Guiding Word Equation Solving using Graph Neural Networks

Parosh Aziz Abdulla[1][0000−0001−6832−6611], Mohamed Faouzi Atig[1][0000−0001−8229−3481], Julie Cailler[2][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
parosh.abdulla@it.uu.se
mohamed_faouzi.atig@it.uu.se
chencheng.liang@it.uu.se
[2] University of Regensburg, Regensburg, Germany
philipp.ruemmer@ur.de
julie.cailler@ur.de

Abstract. This paper proposes a Graph Neural Network-guided algorithm for solving word equations, based on the well-known Nielsen transformation for splitting equations. The algorithm iteratively rewrites the first terms of each side of an equation, giving rise to a tree-like search space. The choice of path at each split point of the tree significantly impacts solving time, motivating the use of Graph Neural Networks (GNNs) for efficient split decision-making. Split decisions are encoded as multi-classification tasks, and five graph representations of word equations are introduced to encode their structural information for GNNs. The algorithm is implemented as a solver named DragonLi. Experiments are conducted on artificial and real-world benchmarks. The algorithm performs particularly well on satisfiable problems. For single word equations, DragonLi can solve significantly more problems than well-established string solvers. For the conjunction of multiple word equations, DragonLi is competitive with state-of-the-art string solvers.

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

1 Introduction

Over the past few years, reasoning within specific theories, including arithmetic, arrays, or algebraic data structures, has become one the main challenges in automated reasoning. To address the needs of modern applications, new techniques have been developed, giving rise to *SMT* (Satisfiability Modulo Theories) solvers. SMT solvers implement efficient decision procedures and reasoning methods for a wide range of theories, and are used in applications such as verification.

Among the theories supported by SMT solvers, the theory of *strings* has in particular received attention in the last years. Strings represent one of the most important data-types in programming, and string constraints are therefore relevant in various domains, from text processing to database management systems and web applications. One of the simplest kind of constraints supported by the SMT-LIB theory of strings [12] are *word equations,* i.e., equations in a free semigroup [30]. Makanin's work [36] demonstrated the decidability of the word equation problem, which was later confirmed to be in PSPACE [44]. However, even the leading SMT solvers with support for string constraints (including cvc5 [10], Z3 [39], Norn [5], TRAU [6], Ostrich [16], Woorpje [19], and Z3-Noodler [17]) tend to be incomplete for proving the unsatisfiability of word equations, illustrating the hardness of the theory.

Solving a word equation is to check for the existence of string values for variables that make equal both sides of the equation. For example, consider the equation $Xab = YaZ$, where X, Y, and Z are variables ranging over strings, and a and b are letters. This equation is satisfiable and has multiple solutions. For example, assigning a to both X and Y and b to Z results in $aab = aab$.

This paper presents an algorithm that makes use of *Graph Neural Networks* (GNN) [13] in order to solve word equations. It is an extension of the method proposed in [4] and implemented in Norn [5], referred to as the *split algorithm.* The split algorithm is, in turn, based on the well-known Nielsen transformation [40]. It builds a proof tree by iteratively applying a set of inference (split) rules on a word equation.

One critical aspect of the algorithm lies in selecting the next branch to be explored while constructing the proof tree, which significantly influences the solving time. To address this, we present a heuristic that leverages deep learning to determine the exploration order of branches. GNNs [13] represent one of the paradigms in neural network research, tailored for non-Euclidean, graph-structured data. This makes them suited for scenarios where data points are interconnected, such as social networks [22], molecular structures [23], programs [38,9], and logical formulae [50,29,41,18]. Our work represents, to the best of our knowledge, the first use of deep learning in the context of word equations.

Figure 1 illustrates the workflow of our approach. During the training stage, we initially employ the split algorithm (without GNN guidance) to solve word equation problems drawn from a training dataset. For each satisfiable (SAT) problem, we generate a corresponding proof tree. Within this tree, each branch (pair of a node and a direct child) is evaluated to determine whether it leads to a solution, as well as its distance to the corresponding leaf. Based on this information, we assign labels to each branch to indicate whether it is a favorable choice for reaching a solution. Subsequently, we model each branching point comprising the node and its child nodes as a graph. These graph representations, along with their associated labels, are then used to train the GNN.

In the prediction stage, word equations from an evaluation dataset are processed using the split algorithm, now guided by GNN. At each branching point, the current branch is first transformed into a graph representation, which is in turn fed to the trained GNN model. The GNN, using its learned insights, advises on which branch should be prioritized and explored first.

We implemented this algorithm in the DragonLi tool. Experiments were conducted using four word equation benchmarks: three of them are artificially gen-

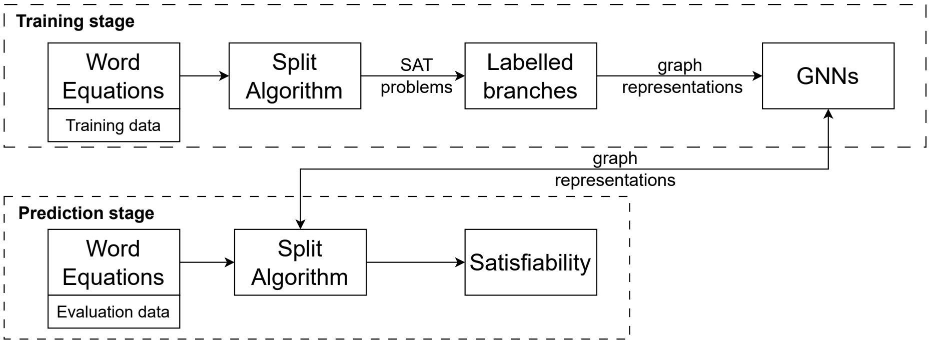

Fig. 1: The workflow diagram for the training and prediction stage

erated and inspired by Woorpje [19]; a fourth one is extracted from SMT-LIB benchmarks [1] and encodes real-world problems. Results show that for SAT problems, the pure split algorithm without GNNs is already competitive with some leading string solvers (Z3 [39], cvc5 [10], Ostrich [16], Woorpje [19], Z3-Noodler [17]), while it performs less well on UNSAT problems. We conjecture that this is due to the (relatively straightforward) depth-first search performed by our implementation of the split algorithm, which is a good strategy for finding solutions, whereas other solvers devote more time (e.g., using length reasoning) to show that formulas are unsatisfiable. Enabling GNN guidance in DragonLi uniformly improves performance on SAT problems, allowing it to outperform all other solvers in one specific benchmark. Specifically, in Benchmark 2, the GNN-guided version of DragonLi solves 115% more SAT problems than its non-GNN-guided counterpart and 43.0% more than the next best string solver, Woorpje.

In summary, the contributions of this paper are as follows:

- We define a proof system based on the split algorithm for solving word equations, tailored to combining symbolic reasoning with GNN-based guidance.
- Based on the proof system, we introduce an algorithm for integrating GNN-based guidance with the proof system.
- To train a GNN on the data obtained from solving word equations, we present five possible graph representations of word equations.
- We present an extensive experimental evaluation on four word equation benchmarks, comparing, in particular, the different graph encodings and different backtracking strategies of the algorithm.

