Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology 2510

Learning to Guide Automated Reasoning

A GNN-Based Framework

CHENCHENG LIANG

ACTA UNIVERSITATIS
UPSALIENSIS
2025

ISSN 1651-6214
ISBN 978-91-513-2414-2
urn:nbn:se:uu:diva-551906

UPPSALA
UNIVERSITET

Dissertation presented at Uppsala University to be publicly examined in Häggsalen, Ångströmlaboratoriet, Lägerhyddsvägen 1, Uppsala, Tuesday, 6 May 2025 at 13:15 for the degree of Doctor of Philosophy. The examination will be conducted in English. Faculty examiner: Professor Stephan Schulz (Baden-Wuerttemberg Cooperative State University).

Abstract

Liang, C. 2025. Learning to Guide Automated Reasoning. A GNN-Based Framework. *Digital Comprehensive Summaries of Uppsala Dissertations from the Faculty of Science and Technology* 2510. 62 pp. Uppsala: Acta Universitatis Upsaliensis. ISBN 978-91-513-2414-2.

Symbolic methods play a crucial role in reasoning tasks such as program verification, theorem proving, and constraint solving. These methods rely on heuristic-driven decision processes to efficiently explore large or infinite state spaces. Traditional heuristics, while effective, are manually designed and often struggle with generalization across different problem domains. This dissertation introduces a deep learning-based framework to replace or augment heuristic decision-making in symbolic methods, enabling adaptive and data-driven guidance.

The framework leverages Graph Neural Networks (GNNs) to learn structural patterns from symbolic expressions, formulating decision problems as classification or ranking tasks. We propose novel graph representations for Constrained Horn Clauses (CHCs) and word equations, capturing both syntactic and semantic properties to facilitate effective learning. Various GNN architectures, including Graph Convolutional Networks (GCNs) and Relational Hypergraph Neural Networks (R-HyGNNs), are evaluated for their suitability in different symbolic reasoning tasks.

We implement the framework in a CHC solver and a word equation solver, demonstrating that learning-based heuristics can improve solver efficiency by guiding key decision processes such as clause selection and branch selection. The dissertation also explores different strategies for integrating trained models into solvers, balancing computational overhead with performance gains through caching, hybrid heuristics, and selective model querying.

Experimental results show that our framework consistently enhances solver performance, particularly in challenging problem domains. The findings suggest that deep learning can significantly improve symbolic reasoning by learning adaptive heuristics, paving the way for further integration of machine learning in formal methods.

Future research directions include extending the word equation solver, optimizing GNN architectures, and expanding training data sources.

Keywords: Graph Neural Network, CHC Solver, Word Equation

Chencheng Liang, Department of Information Technology, Computer Systems, Box 337, Uppsala University, SE-75105 Uppsala, Sweden.

ISSN 1651-6214
ISBN 978-91-513-2414-2
URN urn:nbn:se:uu:diva-551906 (http://urn.kb.se/resolve?urn=urn:nbn:se:uu:diva-551906)

To My Family and Friends

List of Papers

This thesis is based on the following papers, which are referred to in the text by their Roman numerals.

I **Exploring Representation of Horn Clauses Using GNNs**. Chencheng Liang, Philipp Rümmer, Marc Brockschmidt. *In Proceedings of 8th Workshop on Practical Aspects of Automated Reasoning (PAAR)*, 2022.

II **Boosting Constrained Horn Solving by Unsat Core Learning**. Parosh Aziz Abdulla, Chencheng Liang, Philipp Rümmer. *In Proceedings of 25th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2024.

III **Guiding Word Equation Solving Using Graph Neural Networks**. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Julie Cailler, Chencheng Liang, Philipp Rümmer. *In Proceedings of 22nd International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2024.
An extended version that corrects an error in the published version can be found at arXiv `https://arxiv.org/abs/2411.15194`

IV **When GNNs Met a Word Equations Solver: Learning to Rank Equations**. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Julie Cailler, Chencheng Liang, Philipp Rümmer. *Under Submission to 30th International Conference on Automated Deduction (CADE)*, 2025.

Reprints were made with permission from the publishers. Authors are listed in alphabetical order. I served as the primary author and led the experimental investigations.

Other Peer Reviewed Papers

- **Adaptive Fuzzy Game-Based Energy-Efficient Localization in 3D Underwater Sensor Networks**. Yali Yuan, Chencheng Liang, Xu Chen, Thar Baker, Xiaoming Fu. *ACM Transactions on Internet Technology*, 2022, Volume 22, Issue 2.
- **Topology Control for Energy-Efficient Localization in Mobile Underwater Sensor Networks Using Stackelberg Game**. Yali Yuan, Chencheng Liang, Megumi Kaneko, Xu Chen, Dieter Hogrefe. *IEEE Transactions on Vehicular Technology*, 2019, Volume 68, Issue 2.
- **Comments on "Fuzzy Multicriteria Decision Making Method Based on the Improved Accuracy Function for Interval-Valued Intuitionistic Fuzzy Sets" by Ridvan Sahin**. Yong Yang, Chencheng Liang, Shiwei Ji. *Soft Computing*, 2017, Volume 21, Issue 11.
- **Adjustable Soft Discernibility Matrix Based on Picture Fuzzy Soft Sets and its Applications in Decision Making**. Yong Yang, Chencheng Liang, Shiwei Ji, Tingting Liu. *Journal of Intelligent and Fuzzy Systems*, 2015, Volume 29, Issue 4.

Acknowledgements

To My Suervisors
First and foremost, I would like to express my deepest gratitude to my advisors, Philipp Rümmer, Parosh Aziz Abdulla, Marc Brockschmidt, and Yi Wang, for their unwavering guidance and support throughout this journey.

Philipp, thank you for giving me the opportunity to work alongside so many brilliant minds. Your intelligence, philosophical insights, and vast knowledge have profoundly influenced both my academic and personal growth. This experience has been transformative, pushing me to new heights and opening doors to exciting opportunities for the next chapter of my life.

Parosh, your kindness and support have been invaluable. Your research ideas were a lifeline during challenging times, and your meticulous feedback on my writing helped me grow rapidly as a scholar. I am deeply grateful for your mentorship.

Marc and Yi, your expertise and encouragement have been instrumental in shaping my work. Thank you for your patience and for always pushing me to think critically and creatively.

To Colleagues in the Department
I am incredibly fortunate to have worked with such brilliant colleagues. Zafer Essen, thank you for introducing me to countless tools and sharing your knowledge, which made my studies and life in Sweden so much easier. Your friendship and the lively atmosphere you brought to the office made every day enjoyable. You are truly my best buddy here. Peter Backman, your ability to think smartly has been a constant source of inspiration. Thank you for sharing your experiences and knowledge, which have been invaluable to my learning journey.

I would also like to thank all the teachers I have assisted, including Pontus Ekberg, Elias Castegren, and Didem Gürdür Broo, for creating such a supportive and flexible working environment. A special thanks to Olle Gällmo, from whom I learned not only about machine learning but also about the philosophy of teaching. Your mentorship has been invaluable.

Chang Hyun Park and Mairton Barros, thank you for your invaluable advice and support as the senior group. Your guidance not only helped me navigate the challenges of my studies but also eased my anxieties, making this journey much more manageable and rewarding.

A special thanks to Mohamed Faouzi Atig, my teacher and co-author. Your dedication and ability to excel in every situation have been a constant source of inspiration.

To my fika and corridor mates, Xuezhi Niu, Nikolaus Huber, Xin Shen, Duc Anh Nguyen, Alexandros Rouchitsas, Govind Rajanbabu, and Gaoyang Dai, thank you for creating a relaxed and fun atmosphere. You were the source of joy and laughter in the office, and I will always cherish our coffee breaks together.

To the Administrative Staff
A heartfelt thank you to all the administrative and HR staff for your tireless support. Your efforts ensured that every trip and administrative process ran smoothly and efficiently, allowing me to focus on my research.

To My Friends and Family
To all my friends and neighbors in Sweden, thank you for your warmth and companionship. A special thanks to Shenghui Li, Xi Weng, Biqin Fang, Yunyun Zhu, Jing Xu, Weijie Xu, and Yulia Essen for your help and friendship.

As the ~~destroyer~~ founder of the ~~Mahjong~~ Poker club, I want to thank Zheqiang Xu for providing the venue and Xueying Kong for organizing our gatherings. To all my Poker friends, thank you for making life in Sweden so much more enjoyable. Without you, my time here would have been far less colorful.

Special thanks to Sijia Hu for our endless discussions about philosophy. Her companionship, whether through intellectual debates or shared laughter over questionable choices in life, has been invaluable throughout this journey.

Finally, I want to thank my parents for their unconditional love and support. Words cannot fully express my gratitude, but I hope you know how much I love you. This achievement is as much yours as it is mine.

Chencheng Liang
Uppsala, March 2025

Infrastructure and Funding Acknowledgements

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 (VR) through grant agreements no. 2022-06725. The research was also partially supported by the Swedish Research Council through the grant agreement no. 2018-04727 and 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.

