When GNNs Met a Word Equations Solver: Learning to Rank Equations

Parosh Aziz Abdulla[1][0000−0001−6832−6611], Mohamed Faouzi Atig[1], Julie Cailler[3][0000−0002−6665−8089], Chencheng Liang[1][0000−0002−4926−8089], and Philipp Rümmer[1,2][0000−0002−2733−7098]

[1] Uppsala University, Uppsala, Sweden
[2] University of Regensburg, Regensburg, Germany
[3] University of Lorraine, CNRS, Inria, LORIA, Nancy, France

Abstract. Nielsen transformation is a standard approach for solving word equations: by repeatedly splitting equations and applying simplification steps, equations are rewritten until a solution is reached. When solving a conjunction of word equations in this way, the performance of the solver will depend considerably on the order in which equations are processed. In this work, the use of Graph Neural Networks (GNNs) for ranking word equations before and during the solving process is explored. For this, a novel graph-based representation for word equations is presented, preserving global information across conjuncts, enabling the GNN to have a holistic view during ranking. To handle the variable number of conjuncts, three approaches to adapt a multi-classification task to the problem of ranking equations are proposed. The training of the GNN is done with the help of minimum unsatisfiable subsets (MUSes) of word equations. The experimental results show that, compared to state-of-the-art string solvers, the new framework solves more problems in benchmarks where each variable appears at most once in each equation.

Keywords: Word equation · Graph neural network · String theory.

1 Introduction

A *word equation* is an equality between two *strings* that may contain variables representing unknown substrings. Solving a *word equation problem* involves finding assignments to these variables that satisfy the equality. Word equations are crucial in string constraints encountered in program verification tasks, such as validating user inputs, ensuring proper string manipulations, and detecting potential security vulnerabilities like injection attacks. The word equation problem is decidable, as shown by Makanin [33]; while the precise complexity of the problem is still open, it is know to be NP-hard and in PSPACE [38].

Abdulla et al. [11] recently proposed a Nielsen transformation-based algorithm for solving word equation problems [36]. This algorithm solves word equations by recursively applying a set of inference rules to branch and simplify the problem until a solution is reached, in a tableau-like fashion. When multiple word equations are present, the algorithm must select the equation to process next at each proof step. This selection process is critical and heavily influences the performance of the algorithm, as the unsatisfiability of a set of equations can often be shown by identifying a small unsatisfiable core of equations. At the same time, the search tree can contain infinite branches on which no solutions can be found, so that bad decisions can lead a solver astray. The situation is similar to the case of first-order logic theorem provers, where the choice of clauses to process plays a decisive role in determining efficiency. In the latter context, several deep learning techniques have been introduced to guide theorem provers [27,16,44,10,17]. However, for word equation problems, the application of learning techniques for selecting equations remains largely unexplored.

In this work, we employ Graph Neural Networks (GNNs) [15] to guide the selection of word equations at each iteration of the algorithm. Our research complements existing techniques for learning branching heuristics in word equation solvers [11]. We refer to the selection step as the *ranking process*. For this, we enhanced the existing algorithm [11] to enable the re-ordering of conjunctive word equations. The extension preserves the soundness and the completeness (for finding solutions) of the algorithm. We refer to this extended algorithm as the *split algorithm* throughout the paper.

The primary challenge in training a deep learning model to guide the ranking process lies in managing a variable number of inputs. In our work, this specifically involves handling a varying number of word equations depending on the input. Unlike with branching heuristics, which have to handle only a fixed and small number of branches (typically 2 to 3), the ranking process must handle a variable number of conjuncts. To address this challenge, we adapt multi-classification models to accommodate inputs of varying sizes using three distinct approaches. Additionally, to effectively train the GNNs, we enhance the graph representations of word equations from [11] by incorporating global term occurrence information.

Our model is trained using data from two sources: (1) Minimal Unsatisfiable Subsets (MUSes) of word equations computed by other solvers, and (2) data extracted by running the split algorithm with non-GNN-based ranking heuristics. MUSes computed by solvers such as Z3 [35] and cvc5 [14] help detect unsatisfiable conjuncts early, enabling prompt termination and improved efficiency. When the split algorithm tackles conjunctive word equations, each ranking decision creates a branch in a decision tree. By extracting the shortest path from this tree, we obtain the most effective sequence of choices, which we then use as training data.

Moreover, we explore seven options that combine the trained model with both random and manually designed heuristics for the ranking process.

We evaluated our framework on artificially generated benchmarks inspired by [20]. The benchmarks are divided into two categories: *linear* and *non-linear*, where linear means that, within a single equation, a variable can occur only once, while non-linear allows a variable to appear multiple times. Note that this definition of linearity applies to individual equations: in systems with multiple equations, even if each equation is linear, shared variables can cause a variable to appear multiple times within the system.

Finally, we compare our framework with several leading SMT solvers and a word equation solver, including Z3, Z3-Noodler [19], cvc5, Ostrich [18], and Woorpje [20]. The experimental results show that for linear problems, our framework outperforms all leading solvers in terms of the number of solved problems. For non-linear problems, when the occurrence frequency of the same variables (non-linearity) is low, our algorithm remains competitive with other solvers.

In summary, the contributions of this paper are as follows: (i) We adapt the Nielsen transformation-based algorithm [11] to allow control over the ordering of word equations at each iteration. (ii) We develop a framework to train and deploy a deep learning model for ranking and ordering conjunctive word equations within the split algorithm. The model leverages MUSes generated by leading solvers and uses graph representations enriched with global information of the formula. We propose three strategies to adapt multi-classification models for ranking tasks and explore various integration methods within the split algorithm. (iii) Experimental results demonstrate that our framework performs effectively on linear problems, with the deep learning model significantly enhancing performance. However, its effectiveness on non-linear problems is constrained by the limitations of the inference rules.

2 Preliminaries

We first define the syntax of word equations and the concept of satisfiability. Next, we explain the message-passing mechanism of Graph Neural Networks (GNNs) and describe the specific GNN model employed in our experiments.

Word Equations. We assume a finite non-empty alphabet Σ and write Σ^* for the set of all strings (or words) over Σ. The empty string is denoted by ϵ. We work with a set Γ of string variables, ranging over words in Σ^*. The symbol $\cdot$ denotes the concatenation of two strings; in our examples, we often write uv as shorthand for $u \cdot v$. The syntax of word equations is defined as follows, where $X \in \Gamma$ ranges over variables and $c \in \Sigma$ over letters:

$$\begin{array}{rr} \text{Formulae } \phi ::= \mathit{true} \mid e \wedge \phi & \qquad \text{Words } w ::= \epsilon \mid t \cdot w \\ \text{Equations } e ::= w = w & \qquad \text{Terms } t ::= X \mid c \end{array}$$

Definition 1 (Satisfiability of conjunctive word equations). *A formula ϕ is* satisfiable (SAT) *if there exists a substitution $\pi : \Gamma \to \Sigma^*$ such that, when each variable $X \in \Gamma$ in ϕ is replaced by $\pi(X)$, all equations in ϕ hold.*

Definition 2 (Linearity of a word equation). *A word equation is called* linear *if each variable occurs at most once. Otherwise, it is* non-linear.