2 Preliminaries

We start by defining the syntax of word equations, as well as the notion of satisfiability. Then, we explain the fundamental mechanism of Graph Neural Networks (GNNs), along with a description of the specific GNN model we have 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. We work with a set Γ of string variables, ranging over words in Σ^*, and denote the empty string by ϵ. 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:

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

where $X \in \Gamma$ ranges over variables and $c \in \Sigma$ over letters.

Definition 1 (Satisfiability of word equations). *A formula ϕ is* satisfiable *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 are satisfied.*

Graph Neural Networks. A *Graph Neural Network* (GNN) [13] uses *Multi-Layer Perceptrons* (MLPs) to extract features from a given graph. MLPs, also known as multi-layer neural networks [25], transform an input space to make different classes of data linearly separable, and this way learn representations of data with multiple levels of abstraction. Each layer of an MLP consists of neurons that apply a nonlinear transformation to the inputs received from the previous layer. This allows MLPs to learn increasingly complex patterns as data moves from the input layer to the output layer.

Message passing-based GNNs (MP-GNNs) [24] 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. For instance, if we represent variables in a word equation by nodes in a graph, then node features could represent symbol type (i.e., being a variable), possible assignments, or the position in the word equation.

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. 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 $h_v^t \in \mathbb{R}^n$ is the updated representation of node v after t iterations, starting from the initial representation $h_v^0 = x_v$. The node representation of u in the previous iteration $t-1$ is h_u^{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 updated node representation of v at the t-th iteration.

MP-GNNs operate under the assumption that node features can capture structural information from long-distance neighbors by aggregating and updating features of neighboring nodes. After T message-passing steps, an MP-GNN yields an updated node representation (feature) that includes information from neighbors within a distance of T, applicable to various downstream tasks like node or graph classification.

In this study, we choose *Graph Convolutional Networks* (GCNs) [31] to guide our algorithm. In GCNs, the node representation h_v^t of v at step $t \in \{1, ..., T\}$ where $T \in \mathbb{N}$ is computed by

$$h_v^t = \mathrm{ReLU}(\mathrm{MLP}^t(\mathrm{mean}\{h_u^{t-1} \mid u \in N_v \cup \{v\}\})), \tag{2}$$

where each MLP^t is a fully connected neural network, ReLU (Rectified Linear Unit) [8] is the non-linear function $f(x) = max(0, x)$, and $h_v^0 = x_v$.

3 Search Procedure and Split Algorithm

In this section, we define our proof system for word equations, the notion of a proof tree, and show soundness and completeness. We then introduce an algorithm to solve a conjunction of word equations.

3.1 Split Rules

We introduce four types of proof rules in Figure 2, each corresponding to a specific situation. The proof rules are inspired by [4], but streamlined and formulated differently. Each proof rule is of the form:

$$\textit{Name}\;\frac{P}{\begin{array}{c|c|c} [\textit{cond}_1] & & [\textit{cond}_n] \\ C_1 & \cdots & C_n \end{array}}$$

Here, *Name* is the name of the rule, P is the premise, and C_is are the conclusions. Each $\textit{cond}_i$ is a substitution that is applied implicitly to the corresponding conclusion C_i, describing the case handled by that particular branch. In our case, P is a conjunction of word equations and each C_i is either a conjunction of word equations or a final state, SAT or UNSAT.

To introduce our proof rules, we use distinct letters $a, b \in \Sigma$ and variables $X, Y \in \Gamma$, while u and v denote sequences of letters and variables.

Rules R_1, R_2, R_3, and R_4 (Figure 2a) define how to simplify word equations, and how to handle equations in which one side is empty. In R_3, note that the substitution $X \mapsto \epsilon$ is applied to the conclusion ϕ. Rules R_5 and R_6 (Figure 2b) refer to cases in which each word starts with a letter. The rules simplify the current equation, either by removing the first letter, if it is identical on both sides (R_5), or by concluding that the equation is UNSAT (R_6). Rule R_7 (Figure 2c) manages cases where one side begins with a letter and the other one with a

$$R_1\ \frac{\mathit{true}}{\mathrm{SAT}} \qquad R_2\ \frac{\epsilon = \epsilon \wedge \phi}{\phi} \qquad R_3\ \frac{X = \epsilon \wedge \phi}{\begin{array}{c}[X \mapsto \epsilon] \\ \phi\end{array}} \qquad R_4\ \frac{a \cdot u = \epsilon \wedge \phi}{\mathrm{UNSAT}}$$

with $X \in \Gamma$ and $a \in \Sigma$.

(a) Simplification rules

$$R_5\ \frac{a \cdot u = a \cdot v \wedge \phi}{u = v \wedge \phi} \qquad R_6\ \frac{a \cdot u = b \cdot v \wedge \phi}{\mathrm{UNSAT}}$$

with a, b two different letters from Σ.

(b) Letter-letter rules

$$R_7\ \frac{X \cdot u = a \cdot v \wedge \phi}{\begin{array}{c|c}[X \mapsto \epsilon] & [X \mapsto a \cdot X'] \\ u = a \cdot v \wedge \phi & X' \cdot u = v \wedge \phi\end{array}}$$

with X' a *fresh* element of Γ.

(c) Variable-letter rules

$$R_8\ \frac{X \cdot u = Y \cdot v \wedge \phi}{\begin{array}{c|c|c}[X \mapsto Y] & [X \mapsto Y \cdot Y'] & [Y \mapsto X \cdot X'] \\ u = v \wedge \phi & Y' \cdot u = v \wedge \phi & u = X' \cdot v \wedge \phi\end{array}}$$

with X', Y' *fresh* elements of Γ.

(d) Variable-variable rule

Fig. 2: Rules of the proof system for word equations

variable. The rule introduces two branches, since the variable must either denote the empty string ϵ, or its value must start with the same letter as the right-hand side. Rule R_8 (Figure 2d) handles the case in which both sides of an equation start with a variable, implying that either both variables have the same value or the value of one is included in the value of the other.

We implicitly assume symmetric versions of the rules R_3, R_4, and R_7, swapping left-hand side and right-hand side of the equation that is rewritten. For instance, the symmetric rule for R_3 would have premise $\epsilon = X \wedge \phi$.

Although our proof system is not complete for proving the unsatisfiability of word equations, we can observe that the proof rules are sound and locally complete. A proof rule is said to be *sound* if the satisfiability of the premise implies the satisfiability of one of the conclusions. It is said to be *locally complete* if the satisfiability of one of the conclusions implies the satisfiability of the premise.