Contents

1 Introduction 15

1.1 Overview 15

1.2 Motivating Examples 16

1.3 Deep Learning 18

1.4 Research Questions 19

1.5 Learning-Based Framework 21

1.6 Thesis Outline 23

2 Background 25

2.1 Constrained Horn Clauses 25

2.2 Word Equations 28

2.3 Minimal Unsatisfiable Subsets 30

2.4 Deep Neural Networks 31

3 GNN-Based Framework to Rank Formulas 35

3.1 Training Tasks for Decision Problems (RQ1) 35

3.2 Representations of Formulas (RQ2) 37

3.3 GNN Selection (RQ3) 41

3.4 Integrating Strategies (RQ4) 43

4 Summary of Contributions 45

4.1 Paper I 45

4.2 Paper II 47

4.3 Paper III 49

4.4 Paper IV 51

5 Conclusions and Future Work 53

5.1 Conclusions 53

5.2 Future Work 53

Summary in Swedish 55

References 57

List of Acronyms

CHC	Constrained Horn Clause
MUS	Minimal Unsatisfiable Subset
CEGAR	Counterexample-Guided Abstraction Refinement
MLP	Multi-Layer Perceptron
GNN	Graph Neural Network
GCN	Graph Convolutional Network
R-GCN	Relational Graph Convolutional Networks
R-HyGNN	Relational Hypergraph Neural Network
GIN	Graph Isomorphism Network
GAT	Graph Attention Networ
CNN	Convolutional Neural Network
SMT	Satisfiability Modulo Theories
ATP	Automatic Theorem Prover
FOL	First-Order Logic
HOL	Higher-Order Logic
CDCL	Conflict-Driven Clause Learning
ARG	Abstract Reachability Graph
SAT	Satisfiable
UNSAT	Unsatisfiable
CDHG	Control- and Data-Flow Hypergraph
CFHE	Control-Flow Hyperedge
DFHE	Data-Flow Hyperedge
CG	Constraint Graph

1. Introduction

1.1 Overview

Analysis and verification of complex systems often require exploring a vast or even infinite state space to ensure conformance with desired specifications [1, 2]. As these systems grow in complexity, exhaustively enumerating and checking every possible configuration becomes infeasible due to the well-known phenomenon of combinatorial explosion. This challenge has led researchers to develop more efficient approaches, most notably the use of symbolic expressions to represent large or infinite sets of states in a compact form. Symbolic expressions, frequently embodied in logical or algebraic forms, effectively encode multiple states and transitions within a single representation, reducing the memory and computational resources required to handle complex systems. Symbolic methods refer to a class of techniques that operate on symbolic expressions to explore and manipulate collections of states and transitions [3]. They harness the structure of symbolic expression, in particular formulas, to prune infeasible paths or quickly detect contradictions, offering a more feasible alternative to exhaustive enumeration. This makes them both space-efficient and computationally powerful.

Symbolic methods rely on several fundamental techniques, including *deductive reasoning* and *rewriting* [4], which are also known as *automated reasoning*. Deductive reasoning uses logical inference rules to derive conclusions from given premises. For instance, *resolution* is a specific rule of inference used within deductive reasoning that systematically merges clauses to eliminate literals and detect contradictions when they arise. Rewriting simplifies complex expressions through a set of predefined rules.

Typical applications of automated reasoning include program verification and computer computer-assisted proofs in mathematics. For instance, in program verification, the primary goal is to ensure that a program behaves correctly according to its specifications. The reachable states of a program, such as the values of its variables, its control-flow location, and other runtime conditions, are captured as symbolic expressions rather than enumerated explicitly. Therefore, the program's execution paths can be systematically expressed symbolically and explored. Rewriting can simplify symbolic expressions. Deductive reasoning can propagate constraints across symbolic states.

Symbolic methods offer significant advantages for handling large or infinite state spaces, but they also present several challenges [3]:

1. Complex Representations: Symbolic expressions can become very large or intricate, especially when representing complex systems or diverse data types. Managing these expressions efficiently is often non-trivial.

2. Theory Handling: Many real-world problems involve multiple theories (e.g., arithmetic, bit-vectors, arrays), so solvers must handle combinations of logical theories consistently. Extending resolution, rewriting, or other symbolic algorithms to work across different theories can be highly complex.
3. Scalability: Although symbolic techniques can be more efficient than exhaustive enumeration, they still face exponential worst-case scenarios. As systems grow, finding solutions or proofs can remain intractable without careful optimization or heuristics that guide the search process.

In practice, the typical backend tools of symbolic methods include Automatic Theorem Provers (ATPs) [5], SAT/SMT solvers [6, 7], and CHC solvers [8].

ATPs automatically establish the validity of logical statements, operating within formal systems to derive proofs without human intervention. SAT solvers determine the satisfiability of propositional logic formulas. SMT solvers extend SAT solving by incorporating background theories such as arithmetic and equality reasoning. CHC solvers address satisfiability problems within a fragment of first-order logic, particularly useful in program verification and synthesis.

To handle complex representations, many tools represent formulas internally in compact data structures such as Directed Acyclic Graphs (DAGs) and Binary Decision Diagrams (BDDs) [9], allowing shared subexpressions and avoiding repetitive computation.

To handle different theories, SMT solvers extend SAT solving by delegating theory-specific checks to specialized decision procedures. For theories with disjoint signatures, the Nelson–Oppen method [10] can combine decision procedures by exchanging equalities. Problems (especially those involving loops, recursion, and other inductive structures) are encoded as Horn clauses augmented with constraints from a background theory. CHC solvers then handle them using specialized algorithms that accommodate various background theories.

Symbolic methods can explore search spaces systematically. However, they may fail to yield useful results within a reasonable timeframe. To improve the scalability, selecting appropriate heuristics is crucial and often application-dependent. This dissertation focuses on building a general learning-based framework to construct the heuristics.

1.2 Motivating Examples

Example 1. Saturation-based theorem proving (e.g., E [11, 12] and Vampire [13]) is one of the most common and widely used approaches in first-order logic (FOL) theorem proving. At the heart of this approach is the *given-clause algorithm*, which divides the working set of clauses into two groups: an *active set* containing processed clauses and a *passive set* containing clauses that have been generated but not yet used for inference. The prover repeatedly selects a clause from the passive set based on heuristics such as clause weight, term complexity, or relevance to the proof goal. This selected clause is then moved to the active set, where it undergoes inference operations such as resolution or superposition with existing active clauses. The newly generated clauses are added back to the passive set, and redundant or subsumed clauses are removed to maintain efficiency. This process iterates, applying inferences until no new clauses can be derived. If the empty clause appears, the theorem is proved; otherwise, the proof attempt fails.

At each iteration, clause selection is crucial for controlling the proof search and avoiding unnecessary inferences. Various heuristics are proposed to guide this selection based on features of clauses such as age, weight, size, or relevance to the proof goal [14]. The heuristics can be predefined or learning-based. Predefined heuristics include first-in/first-out, symbol counting, etc. In recent years, there have been many successful attempts at learning-based heuristics [15]. For instance, Deepire [16] uses recursive neural networks to classify the clauses based only on their derivation history. ENIGMA [17] uses both Gradient Boosting Decision Trees (GBDTs) [18] and Graph Neural Networks (GNNs) [19] to select the clauses based on the syntax trees of clauses and variable statistics.

Example 2. Many modern SAT solvers, such as MiniSat [20], Glucose [21], and Kissat [22], are based on the Conflict-Driven Clause Learning (CDCL) algorithm [23, 24]. The CDCL algorithm begins by selecting an unassigned variable and assigning it a truth value. It then performs unit propagation to deduce implied assignments. This propagation continues until no further unit clauses remain or a conflict is encountered. If no conflict occurs, the solver checks whether all variables have been assigned. If so, the formula is satisfiable. Otherwise, if a conflict arises, the solver analyzes the sequence of assignments leading to the conflict. Using an implication graph, it identifies the root cause and derives a new clause (the learned clause) to prevent the same conflict in future searches. Instead of backtracking to the most recent decision, the solver performs non-chronological backtracking, known as backjumping, directly to the decision level responsible for the conflict. This avoids redundant exploration and accelerates the search. The process iterates until the solver either finds a satisfying assignment or proves the formula unsatisfiable. To escape unproductive regions of the search space, CDCL solvers periodically restart the search while retaining learned clauses, balancing exploration and exploitation.

CDCL involves multiple decision problems, such as selecting an unassigned variable for branching, determining which learned clauses to prune, and deciding when to restart. Branching variable selection has been extensively studied, with many predefined heuristics proposed over the decades [25]. One such heuristic, Exponential Variable State-Independent Decaying Sum

(EVSIDS) [24], assigns scores to variables based on the number of conflicts they have participated in, with more recent conflicts weighted significantly higher than past conflicts. In recent years, learning-based approaches have gained attention. For instance, NeuroSAT [26] periodically resets EVSIDS scores based on the predictions of a message-passing neural network. For restarting heuristics, Liang et al. [27] propose a machine learning-based restart policy that predicts the quality of the next learned clause by analyzing the history of previously learned clauses.

Summary of Examples. Beyond the previously mentioned examples, nearly all backend tools of symbolic methods incorporate multiple heuristic-driven decision processes. These decision processes can be viewed as functions, which may either be predefined or dynamically adapted through learning-based approaches.

In recent years, with the remarkable advancements in deep learning [28], there has been a growing amount of research dedicated to integrating deep learning techniques into symbolic reasoning tools. These efforts aim to enhance decision-making processes by leveraging learned representations, enabling adaptive heuristics that outperform static strategies. This paradigm shift has demonstrated significant potential in improving the efficiency of symbolic reasoning systems. However, the heuristics presented in these studies are tailored to specific cases and do not generalize well across different problem domains.

This dissertation presents a general deep learning-based framework that provides a complete pipeline for training deep learning models to replace the decision processes in symbolic methods. By making these processes data-driven, the framework eliminates the need for manually specifying domain-specific knowledge. We apply our framework to the CHC solver Eldarica [29] and the word equation solver DragonLi (Paper III) to demonstrate its generalization capability and effectiveness.

1.3 Deep Learning

The remarkable achievements of deep learning are exemplified by systems like AlphaGo [30] in the game of Go, AlphaFold [31] in biology, and ChatGPT [32] in natural language processing. In this dissertation, deep learning refers to neural networks with multiple layers.