Graph Neural Networks. *Message Passing-based GNNs* (MP-GNNs) [23] are designed to learn features of graph nodes (and potentially the entire graph) by iteratively aggregating and transforming feature information from the neighborhood of a node. Consider a graph $G = (V, E)$, with V as the set of nodes and $E \subseteq V \times V$ as the set of edges. Each node $v \in V$ has an initial representation $x_v \in \mathbb{R}^n$ and a set of neighbors $N_v \subseteq V$. In an MP-GNN comprising T message-passing steps, node representations are iteratively updated. The initial node representation of v at time step 0 is $H_v^0 = x_v$. At each step t, the representation of node v, denoted as H_v^t, is updated using the equation:

$$H_v^t = \eta_t(\rho_t(\{H_u^{t-1} \mid u \in N_v\}), H_v^{t-1}), \tag{1}$$

where H_u^{t-1} is the node representation of u in the previous iteration $t-1$, and node u is a neighbor of node v. In this context, $\rho_t : (\mathbb{R}^n)^{|N_v|} \to \mathbb{R}^n$ is an aggregation function with trainable parameters (e.g., an MLP followed by sum, mean, min, or max) that aggregates the node representations of v's neighboring nodes at the t-th iteration. Along with this, $\eta_t : (\mathbb{R}^n)^2 \to \mathbb{R}^n$ is an update function with trainable parameters (e.g., an MLP) that takes the aggregated node representation from ρ_t and the node representation of v in the previous iteration as input, and outputs the node representation of v at the t-th iteration.

In this study, we employ *Graph Convolutional Networks* (GCNs) [29] to guide our algorithm due to their computational efficiency to generalize across tasks without the need for task-specific architectural modifications. In GCNs, the node representation H_v^t of v at step $t \in \{1, ..., T\}$ where $T \in \mathbb{N}$ is computed by

$$H_v^t = \text{ReLU}(\text{MLP}^t(\text{mean}\{H_u^{t-1} \mid u \in N_v \cup \{v\}\})), \tag{2}$$

where each MLP^t is a fully connected neural network, ReLU (Rectified Linear Unit) [13] is the non-linear function $f(x) = max(0, x)$, and $H_v^0 = x_v$.

3 Split Algorithm with Ranking

Split Algorithm. Algorithm 1, splitEquations, determines the satisfiability of a word equation formula ϕ by recursively applying inference rules from [11].

The algorithm begins by checking the satisfiability of the conjunctive formula (Line 2). If all word equations can be eliminated in this way, then ϕ is SAT. If any conjunct is unsatisfiable (UNSAT), then ϕ is UNSAT. Otherwise, the satisfiability status remains *unknown* (UKN). If ϕ is in one of the first two cases, its status is returned (Line 3).

Otherwise (Line 4), RankEqs orders all conjuncts using either manually designed or data-driven methods. Next, the function ApplyRules matches and applies the corresponding inference rules to generate branches—alternative prospective solving paths for the same equation. This step is called the *branching process*. Notably, rules R_7 and R_8 generate two and three branches, respectively, while all the other rules do not cause any branching.

Next, the splitEquations call (Line 9) recursively checks the satisfiability of each branch. Let $\{b_1, \ldots, b_n\}$ be the set of branches. The formula ϕ has status SAT if at least one branch b_i is satisfiable, UNSAT if all branches are unsatisfiable, and UKN otherwise.

```
   Data: A formula $\phi$
   Result: The satisfiability status of $\phi$ (i.e., SAT, UNSAT, or UKN) and the
           simplified version of $\phi$
 1 begin
 2     $res \leftarrow$ CheckFormulaSatisfiability($\phi$)
 3     if $res \neq UKN$ then return res, $\phi$
 4     else
 5         $\phi_s$ = RankEqs($\phi$)                        // Ranking process
 6         $Branches$ = applyRules($\phi_s$)                 // Branching process
 7         $uknFlag \leftarrow 0$
 8         for $b$ in $Branches$ do
 9             $res_b, \phi_b$=splitEquations($b$)
10             if $res_b = SAT$ then return $SAT$, $\phi_b$
11             if $res_b = UKN$ then $uknFlag \leftarrow 1$
12         if $uknFlag = 1$ then return $UKN$, $\phi$
13         else return $UNSAT$, $\phi$
```

Algorithm 1: SplitEquations algorithm.

Since the inference rules apply to the leftmost equation, the performance of the algorithm is strongly influenced by both the order in which branches are processed (Line 8) and the ordering of equations in ϕ (Line 5). While the impact of branch ordering has been studied in [11], this paper explores whether employing a data-driven heuristic in RankEqs can enhance termination.

The baseline option to implement RankEqs is referred to as **RE1: Baseline**. It computes the priority of a word equation p using the following definition:

$$p = \begin{cases} 1 & \text{if } \epsilon = \epsilon \\ 2 & \text{otherwise, if } \epsilon = u \cdot v \text{ or } u \cdot v = \epsilon \\ 3 & \text{otherwise, if } a \cdot u = b \cdot v \text{ or } u \cdot a = v \cdot b \\ 4 & \text{otherwise, if } a \cdot u = a \cdot v \\ 5 & \text{otherwise} \end{cases}$$

where $a, b \in \Sigma$, and u, v are sequences of variables and letters. Smaller numbers indicate higher priority, assigning greater precedence to simpler cases where satisfiability is obvious. Word equations with the same priorities between 1 and 4 are further ordered by their length (i.e., the number of terms), with shorter equations taking precedence. For word equations with a priority of 5, the original input order is maintained. The newly created equations inherit the ranking of their parents. We refer to the split algorithm using **RE1** for RankEqs as DragonLi. The correctness of Algorithm 1 follows directly from the soundness and local completeness of the inference rules in [11]:

Lemma 1 (Soundness of Algorithm 1). *For a conjunctive word equation formula ϕ, if Algorithm 1 terminates with the result* SAT *or* UNSAT*, then ϕ is* SAT *or* UNSAT*, respectively.*

AND-OR Tree. The search tree explored by the algorithm can be represented as an AND-OR tree, as shown in Figure 1. The example illustrates the three paths, each placing different equations in the first position, generated by the ranking and branching process to solve the word equation $\phi = (Xb = bXX \wedge \epsilon = \epsilon \wedge X = a)$, where $a, b \in \Sigma$ and $X \in \Gamma$.

Example 1. In the first step, ϕ can be reordered in three distinct ways by prioritizing one conjunct to occupy the leftmost position (we ignore the order of the rest two equations, as their order does not influence the next rule application). Thus, the root of the tree branches into three paths. For each ranked formula, the inference rules are then applied to execute the branching process. By iterating these two steps alternately, the complete AND-OR tree is constructed. Notably, continuously selecting the leftmost branch that prioritizes $Xb = bXX$ at the root and applying the left branch of R_7 may lead to non-termination, as the length of the word equation keeps increasing. In contrast, prioritizing $X = a$ at the root results in a solution (UNSAT) at a relatively shallow depth, avoiding the risk of non-termination caused by further ranking and branching. In this case, exploring only a single branch during the ranking process suffices to determine the satisfiability of ϕ. This optimal path is highlighted with solid edges.