Lemma 1. *The proof rules in Figure 2 are sound and locally complete.*

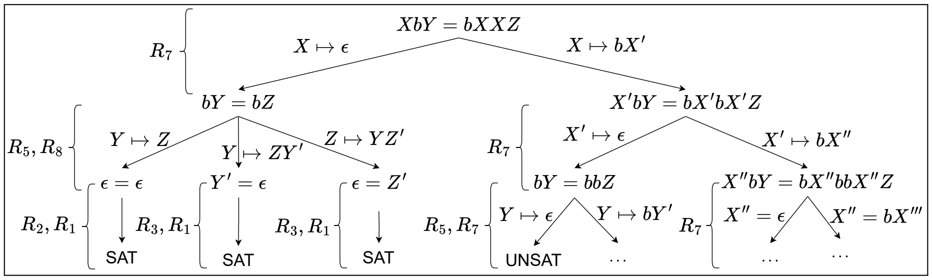

Fig. 3: Proof tree resulting from the word equation $XbY = bXXZ$

3.2 Proof Trees

Iteratively applying the proof rules to a conjunction of word equations gives rise to a *proof tree* growing downwards. Given the proof rules $R_1, \ldots, R_8$ in Figure 2, we represent a proof tree as a tuple $\tau = (N, \alpha, E, \lambda)$ where:

- N is a finite set of nodes;
- $E \subseteq N \times N$ is a set of edges, such that (N, E) is a directed tree. An edge $(n_i, n_j) \in E$ implies that n_j is derived from n_i by applying a proof rule;
- $\alpha : N \to \mathit{For} \cup \{\mathrm{SAT}, \mathrm{UNSAT}\}$ is a function mapping each node $n \in N$ to a formula or to a label SAT, UNSAT;
- $\lambda : E \to R$ is a function that assigns to each edge a proof rule.

A *path* in the proof tree is a sequence of edges starting from the root and ending with a leaf node. Due to local completeness, if there is a leaf node that is SAT, then the word equation at the root node is satisfiable. Due to soundness, if all the leaf nodes are UNSAT, then the formula at the root node is unsatisfiable.

Figure 3 illustrates the proof tree generated by applying the proof rules in Figure 2 on the word equation $\phi = (XbY = bXXZ)$. In this example, $b \in \Sigma$ and $X, Y, Z \in \Gamma$. The application of R_7 on the root generates two branches. While exploring the left branch first yields a solution (SAT) at a depth of 3, iteratively navigating through the right branch of R_7 leads to non-termination since the length of the word equation keeps increasing.

3.3 GNN-Guided Split Algorithm

We use the proof rules in Figure 2 and the idea of iterative deepening from [32] (combination of depth- and breadth-first search in a tree) to solve word equations, as shown in Algorithm 1. This algorithm aims to check the satisfiability of word equations $\phi = (\bigwedge_{i=1}^{n} w_i^l = w_i^r)$.

Algorithm 1 receives as parameter a backtrack strategy $BT \in \{BT_1, BT_2, BT_3\}$, which determines when to stop exploring a path of the proof tree and return to a previous branching point. The algorithm calls the function *solveEqsRec* (Algorithm 2), which returns the satisfiability status by exploring a proof tree recursively. At each branching point in this tree, if at least one child node is SAT, the algorithm concludes that ϕ is SAT and terminates (Line 13 of *solveEqsRec*). Conversely, if every child node is UNSAT, the current node is marked as UNSAT (Line 19 of *solveEqsRec*), and the algorithm backtracks to the last branching point. A formula ϕ is considered UNSAT only after all branches have been checked and found to be UNSAT.

We explore three different backtrack strategies, BT_1, BT_2, and BT_3:

- BT_1: This strategy performs depth-first search until it finds a SAT node or exhausts all branches to conclude UNSAT. It may lead to non-termination of the algorithm in proof trees with infinite branches, and can miss solutions; an example for this is the rightmost branch of Figure 3.
- BT_2: This hybrid strategy imposes a limit, l_{BT_2}, on the depth to which a proof branch is explored. When the maximum depth l_{BT_2} is reached, the proof search backtracks to the last branching point, and l_{BT_2} is globally increased by $l_{BT_2}^{step}$ (line 3 of *solveEqsRec*). Similarly as BT_1, this strategy can miss solutions of word equations.
- BT_3: This strategy performs the classical depth-first search with iterative deepening, by setting an initial limit l_{BT_3} on the exploration depth. This limit is increased (line 8 of *solveEqs*) when no node with label SAT is found but the tree was not fully explored yet. This strategy is complete in the sense that it will eventually find a solution for every satisfiable formula.

The performance and termination of the algorithm are highly influenced by the order in which we explore the proof tree. This order is determined by the *orderBranches* function (Line 8 of *solveEqsRec*). Our main goal in this paper is to study whether the integration of GNN models within *orderBranches* is able to optimise solving time or make it more likely for the algorithm to terminate.

For a conjunction of multiple word equations, deciding which word equation to work on first is also important for performance. Our current proof rules only rewrite the leftmost equation in a conjunction; reordering word equations is beyond the scope of this paper. We discuss this point further in Section 7.

The correctness of Algorithm 1 directly follows from the soundness and local completeness of the proof rules in Figure 2:

Lemma 2 (Soundness of Algorithm 1). *For a conjunction of word equations ϕ, if Algorithm 1 terminates with the result* SAT *or* UNSAT*, then ϕ is* SAT *or* UNSAT*, respectively.*

4 Guiding the Split Algorithm

This section describes how to train and apply the GNNs in the *orderBranches* function in Algorithm 2. We start by describing five graph representations for a conjunction of word equations, which encode word equations in form of text to graph representations to be readable by GNNs. Then, we explain how to train our classification tasks on GNNs and collect the training data. Finally, we describe different ways to apply the predicted results back to algorithm.

```
Input: Alphabet $\Sigma$ and variables $\Gamma$;
       Word equations $\phi = (\bigwedge_{i=1}^{n} w_i^l = w_i^r)$;
       Backtrack strategy $BT \in \{BT_1, BT_2, BT_3\}$;
       Global backtrack limits $l_{BT_2}, l_{BT_2}^{step}, l_{BT_3}$.
Output: The status of $\phi$: SAT, UNSAT, or UNKNOWN
begin
    $res \leftarrow$ UNKNOWN
    if $BT \in \{BT_1, BT_2\}$ then
        $res \leftarrow solveEqsRec(\phi, 0, BT, \Sigma, \Gamma)$
    if $BT = BT_3$ then
        do
            $res \leftarrow solveEqsRec(\phi, 0, BT_3, \Sigma, \Gamma)$
            $l_{BT_3} \leftarrow l_{BT_3} + 1$
        while $res \neq$ UNKNOWN
    return $res$
```

Algorithm 1: The top-level algorithm *solveEqs* for word equations

```
Input: Alphabet $\Sigma$ and variables $\Gamma$;
       Word equations $\phi = (\bigwedge_{i=1}^{n} w_i^l = w_i^r)$;
       Backtrack strategy $BT \in \{BT_1, BT_2, BT_3\}$;
       Current exploration depth $currentDepth$;
       Global backtrack limits $l_{BT_2}, l_{BT_2}^{step}, l_{BT_3}$.
Output: The state of $\phi$: SAT, UNSAT, or UNKNOWN
begin
    if $BT = BT_2 \wedge currentDepth \geq l_{BT_2}$ then
        $l_{BT_2} \leftarrow l_{BT_2} + l_{BT_2}^{step}$
        return UNKNOWN
    if $BT = BT_3 \wedge currentDepth \geq l_{BT_3}$ then
        return UNKNOWN
    $branches \leftarrow applyRules(\phi, \Sigma, \Gamma)$
    $branches \leftarrow orderBranches(branches)$
    $unknownFlag \leftarrow false$
    foreach $child$ in $branches$ do
        $res \leftarrow solveEqsRec(child, currentDepth + 1, BT, \Sigma, \Gamma)$
        if $res =$ SAT then
            return SAT
        if $res =$ UNKNOWN then
            $unknownFlag \leftarrow true$
    if $unknownFlag$ then
        return UNKNOWN
    else
        return UNSAT
```

Algorithm 2: Recursive exploration *solveEqsRec* of word equations

4.1 Representing Word Equations by Graphs

Graph representations can capture the structural information in word equations and are the standard input format for GNNs. To understand the impact of the graph structure on our framework, we have designed five graph representations for word equations.

