selector. *Journal of Artificial Intelligence Research* (2015).

[31] Andrea Loreggia, Yuri Malitsky, Horst Samulowitz, and Vijay Saraswat. 2016. Deep learning for algorithm portfolios. In *Proceedings of the 30th AAAI Conference on Artificial Intelligence*.

[32] Yuri Malitsky, Ashish Sabharwal, Horst Samulowitz, and Meinolf Sellmann. 2013. Boosting sequential solver portfolios: Knowledge sharing and accuracy prediction. In *Proceedings of the 7th International Conference on Learning and Intelligent Optimization*.

[33] Joao Marques-Silva, Inês Lynce, and Sharad Malik. 2021. Conflict-driven clause learning SAT solvers. In *Handbook of satisfiability*. IOS press, 133–182.

[34] Mladen Nikolić, Filip Marić, and Predrag Janičić. 2013. Simple algorithm portfolio for SAT. *Artificial Intelligence Review* (2013).

[35] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*.

[36] Fabian Pedregosa, Gaël Varoquaux, Alexandre Gramfort, Vincent Michel, Bertrand Thirion, Olivier Grisel, Mathieu Blondel, Peter Prettenhofer, Ron Weiss, Vincent Dubourg, et al. 2011. Scikit-learn: Machine learning in Python. *the Journal of machine Learning research* 12 (2011), 2825–2830.

[37] Daniel Selsam and Nikolaj Bjørner. 2019. Guiding high-performance SAT solvers with unsat-core predictions. In *Proceedings of the 22nd International Conference on the Theory and Applications of Satisfiability Testing*. Springer, 336–353.

[38] Daniel Selsam, Matthew Lamm, Benedikt Bünz, Percy Liang, Leonardo de Moura, and David L Dill. 2019. Learning a SAT solver from single-bit supervision. In *Proceedings of the 7th International Conference on Learning Representations*.

[39] Emina Torlak. 2009. *A constraint solver for software engineering: finding models and cores of large relational specifications*. Ph. D. Dissertation. Massachusetts Institute of Technology.

[40] Grigori S Tseitin. 1983. On the complexity of derivation in propositional calculus. *Automation of Reasoning 2: Classical Papers on Computational Logic 1967–1970* (1983), 466–483.

[41] Ashish Vaswani, Noam Shazeer, Niki Parmar, Jakob Uszkoreit, Llion Jones, Aidan N Gomez, Łukasz Kaiser, and Illia Polosukhin. 2017. Attention is all you need. In *Proceedings of the 30th International Conference on Neural Information Processing Systems*.

[42] Junshan Wang, Guojie Song, Yi Wu, and Liang Wang. 2020. Streaming graph neural networks via continual learning. In *Proceedings of the 29th ACM International Conference on Information and Knowledge Management*. 1515–1524.

[43] Minjie Yu Wang. 2019. Deep Graph Library: Towards efficient and scalable deep learning on graphs. In *ICLR Workshop on Representation Learning on Graphs and Manifolds*.

[44] Weihuang Wen and Tianshu Yu. 2023. W2SAT: Learning to generate SAT instances from Weighted Literal Incidence Graphs. arXiv:2302.00272 [cs.LG]

[45] Lin Xu, Frank Hutter, Holger Hoos, and Kevin Leyton-Brown. 2012. Evaluating component solver contributions to portfolio-based algorithm selectors. In *Proceedings of the 15th International Conference on the Theory and Applications of Satisfiability Testing*.

[46] Lin Xu, Frank Hutter, Holger H. Hoos, and Kevin Leyton-Brown. 2008. SATzilla: Portfolio-Based Algorithm Selection for SAT. *Journal of Artificial Intelligence Research* (2008).

[47] Lin Xu, Frank Hutter, Holger H Hoos, and Kevin Leyton-Brown. 2009. SATzilla2009: an automatic algorithm portfolio for SAT. (2009).

[48] Emre Yolcu and Barnabás Póczos. 2019. Learning Local Search Heuristics for Boolean Satisfiability. In *Proceedings of the 32nd International Conference on Neural Information Processing Systems*.

[49] Jiaxuan You, Haoze Wu, Clark Barrett, Raghuram Ramanujan, and Jure Leskovec. 2019. G2SAT: Learning to Generate SAT Formulas. In *Proceedings of the 32th International Conference on Neural Information Processing Systems*.

[50] Chuxu Zhang, Dongjin Song, Chao Huang, Ananthram Swami, and Nitesh V Chawla. 2019. Heterogeneous graph neural network. In *Proceedings of the 25th ACM SIGKDD international conference on knowledge discovery & data mining*. 793–803.

Table 7: Node features used in our graphical representation of a SAT instance. We report the feature dimension, feature name and a description.

Positive literal node features		
1	pos_lit_degree	Number of clauses the positive literal appears in, divided by the total number of clauses.
1	pos_lit_num_horn	Number of Horn clauses the positive literal appears in, divided by the total number of clauses.
1	lit_pos_neg_ratio	Number of clauses the positive literal appears in, divided by the number of clauses its negation appears in plus 1.
Negative literal node features		
1	neg_lit_degree	Number of clauses the negative literal appears in, divided by the total number of clauses.
1	neg_lit_num_horn	Number of Horn clauses the negative literal appears in, divided by the total number of clauses.
1	lit_pos_neg_ratio	Number of clauses the positive literal appears in, divided by the number of clauses its negation appears in plus 1.
Clause node features		
1	clause_is_horn	Is the clause Horn?
1	clause_degree	Number of literals in the clause, divided by the total number of variables in the instance.
1	clause_is_binary	Is the clause composed of two literals?
1	clause_is_ternary	Is the clause composed of three literals?
1	clause_pos_num	Number of positive literals divided by the total number of literals in the clause.
1	clause_neg_num	Number of negative literals divided by the total number of literals in the clause.
1	clause_pos_neg_ratio	Number of postivive literals, divided by the number of negative literals in the clause plus 1.
10	clause_pe	Positional encoding (see Subsection 3.2.1 in the main paper.)

A NODE FEATURES

We provide in Table 7 a summary of the features in our graph representations, for each node type.