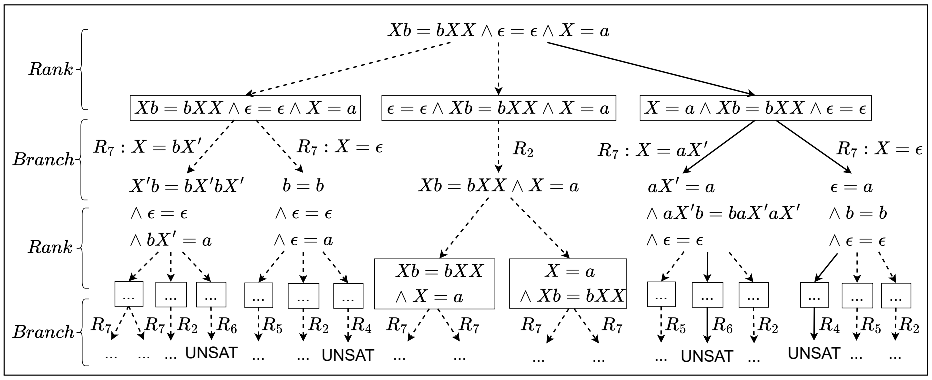
Rank
Branch
Rank
Branch
$Xb = bXX \wedge \epsilon = \epsilon \wedge X = a$
$Xb = bXX \wedge \epsilon = \epsilon \wedge X = a$
$\epsilon = \epsilon \wedge Xb = bXX \wedge X = a$
$X = a \wedge Xb = bXX \wedge \epsilon = \epsilon$
$R_7 : X = bX'$
$R_7 : X = \epsilon$
R_2
$R_7 : X = aX'$
$R_7 : X = \epsilon$
$X'b = bX'bX' \wedge \epsilon = \epsilon \wedge bX' = a$
$b = b \wedge \epsilon = \epsilon \wedge \epsilon = a$
$Xb = bXX \wedge X = a$
$aX' = a \wedge aX'b = baX'aX' \wedge \epsilon = \epsilon$
$\epsilon = a \wedge b = b \wedge \epsilon = \epsilon$
$Xb = bXX \wedge X = a$
$X = a \wedge Xb = bXX$
...
R_7 R_7 R_2 R_6 R_5 R_2 R_4 R_7 R_7 R_7 R_7 R_5 R_6 R_2 R_4 R_5 R_2
UNSAT
UNSAT
UNSAT
UNSAT

Fig. 1: AND-OR tree resulting from the word equation $Xb = bXX \wedge \epsilon = \epsilon \wedge X = a$. The formulas enclosed in boxes are generated by RankEqs, while the formulas without boxes are obtained from ApplyRules.

4 Guiding the Split Algorithm

This section details the training and application of a GNN model in Algorithm 1. We first describe the process of collecting training data, followed by the graph-based representation of each word equation. Next, we outline three model structures for ranking a set of word equations. Finally, we discuss methods for integrating the trained model back into the algorithm.

4.1 Training Data Collection

Assume that ϕ is an unsatisfiable conjunctive word equation consisting of a set of conjuncts $\mathcal{E}$.

Definition 3 (Minimal Unsatisfiable Set). *A subset $U \subseteq \mathcal{E}$ is a* Minimal Unsatisfiable Set *(MUS) if the conjunction of U is unsatisfiable, and for all conjuncts $e \in U$, the conjunction of subset $U \setminus \{e\}$ is satisfiable.*

We collect training data from two sources: (1) MUSes extracted by other solvers, including Z3, Z3-Noodler, cvc5, and Ostrich; and (2) formulas from the ranking process that lie on the shortest path from the subtree leading to UNSAT in the AND-OR trees. A numerical example of these two sources is provided in Section 5.2.

For training data from source (1), we first pass all problems to DragonLi. Next, we identify unsolvable problems and forward them to other solvers. If any solver successfully solves a problem, we select the one that finds a solution in the shortest time. This solver is then used to extract the MUS by exhaustively checking the satisfiability of all subsets of the conjuncts. Finally, each conjunct within a set of word equations is labeled based on its membership in the MUS and its length.

Formally, given a formula $\phi = e_1 \wedge \cdots \wedge e_n$, its conjuncts are denoted $\mathcal{E} = \{e_1, \ldots, e_n\}$, and an MUS $U \subseteq \mathcal{E}$. The corresponding labels of $e_i \in \mathcal{E}$ are $Y_n = \{y_1, \ldots, y_n\}$, where $y_i \in \{0, 1\}$, and their length is denoted $|e_i|$. The label y_i is computed as follows:

$$y_i = \begin{cases} 1 & \text{if } e_i \in U \text{ and } |e_i| = \min\big(\{|e| \mid e \in U\}\big), \\ 0 & \text{otherwise.} \end{cases} \tag{3}$$

We assign label 1 only to the shortest equation in the MUS, rather than labeling all MUS equations as 1 and non-MUS equations as 0, because the algorithm selects only one equation to proceed at each iteration. Our goal is to identify the most efficient choice. We assume that the shortest equation in the MUS is more likely to lead to quicker termination, as the branching process aims to reduce equation length until a form is reached where satisfiability (or unsatisfiability) can be easily concluded.

To collect training data from source (2), we pass the problems, along with the MUS extracted from other solvers, to DragonLi. If DragonLi solves the problem, multiple paths to UNSAT are generated by sequentially prioritizing each equation at the leftmost position in the ranked word equation.

Subsequently, we export and label each conjunctive word equation along the shortest path in the subtree leading to UNSAT. Formally, given a set of conjuncts $\mathcal{E} = \{e_1, \ldots, e_n\}$ of a conjunctive word equation, the corresponding labels $Y_n = (y_1, \ldots, y_n)$ are computed by

$$y_i = \begin{cases} 1 & \text{if } e_i \text{ in the shortest path of a subtree leading to UNSAT,} \\ 0 & \text{otherwise.} \end{cases} \tag{4}$$

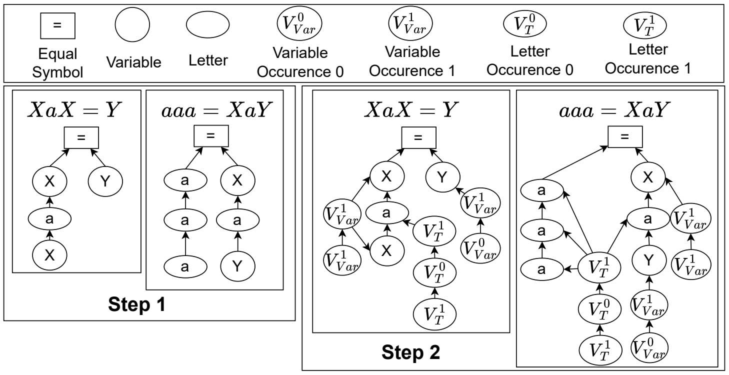

Fig. 2: The steps for constructing graph representation for the conjunctive word equations $XaX = Y \wedge aaa = XaY$ where X, Y are variables and a is a letter.