In order to extract a single graph from the equations, we first translate a conjunction $\bigwedge_{i=1}^{n} w_i^l = w_i^r$ of word equations to a single word equation, by inserting a distinguished letter $\# \notin \Sigma$ as follows:

$$w_1^l \# w_2^l \# ... \# w_n^l = w_1^r \# w_2^r \# ... \# w_n^r, \tag{3}$$

Then, we construct the graph representations for the word equation in (3). A graph representation $G = (V, E, V_{\mathrm{T}}, V_{\mathrm{Var}})$ of a word equation consists of a set of nodes V, a set of edges $E \subseteq V \times V$, a set of terminal nodes $V_{\mathrm{T}} \subseteq V$, and a set of variable nodes $V_{\mathrm{Var}} \subseteq V$. We start constructing the graph by drawing the "=" symbol as the root node. Its left and right children are the leftmost terms of both sides of the equation, respectively. The rest of the graph is built following the choice of the graph type:

- **Graph 1:** Inspired by Abstract Syntax Trees (ASTs). Each letter and variable is represented by its own node, and words are represented by singly-linked lists of nodes.
- **Graph 2:** An extension of Graph 1, introducing additional edges from each term node back to the root node.
- **Graph 3:** An extension of Graph 1 which incorporates unique variable nodes. In this design, nodes representing variables are added, which are connected to their respective occurrences in the linked lists. This representation aims at facilitating the learning of long-distance variable relationships by GNNs.
- **Graph 4:** Similar in approach to Graph 3, but introducing unique nodes for letters instead of variables.
- **Graph 5:** A composite structure that merges the concepts of Graphs 3 and 4. It includes unique nodes for both variables and letters..

Figure 4 illustrates the five graph representations of the conjunction of word equation $aXY\#bc = XY\#Zc$, where $\{X, Y, Z\} \subseteq \Gamma$ and $\{a, b, c\} \subseteq \Sigma$.

4.2 Training of Graph Neural Networks

Forward Propagation. In the *orderBranches* function of Algorithm 1, we sort the branches by using the predictions from a trained GNN model. This GNN model performs a multi-classification task. Given a list of branches $(b_1, \ldots, b_n)$ resulting from a rule application, we expect the trained GNN model to output a list of floating-point numbers $\hat{Y}_n = (\hat{y}_1, \ldots, \hat{y}_n)$, representing priorities of the branches. A higher value for $\hat{y}_i$ indicates a higher priority of the branch. For instance, given a node with two children b_1 and b_2, the output from the model

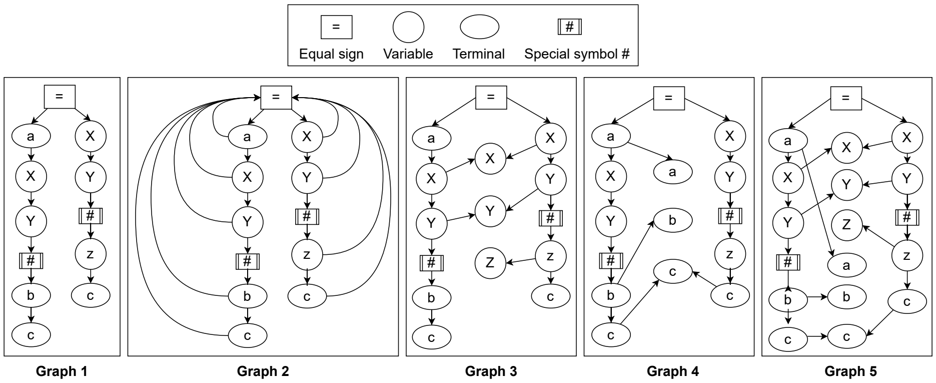

Fig. 4: The five graph representations for the word equation $aXY\#bc = XY\#Zc$ where X, Y, Z are variables and a, b, c are terminals

could be $\hat{Y_2} = (0.3, 0.7)$, expressing the prediction that b_2 will lead to a solution more quickly than b_1 and should be explored first. We detail the process of deriving $\hat{Y_n}$ at each split point using GNNs, exemplified by using $n = 2$.

Propagation on Graphs. To explain forward propagation, suppose a node labelled with formula ϕ_0 in the proof is rewritten by applying rule R_7, resulting in direct children labelled with ϕ_1, ϕ_2. The situation is similar for applications of R_8.

Formulas ϕ_0, ϕ_1, and ϕ_2 are transformed to graphs $G_0 = (V^0, E^0, V^0_{\mathrm{T}}, V^0_{\mathrm{Var}})$, $G_1 = (V^1, E^1, V^1_{\mathrm{T}}, V^1_{\mathrm{Var}})$, and $G_2 = (V^2, E^2, V^2_{\mathrm{T}}, V^2_{\mathrm{Var}})$, respectively, according to one of the encodings in Section 4.1. Each node in those graphs is then assigned an initial node representation in $\mathbb{R}^m$, which is determined by the node type: variable, letter, $=$, or $\#$. This gives rise to three initial node representation functions $H^0_i : V^i \to \mathbb{R}^m$ for $i \in \{1, 2, 3\}$, mapping the nodes of the graphs to vectors of real numbers.

Equation (2) defines how node representations are updated. Iterating the update rule, we obtain node representations $H^t_i = \mathrm{GCN}(H^{t-1}_i, E^i)$ for $i \in \{1, 2, 3\}$ and $t \in \{1, \ldots, T\}$, where the relation E^i is used to identify neighbours. Subsequently, representation of the graphs as a whole are derived by summing up the node representations at point T, resulting in $H_{G_i} = \sum_{v \in V^i} H^T_i(v)$.

Finally, these graph representations are concatenated and fed to a classifier $\mathrm{MLP} : (\mathbb{R}^m)^3 \to \mathbb{R}^2$ to calculate scores $\hat{Y_2} = \mathrm{MLP}(H_{G_0} || H_{G_1} || H_{G_2})$, where $||$ denotes concatenation of vectors. The whole process generalizes in a straightforward way to branching points in the proof tree with n children.

Backward Propagation. The trainable parameters of the model, as described above, are the initial node representations for the four types of graph nodes and the parameters of 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 following equation:

$$loss = -\frac{1}{N}\sum_{1}^{N} y_i \log(\hat{y}_i) \tag{4}$$

where N is the number of split points in a training batch. We explain how to collect the training data Y_n in the next section.

4.3 Training Data Collection

With our current algorithm, UNSAT problems always require an exhaustive exploration of a proof tree; branch ordering therefore does not affect the solving time. We have thus focused on optimizing the process of finding solutions and only extract training data from SAT problems.

To collect our training labels, we construct the complete proof tree for given conjunctions of word equations, up to a certain depth. The tree enables us to identify cases of multiple SAT pathways within the tree, and to identify situations where one branch leads to a solution more quickly than other branches.