The success of these applications is underpinned by the Universal Approximation Theorem [33], which asserts that feedforward neural networks with at least one hidden layer can approximate any continuous function to a desired degree of accuracy, given sufficient parameters. This theorem provides a theoretical foundation for the versatility of neural networks in modeling complex, non-linear relationships inherent in various data types. However, this is not the only reason why deep learning works so well. For instance, as an approximator, Support Vector Machines (SVMs) [34] with certain kernels, such as the Radial Basis Function (RBF) [35], polynomial kernels, and specific dot product kernels, have also been demonstrated to possess universal approximation capabilities.

Another key to the success of deep learning is that it works as an effective automatic feature extractor [36, 37]. Through the training process, it learns to identify and represent abstract features from raw data without manual intervention. This capability allows it to adapt to diverse tasks, from strategic game playing and scientific discovery to generating human-like text, by capturing intricate patterns and structures within the data.

Deep Learning in Symbolic Methods. In symbolic methods, deep learning serves two primary roles: as an approximator for classification and regression tasks and as a feature extractor for symbolic expressions. The main motivation for integrating deep learning is its ability to autonomously extract abstract features from symbolic expressions, particularly formulas. This allows the model to learn essential patterns for a given task without requiring explicit domain-specific knowledge.

For example, consider designing a heuristic to guide clause selection in saturation-based ATPs. A common domain-specific heuristic selects clauses based on statistics from the passive set, favoring smaller clauses first, as they are more likely to resolve to the empty clause quickly.

When deep learning is used as a feature extractor, we assume it can learn abstract features (properties) such as the likelihood that a clause will lead to a conflict. A downstream classifier can then leverage these extracted features to determine which clause to select more efficiently.

However, learning such useful abstract features requires well-designed and tailored training data and tasks for specific decision processes. A key question arises: Assume that the decision processes in symbolic methods take formulas as input and produce decisions as output.
Can we generalize the working pipeline so that decision processes can be systematically improved by deep learning within a unified framework?

This leads to our research questions.

1.4 Research Questions

Research Question 1 (RQ1): What are good encodings of symbolic decision processes as training tasks, so that deep learning is effectively able to learn decision heuristics?

The inputs to the training task are symbolic expressions, but they can incorporate various types of information. For example, in clause selection for saturation-based ATPs, the input may consist solely of the clauses in the passive set or include all clauses from both the passive and active sets. Additionally, the input can be enriched with metadata, such as the age of each clause.

The structure of the training task depends on the output format. If the goal is to select a single clause from a set, we have two approaches. One option is to treat each clause individually by performing binary classification or regression tasks multiple times to assign scores and then rank the clauses accordingly. Alternatively, we can consider all clauses simultaneously and perform a single multi-class classification.

Another aspect of deciding what is the best encoding of a decision process in symbolic methods into training tasks is where to collect the training data. Typically, training data is generated from the traces of successfully solved problems. However, a single problem can have multiple solution traces within the same solver, and different solvers may produce distinct traces. The source and quality of the training data significantly affect the performance of the trained model.

Research Question 2 (RQ2): What is the most effective format for representing formulas in deep learning?

Formulas are typically stored in text-based formats, such as SMT-LIB and TPTP. However, most solvers first transform these formulas into structural representations, such as Abstract Syntax Trees (ASTs) or Directed Acyclic Graphs (DAGs), to better capture the relationships between elements. This transformation aligns with the core principles of symbolic methods, which operate on and manipulate symbols while preserving logical or algebraic correctness. Consequently, when applying deep learning to extract features from formulas, the prevailing approach is to represent them as graphs.

Despite this, the optimal way to structure formulas as graphs for deep learning models to effectively extract features remains an open question, as different representations impact both efficiency and learning capabilities. For example, consider the formula $(x \rightarrow y) \ \wedge \ (x \rightarrow z)$. One question is how to represent the identical symbol x. Merging occurrences of x in a DAG can reduce redundancy but may obscure some structural details. Alternatively, introducing a hyperedge to explicitly represent that one element implies multiple others can better capture multi-variable relationships, but this approach increases processing complexity.

Research Question 3 (RQ3): Which deep learning technique is best suited for feature extraction from formulas?

With recent breakthroughs in natural language processing, we cannot rule out the possibility that large language models may capture the semantic features of symbolic expressions more accurately through textual representations. However, as previously discussed, modern solvers primarily rely on operating on the relationships between symbols. Therefore, this thesis focuses on the case where symbolic expressions are represented as graph-based inputs.

Graph neural networks (GNNs) are specialized neural networks designed to process data structured as graphs. They are good at capturing complex relationships between entities represented as nodes and edges. For graph-based inputs, GNNs serve as the most effective feature extractors. There are various types of GNNs, such as Graph Convolutional Network (GCN), Graph Isomorphism Network (GIN), and Graph Attention Network (GAT). The choice of an appropriate GNN architecture as a feature extractor depends on the structural characteristics of the input graph. For instance, GCN is computationally efficient for homophilous graphs (where connected nodes have similar features) due to its simple aggregation mechanism, that averages neighboring node features. In contrast, GAT is more suitable for heterophilous graphs (where neighboring nodes have distinct features) as it learns adaptive weights for neighbors through an attention mechanism, allowing it to capture complex relationships.

Research Question 4 (RQ4): What are the methods for integrating the trained model into algorithms?

Under the assumption that prediction data and training data follow similar distributions, a condition typically achieved through large-scale training, external models may generate more effective heuristics than predefined rules, as they are explicitly optimized for the target data. However, practical implementations face inherent latency challenges. Querying external models involves computational overhead from converting symbolic representations into graph structures for model input and performing matrix operations during inference, making it significantly slower than predefined heuristic operations. As a result, an important consideration is how to leverage these models effectively while minimizing overhead to achieve the greatest possible performance improvement.

To address this, three principal strategies can be explored. First, randomness can be introduced by defining parameters that probabilistically determine when the model should be queried. Second, the model can be combined with existing heuristics, selectively invoking it based on real-time information. Third, the model can be queried only once at the beginning, as the initial search direction often has the greatest impact on the final outcome. These strategies help balance efficiency and effectiveness, ensuring that model-based heuristics improve performance without introducing excessive computational overhead.

1.5 Learning-Based Framework

Figure 1.1 provides an overview of the learning-based framework. The input consists of a set of problems used as training data, and the output is a deep learning model incorporating GNNs. Ultimately, the trained model is employed to either replace or collaborate with a solver's decision-making process, guiding the solving procedure for unseen problems.

The process begins by using one or more solvers to solve the problems in the training dataset. For each solved problem, one or multiple traces lead to the solution. Training data is then constructed from these traces, where symbolic expressions are encoded as graphs, and labels are assigned based on the most efficient trace—typically the one requiring the fewest iterations to reach a solution.

Once the training data is prepared, we train the model. The end-to-end model generally consists of two components: (1) a feature extractor based on GNNs, and (2) an approximator, which performs classification or regression tasks to generate the final decision.

When encountering new, unseen problems, the solver encodes the current states or traces as a graph representation and queries the trained model to guide decision-making. This process continues iteratively until a solution is found or a timeout occurs.

Although a problem-solving trace may contain extensive information, not all of it is necessary for learning. In other words, the trained model does not need to fully control the guiding process. For example, in clause selection for ATPs, rather than dictating the best clause to choose, the model can complement existing heuristics, such as those based on clause age, by providing additional information. For instance, the model can numerically characterize the "shape" of each clause, assigning it a weight that influences the selection process. This way, the model acts as an auxiliary source of information rather than directly controlling the selection.

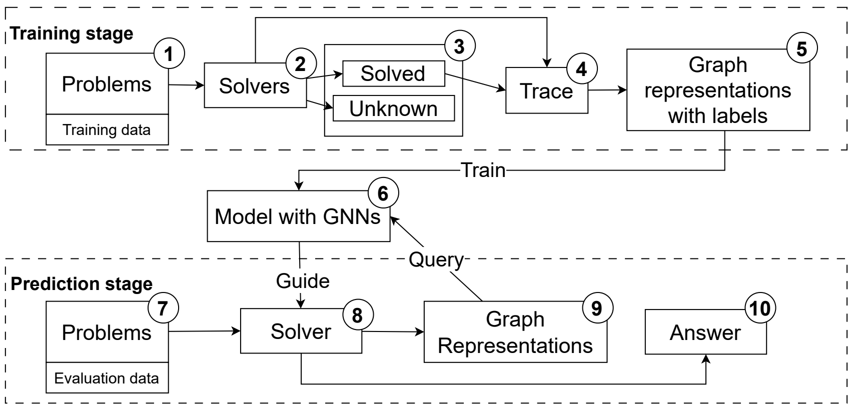


Figure 1.1. Workflow diagram for the learning-based framework.

Instances of the Framework. We instantiate the framework in two specific scenarios:

1. The Extended CHC Solver Eldarica [29]: Eldarica uses a Counterexample-Guided Abstraction Refinement (CEGAR)-based algorithm [38]. It incorporates various decision processes and provides a rich environment of traces for exploring how to adapt our framework. For instance, it solves CHCs by constructing an Abstract Reachability Graph (ARG). At each iteration, a clause must be selected for expansion in the ARG, allowing us to guide the clause selection. Moreover, during the CEGAR process, Craig interpolation generates predicates that represent system abstractions. This process relies on heuristics to focus on the most relevant parts of a proof. The selection of generated predicates that effectively capture essential system properties is also crucial.
2. The Word Equation Solver DragonLi [39]: In the context of SMT solvers dealing with string theory, solving word equations is one of the fundamental challenges. We have developed a calculus based on Nielsen transformations [40] to solve word equations. This approach recursively applies a set of inference rules to simplify and split the equations until an trivial solution is reached. At each iteration, decisions regarding which equation to branch on and which branch to explore first significantly impact performance. We chose this scenario because deep learning has not yet been applied to word equation solving. Moreover, the word equation problem is known to be NP-hard [41], so it heavily relies on heuristics, making it a suitable testbed for our framework.