For both sources, when $\sum_{i=1}^{n} y_i > 1$, we keep the first equation with label 1 and discard the rest equations with label 1 to ensure $\sum_{i=1}^{n} y_i = 1$. When $\sum_{i=1}^{n} y_i = 0$, we discard this training data due to no positive label.

4.2 Graph Representation for Conjunctive Word Equations

The graph representation of a single word equation is discussed in [11]. However, since word equations are interconnected through shared variables, ranking them requires not only local information about individual equations but also a global perspective. By considering the entire set of word equations collectively, we can incorporate dependencies and shared structures, improving the ranking process.

To achieve this, we first represent each conjunctive word equation independently. Then, we compute the occurrences of variables and letters across all equations and integrate this global information into each individual graph representation. This enriched representation captures both the complexity of individual equations and their interactions within the system.

In details, the graph representation of a word equation is defined as $G = (V, E, v_=, V_\mathrm{T}, V_\mathrm{Var}, V^0_\mathrm{T}, V^1_\mathrm{T}, V^0_\mathrm{Var}, V^1_\mathrm{Var})$, where V is the set of nodes, $E \subseteq V \times V$ is the set of edges, and $v_= \in V$ is a special node representing the "=" symbol. The sets $V_\mathrm{T} \subseteq V$ and $V_\mathrm{Var} \subseteq V$ contain letter and variable nodes, respectively. Additionally, V^0_T and V^1_T are special nodes representing letter occurrences and V^0_Var and V^1_Var analogously represent variable occurrences.

Figure 2 illustrates the two steps involved in constructing the graph representation of the conjunctive word equations $XaX = Y \wedge aaa = XaY$, where $\{X, Y\} \subseteq \Gamma$ and $a \in \Sigma$:

- **Step 1:** Inspired by Abstract Syntax Trees (ASTs), we begin to build the graph by placing the "=" symbol as the root node. The left and right children of the root represent the leftmost terms of each side of the equation, respectively. Subsequent terms are organized as singly linked lists of nodes.

- **Step 2:** Calculate the number of occurrences of all terms across the conjunctive word equations. In this example, Occurrence$(X) = 3$, Occurrence$(Y) = 2$, and Occurrence$(a) = 5$. Their binary encodings are 11, 10 and 101 respectively. We encode these as sequentially connected nodes: (V^1_{Var}, V^1_{Var}) for X, (V^1_{Var}, V^0_{Var}) for Y, and (V^1_T, V^0_T, V^1_T) for a. Finally, we connect the roots of these nodes to their corresponding variable and letter nodes.

We chose binary encoding because using unary encoding would significantly increase the graph size, making computation inefficient. Higher-base encodings like ternary or decimal tend to blur structural distinctions, i.e., different values may be represented using the same number of nodes, making it hard for the graph structure to reflect meaningful differences. Binary encoding strikes a balance. It keeps the graph size manageable while preserving enough structural information for our GNN to effectively process word equations at the scale we target.

The rationale behind our other choices of graph representation for word equations, along with a discussion of alternative representations, is provided in the repository [2].

4.3 Training of Graph Neural Networks

In the function RankEqs of Algorithm 1, equations can be ranked and sorted based on predicted rank scores from a trained model. Given a conjunctive word equation $\phi = e_1 \wedge \cdots \wedge e_n$, the model outputs a *ranking*, i.e., a list of real numbers $\hat{Y}_n = (\hat{y}_1, \ldots, \hat{y}_n)$ in which a higher value indicates a higher rank. For example, for a conjunctive word equation $e_1 \wedge e_2$, the model might output $\hat{Y}_2 = (0.3, 0.7)$, indicating that e_2 is expected to lead to a solution more quickly than e_1, and the equations should be reordered as $e_2 \wedge e_1$.

Forward Propagation. To compute this ranking, we first transform the word equations $\{e_1, ..., e_n\}$ to their graph representations $G = \{G_1, ..., G_n\}$ where $G_i = (V, E, v_=, V_\mathrm{T}, V_\mathrm{Var}, V^0_\mathrm{T}, V^1_\mathrm{T}, V^0_\mathrm{Var}, V^1_\mathrm{Var})$. Each node $v \in V$ is first assigned an integer representing the node type: $v \in \bigcup\{V_T, V_{Var}, V^0_T, V^1_T, V^0_{Var}, V^1_{Var}\} \cup \{v_=\}$. Those integers are then passed to a trainable embedding function $\mathrm{MLP}_0 : \mathbb{Z} \to \mathbb{R}^m$ to compute all the initial node representations H^0_i in G_i.

Equation (2) defines how node representations are updated. By iterating this update rule, we obtain the node representations $H^t_i = \mathrm{GCN}(H^{t-1}_i, E)$ for $t \in \{1, \ldots, T\}$, where the relation E is used to identify neighbors. Subsequently, the representation of the entire graph is obtained by summing the node representations at time step T, resulting in $H_{G_i} = \frac{1}{n}\sum_{i=1}^{n} H^T_i$.

Then, we introduce three ways to compute the $\hat{Y}_n$:

- **Task 1**: Each graph representation H_{G_i} is given to a trainable classifier $\mathrm{MLP}_1 : \mathbb{R}^m \to \mathbb{R}^2$, which outputs $\mathbf{z}_i = \mathrm{MLP}_1(H_{G_i}) = (z_1, z_2)$. The score for graph i is then computed as $y_i = \mathit{softmax}(\mathbf{z}_i)_1$ for $y_i \in \hat{Y}_n$ where $\mathit{softmax}(\mathbf{z}_i) = (\frac{e^{z_1}}{\sum_{j=1}^{n} e^{z_j}}, \ldots, \frac{e^{z_n}}{\sum_{j=1}^{n} e^{z_j}})$ and $\mathit{softmax}(\cdot)_1$ is the first element of $\mathit{softmax}(\cdot)$. It represents the probability of the class in the first index.

- **Task 2**: All graph representations in a conjunctive word equations are first aggregated by $H_G = \frac{1}{n}\sum_{i=1}^{n} H_{G_i}$. Then, we compute the score by of each graph by $y_i = \mathit{softmax}(\text{MLP}_2(H_{G_i}||H_G))_1$ for $y_i \in \hat{Y}_n$ where $\text{MLP}_2 : \mathbb{R}^{2m} \rightarrow \mathbb{R}^2$ is a trainable classifier and $||$ denotes concatenation of two vectors.
- **Task 3**: We begin by fixing a limit n of equation within a conjunctive word equation. For conjunctive word equations containing more than n word equations, we first sort them by length (in ascending order) and then trim the list to n equations. Next, we compute scores for resulting equations using $\hat{Y}_n = \text{MLP}_3(H_{G_1}, \dots, H_{G_n})$ where $\text{MLP}_3 : \mathbb{R}^{nm} \rightarrow \mathbb{R}^n$ is a trainable classifier. Scores for any trimmed word equations are set to 0. If a conjunctive word equations contains fewer than n word equations, we fill the list with empty equations to reach n, and then compute $\hat{Y}_n$ in the same way.