Each node v of the proof tree with multiple children is labelled based on two criteria: the satisfiability status (SAT, UNSAT, or UNKNOWN) of the formula, and the size of the proof sub-tree underneath each of the direct children. Assume that node v has n children, each of which has status SAT, UNSAT, or UNKNOWN, respectively. If there is exactly one child of v, say the i'th child, that is SAT, then the label of v is a list of integers $(x_1, \ldots, x_n)$ with $x_i = 1$ and $x_j = 0$ for $j \neq i$. If multiple children are SAT, we examine the size of the sub-tree underneath each of those children, and label all children with minimal sub-trees with 1 in the list $(x_1, \ldots, x_n)$.

More formally, suppose a proof tree $\tau = (N, \alpha, E, \lambda)$. The satisfiability status $\sigma(v)$ of a node $v \in N$ is determined by:

$$\sigma(v) \;=\; \begin{cases} \alpha(v) & \text{if } \alpha(v) \in \{\mathrm{SAT}, \mathrm{UNSAT}, \mathrm{UNKNOWN}\} \\ \mathrm{SAT} & \text{if there is } u \in V \text{ with } \sigma(u) = \mathrm{SAT} \text{ and } (v,u) \in E \\ \mathrm{UNKNOWN} & \text{otherwise, if there is } u \in V \text{ with } \sigma(u) = \mathrm{UNKNOWN} \\ & \quad \text{and } (v,u) \in E \\ \mathrm{UNSAT} & \text{otherwise} \end{cases} \tag{5}$$

The size $\mathit{\Delta}(v)$ of the sub-tree underneath a node v is defined by:

$$\mathit{\Delta}(v) \;=\; 1 + \sum_{u \in N, (v,u) \in E} \mathit{\Delta}(u)$$

Finally, the label $Y_n^v = (y_1, ..., y_n)$ of a node v with $\sigma(v) = \mathrm{SAT}$ and n children $v_1, \ldots, v_n$, where $y_i \in \{0, 1\}$, is defined by:

$$\begin{aligned} y_i &= \begin{cases} 1 & \text{if } \sigma(v_i) = \mathrm{SAT} \text{ and } \varDelta(v_i) = \min S \\ 0 & \text{otherwise} \end{cases} \\ S &= \{\varDelta(v_i) \mid \sigma(v_i) = \mathrm{SAT}\} \end{aligned}$$

When $\sum_{i=1}^{n} y_i > 1$, we discard some children with label 1 until $\sum_{i=1}^{n} y_i = 1$ to make sure that the label for each split point is consistent.

4.4 Guidance for the Split Algorithm using the GNN Model

In Algorithm 1, we introduce five strategies for the *orderBranches* function implementation, designed to evaluate the efficiency of deterministic versus stochastic methods in branch ordering and to investigate the interplay between fixed and variable branch ordering approaches:

- **Fixed Order:** Use a predetermined branch order, defined before execution. In our experiments, we simply use the order in which the branches are displayed in Figure 2.
- **Random Order:** Reorder branches randomly.
- **GNN (S1):** Exclusively use the GNN model for branch ordering.
- **GNN-fixed (S2):** A balanced approach with a 50% chance of using the GNN model and a 50% chance of using the fixed order.
- **GNN-random (S3):** Similar to S2, but with the alternative 50% chance dedicated to random ordering.

5 Experimental Results

This section presents the benchmarks used for our experiments and details the results with the different versions of our algorithm. It also provides a comprehensive comparison with other state-of-the-art solvers.

5.1 Implementation of DragonLi

DragonLi [2] is developed from scratch using `Python 3.8` [49]. We train the models with `PyTorch` [42] and construct the GNNs using the `Deep Graph Library (DGL)` [51]. For tracking and visualizing training experiments, `mlflow` [15] is employed. Proof trees and graph representations of word equations are stored in `JSON` [43] format, while `graphviz` [21] is utilized for their tracking and visualization.

Table 1: Number of SAT (✓), UNSAT (×), UNKNOWN (∞), and evaluation (Eval) problems in the four benchmarks

Benchmark 1 Total: 3000	Benchmark 2 Total: 21000	Benchmark 3 Total: 41000	Benchmark 4 Total: 2310

2000			Eval	20000			Eval	40000			Eval	1855			Eval
✓	×	∞		✓	×	∞		✓	×	∞		✓	×	∞	
1997	0	3	1000	1293	0	18707	1000	1449	1137	37414	1000	1673	16	166	455

5.2 Benchmarks Selection

We consider two kinds of benchmarks: benchmarks that are artificially generated based on the benchmarks used to evaluate the solver Woorpje [19], as well as benchmarks extracted from the non-incremental QF_S, QF_SLIA, and QF_SNLIA track of the SMT-LIB benchmarks [1]. We summarize the benchmarks as following:

- **Benchmark 1** is generated by the mechanism used in Woorpje track I. Given finite sets of letters C and variables V, we construct a string s with maximum length of k by randomly concatenating selected letters from C. We then form a word equation $s = s$ and repeatedly replace substrings in s with the concatenation of between 1 and 5 fresh variables. This procedure guarantees that the constructed word equation is SAT.
- **Benchmark 2** is generated by the mechanism used in Woorpje track III. It first generates a word equation using the following definition:

$$\begin{aligned} &X_n a X_n b X_{n-1} b X_{n-2} \cdots b X_1 = \\ &\qquad a X_n X_{n-1} X_{n-1} b X_{n-2} X_{n-2} b \cdots b X_1 X_1 b a a \end{aligned} \tag{6}$$

  where $X_1, ..., X_n$ are variables and a and b are letters. We then generate a word equation using the mechanism for Benchmark 1, and replace letters b in (6) randomly with the left-hand side or the right-hand side of that equation.
- **Benchmark 3** is generated by conjoining multiple word equations that were randomly generated using the mechanism described in Benchmark 1. This procedure mainly produces benchmarks that are UNSAT.
- **Benchmark 4** is extracted from benchmarks from the non-incremental QF_S, QF_SLIA, and QF_SNLIA tracks of SMT-LIB. We obtain word equations by removing length constraints, regular expressions, and unsupported Boolean operators, which are not considered in this paper. As a result, benchmarks after transformation can be SAT even if the original SMT-LIB benchmarks were UNSAT.

Table 1 presents the number of problems in each benchmark. Benchmark 4 originates from a collection of 100805 SMT-LIB problems; after transformation, we obtain 2310 problems. For evaluation, we selected hold-out sets of 1000

(Benchmarks 1–3) and 455 (Benchmark 4) problems were selected uniformly at random; those sets were exclusively used for evaluation, not for training or for tuning hyper-parameters. All benchmarks, as well as our implementation and chosen hyper-parameters are available on Zenodo [3].

We then applied the split algorithm (Algorithm 1) to all benchmarks with the *fixed* reordering strategy, to determine the number of SAT, UNSAT and UNKNOWN problems. After the dispatch phase, we only retained SAT problems for the construction of the training dataset.

5.3 Experimental Settings

DragonLi was parametrized with values $l_{BT2} = 500$, $l_{BT2}^{step} = 250, l_{BT3} = 20$ for Algorithm 1. In addition, we chose a hidden layer of size 128 for all neural networks, and used a two message-passing layer (i.e. $t = 2$ in Equation 1) for the GNN. 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 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.