1.6 Thesis Outline

The rest of this dissertation is organized as follows: Chapter 2 provides the necessary background concepts essential for understanding our framework. In Chapter 3, we delve into the details of our framework by addressing the four research questions. Chapter 4 summarizes the contributions of both published and under-reviewed papers related to this work. Finally, Chapter 5 concludes our research and proposes several future research directions in this domain.

2. Background

In this chapter, we introduce key concepts, including constrained Horn clauses, word equations, minimal unsatisfiable subsets, and deep neural networks. For each topic, we first provide the necessary background, then present its formal definition, and finally explain its relevance to this dissertation.

2.1 Constrained Horn Clauses

Classical Horn Clauses were introduced by mathematician Alfred Horn in the 1950s [42]. They are restricted to at most one positive literal and became foundational for logic programming languages like Prolog [43]. Constrained Horn Clauses (CHCs) extend classical Horn Clauses by incorporating constraints from various domains, such as integers and real numbers. CHCs provide a natural way to express the properties of programs, especially those involving loops and recursion, making them highly useful in program verification.

2.1.1 Definitions

To formally define CHCs, we assume a fixed signature $\Lambda = (\mathcal{S}, \mathcal{R}, \mathcal{F}, \mathcal{X})$ and a unique structure $\mathcal{M}$ over Λ as the background theory where $\mathcal{S}, \mathcal{R}, \mathcal{F}, \mathcal{X}$ are sorts, relations, functions, and variables, respectively, and $\mathcal{M}$ is a fixed interpretation of the symbols in the signature Λ. In this dissertation, we restrict the background theory to Linear Integer Arithmetic (LIA), following the SMT-LIB standard [44]. Additionally, we assume a set of relation symbols $\mathcal{R}_c$ disjoint from $\mathcal{R}$, which will be used to define the head and body of clauses. A *closed formula* is a logical formula that contains no free variables. A *constraint formula* is a special type of formula that belongs to the fixed background theory.

Definition 1 (Constrained Horn Clauses) *Given signature Λ and the relation symbol set $\mathcal{R}_c$, a Constrained Horn Clause (CHC) is a closed formula in the form*

$$\forall \bar{x}.\ H \leftarrow B_1 \wedge B_2 \wedge \cdots \wedge B_n \tag{2.1}$$

where $\bar{x}$ is a vector of variables; H is either false or an atomic formula and $B_1, B_2, \ldots, B_n$ are atomic formulas or constraint formulas over Λ. An atomic formula is a formula in form $p(t_1, \ldots, t_n)$ where $p \in \mathcal{R}_C$. We call H the head *and $B_1, B_2, \ldots, B_n$ the* body *of the clause, respectively.*

For simplicity, we often omit the quantifiers $\forall \bar{x}$ when writing clauses. A CHC with an empty body ($n = 0$) is called a *fact*. A CHC is *linear* if its body contains at most one atom; otherwise, it is *non-linear*.

We use the Fibonacci function as an example to illustrate how CHCs encode program logic. The logic of the Fibonacci function can be written as

$$\begin{aligned}
&\mathrm{fib}(0) = 0\\
&\mathrm{fib}(1) = 1\\
&\mathrm{fib}(n) = \mathrm{fib}(n-1) + \mathrm{fib}(n-2), \quad \text{for } n > 1
\end{aligned}$$

Its CHC encoding can be expressed as

$$\begin{aligned}
&F(0,0) \leftarrow \mathit{true}\\
&F(1,1) \leftarrow \mathit{true}\\
&F(x,y) \leftarrow x > 1 \land x_1 = x - 1 \land x_2 = x - 2\\
&\qquad\qquad \land F(x_1,y_1) \land F(x_2,y_2) \land y = y_1 + y_2
\end{aligned}$$

where the predicate $F(x,y)$ indicates that y is the Fibonacci number corresponding to index x.

To encode assertions such as "the Fibonacci function does not return negative numbers", we introduce another predicate, *false*, which represents an assertion violation. The CHCs for the assertion are:

$$\mathit{false} \leftarrow F(x,y) \land y < 0$$

This clause says: "There are no negative Fibonacci numbers".

Solving CHCs means to search for interpretations of the relation symbols $\mathcal{R}_C$ that satisfy the CHCs, assuming that all background symbols from Λ are interpreted by the fixed structure $\mathcal{M}$:

Definition 2 (Satisfiability of CHCs) *A Constrained Horn Clause (CHC) h is* satisfiable *if there exists a structure $\mathcal{M}_C = (\mathcal{U}, I_C)$ that provides an interpretation for the extended signature $\Lambda_C = (\mathcal{S}, \mathcal{R} \cup \mathcal{R}_C, \mathcal{F}, \mathcal{X})$ such that:*

1. *The interpretation I_C* agrees *with the original interpretation I on the symbols in the signature Λ.*
2. *The structure $\mathcal{M}_C$* satisfies *the CHC h, meaning that the assigned meanings of functions, relations, and variables make the clause hold true.*

A set $\mathcal{C}$ of CHCs is satisfiable *if there is an extended structure $\mathcal{M}_C$ simultaneously satisfying all clauses in $\mathcal{C}$.*

SAT Example: Consider a model:

$$F(x,y) \equiv x \geq 0 \ \land \ y \geq 0$$

Replacing F with this formula will make all clauses valid which means the system is **satisfiable (SAT)**.

2.1.2 Solving CHCs

A *CHC system* consists of a set of CHCs. Techniques for solving CHC systems include symbolic execution, abstract interpretation, and Counterexample-Guided Abstraction Refinement (CEGAR) [38], among others. These techniques are often used in combination rather than in isolation. Modern CHC solvers integrate multiple approaches to enhance efficiency and precision. For example, the CHC solver Eldarica [29] contains both CEGAR- and symbolic execution-based algorithms.

The CEGAR-based algorithm uses predicates to represent system abstractions. Initially, the abstraction is coarse, typically it is generated by a single trivial predicate such as *true* or no predicate. The solver checks the satisfiability of the clauses under this abstraction. If it finds a counterexample trace leading to an undesirable state (e.g., a violation of the CHC system), it determines whether the trace is spurious. A spurious counterexample is one that violates the abstracted system but not the concrete system. If the counterexample is spurious, the solver refines the abstraction by introducing more precise predicates. This process repeats until either a real violation is identified or the abstraction becomes precise enough to prove the CHC system's satisfiability.

In the symbolic execution-based algorithm (Paper II), a clause c with no atomic formula in its body is selected from a CHC system $\mathcal{C}$. The atomic formula in the head of c is then replaced in the remaining clauses $\mathcal{C} \setminus c$ with the constraint formula in c's body, generating new clauses to update the system. This process, akin to resolution, repeats until either a contradiction is found or the CHC system is proven to hold.

2.1.3 Learning-Based Methods

In recent years, learning-based techniques have increasingly been introduced for solving CHCs. For example, Code2Inv [45] proposed an end-to-end learning framework for synthesizing loop invariants, which are crucial for solving CHCs [8]. Luo et al. [46] cast CHC solving as a symbolic classification task and solve it by learning partitions of generalized reachable states to derive CHC interpretations. ROPEY [47] trains a neural network to capture co-occurrences of literals in lemmas over historical runs of the CHC solver SPACER [48] to guide the inductive generalization process.

This dissertation investigates various aspects of learning-based methods for CHCs, including graph representations of CHCs, the ability of Graph Neural Networks (GNNs) [19] to learn from these representations, the most suitable GNN architectures for capturing structural information of CHCs such as control and data flow, and the integration of learned heuristics into a CHC solver.

2.2 Word Equations

A *word equation* is an equation of the form $u = v$ where u and v are strings composed of variables and constants (fixed letters from a given alphabet). The *word equation problem* asks whether there exist assignments of concrete strings to variables such that the equation holds. If such assignments exist, the equation is *satisfiable*, and the assignments are returned; otherwise, it is *unsatisfiable*.

For instance, consider the word equation $Xab = YaZ$, where a and b are letters, and X, Y, and Z are variables ranging over strings of these letters. This equation is satisfiable because assigning $X = a$, $Y = a$, and $Z = b$ yields $aab = aab$.

The earliest study related to word equations was conducted by Axel Thue [49] who investigated properties of word transformations and substitutions, laying the foundation for what would later become combinatorics on words [50]. Makanin [51] proved that the satisfiability of word equations is decidable. Today, the exact complexity of the problem remains an open question, but it is known to be NP-hard and in PSPACE [41].

2.2.1 Definitions

To formally define a word equation system, we assume a finite, non-empty alphabet Σ and denote by Σ^* the set of all strings (or words) over Σ. We work with a set Γ of string variables, which range over words in Σ^*, and use ε to denote the empty string. String concatenation is represented by $\cdot$, though we often write uv as shorthand for $u \cdot v$ in examples.

The syntax of word equations used in this dissertation is defined as follows:

$$\begin{aligned} &\text{Formulae } \phi ::= \mathit{true} \mid e \wedge \phi, && \text{Words } w ::= \varepsilon \mid t \cdot w, \\ &\text{Equations } e ::= w = w, && \text{Terms } t ::= X \mid c \end{aligned}$$

where $X \in \Gamma$ represents variables and $c \in \Sigma$ represents constants. In this dissertation, a word equation system refers to a set of word equations to be solved simultaneously.

Definition 3 (Satisfiability of a Word Equation) *A word equation e 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 $\pi(X)$, the equation e holds.*

Definition 4 (Satisfiability of a Word Equation System) *A word equation system ϕ is* satisfiable *if all equations in ϕ are satisfiable.*

Definition 5 (Linearity of a Word Equation) *A word equation is called* linear *if each variable occurs at most once. Otherwise, it is* non-linear.

Note that this definition of linearity applies only to single-word equations. In a word equation system with multiple equations, a variable may still appear multiple times across different equations, even if each individual equation is linear.