Backward Propagation. The trainable parameters of the model include the weights of the embedding function MLP_0, the classifiers MLP_1, MLP_2, MLP_3, and the GCNs. Those trainable parameters are optimized together by minimizing the categorical cross-entropy loss between the predicted label $\hat{y}_i \in \hat{Y}_n$ and the true label $y_i \in Y_n$, using the equation $\mathit{loss} = -\frac{1}{n}\sum_{i=1}^{n} y_i \log(\hat{y}_i)$ where n is the number of conjuncts in the conjunctive word equations.

4.4 Ranking Options

In Algorithm 1, we introduce seven implementations of RankEqs, aimed at evaluating the efficiency of deterministic versus stochastic ranking methods.

- **RE1, Baseline:** A baseline defined in Section 3.
- **RE2, Random: RE1** is first used to compute the priority of each word equation, and then equations with a priority of 5 are randomly ordered. This approach aims to add some randomness to the baseline.
- **RE3, GNN:** Equations ranked at 5 by **RE1** are then ranked and sorted using the GNN model. While this option incurs higher overhead due to frequent use of the GNN model, it provides the most fine-grained guidance.
- **RE4, GNN-Random:** Based on **RE3**, there is a 50% chance of invoking the GNN model and a 50% chance of randomly sorting word equations with a priority of 5. This option provides insight into the performance when introducing a random process into GNN-based ranking.
- **RE5, GNN-one-shot:** Based on the priority assigned by **RE1**, the GNN model is used to rank and sort equations with a priority of 5 the first time they occur, while it is managed by **RE1** in subsequent iterations. This option invokes the GNN only once to minimize its overhead, while still maintaining its influence on subsequent iterations. Ranking and sorting the word equations early in the process has a greater impact on performance than doing them later.
- **RE6, GNN-each-n-iteration:** Based on **RE3**, instead of calling the GNN model each time multiple word equations have priority 5, it is invoked only

every $n = 5000$ calls to the RankEqs function. This option explores a balance between **RE3** and **RE5**.

- **RE7, GNN-formula-length:** Based on **RE3**, instead of calling the GNN model each time multiple word equations have priority 5, it is invoked only after $n = 1000$ calls to the RankEqs function when the length of the current word equation does not decrease. This option introduces dynamic control over calling the GNN model.

5 Experimental Results

This section describes the benchmarks and the methods used for training data collection. We also compare our evaluation data with leading solvers. The training and prediction workflow can be find in the repository [5].

5.1 Benchmarks

We initially transformed real-world benchmarks from the non-incremental QF_S, QF_SLIA, and QF_SNLIA tracks of the SMT-LIB benchmark suite [6], as well as those from the Zaligvinder benchmark suite [7], into word equation problems by removing length constraints, boolean operators, and regular expressions. However, these transformed problems were overly simplistic, as most solvers, including DragonLi, solved them easily. Consequently, we shifted to evaluating solvers using artificially generated word equation problems inspired by prior research [20,11]. We summarize the benchmarks as follows:

- **Benchmark A1:** Given a finite set of letters T and a set of variables V, the process begins by generating individual word equations of the form $s = s$, where s is a string composed of randomly selected letters from T. The maximum length of s is capped at 60. Next, substrings in s on both sides of the equation are replaced n times with the concatenation of m fresh variables from V. Here $|T| = 6$, $|V| = 10$, $n \in [0, 5]$, and $m \in [1, 5]$. Finally, multiple such word equations are conjoined to form a conjunctive word equation problem. The number of equations to be conjoined is randomly selected between 1 and 100. Since each replacement variable is a fresh variable from V, individual equations in the problem remain linear.
- **Benchmark A2:** This benchmark is generated using the same method as Benchmark A1; however, different parameters are employed to increase the difficulty while ensuring that the problem remains linear. Specifically, we use $|T| = 26$, $|V| = 100$, $n \in [0, 16]$, and $m = 1$.
- **Benchmark B:** This benchmark is generated by the same method as Benchmark A1, except it does not use fresh variables to replace substrings in s. This causes a single variable to potentially occur multiple times in an equation, making the problem non-linear. The number of equations to be conjoined is randomly picked between 2 and 50, and the maximum length of s is capped at 50. In this benchmark, we use $|T| = 10$, $|V| = 10$, $n \in [0, 5]$, and $m = 1$.

- **Benchmark C:** We first generate a word equation in the following format:

$$X_n a X_n b X_{n-1} \cdots b X_1 = a X_n X_{n-1} X_{n-1} b \cdots X_1 X_1 baa$$

where $X_1, ..., X_n$ are variables and a and b are letters. Then, we replace each b with one side of an individual equation generated by Benchmark A1. Finally, we join the individual equations to form a conjunctive word equation problem, with the maximum number of conjuncts capped at 100. This method ensures that the resulting benchmark is highly non-linear.

The statistics of the evaluation data for benchmarks is shown in the repository [3].

5.2 Training Data Collection

Table 1 outlines the training data collection. We generate 60,000 problems per benchmark and check their satisfiability with DragonLi. For instance, Benchmark A1 contains 1,859 unsolved problems, which are then passed to solvers such as Z3, Z3-Noodler, cvc5, and Ostrich. Together, these solvers identify 181 SAT and 1,678 UNSAT problems, with no single tool able to solve them all.

For UNSAT problems, we extract Minimal Unsatisfiable Subsets (MUSes) using the fastest solver. This yields 909 problems with extractable MUSes, as detailed in the repository [4]. We rank word equations within each problem based on their presence in the MUS and their length, then pass the ranked problems back to DragonLi. This allows DragonLi to prioritize word equations appearing in the MUS, enabling it to solve 518 new problems. Problems in the row *Have MUS* are transformed into a single labeled data (a conjunctive word equation). Problems in the row *DragonLi using MUS* are transformed into multiple labeled data, each representing a ranking process step on the shortest path to the solution.

The ranking heuristic's effectiveness varies with problem benchmarks. For Benchmarks A1 and A2, 57% to 58% of problems with MUSes are solved. In Benchmark B, the success rate drops to 20%, while for Benchmark C, the heuristic has no effect, solving 0 additional problems. Consequently, no training data or model was generated for Benchmark C.

6 Experimental Settings

To better investigate the influence of conjuncts order at a conjunctive word equations, we fixed the branch order for all inference rules. Additionally, we fixed the inference rule to the prefix version, meaning it always simplifies the word equation starting from the leftmost term.

Benchmarks were split uniformly into training, validation, and test sets, following standard deep-learning practice. We save the model from the epoch with the highest validation accuracy.

Table 1: Number of problems solved by different solvers and having extracted MUS. The row *Other solvers* shows the number of solved problem in total by Z3, Z3-Noodler, cvc5, and Ostrich where ✓, ×, and ∞ denotes SAT, UNSAT, and UKN respectively. The row *DragonLi using MUS* is the number of problems solved by DragonLi when using MUS to rank word equations in the first iteration.

Type	Linear				Non-linear			
Bench	A1		A2		B		C	
Total	60000		60000		60000		60000	
DragonLi	Solved	∞	Solved	∞	Solved	∞	Solved	∞
	58141	1859	50610	9390	52056	7944	31	59969
Other solvers	✓	×	✓	×	✓	×	✓	×
	181	1678	667	4167	640	7304	383	58259
Have MUS		909		1024		2996		15875
DragonLi using MUS		518		594		607		0

All training records and corresponding hyperparameters, such as a hidden layer size of 128 for all neural networks and number of message passing rounds are available in our repository [8]. For example, the experimental results for Benchmark A and Task 2 can be found in [1].