5.4 Comparison with Other Solvers

In Table 2, we evaluate three versions of our algorithm based on their implementation of the *orderRules* function: the fixed, random, and GNN-guided order versions (listed in Section 4.4). The performance of the GNN-guided DragonLi (row GNN in Table 2) for each benchmark is selected from the best results out of 45 experiments (see Table 3). These experiments use different combinations of five graph representations, three backtrack strategies, and three GNN guidance strategies, as shown in bold text in Table 3. We compare the results with those of five other solvers: Z3 (v4.12.2) [39], Z3-Noodler (v1.1.0) [17], cvc5 (v1.0.8) [10], Ostrich (v1.3) [16], and Woopje (v0.2) [19].

The primary metric is the number of solved problems. DragonLi outperforms all other solvers on SAT problems in benchmark 2. Notably, the GNN-based DragonLi solves the highest number of SAT problems. For the conjunction of multiple word equations (benchmark 3 and 4), DragonLi's performance is comparable to the other solvers. The order of processing word equations is crucial for those problems; currently, our solver uses only a predefined sequence, indicating significant potential for improvement. A limitation of DragonLi is determining UNSAT cases, as it requires an exhaustive check of all nodes in the proof tree.

In terms of average solving time for solved problems, GNN-based DragonLi does not hold an advantage. This is mainly due to the overhead associated with encoding equations into a graph at each split point and invoking GNNs. Furthermore, DragonLi is written in Python and not particularly optimized at this point, so that there is ample room for improvement in future efforts.

The measurement of the average number of splits in solved problems is used to gain insight into the efficiency of the different versions of our algorithm. For

Table 2: Evaluation on three metrics for different solvers. The labels SAT, UNS, UNI, CS, and CU are abbreviation of SAT, UNSAT, unique solved, commonly solved SAT, and commonly solved UNSAT, respectively. The labels Fixed, Random, and GNN are the three variations of DragonLi in Section 4.4. Entries marked "-" do not apply. Values less than 0.1 are rounded to 0.1. GNN rows for benchmarks 1-4 share the configuration $(BT_2, S1, G5)$.

Bench	Solver	Number of solved problems					Average solving time (split numbet)			
		SAT	UNS	UNI	CS	CU	SAT	UNS	CS	CU
1 (1000 SAT)	Fixed	999	-	0	777	0	4.1 (182.0)	- (-)	4.0 (169.0)	- (-)
	Random	996	-	0			4.2 (349.6)	- (-)	4.1 (269.8)	- (-)
	GNN	995	-	0			7.6 (215.7)	- (-)	7.0 (162.3)	- (-)
	cvc5	**1000**	-	0			0.1 (-)	- (-)	0.1 (-)	- (-)
	Ostrich	918	-	0			20.4 (-)	- (-)	19.6 (-)	- (-)
	Woorpje	967	-	0			1.6 (-)	- (-)	0.5 (-)	- (-)
	Z3	902	-	0			3.4 (-)	- (-)	2.4 (-)	- (-)
	Z3-Noodler	935	-	0			1.9 (-)	- (-)	1.1 (-)	- (-)
2 (1000 in total)	Fixed	33	0	10	1	0	13.2 (1115.2)	- (-)	4.8 (7)	- (-)
	Random	41	0	6			11.7 (3879.5)	- (-)	4.2 (60)	- (-)
	GNN	**71**	0	27			46.0 (1813.5)	- (-)	5.1 (5)	- (-)
	cvc5	4	2	4			2.0 (-)	0.1 (-)	0.1 (-)	- (-)
	Ostrich	14	**43**	44			40.7 (-)	31.8 (-)	2.5 (-)	- (-)
	Woorpje	23	0	2			38.3 (-)	- (-)	0.1 (-)	- (-)
	Z3	6	0	2			0.1 (-)	- (-)	4.2 (-)	- (-)
	Z3-Noodler	19	0	0			45.8 (-)	- (-)	4.2 (-)	- (-)
3 (1000 in total)	Fixed	32	79	0	23	50	5.2 (1946.2)	65.8 (4227.0)	3.6 (57.0)	38.3 (796.6)
	Random	32	79	0			9.5 (3861.8)	65.0 (4227.0)	3.8 (61.7)	38.5 (796.6)
	GNN	32	65	0			214.3 (1471.2)	1471.2 (1471.2)	4.6 (63.7)	84.0 (796.6)
	cvc5	32	943	2			0.1 (-)	0.3 (-)	0.1 (-)	0.3 (-)
	Ostrich	27	926	0			5.8 (-)	4.7 (-)	4.6 (-)	4.5 (-)
	Woorpje	**34**	723	1			12.4 (-)	12.3 (-)	0.1 (-)	23.2 (-)
	Z3	26	**953**	10			5.6 (-)	0.5 (-)	4.7 (-)	0.1 (-)
	Z3-Noodler	28	926	0			22.7 (-)	0.3 (-)	8.9 (-)	0.1 (-)
4 (455 in total)	Fixed	416	6	0	403	2	5.1 (105.5)	17.7 (17119.5)	5.1 (51.0)	5.0 (246)
	Random	415	6	0			4.9 (61.3)	17.9 (17119.5)	4.9 (38.1)	4.4 (246)
	GNN	418	5	0			5.5 (118.3)	31.8 (5019.6)	5.3 (49.0)	8.8 (246)
	cvc5	406	34	0			0.1 (-)	0.1 (-)	0.1 (-)	0.1 (-)
	Ostrich	406	6	0			1.4 (-)	1.2 (-)	1.4 (-)	1.2 (-)
	Woorpje	**420**	2	0			0.2 (-)	3.6 (-)	0.2 (-)	3.6 (-)
	Z3	**420**	10	0			0.1 (-)	0.1 (-)	0.1 (-)	0.1 (-)
	Z3-Noodler	**420**	**35**	1			0.1 (-)	0.1 (-)	0.1 (-)	0.1 (-)

Table 3: Detailed results for number of solved problems of DragonLi in terms of five graph representations (G1 to G5 represent Graph 1 to 5 in Section 4.1), three strategies to apply GNN back to the algorithm (S1, S2, S3 in Section 4.4), and three backtracking strategies (BT_1, BT_2, BT_3 in Section 3.3). The column GT denotes graph type.

GT	Benchmark 1									Benchmark 2								
	BT_1			BT_2			BT_3			BT_1			BT_2			BT_3		
	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3
G1	991	**1000**	996	997	999	999	584	627	624	57	48	44	53	45	40	3	3	3
G2	998	1000	1000	998	1000	998	584	627	624	49	46	41	53	40	50	3	3	3
G3	997	1000	998	998	1000	999	584	627	624	53	43	49	65	46	55	3	3	3
G4	985	999	997	984	999	995	584	627	624	65	52	38	59	54	44	3	3	3
G5	995	1000	999	995	1000	995	584	627	624	64	46	44	**71**	54	50	3	3	3
GT	Benchmark 3									Benchmark 4								
	BT_1			BT_2			BT_3			BT_1			BT_2			BT_3		
	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3	S1	S2	S3
G1	32	32	31	32	32	32	14	16	16	413	415	413	417	417	416	400	404	402
G2	34	32	32	34	32	34	13	16	16	416	416	416	418	418	417	400	402	403
G3	32	32	32	31	32	33	13	16	16	416	416	417	418	417	415	400	402	404
G4	**35**	32	32	34	32	32	14	16	16	414	413	415	417	417	416	400	403	403
G5	31	31	31	32	31	32	14	16	16	415	416	414	**418**	417	416	400	402	402

benchmark 1 and 2, the GNN-guided version outperforms the others on the commonly solved problems. However, for benchmarks 3 and 4, the GNN-guided version does not show advantages. This is for the same reason mentioned in the metric of the number of solved problems; namely, the performance is also influenced by the order of processing equations when dealing with the conjunction of multiple word equations.