2.2.2 Solving Word Equations

One of the earliest insights into solving word equations comes from Levi's Lemma [40] (also known as the Nielsen transformation in group theory) [52]. This combinatorial lemma states that if a concatenation of two strings equals another concatenation (if $u \cdot v = x \cdot y$), then u is a prefix of x, or vice versa. In practical terms, it implies that for any equation $U = V$ (with U and V sequences of constants and variables), one can compare the prefixes of U and V to systematically break the problem into cases:

1. If both sides start with different constant letters, the equation is unsatisfiable.
2. If one side starts with a constant letter a and the other with a variable X, then in any solution X must begin with a or be equal to ε. We can assign $X = aX'$ (for a fresh variable X') or $X = \varepsilon$ and reduce the equation accordingly.
3. If both sides start with variables (say X and Y), there are two sub-cases by Levi's lemma: either X's substitution is a prefix of Y's, or vice versa. For example, if the equation is $X \cdot \alpha = Y \cdot \beta$ (where α, β are the remaining parts), then either we set $Y = XY'$ (meaning X and Y start with the same string so we shorten Y) or $X = YX'$. Each case yields a simpler equation to solve (with one variable occurrence effectively shortened or eliminated).

Using these case distinctions, one can perform a backtracking search for a solution. The search builds a tree of subproblems, where each node is a simpler word equation derived by one of the above prefix cases. However, ensuring termination in the general case is non-trivial. Naive backtracking can loop or expand infinitely because variables can substitute arbitrarily long strings.

The first general decision procedure for word equations was given by Makanin [51]. At a high level, it repeatedly applies transformations similar to the prefix-case analysis above, but in a controlled way that avoids infinite loops.

Another approach for solving word equations is compression-based algorithms pioneered by Plandowski [41]. The key idea is to represent long intermediate strings implicitly to avoid exponential blow-up. Instead of explicitly constructing potentially huge solution words, the algorithm works on a compressed representation. Artur [53] further refined the recompression approach based on local modification of variables and iterative replacement of pairs of letters appearing in the equation with a "fresh" letter.

In practice, modern string constraint solvers and SMT (Satisfiability Modulo Theories) solvers implement their own methods to deal with word equations, often drawing on the theoretical insights above with additional techniques such as length constraints and automata to prune the search space [54, 55]. Many leading solvers such as Z3 [56] and cvc5 [57] are incomplete for proving the unsatisfiability of word equations.

2.2.3 Learning-Based Methods

SMT solvers like Z3 [56] and cvc5 [57] include dedicated procedures for the theory of strings, which handle constraints over string variables (e.g. word equations, substring operations, regex membership). Some of them use deep learning methods to guide their general solving process instead of optimizing a particular field of problems such as word equations. For instance, Piepenbrock et al. [58] use a GNN to guide the selection of quantified formulas and term instantiations during solving. At each step, the network scores all candidate instantiations, replacing or augmenting cvc5's built-in heuristics. This was trained on proof logs so that the network learns instantiation choices that tend to lead to proof. While not specific to strings, it shows deep learning integration in an SMT solver's workflow.

This dissertation presents the first application of deep learning to guide word equation solving. We develop a solver, DragonLi, that integrates a GNN into the solving process. Built upon the classic Nielsen transformation, it leverages a GNN to predict the optimal branch to explore first. Each word equation is encoded as a graph, and the GNN models branch selection as a multi-class classification problem.

2.3 Minimal Unsatisfiable Subsets

A Minimal Unsatisfiable Subset (MUS) is a minimal set of constraints (or clauses) from an unsatisfiable formula that is itself unsatisfiable, but becomes satisfiable if any constraint from the set is removed. MUS plays a crucial role in several domains, particularly in constraint solving and formal verification [59]. For instance, in MaxSAT solvers, finding an MUS helps determine the smallest subset of constraints responsible for infeasibility, guiding optimization techniques to resolve conflicts. In software verification, it helps pinpoint inconsistencies in system properties.

Definition 6 (Minimal Unsatisfiable Subset (MUS)) *Let ϕ be a set of formulas such that ϕ is* ***unsatisfiable****. A subset $\phi' \subseteq \phi$ is called a Minimal Unsatisfiable Subset (MUS) if it satisfies the following conditions:*

- ***Unsatisfiability:*** *ϕ' is unsatisfiable.*
- ***Minimality:*** *For every proper subset $\phi'' \subsetneq \phi'$, ϕ'' is satisfiable, i.e.,*

$$\forall \phi'' \subsetneq \phi', \quad \phi'' \textit{ is satisfiable.}$$

An unsatisfiable formula may have multiple distinct MUSes.

In this dissertation, we study the MUSes of CHCs and word equations. Identifying MUSes in both CHC and word equation systems allows solvers to focus on critical constraints and eliminating redundant computations. To train a model that learns to recognize MUSes, we incorporate *single*, *union*, and *intersection* of MUSes as training data.

Formally, let ϕ be a finite set of constraints (such as CHCs or conjunctive word equations), and assume that ϕ is **unsatisfiable**.

Definition 7 (Single MUS) *A **Single MUS** of ϕ is any minimal subset $\phi' \subseteq \phi$ such that:*

- *ϕ' is unsatisfiable.*
- *Any strict subset $\phi'' \subsetneq \phi'$ is satisfiable.*

Definition 8 (Union MUS) *Let $\mathcal{M} = \{\phi'_1, \phi'_2, \ldots, \phi'_k\}$ be the collection of all MUSes of ϕ. The **Union MUS** is defined as:*

$$UnionMUS(\mathcal{M}) = \bigcup_{i=1}^{k} \phi'_i.$$

This union represents the combined constraints of multiple MUSes, which may not be an MUS.

Definition 9 (Intersection MUS) *Let $\mathcal{M} = \{\phi'_1, \phi'_2, \ldots, \phi'_k\}$ be the collection of all MUSes of ϕ. The **Intersection MUS** is defined as:*

$$IntersectionMUS(\mathcal{M}) = \bigcap_{i=1}^{k} \phi'_i.$$

This intersection captures the core constraints that appear in all MUSes, ensuring that it remains an unsatisfiable set. However, it may not be an MUS.

2.4 Deep Neural Networks

To introduce GNNs, we first need to recall the main ideas of its basic building blocks: Multi-Layer Perceptrons (MLPs) [60], and the backpropagation mechanism, which is the core concept of deep learning.

A MLP is a universal function approximator consisting of multiple layers of interconnected neurons. A *neuron* applies a linear transformation to an input vector by multiplying each element by its corresponding weight and computing the weighted sum of the inputs. A bias term is then added to this sum, and the result is passed through an *activation function* to introduce non-linearity. The output of each layer is either propagated to the next layer or used as the MLP's output.

Formally, given an input vector $x \in \mathbb{R}^n$, the output of the first layer is computed as:

$$z^1 = act(W^1 x + b^1), \tag{2.2}$$

where $W^1 \in \mathbb{R}^{m \times n}$ is the weight matrix, $b^1 \in \mathbb{R}^m$ is the bias vector, and $act(\cdot)$ is an activation function such as ReLU [61] ($\text{ReLU}(x) = \max(0, x)$) or Sigmoid [62] ($\text{Sigmoid}(x) = 1/(1 + e^{-x})$).

This process is repeated for each hidden layer $l = 2, \ldots, L-1$, leading to the final output computed as:

$$\hat{y} = act(W^L z^{L-1} + b^L). \tag{2.3}$$

The *backpropagation* algorithm minimizes the *loss* (difference) between the expected output $\hat{y}$ and the actual output y using a loss function, such as squared error $D = |y - \hat{y}|^2$ and binary cross-entropy $D = -[y\ log(\hat{y}) + ((1-y)log(1-\hat{y}))]$. The MLP parameters (weights and biases) are updated by computing the gradients of the loss function with respect to these parameters.

Formally, given a training pair (x, y) with $x \in \mathbb{R}^n$ and $y \in \mathbb{R}$, we initialize an MLP with random weights $w \in W$ and biases $b \in B$. The forward propagation computes $\hat{y}$ using Equations 2.2 and 2.3. The weights and biases are updated using gradient descent:

$$W := W - \eta \frac{\partial D}{\partial w}, \quad B := B - \eta \frac{\partial D}{\partial b}, \tag{2.4}$$

where $\eta \in [0, 1]$ is the learning rate which determines the step size taken in the direction of the gradient during optimization. The D is a loss function. The trained MLP with optimized parameters serves as the final model.

With backpropagation, an MLP can approximate any real-valued continuous function to an arbitrary degree of accuracy, given sufficient neurons and appropriate weights [33].

2.4.1 Graph Neural Networks

A Graph Neural Network (GNN) [19] consists of Multi-Layer Perceptrons (MLPs) with a special structure, specifically designed to process graph-structured data with nodes and edges. A GNN takes a set of typed nodes and edges as input and produces a set of feature representations (vectors of real numbers) associated with the properties of the nodes. We refer to these as *node representations*.

The Message-Passing based GNN (MP-GNN) [63] is a type of GNN model. It utilizes an iterative message-passing algorithm in which each node in the graph aggregates messages from its neighboring nodes to update its own node representation. This mechanism assists in identifying the inner connections within substructures, such as terms and atoms, in graph-represented formulas.