Each problem in the benchmarks is evaluated on a computer equipped with two Intel Xeon E5 2630 v4 at 2.20 GHz/core and 128GB memory. The GNNs are trained on NVIDIA A100 GPUs. We measured the number of solved problems and the average solving time (in seconds), with timeout of 300 seconds for each proof attempt.

6.1 Comparison with Other Solvers

Table 2 compares the results of three RankEqs options, **RE1**, **RE2**, and **RE5** (corresponding to DragonLi, Random-DragonLi, and GNN-DragonLi), against five solvers: Z3 [35], Z3-Noodler [19], cvc5 [14], Ostrich [18], and Woopje [20].

The primary metric is the number of solved problems. In Benchmark A1, GNN-DragonLi achieves the best performance for both SAT and UNSAT problems. For Benchmark A2, GNN-DragonLi solves the most problems overall (895 problems solved), despite not being the best in either category individually. GNN-DragonLi outperforms both DragonLi and Random-DragonLi, showing the effectiveness of data-driven heuristics over fixed and random heuristics.

As problem non-linearity increases (in Benchmark B), some solvers outperform all DragonLi options. For highly non-linear problems (Benchmark C), DragonLi solves almost no problems, regardless of the options. This is an effect entirely orthogonal to the ranking problem, however: for non-linear equations, substituting variables that appear multiple times can increase equation length, resulting in mostly infinite branches in the search tree. It then becomes more important to implement additional criteria to detect unsatisfiable equations, for instance in terms of word length or letter count (e.g., [30]), which are present in other

Table 2: Number of problems, average solving time, and average split counts for solvers across four benchmarks. The GNN model used in this table is trained on Task 2. Columns "UNI", "CS", and "CU" indicate uniquely solved, common SAT, and common UNSAT problems, respectively. The "-" denotes unavailable data. Each benchmark consists of 1000 problems.

Bench	Solver	Number of solved problems					Average solving time (split number)			
		SAT	UNSAT	UNI	CS	CU	SAT	UNSAT	CS	CU
A1	DragonLi	**24**	955	0	13	678	5.6 (244.8)	6.5 (1085.3)	5.0 (94.4)	5.7 (126.3)
	Random-DragonLi	22	944	0			5.6 (198.8)	6.3 (932.6)	5.6 (137.6)	5.7 (180.5)
	GNN-DragonLi	**24**	**961**	0			6.1 (164.7)	7.5 (1974.8)	6.1 (96.4)	6.3 (**60.5**)
	cvc5	**24**	952	1			0.5	0.6	0.1	0.3
	Z3	17	960	0			8.7	0.4	1.1	0.1
	Z3-Noodler	22	939	2			5.7	0.3	4.8	0.1
	Ostrich	17	931	0			15.0	5.5	8.0	4.7
	Woorpje	23	744	0			3.0	12.5	0.1	12.2
A2	DragonLi	59	824	0	3	0	8.5 (4233.4)	11.8 (1231.3)	4.7 (27.3)	-
	Random-DragonLi	44	806	1			24.7 (29779.6)	6.2 (210.9)	4.6 (27.3)	-
	GNN-DragonLi	59	836	4			8.4 (1330.6)	11.6 (1074.1)	5.9 (27.3)	-
	cvc5	**67**	142	15			0.6	56.0	0.1	-
	Z3	8	**870**	10			1.1	0.6	0.1	-
	Z3-Noodler	22	7	1			15.4	3.8	0.4	-
	Ostrich	13	18	2			24.8	38.8	8.6	-
	Woorpje	0	0	0	-	-	-	-	-	-
B	DragonLi	11	805	0	4	294	4.9 (62.5)	5.2 (81.5)	4.9 (29.2)	5.3 (82.4)
	Random-DragonLi	10	894	0			5.0 (58.7)	5.8 (295.2)	5.0 (27.25)	5.2 (73.1)
	GNN-DragonLi	11	821	0			6.5 (65.1)	6.8 (70.0)	6.5 (28.25)	6.8 (**60.2**)
	cvc5	12	915	0			0.1	0.6	0.1	0.7
	Z3	11	859	3			0.1	0.2	0.1	0.1
	Z3-Noodler	**24**	911	1			4.9	0.4	1.3	0.4
	Ostrich	12	**917**	2			6.9	3.7	3.3	4.2
	Woorpje	19	330	1			29.5	6.0	0.2	5.0
C	DragonLi	2	0	0	-	-	5.1 (85.5)	-	-	-
	Random-DragonLi	2	0	0	-	-	5.0 (85.5)	-	-	-
	GNN-DragonLi	-	-	-	-	-	-	-	-	-
	cvc5	0	**909**	17	-	1	-	46.9	-	17.3
	Z3	1	821	12	1		0.8	1.7	0.8	0.1
	Z3-Noodler	**7**	657	4	1		0.2	94.1	0.1	1.0
	Ostrich	0	61	0	-		-	77.2	-	27.1
	Woorpje	3	62	0	1		65.0	28.4	0.2	223.1

solvers. DragonLi deliberately does not include such optimizations, as we aim at investigating the ranking problem in a controlled setting.

For commonly solved problems, the average solving time provides sufficient data only for Benchmarks A1 and B (678 and 294 problems, respectively). In these cases, DragonLi shows no time advantage, partly due to its implementation in Python. Re-implementing the algorithm in a more efficient language, such as Rust [9], can yield over a 100x speedup for single word equation problems.

We also measure the average number of splits in solved problems to evaluate ranking efficiency. GNN-DragonLi demonstrates fewer average splits compared to other options, indicating higher problem-solving efficiency in Benchmarks A1 and B. Our results can be summarized as follows:

1. For linear problems, all DragonLi ranking options perform competitively, with GNN-DragonLi solving the highest number of problems.
2. For moderately non-linear problems (Benchmark B), DragonLi shows moderate performance, but the ranking heuristic offers limited benefits to GNN-DragonLi, leading to reduced performance compared to other options.
3. For highly non-linear problems (Benchmark C), DragonLi fails to solve most problems due to limitations in its calculus.
4. The current implementation of DragonLi offers no time advantage for commonly solved problems, though significant improvements are achievable by reimplementation.

Increasing training data for Benchmark A2 from 20,000 to 60,000 allowed GNN-DragonLi to solve additional problems, suggesting that larger training sets may enhance performance. An ablation study on alternative RankEqs options is provided as an appendix in the repository [2]. All benchmarks, evaluation results, and implementation details, including hyperparameters, are available in our GitHub repository [8].