We summarize the experimental results compared with some of the leading string solvers as follows:

1. DragonLi shows better or comparable performance on SAT problems but is limited on UNSAT problems. This occurs because the split algorithm concludes SAT upon finding one SAT node, but can conclude UNSAT only after exhaustively exploring the proof tree. In contrast, other solvers may invest more time in proving UNSAT cases, for instance by reasoning about string length or Parikh vectors.
2. DragonLi performs better than other solvers on single word equations (e.g., benchmark 2) and comparably on conjunctions of multiple word equations. This performance difference is because the initial choice of word equation for splitting is crucial for the split algorithm, and this aspect is not optimized currently.

3. Incorporating GNN guidance into the proof tree search enhances the performance of the pure split algorithm for SAT problems, but currently does not lead to an improvement for UNSAT problems.

5.5 Ablation Study

Table 3 displays the number of problems solved in 45 experiments across all benchmarks using the GNN-guided version.

In terms of backtrack strategies, BT_1 performs a pure depth-first search, but it already has good performance. BT_2 performs a depth-first search controlled by parameters l_{BT_2} and $l^{step}_{BT_2}$, and in many cases, it delivers the best performance. BT_3 conducts a systematic search on the proof tree, which is complete for proving problems SAT, but turns out to be relatively inefficient in the experiments and solves the fewest problems given a fixed timeout. This indicates that more sophisticated search strategies may lead to even better performance.

In terms of the guiding strategies (S1, S2, S3), using the GNN alone (S1) to guide the branch order is better than combining it with predefined and random orders (S2 and S3) in most cases. This indicates that the GNN model successfully learns useful patterns at each split point and can be used as a stand-alone heuristic for branching.

In terms of the five graph representations, Graph 1 has the simplest structure, which represents the syntactic information of the word equations and thus incurs the least overhead when we call the model at each split point. This yields average performance compared to other graph representations. The performances of Graph 2 are weaker than others; this is probably due to the extra edges not providing any benefits for prediction, but leading to additional computational overhead. Graphs 3 and 4 emphasize the relationships between terminals and variables, respectively, thus the performance is biased by individual problems. Graph 5 considers the relationships for both terminals and variables, thus it has bigger overhead than Graphs 1, 3, and 4, but it offers relatively good performance. This shows that representing semantic information of the word equations well in graphs helps the model to learn important patterns. In summary, the setting $(BT_2, S1, Graph5)$ performs the best.

6 Related Work

There are many techniques within solvers supporting word equations, as well as in stand-alone word equation solvers. For instance, the SMT solvers Norn [5] and TRAU [6] introduce several improvements on the inference rules [4], including length-guided splitting of equalities and a more efficient way to handle disequalities. The stand-alone word equation solver Woorpje [19] reformulates the word equation problem as a reachability problem for nondeterministic finite automata, then encodes it as a propositional satisfiability problem which can be handled by SAT solvers. In [20], the authors propose a transformation system that extends the Nielsen transformation [34] to work with linear length constraints.

This transformation system can be integrated into existing string solvers such as Z3STR3 [14], Z3SEQ [39], and CVC4 [11], thereby advancing the efficiency of word equation resolution.

GNNs excel at analyzing the graph-like structures of logic formulae, offering a complementary approach to formal verification. FormulaNet [50] is an early study guiding the premise selection for Automated Theorem Provers (ATPs). It uses MP-GNNs [24] to process the graph representation of the formulae in the proof trace extracted from HOL Light [26]. With more studies [29,41,18] exploring this path, this trend quickly expands to related fields. For instance, for SAT solvers [52,33], NeuroSAT [45,46] predicts the probability of variables appearing in unsat cores to guide the variable branching decisions for Conflict-Driven Clause Learning (CDCL) [37]. Moreover, GNNs have been combined with various formal verification techniques, such as scheduling SMT solvers [28], loop invariant reasoning [47,48], or guiding Constraint Horn Clause (CHC) solvers [7,35,27]. They provide the empirical foundations for designing the learning task in Section 4, such as the graph representation of word equations and forming the learning task in split points.

7 Conclusion and Future Work

This study introduces a GNN-guided split algorithm for solving word equations, along with five graph representations to enhance branch ordering through a multi-classification task at each split point of the proof tree. We developed our solver from scratch instead of modifying a state-of-the-art SMT solver. This decision prevents the confounding influences of pre-existing optimizations in state-of-the-art SMT solvers, allowing us to isolate and evaluate the specific impact of GNN guidance more effectively.

We investigate various configurations, including graph representations, backtrack strategies, and the conditions for employing GNN-guided branches, aiming to analyze the behaviors of the algorithm across different settings.

The evaluation tables reveal that while the split algorithm effectively solves single word equations, it does not demonstrate marked improvements for multiple conjunctive word equations relative to other solvers. This discrepancy is attributed to the significance of the processing order for conjunctive word equations, where our current solver employs a predefined order. It is possible to make use of a GNN to compute the best equation to start with. However, this involves ranking a list of elements with variable lengths, rather than performing a fixed-category classification task, and requires completely different training for this specific task. Consequently, as future work, we aim to investigate both deterministic and stochastic strategies to optimize the ordering of conjunctive word equations for the split algorithm. Our algorithm is also limited in handling UNSAT problems because it can only conclude UNSAT by exhausting the proof tree. This can be improved in future work.

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, the Swedish Foundation for Strategic Research (SSF) under the project WebSec (Ref. RIT17-0011), and the Wallenberg project UPDATE.

References

1. The satisfiability modulo theories library (SMT-LIB), accessed: 2024-04-25, https://smtlib.cs.uiowa.edu/benchmarks.shtml
2. DragonLi github repository (2024), accessed: 2024-06-28. https://github.com/ChenchengLiang/boosting-string-equation-solving-by-GNNs
3. Zenodo record of DragonLi (2024), accessed: 2024-08-21. https://zenodo.org/records/13354774
4. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification. pp. 150–166. Springer International Publishing, Cham (2014)
5. Abdulla, P.A., Atig, M.F., Chen, Y.F., Holík, L., Rezine, A., Rümmer, P., Stenman, J.: Norn: An SMT solver for string constraints. In: Kroening, D., Păsăreanu, C.S. (eds.) Computer Aided Verification. pp. 462–469. Springer International Publishing, Cham (2015)
6. Abdulla, P.A., Faouzi Atig, M., Chen, Y.F., Diep, B.P., Holík, L., Rezine, A., Rümmer, P.: TRAU: SMT solver for string constraints. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–5 (2018). https://doi.org/10.23919/FMCAD.2018.8602997
7. 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)
8. Agarap, A.F.: Deep Learning using Rectified Linear Units (ReLU) arXiv:1803.08375 (Mar 2018). https://doi.org/10.48550/arXiv.1803.08375
9. Allamanis, M., Brockschmidt, M., Khademi, M.: Learning to represent programs with graphs. CoRR **abs/1711.00740** (2017), http://arxiv.org/abs/1711.00740
10. 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)
11. Barrett, C., Conway, C.L., Deters, M., Hadarean, L., Jovanović, D., King, T., Reynolds, A., Tinelli, C.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) Computer Aided Verification. pp. 171–177. Springer Berlin Heidelberg, Berlin, Heidelberg (2011)
12. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Tech. rep., Department of Computer Science, The University of Iowa (2017), available at `www.SMT-LIB.org`
13. 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
14. Berzish, M., Ganesh, V., Zheng, Y.: Z3str3: A string solver with theory-aware heuristics. In: 2017 Formal Methods in Computer Aided Design (FMCAD). pp. 55–59 (2017). https://doi.org/10.23919/FMCAD.2017.8102241