Formally, let $G = (V, E)$ be a graph, where V is the set of nodes and E is the set of edges. Let x_v be the initial node representation (a vector of random real numbers) for node v in the graph, and let N_v be the set of neighbors of node v. An MP-GNN consists of a series of T message-passing steps. At each step t, every node v in the graph updates its node representation as follows:

$$h_v^t = \phi_t(\rho_t(\{h_u^{t-1} \mid u \in N_v\}), h_v^{t-1}), \tag{2.5}$$

where $h_v^t \in \mathbb{R}^n$ is the updated node representation for node v after t iterations. The initial node representation, h_v^0, is usually derived from the node type and given by 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. The $\rho_t : (\mathbb{R}^n)^{|N_v|} \rightarrow \mathbb{R}^n$ is a aggregation function with trainable parameters (e.g., a MLP followed by sum, min, or max) that aggregates the node representations of v's neighboring nodes at the t-th iteration. The $\phi_t : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is a function with trainable parameters (e.g., a MLP) that takes the aggregated node representation from ρ_t and the node representation of v in previous iteration as input, and outputs the updated node representation of v at the t-th iteration. MP-GNN assumes that a node can capture local structural information from neighbors t steps away by updating the node representation using aggregated representations of the neighbor nodes.

The final output of the MP-GNN could be the set of updated node representations for all nodes in the graph after T iterations. These node representations can be used for a variety of downstream tasks, such as node classification or graph classification.

The GNN definition in Equation 2.5 is abstract. It can be instantiated to different types of GNNs. For instance, the node representation of t-th layer for Message Passing Neural Networks (MPNNs) [63] is

$$h_v^t = U_t(h_v^{t-1}, \sum_{u \in N_v^r} M_t(h_v^{t-1}, h_u^{t-1}, e_{vu})), \tag{2.6}$$

where M_t and U_t are message updating function and vertex update function respectively. The e_{vu} is the edge feature for the edge between v and u. The $h_v^0 = x_v$ is the initial feature vector. The summation for all updated representation from M_t is the aggregation function.

The node updating function of MPNNs can be abstracted to

$$h_v^t = \phi^u(h_v^{t-1}, \rho(\{\phi^m(h_u^{t-1}, h_v^{t-1}, e_{vu}) | u \in N_v\})), \tag{2.7}$$

where ρ is the summation function for all neighbor of v, and ϕ^m and ϕ^u are message update function M_t and U_t respectively.

Graph Convolutional Networks (GCNs) [64] are also used in our experiments. Their node updating function is

$$h_v^t = \sigma\left(\sum_{u \in N_v \cup \{v\}} \frac{1}{c_{vu}} W^t h_u^{t-1}\right), \tag{2.8}$$

where W^t is a trainable weight matrix applied to transform node representations. The normalization factor c_{vu} is typically defined as $c_{vu} = \sqrt{|\mathcal{N}_v|} \cdot \sqrt{|\mathcal{N}_u|}$ to ensure numerical stability. The activation function σ is a ReLU function. The aggregation function sums the normalized representations of the neighbors and the node itself, followed by a transformation with W^t.

2.4.2 Learning Symbolic Expressions by GNNs

GNNs have recently been explored as a powerful tool for learning structural features from symbolic expressions in various solvers. By encoding logical formulas, proof states, or program structures into graph representations, GNNs help improve heuristics, decision-making, and efficiency in automated reasoning.

Graph-Q-SAT uses a GNN-based deep Q-learning approach to learn branching heuristics for a CDCL SAT solver, reducing search iterations and improving generalization to larger instances [65]. Bansal et al. [66] model higher-order logic formulas as graphs (instead of trees) and use GNNs to encode conjectures and premises for guiding proof search for a neural theorem prover (DeepHOL) [67]. Hůla et al. [68] apply a GNN to learn representations of SMT formulas (encoded as graphs) for per-instance solver portfolio selection.

In this dissertation we also use GNN to extract features for graph represented CHCs and word equations. In particular, we introduced a GNN called R-HyGNN to handle graph with hyper-edges. Its updating rule for node representation at time step t is defined as

$$h_v^t = \mathrm{ReLU}(\sum_{r \in R} \sum_{p \in P_r} W_{r,p}^t \cdot \|\{h_u^{t-1} \mid u \in \{N_v^{r,p} \cup v\}\}), \tag{2.9}$$

where $\|\{\cdot\}$ means concatenation of all elements in a set, $r \in R = \{r_i \mid i \in \mathbb{N}\}$ is the set of edge types (relations), $p \in P_r = \{p_j \mid j \in \mathbb{N}\}$ is the set of node positions under edge type r, $W_{r,p}^t$ denotes learnable parameters when the node is in the pth position with edge type r, and $N_v^{r,p}$ denotes the set of neighbor nodes of v in edge type r when v is in pth position.

3. GNN-Based Framework to Rank Formulas

Our four papers collectively developed and established a GNN-based framework for learning symbolic expressions to guide decision processes in symbolic methods. In this chapter, we discuss the applications of our framework presented in these four papers, addressing the four research questions introduced in Chapter 1.4. For each research question, we begin by summarizing the answers from related works, then we outline our approach. Finally, we present our findings based on our work.

3.1 Training Tasks for Decision Problems (RQ1)

We can broadly categorize training tasks for decision problems in symbolic methods into three types: classification-based, ranking-based, and sequential decision tasks.

The classification-based approach is the most straightforward way to model the decision process. It employs binary or multi-class classification to resolve decision problems that involve a fixed set of choices. For instance, Selsam et al. [69] model a GNN-based SAT solver as a binary classification task, where the model predicts whether a given query is satisfiable or unsatisfiable. Similarly, Zhang et al. [70] formulate solver selection as a multi-class classification problem, where the model selects the most suitable solver from a portfolio based on which one is expected to solve the given instance fastest.

The ranking-based approach is typically used when the input has a variable size. A common method is to decompose the problem into repeated binary classification tasks multiple times and then aggregate the results to establish a ranking. For example, DeepMath [71] and FormulaNet [72] provide early explorations of this approach by modeling premise selection in automated theorem proving as a series of binary classification tasks.

When the decision process involves a sequence of inter-dependent decisions with temporal dependencies, a sequential decision approach is required. For instance, TRAIL [73] trains a neural policy using self-play (similar to AlphaGo Zero [30]) to guide the theorem proving process.

3.1.1 Our Works Regarding to RQ1

Our work explores both classification-based and ranking-based tasks.

In Paper I, we investigate five training tasks (**Task 1–5**) based on the graph representations of CHCs.

- **Task 1** involves identifying different node types in the graph representation. In our CHC graph representation, node types correspond to the types of symbols in CHCs. For instance, this task determines whether a symbol in a CHC is an argument of an atomic formula. As a binary classification task, it evaluates whether the selected GNN, given the current graph representation, can effectively learn the basic syntax of CHCs.
- **Task 2** focuses on predicting the occurrence of an element in a CHC system. Unlike the previous task, this is a regression task that outputs numerical values rather than discrete classes. It demonstrates the potential of the GNN-based framework to provide direct numerical predictions.
- **Task 3** aims to determine whether an element in a CHC system is involved in a cycle by checking its presence in a strongly connected component within the graph representation. This binary classification task highlights the framework's ability to recognize fundamental graph structures such as cycles.
- **Task 4** predicts the existence of bounds for the arguments of atomic formulas. As a binary classification task, it showcases the framework's applicability to undecidable problems.
- **Task 5** predicts whether a clause belongs to a Minimal Unsatisfiable Subset (MUS). This binary classification task demonstrates the framework's potential in handling ranking-based tasks.

Paper I focuses on exploring various training tasks in the framework's training phase, highlighting its robustness across different tasks and laying the foundation for subsequent research.

Paper II extends **Task 5** from Paper I, specifically targeting the identification of MUSes in unsatisfiable CHC systems. The primary challenge in this extension lies in collecting high-quality training data. To address this, we gather training data from single MUSes, their unions, and their intersections.

Paper III integrates two models trained on classification tasks to guide branching selection in a Nielsen transformation-based algorithm for solving word equations. Since the branching selection problem has a fixed number of possible choices, framing it as a classification task is a natural choice.

In Paper IV, the objective is to rank word equations at each iteration of the Nielsen transformation-based algorithm. Instead of simply decomposing the ranking problem into multiple binary classification tasks, we propose two alternative approaches to encode the problem: 1. Enhancing the binary classification task by incorporating global information into each binary query. 2. Reformulating the problem as a multi-class classification task with a fixed number of possible outcomes. If the input size exceeds this fixed limit, we truncate it; if it is below the limit, we pad the input accordingly.

To ensure high-quality training data, we extract data from the shortest paths within the unsatisfiable subtree.

3.1.2 Answer to RQ1

Our empirical answer to **RQ1** is as follows:

- If the decision process has a well-defined target with a fixed number of choices, such as “At this point, branch left or right” or “This variable has or does not have a bound,” then encoding the problem as a classification-based task is the optimal choice.
- If the decision process must be modeled as a ranking task due to a variable number of inputs, it is crucial to incorporate global information into the input representation.

For training data collection, the key strategy is to first expand the sources of training data and then select the highest-quality data from the best source. When extracting training data from traces of problem-solving processes, generating multiple traces for the same problem and selecting the best trace—i.e., the one that minimizes time or the number of steps required to solve the problem—ensures higher-quality data. If the training data consists of standard intermediate elements, collecting them from multiple solvers can further improve data quality.

3.2 Representations of Formulas (RQ2)

The early research efforts that introduced deep learning into symbolic reasoning tasks primarily relied on handcrafted features or encoded logical formulas as sequences or even images. For instance, Loreggia et al. [74] approached SAT solver selection by converting the ASCII text of SAT and Constraint Satisfaction Problem (CSP) instances into fixed-size grayscale images, subsequently applying a Convolutional Neural Network (CNN) to predict the optimal solver. Similarly, Wang et al. [75] encoded Conjunctive Normal Form (CNF) formulas as compact 2D images and applied CNNs to process them. Grozea et al. [76] trained recurrent networks on handcrafted features to guide branch decisions in the DPLL algorithm [77] for the 3-SAT problem [78], aiming to reduce the search tree size.