7 Related Work

Axel Thue [39] laid the theoretical foundation of word equations by studying the combinatorics of words and sequences, providing an initial understanding of repetitive patterns. The first deterministic algorithm to solve word equations was proposed by Makanin [33], but the complexity is non-elementary. Plandowski [38] designed an algorithm that reduces the complexity to PSPACE by using a form of run-length encoding to represent strings and variables more compactly during the solving process. Artur Jeż [28] proposed a nondeterministic algorithm that runs in $O(n \log n)$ space. Closer to our approach, recent research has focused on improving the practical efficiency of solving word equations. Perrin and Pin [37] offered an automata-based technique that represents equations in terms of states and transitions. This allows the automata to capture the behavior of strings satisfying the equation. Markus et al. [22] explored graph representations and graph traversal methods to optimize the solving process for word equations, while Day et al. [20] reformulated the word equation problem as a reachability problem for nondeterministic finite automata, then encoded it as a propositional satisfiability problem that can be handled by SAT solvers. Day et al. [21] proposed a transformation system that extends the Nielsen transformation [31] to work with linear length constraints.

Deep learning [24] has been integrated with various formal verification techniques, such as scheduling SMT solvers [26], loop invariant reasoning [42,43], and guiding premise selection for Automated Theorem Provers (ATPs) [27,45]. Closely related work in learning from Minimal Unsatisfiable Subsets (MUSes) includes NeuroSAT [40,41], which utilizes GNNs to predict the probability of variables appearing in unsat cores, guiding variable branching decisions for Conflict-Driven Clause Learning (CDCL) [34]. Additionally, some recent works [12,32] explore learning MUSes to guide CHC [25] solvers.

8 Conclusion and Future Work

In this work, we extend a Nielsen transformation based algorithm [11] to support the ranking of conjunctive word equations. We adapt a multi-classification task to handle a variable number of inputs in three different ways in the ranking task. The model is trained using MUSes to guide the algorithm in solving UNSAT problems more efficiently. To capture global information in conjunctive word equations, we propose a novel graph representation for word equations. Additionally, we explore various options for integrating the trained model into the algorithms. Experimental results show that, for linear benchmarks, our framework outperforms the listed leading solvers. However, for non-linear problems, its advantages diminish due to the inherent limitations of the inference rules. Our framework not only offers a method for ranking word equations but also provides a generalized approach that can be extended to a wide range of formula ranking problems which plays a critical role is symbolic reasoning.

As future work, we aim to optimize GNN overhead, integrate GNN guidance for both branching and ranking, and extend the solver to support length constraints and regular expressions for greater real-world applicability. Our framework can be generalized to handle more decision processes in symbolic methods that take symbolic expressions as input and output a decision choices.

Disclosure of Interests. The authors have no competing interests to declare that are relevant to the content of this article.

Acknowledgement. The computations were enabled by resources provided by the National Academic Infrastructure for Supercomputing in Sweden (NAISS) at Chalmers Centre for Computational Science and Engineering (C3SE) and Uppsala Multidisciplinary Center for Advanced Computational Science (UPPMAX) partially funded by the Swedish Research Council through grant agreement no. 2022-06725. The research was also partially supported by the Swedish Research Council through grant agreement no. 2021-06327, by a Microsoft Research PhD grant, and the Wallenberg project UPDATE.

References

1. DragonLi solver experimental report for benchmark a and task 2, accessed: 2025-05-16, https://github.com/ChenchengLiang/DragonLi/tree/rank/experimental_results_tables/eval_data_GNN/A1/task_2/model
2. Github repository for the ablastion study, accessed: 2025-06-30, https://github.com/ChenchengLiang/DragonLi/blob/rank/Appendix/Ablation-study.md
3. Github repository for the statistics of evaluation data, accessed: 2025-06-30, https://github.com/ChenchengLiang/DragonLi/blob/rank/Appendix/Statistics-of-evaluation-data.md
4. Github repository for the statistics of muse, accessed: 2025-06-30, https://github.com/ChenchengLiang/DragonLi/blob/rank/Appendix/Statistics-of-MUSes.md
5. Github repository for the workflow, accessed: 2025-06-30, https://github.com/ChenchengLiang/DragonLi/blob/rank/Appendix/Workflow.md
6. The satisfiability modulo theories library (SMT-LIB), accessed: 2025-05-16, https://smtlib.cs.uiowa.edu/benchmarks.shtml
7. Zaligvinder: A string solving benchmark framework, accessed: 2025-05-16, https://zaligvinder.github.io
8. DragonLi github repository branch:rank (2025), accessed: 2025-05-16, https://github.com/ChenchengLiang/boosting-string-equation-solving-by-GNNs/tree/rank
9. wordeq_solver (2025), accessed: 2025-05-16. https://github.com/tage64/wordeq_solver
10. Abdelaziz, I., Crouse, M., Makni, B., Austil, V., Cornelio, C., Ikbal, S., Kapanipathi, P., Makondo, N., Srinivas, K., Witbrock, M., Fokoue, A.: Learning to guide a saturation-based theorem prover (2021), https://arxiv.org/abs/2106.03906
11. Abdulla, P.A., Atig, M.F., Cailler, J., Liang, C., Rümmer, P.: Guiding word equation solving using graph neural networks. In: Akshay, S., Niemetz, A., Sankaranarayanan, S. (eds.) Automated Technology for Verification and Analysis. pp. 279–301. Springer Nature Switzerland, Cham (2025)
12. Abdulla, P.A., Liang, C., Rümmer, P.: Boosting constrained Horn solving by unsat core learning. In: Dimitrova, R., Lahav, O., Wolff, S. (eds.) Verification, Model Checking, and Abstract Interpretation. pp. 280–302. Springer Nature Switzerland, Cham (2024)
13. Agarap, A.F.: Deep Learning using Rectified Linear Units (ReLU) arXiv:1803.08375 (Mar 2018). https://doi.org/10.48550/arXiv.1803.08375
14. Barbosa, H., Barrett, C., Brain, M., Kremer, G., Lachnitt, H., Mann, M., Mohamed, A., Mohamed, M., Niemetz, A., Nötzli, A., Ozdemir, A., Preiner, M., Reynolds, A., Sheng, Y., Tinelli, C., Zohar, Y.: cvc5: A versatile and industrial-strength SMT solver. In: Fisman, D., Rosu, G. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 415–442. Springer International Publishing, Cham (2022)
15. Battaglia, P.W., Hamrick, J.B., Bapst, V., Sanchez-Gonzalez, A., Zambaldi, V.F., Malinowski, M., Tacchetti, A., Raposo, D., Santoro, A., Faulkner, R., Gülçehre, Ç., Song, H.F., Ballard, A.J., Gilmer, J., Dahl, G.E., Vaswani, A., Allen, K.R., Nash, C., Langston, V., Dyer, C., Heess, N., Wierstra, D., Kohli, P., Botvinick, M., Vinyals, O., Li, Y., Pascanu, R.: Relational inductive biases, deep learning, and graph networks. CoRR **abs/1806.01261** (2018), http://arxiv.org/abs/1806.01261