15. Chen, A., Chow, A., Davidson, A., DCunha, A., Ghodsi, A., Hong, S.A., Konwinski, A., Mewald, C., Murching, S., Nykodym, T., Ogilvie, P., Parkhe, M., Singh, A., Xie, F., Zaharia, M., Zang, R., Zheng, J., Zumar, C.: Developments in mlflow: A system to accelerate the machine learning lifecycle. In: Proceedings of the Fourth International Workshop on Data Management for End-to-End Machine Learning. DEEM '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3399579.3399867, https://doi.org/10.1145/3399579.3399867
16. 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
17. 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)
18. Chvalovsky, K., Korovin, K., Piepenbrock, J., Urban, J.: Guiding an instantiation prover with graph neural networks. 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. 112–123. EasyChair (2023). https://doi.org/10.29007/tp23, https://easychair.org/publications/paper/5z94
19. 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)
20. 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
21. Ellson, J., Gansner, E.R., Koutsofios, E., North, S.C., Woodhull, G.: Graphviz and Dynagraph — Static and Dynamic Graph Drawing Tools, pp. 127–148. Springer Berlin Heidelberg, Berlin, Heidelberg (2004). https://doi.org/10.1007/978-3-642-18638-7_6, https://doi.org/10.1007/978-3-642-18638-7_6
22. Fan, W., Ma, Y., Li, Q., He, Y., Zhao, E., Tang, J., Yin, D.: Graph neural networks for social recommendation. In: The World Wide Web Conference. pp. 417–426. WWW '19, Association for Computing Machinery, New York, NY, USA (2019). https://doi.org/10.1145/3308558.3313488
23. Gilmer, J., Schoenholz, S.S., Riley, P.F., Vinyals, O., Dahl, G.E.: Neural message passing for Quantum chemistry. In: Proceedings of the 34th International Conference on Machine Learning - Volume 70. pp. 1263–1272. ICML'17, JMLR.org (2017)
24. 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
25. Goodfellow, I.J., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge, MA, USA (2016), http://www.deeplearningbook.org
26. Harrison, J.: Hol light: An overview. In: Proceedings of the 22nd International Conference on Theorem Proving in Higher Order Logics. pp. 60–66. TPHOLs '09, Springer-Verlag, Berlin, Heidelberg (2009)

27. Hojjat, H., Rümmer, P.: The ELDARICA Horn solver. In: 2018 Formal Methods in Computer Aided Design (FMCAD). pp. 1–7 (2018). https://doi.org/10.23919/FMCAD.2018.8603013
28. 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
29. 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. pp. 448–463. Springer International Publishing, Cham (2020)
30. Khmelevskii, Y.I.: Equations in a free semigroup. Proc. Steklov Inst. Math. **107**, 1–270 (1971)
31. 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
32. Korf, R.E.: Depth-First Iterative-Deepening: An Optimal Admissible Tree Search. Artificial Intelligence **27**(1), 97–109 (1985)
33. Kurin, V., Godil, S., Whiteson, S., Catanzaro, B.: Improving SAT solver heuristics with graph networks and reinforcement learning (2020), https://openreview.net/forum?id=B1lCn64tvS
34. Levi, F.W.: On semigroups. Bull. Calcutta Math. Soc **36**(141-146), 82 (1944)
35. 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
36. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Math. Sb. (N.S.) **103(145)**(2(6)), 147–236 (1977)
37. 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
38. Mou, L., Li, G., Zhang, L., Wang, T., Jin, Z.: Convolutional neural networks over tree structures for programming language processing. In: Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence. pp. 1287–1293. AAAI'16, AAAI Press (2016)
39. 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)
40. Nielsen, J.: Die Isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden. Mathematische Annalen **78**, 385–397 (1917), https://api.semanticscholar.org/CorpusID:119726936
41. Paliwal, A., Loos, S.M., Rabe, M.N., Bansal, K., Szegedy, C.: Graph representations for higher-order logic and theorem proving. CoRR **abs/1905.10006** (2019)
42. Paszke, A., Gross, S., Massa, F., Lerer, A., Bradbury, J., Chanan, G., Killeen, T., Lin, Z., Gimelshein, N., Antiga, L., Desmaison, A., Kopf, A., Yang, E., DeVito, Z., Raison, M., Tejani, A., Chilamkurthy, S., Steiner, B., Fang, L., Bai, J., Chintala, S.: Pytorch: An imperative style, high-performance deep learning library. In:

Advances in Neural Information Processing Systems 32, pp. 8024–8035. Curran Associates, Inc. (2019), http://papers.neurips.cc/paper/9015-pytorch-an-imperative-style-high-performance-deep-learning-library.pdf

43. Pezoa, F., Reutter, J.L., Suarez, F., Ugarte, M., Vrgoč, D.: Foundations of json schema. In: Proceedings of the 25th International Conference on World Wide Web. pp. 263–273. International World Wide Web Conferences Steering Committee (2016)
44. Plandowski, W.: An efficient algorithm for solving word equations. In: Proceedings of the Thirty-Eighth Annual ACM Symposium on Theory of Computing. pp. 467—-476. STOC '06, Association for Computing Machinery, New York, NY, USA (2006). https://doi.org/10.1145/1132516.1132584
45. Selsam, D., Bjørner, N.: Neurocore: Guiding high-performance SAT solvers with unsat-core predictions. CoRR **abs/1903.04671** (2019)
46. 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
47. 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)
48. 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)
49. Van Rossum, G., Drake, F.L.: Python 3 Reference Manual. CreateSpace, Scotts Valley, CA (2009)
50. 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)
51. Wang, M., Zheng, D., Ye, Z., Gan, Q., Li, M., Song, X., Zhou, J., Ma, C., Yu, L., Gai, Y., Xiao, T., He, T., Karypis, G., Li, J., Zhang, Z.: Deep graph library: A graph-centric, highly-performant package for graph neural networks. arXiv preprint arXiv:1909.01315 (2019)
52. Wang, W., Hu, Y., Tiwari, M., Khurshid, S., McMillan, K.L., Miikkulainen, R.: Neurocomb: Improving SAT solving with graph neural networks. CoRR **abs/2110.14053** (2021), https://arxiv.org/abs/2110.14053