However, representing formulas as text or flat feature vectors failed to capture their rich structural properties. It became evident that more structured representations were necessary, as naive sequence encodings struggled to effectively preserve the variable–clause relationships and term structures inherent in logical formulas. This realization prompted a shift toward graph-based representations, which naturally reflect the underlying structure of symbolic expressions and have gained prominence in recent years.

For example, in NeuroSAT [69], literals are represented as nodes, and edges connect literals based on clause co-occurrence. Graph-Q-SAT [65] extends this idea by representing both clauses and variables as nodes, with an undirected edge linking a variable node to a clause node if the variable appears in the clause. If the variable is negated in the clause, the edge stores this information as a feature.

Paliwal et al. [66] proposed a graph-based representation for Higher-Order Logic (HOL) formulas using bidirectional syntax trees, where subexpressions and terms are treated as nodes connected by bidirectional edges. Nodes corresponding to the same variable (even if bound by different quantifiers) are merged, while nodes and edges are annotated with type information relevant to the terms they represent. Additionally, special nodes are introduced to explicitly capture function applications.

3.2.1 Our Works Regarding to RQ2

In Papers I and II, we propose two graph representations for the CHC system.

One of these representations is the Control- and Data-Flow Hypergraph (CDHG), which models all symbols as nodes. The relationships between symbols are captured using various types of binary and hyper-edges. An example illustrating this encoding is presented in Figure 3.1. The core idea of representing the CHC system in this manner is to capture its control flow through implications within each clause and its data flow across the system. Finally, nodes representing the same symbol are merged.

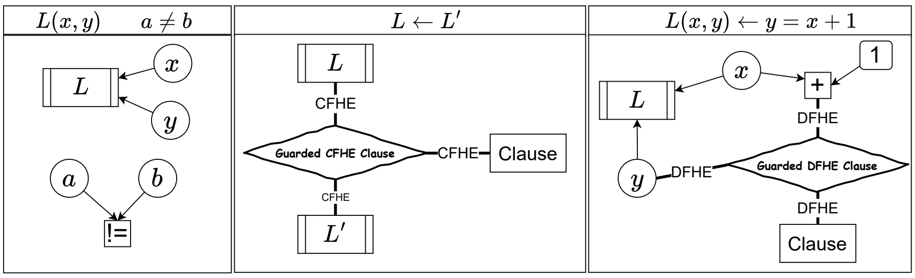


Figure 3.1. Graph components of CDHG. In the left figure, atomic formulas of the form $L(x,y)$ are encoded by representing their arguments and relation symbols as nodes, with binary edges connecting the arguments to the relation symbol to capture their relationships. Constraints such as $a \neq b$ are encoded using a syntax tree, where operators and elements are represented as nodes, connected by binary edges to reflect their hierarchical structure. In the middle figure, the control flow of the implication relation $L \leftarrow L'$ is encoded using a Control Flow Hyperedge (CFHE). The node labeled "clause" serves as an abstract symbol acting as a clause identifier in a CHC system. The right figure illustrates how data flow is represented in a CHC. The rule $L(x,y) \leftarrow y = x+1$ is encoded by introducing a Data Flow Hyperedge (DFHE), which connects the result of $x+1$ to y, thereby capturing the data dependency from $x+1$ to y. The DFHE also connects to the "clause" node to distinguish this assignment from others.

Another graph representation is the Constraint Graph (CG). The main idea is to divide each CHC into three layers, each representing a different aspect of the CHC. An example is shown in Figure 3.2. The *predicate layer* represents all atomic formulas in the CHC system. The *clause layer* encodes the implication, capturing the relationship between the body's atomic formulas and the head for each clause. The *constraint layer* represents the syntax tree of the constraints appearing in the body. These three layers are connected through additional edges, forming the final CG.

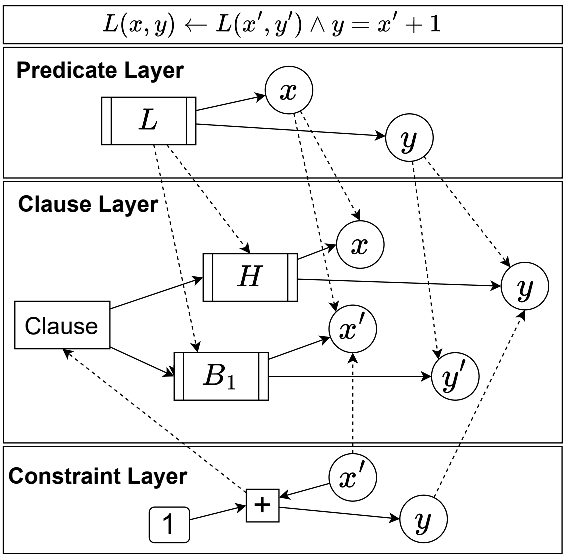


Figure 3.2. Graph representation of the CG for a CHC of the form $L(x,y) \leftarrow L(x',y') \wedge y = x' + 1$. The top figure represents the *predicate layer*, which encodes the atomic formulas in the CHC. Relation symbols and their arguments are represented as nodes, with binary edges capturing their relationships. The same relation symbols appearing in different atomic formulas are merged. The middle figure illustrates the *clause layer*, where a clause node serves as an identifier for the clause. The head and body atomic formulas, along with their arguments, are represented as nodes. The bottom figure depicts the *constraint layer*, which encodes the constraint in the body as a syntax tree. The same elements in the three layers are connected by additional edges to form a complete CG.

For both CDHG and CG, we illustrate only their core concepts by presenting their main components. The complete formal definitions can be found in Paper I.

In Paper III and IV, we propose six graph representations for word equations, all of which are built upon the syntax tree of a word equation. In these representations, all symbols are represented as nodes, while operators, such as "=" and concatenation, are represented as edges.

For example, consider the word equation $aXa = bYX$, where a and b are letters, and X and Y are variables. The construction of the syntax tree begins by placing the "=" symbol as the root. The nodes a and b are then connected as the left and right children, respectively. The following elements X and Y are subsequently connected to a and b as child nodes.

In Figure 3.3, we illustrate the syntax tree of word equations along with two variations. One variation extends the syntax tree by adding extra nodes to indicate both the type of each symbol and occurrences of the same symbol. The other variation introduces special nodes, $0_\Sigma, 1_\Sigma, 0_\Gamma, 1_\Gamma$, to represent the occurrences of each letter and variable in binary. The remaining four graph representations are derived as variations of these two graphs.

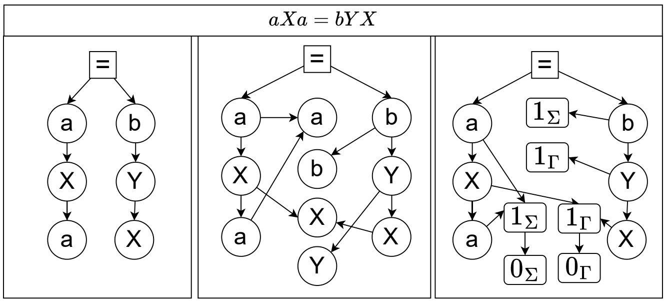


Figure 3.3. Three graph representations of the word equation $aXa = bYX$ are shown. The left figure depicts the syntax tree of the word equation. The middle figure presents a variation of the syntax tree that extends it by adding extra nodes to indicate both the type of each symbol and occurrences of the same symbol. The right figure introduces another variation, which incorporates four special nodes, $0_\Sigma, 1_\Sigma, 0_\Gamma, 1_\Gamma$, to represent the occurrences of each letter and variable in binary.

3.2.2 Answer to RQ2

According to the experimental results from our ablation studies comparing these graph representations, our answer to **RQ2** is as follows:

- The graph representation must include all explicit syntactic elements of the formula.
- A heterogeneous graph with hyperedges and merged identical nodes can represent structural information more efficiently and encode information in a compact manner, resulting in better performance. However, designing such a representation requires greater domain-specific expertise.
- The graph representation should include only the information necessary for the task. Redundant information not only increases computational overhead but can also mislead the training process, ultimately degrading model performance. For instance, including element occurrence counts in the graph representation of a word equation is unnecessary for the branch selection task in a Nielsen transformation-based algorithm. The decision on which transformation to apply depends on the structural properties

of the equation (e.g., variable dependencies, and cyclic dependencies), rather than the frequency of element occurrences.

3.3 GNN Selection (RQ3)

In the field of symbolic reasoning, Graph Neural Networks (GNNs) have emerged as a powerful tool for learning structural features of logical symbolic expressions from their graph representations. Simple architectures, such as Graph Convolutional Networks (GCN) [64], laid the foundation by efficiently encoding logical structures. More advanced models, including Graph Isomorphism Network (GIN) [79], Graph Attention Networks (GAT) [80], and Relational Graph Convolutional Networks (R-GCN) [81], introduce greater expressiveness, enabling the differentiation of complex logical patterns and heterogeneous relationships.

GCNs are among the most widely used GNN architectures and are often chosen as a baseline or starting point for symbolic reasoning tasks due to their stable performance and low computational complexity. For example, in early work on premise selection for theorem proving, Wang et al. [72] proposed FormulaNet, a GCN-based deep graph embedding of logical formulas. Its simplicity allowed the focus to remain on representing formula structures as graphs, where nodes represent symbols and clauses, without requiring extensive model tuning. Similarly, in an SMT solver scheduling approach, Hula et al. [68] initially experimented with more advanced GNNs but found that a basic GCN achieved comparable accuracy.

The GIN is designed to approximate the Weisfeiler–Lehman graph isomorphism test. It employs an injective aggregation mechanism, where neighbor features are summed with a learnable weight and then passed through a nonlinear transformation. This ensures that different multisets of neighbors produce distinct node embeddings. Essentially, GIN's sum-and-MLP update enables it to capture fine-grained structural differences that simpler averaging methods, such as those used in GCNs, might fail to distinguish. In other words, GIN can differentiate nearly identical graph structures. Li et al. [82] demonstrated that on a specially constructed SAT dataset where formulas differ by only one literal, most models struggled, but GIN achieved the top performance for GNN-based SAT solvers.