16. Bártek, F., Suda, M.: Neural precedence recommender. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction – CADE 28. Lecture Notes in Computer Science, vol. 12699, pp. 503–520. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-79876-5“˙30, https://doi.org/10.1007/978-3-030-79876-5_30
17. Bártek, F., Suda, M.: How much should this symbol weigh? a GNN-advised clause selection. In: Piskac, R., Voronkov, A. (eds.) Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning. EPiC Series in Computing, vol. 94, pp. 96–111. EasyChair (2023). https://doi.org/10.29007/5f4r, /publications/paper/2BSs
18. Chen, T., Hague, M., Lin, A.W., Rümmer, P., Wu, Z.: Decision procedures for path feasibility of string-manipulating programs with complex operations. Proc. ACM Program. Lang. **3**(POPL), 49:1–49:30 (2019). https://doi.org/10.1145/3290362, https://doi.org/10.1145/3290362
19. Chen, Y.F., Chocholatý, D., Havlena, V., Holík, L., Lengál, O., Síč, J.: Z3-noodler: An automata-based string solver. In: Finkbeiner, B., Kovács, L. (eds.) Tools and Algorithms for the Construction and Analysis of Systems. pp. 24–33. Springer Nature Switzerland, Cham (2024)
20. Day, J.D., Ehlers, T., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: On solving word equations using SAT. In: Filiot, E., Jungers, R., Potapov, I. (eds.) Reachability Problems. pp. 93–106. Springer International Publishing, Cham (2019)
21. Day, J.D., Kulczynski, M., Manea, F., Nowotka, D., Poulsen, D.B.: Rule-based word equation solving. In: Proceedings of the 8th International Conference on Formal Methods in Software Engineering. p. 87–97. FormaliSE ’20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372020.3391556
22. Diekert, V., Lohrey, M.: Word equations over graph products. In: Pandya, P.K., Radhakrishnan, J. (eds.) FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science. pp. 156–167. Springer Berlin Heidelberg, Berlin, Heidelberg (2003)
23. Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for quantum chemistry. CoRR **abs/1704.01212** (2017), http://arxiv.org/abs/1704.01212
24. Goodfellow, I.J., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge, MA, USA (2016), http://www.deeplearningbook.org
25. Horn, A.: On sentences which are true of direct unions of algebras. Journal of Symbolic Logic **16**(1), 14–21 (1951). https://doi.org/10.2307/2268661
26. Hůla, J., Mojžíšek, D., Janota, M.: Graph neural networks for scheduling of SMT solvers. In: 2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI). pp. 447–451 (2021). https://doi.org/10.1109/ICTAI52525.2021.00072
27. Jakubův, J., Chvalovský, K., Olšák, M., Piotrowski, B., Suda, M., Urban, J.: ENIGMA Anonymous: Symbol-Independent Inference Guiding Machine (System Description). In: Peltier, N., Sofronie-Stokkermans, V. (eds.) Automated Reasoning. IJCAR 2020. Lecture Notes in Computer Science, vol. 12167, pp. 448–463. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1“˙29, https://doi.org/10.1007/978-3-030-51054-1_29
28. Jeż, A.: Recompression: a simple and powerful technique for word equations (2014), https://arxiv.org/abs/1203.3705

29. Kipf, T.N., Welling, M.: Semi-supervised classification with graph convolutional networks. In: 5th International Conference on Learning Representations, ICLR 2017, Toulon, France, April 24-26, 2017, Conference Track Proceedings. OpenReview.net (2017), https://openreview.net/forum?id=SJU4ayYgl
30. Kumar, A., Manolios, P.: Mathematical programming modulo strings. In: Formal Methods in Computer Aided Design, FMCAD 2021, New Haven, CT, USA, October 19-22, 2021. pp. 261–270. IEEE (2021). https://doi.org/10.34727/2021/ISBN.978-3-85448-046-4“¨36, https://doi.org/10.34727/2021/isbn.978-3-85448-046-4_36
31. Levi, F.W.: On semigroups. Bull. Calcutta Math. Soc **36**(141-146), 82 (1944)
32. Liang, C., Rümmer, P., Brockschmidt, M.: Exploring representation of horn clauses using GNNs. In: Konev, B., Schon, C., Steen, A. (eds.) Proceedings of the Workshop on Practical Aspects of Automated Reasoning Co-located with the 11th International Joint Conference on Automated Reasoning (FLoC/IJCAR 2022), Haifa, Israel, August, 11 - 12, 2022. CEUR Workshop Proceedings, vol. 3201. CEUR-WS.org (2022), https://ceur-ws.org/Vol-3201/paper7.pdf
33. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Math. Sb. (N.S.) **103(145)**(2(6)), 147–236 (1977)
34. Marques-Silva, J., Sakallah, K.A.: Grasp: A search algorithm for propositional satisfiability. IEEE Trans. Computers **48**, 506–521 (1999), https://api.semanticscholar.org/CorpusID:13039801
35. de Moura, L., Bjørner, N.: Z3: an efficient SMT solver. In: 2008 Tools and Algorithms for Construction and Analysis of Systems. pp. 337–340. Springer, Berlin, Heidelberg (March 2008)
36. Nielsen, J.: Die Isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden. Mathematische Annalen **78**, 385–397 (1917), https://api.semanticscholar.org/CorpusID:119726936
37. Pin, J.E., Perrin, D.: Infinite Words: Automata, Semigroups, Logic and Games. Elsevier (2004), https://hal.science/hal-00112831
38. Plandowski, W.: Satisfiability of word equations with constants is in PSPACE. In: 40th Annual Symposium on Foundations of Computer Science (Cat. No.99CB37039). pp. 495–500 (1999). https://doi.org/10.1109/SFFCS.1999.814622
39. Power, J.F.: Thue's 1914 paper: a translation. CoRR **abs/1308.5858** (2013), http://arxiv.org/abs/1308.5858
40. Selsam, D., Bjørner, N.: Neurocore: Guiding high-performance SAT solvers with unsat-core predictions. CoRR **abs/1903.04671** (2019)
41. Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., Dill, D.L.: Learning a SAT solver from single-bit supervision. In: 7th International Conference on Learning Representations, ICLR 2019, New Orleans, LA, USA, May 6-9, 2019. OpenReview.net (2019), https://openreview.net/forum?id=HJMC_iA5tm
42. Si, X., Dai, H., Raghothaman, M., Naik, M., Song, L.: Learning loop invariants for program verification. In: Bengio, S., Wallach, H., Larochelle, H., Grauman, K., Cesa-Bianchi, N., Garnett, R. (eds.) Advances in Neural Information Processing Systems. vol. 31. Curran Associates, Inc. (2018)
43. Si, X., Naik, A., Dai, H., Naik, M., Song, L.: Code2inv: A deep learning framework for program verification. In: Computer Aided Verification: 32nd International Conference, CAV 2020, Los Angeles, CA, USA, July 21–24, 2020, Proceedings, Part II. p. 151–164. Springer-Verlag, Berlin, Heidelberg (2020)
44. Suda, M.: Improving enigma-style clause selection while learning from history. In: Platzer, A., Sutcliffe, G. (eds.) Automated Deduction – CADE 28. pp. 543–561. Springer International Publishing, Cham (2021)

45. Wang, M., Tang, Y., Wang, J., Deng, J.: Premise selection for theorem proving by deep graph embedding. In: Proceedings of the 31st International Conference on Neural Information Processing Systems. pp. 2783—2793. NIPS'17, Curran Associates Inc., Red Hook, NY, USA (2017)