The GAT introduces an attention mechanism, allowing the model to assign data-driven weights during message passing. This enables the network to focus on the most relevant parts of the graph. In complex graph representations of logical formulas, such as those with many clauses, not all neighboring nodes contribute equally to a given inference. Researchers have leveraged GAT to allow the model to learn which symbols or clauses are most influential. Selsam et al. [26] demonstrated that incorporating an attention layer (a variant known as GATv2) into their models for guiding SAT solvers, such as Z3 [56] and Glucose [83], significantly improved the accuracy of satisfiability predictions.

The R-GCN extends GCNs to graphs with multiple edge types or relations. Symbolic problems often involve heterogeneous graphs. For instance, first-order logic formulas can be represented as heterogeneous structures consisting of multiple types of nodes to represent terms, predicates, and clauses, as well as edges to capture their relationships. R-GCNs are particularly well-suited for leveraging this structural richness. By assigning separate trainable parameters (weights) to each logical relation (e.g., "parent clause" vs. "sibling argument"), R-GCNs can preserve semantic distinctions that a homogeneous model might blur. For instance, in theorem proving, Olsak et al. [84] utilized a relational clause-term graph, where edges such as "literal in clause" were distinguished from "term in literal," enabling a name-invariant embedding of formulas.

3.3.1 Our Works Regarding to RQ3

In Papers I and II, we developed a Relational Hypergraph Neural Network (R-HyGNN), which extends R-GCN to handle hyperedges and typed relations. This model efficiently captures n-ary relations in the graph representation of the CHC system. Each edge type (e.g., data-flow edges and syntactic AST edges) is assigned its own trainable parameters (weights). This design was motivated by the need to learn both program semantics (data and control flow) and syntax. R-HyGNN achieved over 90% accuracy in predicting which clauses appear in counterexamples, significantly aiding in the identification of unsolvable verification conditions.

In Papers III and IV, we experimented with GCN, GIN, and GAT to extract structural information from the graph representation of word equations to guide a Nielsen transformation-based algorithm. Among these models, GCN achieved the best balance between performance and computational overhead. Note that the experimental results for GIN and GAT are not included in the published paper. However, their existence helps us to understand which model is best suited for the particular tasks discussed in our paper.

3.3.2 Answer to RQ3

Based on both the literature review and our experimental results, the answer to **RQ3** is as follows:

- GCN should always be considered and serve as a baseline.
- If the graph of the symbolic expression is sparse or too large, GAT usually achieves the best performance.
- If the graphs contain similar substructures, GIN is typically the best choice.
- For graphs with complex structures, R-HyGNN is the preferred option, as it can effectively handle both heterogeneous structures and hyperedges.

3.4 Integrating Strategies (RQ4)

Traditional solver heuristics, such as Variable State Independent Decaying Sum (VSIDS) [24] in SAT solvers, are extremely fast since they rely on simple arithmetic operations or lookups. In contrast, neural model inference involves substantial computation (e.g., a forward pass through a GNN or Transformer [85]), which can slow down the solver if invoked too frequently.

One approach to mitigate this overhead is to reduce the frequency of model calls during solving, using the model only at selective points or shifting its use to an offline stage. For example, NeuroBack [86] follows this strategy by performing offline model inference. It runs the neural model before the solving process to gather guidance, which is then utilized during the solver's search without requiring further expensive model calls.

To prevent solver slowdowns while a model runs, researchers sometimes deploy the model in a separate process or server, allowing the solver and model to operate in parallel. In this setup, the solver sends queries (such as the current state or extracted features) to a model server—potentially running on a GPU or as a separate thread/process—and receives the model's decisions asynchronously. For instance, Karel [87] integrates a GNN into the prover's workflow via a client-server architecture. During the proof search, the prover identifies candidate instantiations and sends their representations to the server. The server processes these representations through the GNN and returns scores to the prover.

A complementary strategy is to optimize the model itself for faster execution. This can involve using a smaller neural network, applying model distillation [88], or even converting the learned policy into a symbolic form. A recent example is the Symb4CO framework [89], which employs a large neural network offline to discover a compact symbolic formula that can replace the neural model at runtime.

3.4.1 Our Works Regarding to RQ4

In Papers I and II, we employ a "one-shot" prediction strategy before invoking the solver. This approach eliminates the overhead caused by frequent model calls. The trade-off is that the guidance remains static (computed once) and cannot adapt mid-search. However, in many cases, this "one-shot" advice is sufficient to improve performance without requiring continuous model inference.

In Papers III and IV, we optimize the overhead of calling GNNs at each iteration by caching neural model outputs and intermediate computations. This allows the solver to reuse previously computed results instead of recomputing from scratch on similar states. Specifically, we construct a hash table to store the embeddings of word equations. At each branch point, only equations with modified variables need to be updated, enabling most word equation embeddings to be reused.

For both “one-shot” and continuous model calls, we experimented with three strategies:

- **Model-only guidance:** In this approach, the solver’s decisions are entirely determined by the model’s predictions.
- **Combining the model with randomness:** At each prediction point, we alternately use the model’s prediction and a random choice. This stochastic element helps escape local minima induced by the model’s deterministic guidance.
- **Combining the model with existing heuristics:** For both “one-shot” and continuous prediction, existing heuristics can be numerically integrated with the model’s predictions to balance predefined knowledge with learned distributions.

3.4.2 Answer to RQ4

Based on the experimental results of our research, the answer to **RQ4** is as follows:

- When the model’s input features can be updated incrementally or when solvers frequently encounter related subproblems, caching intermediate embeddings can significantly reduce the overhead of model calls.
- Introducing randomness can be beneficial, but the randomization rate needs careful tuning. If the model tends to cause the solver to get stuck in local minima, an appropriately adjusted level of randomness can help mitigate this issue.
- Numerically combining the model with predefined heuristics did not yield the best performance across all our experiments. We hypothesize that this is due to one of two reasons: (1) the learned knowledge and predefined heuristics may sometimes lead to conflicting behaviors, negatively impacting performance, or (2) the model may learn behaviors similar to the predefined heuristics, resulting in limited performance improvements.

4. Summary of Contributions

4.1 Paper I

Exploring Representation of Horn Clauses Using GNNs. Chencheng Liang, Philipp Rümmer, Marc Brockschmidt. *In Proceedings of 8th Workshop on Practical Aspects of Automated Reasoning (PAAR)*, 2022.

Summary

This work explores the application of deep learning techniques to learn program features for solving program verification problems, with a particular focus on Constrained Horn Clauses (CHCs). CHCs are a widely used intermediate representation in program verification, and many of the benchmarks in the CHC-COMP repository originate from real-world program verification tasks. In this study, we propose two novel graph representations for CHCs: the Constraint Graph (CG) and the Control- and Data-Flow Hypergraph (CDHG). These representations are designed to capture different aspects of program information: the CG emphasizes abstract program syntax, while the CDHG focuses on semantic information, such as control and data flow within the program.

To effectively process the CDHG, which involves hyperedges (edges that connect multiple nodes), we extend the Relational Graph Convolutional Network (R-GCN), a message-passing-based Graph Neural Network (GNN), into a new architecture called the Relational Hypergraph Neural Network (R-HyGNN). This extension enables the model to handle multiple types of edges, including hyperedges, and learn from the complex relational structures present in the CDHG.

We then train GNN models on five proxy tasks designed to systematically evaluate the model's ability to learn both syntactic and semantic information from CHCs. These tasks are ordered by increasing difficulty:

1. Argument Identification: Classify whether a node represents an argument of a relation symbol.
2. Relation Symbol Occurrence Count: Predict how many times a relation symbol appears in all clauses.
3. Relation Symbol Occurrence in Strongly Connected Components (SCCs): Determine if a relation symbol is part of a cycle in the graph (i.e., belongs to an SCC).
4. Existence of Argument Bounds: Predict whether an argument has a lower or upper bound, which is an undecidable property in general.

5. Clause Occurrence in Counterexamples: Predict whether a clause appears in a counter-example, which is closely related to the satisfiability of the CHCs.

The first three tasks focus on syntactic and structural properties of the CHCs, such as identifying elements of the formula, counting symbol occurrences, and recognizing graph-theoretic patterns such as SCCs. In contrast, the last two tasks require the model to learn semantic information that is crucial for solving program verification problems, such as predicting argument bounds and identifying clauses involved in counter-examples.

Finally, we evaluate the performance of our graph representations (CG and CDHG) combined with R-HyGNN on these tasks. The experimental results demonstrate that the framework is capable of extracting intricate semantic information from CHCs, achieving high accuracy on tasks that require understanding complex program features. Notably, the model shows promising potential to generate effective heuristics for program verification, particularly in guiding Horn clause solvers and other verification tools.

Author Contributions

The theoretical framework presented in this paper was developed collaboratively by Rümmer, Brockschmidt, and Liang, with equal contributions. The implementation was primarily carried out by Liang, with contributions from Rümmer and consultations with Brockschmidt. The evaluation of the framework was conducted by Liang. The paper was written by Liang and reviewed and corrected by Rümmer and Brockschmidt.

4.2 Paper II

Boosting Constrained Horn Solving by Unsat Core Learning. Parosh Aziz Abdulla, Chencheng Liang, Philipp Rümmer. *In Proceedings of 25th International Conference on Verification, Model Checking, and Abstract Interpretation (VMCAI)*, 2024.

Summary

In this work, we introduce a prototype of a GNN-based heuristic framework aimed at enhancing the decision-making processes of symbolic solvers. The framework comprises the following key steps:

1. **Modeling and Data Collection:** We identify the critical decision points in the solver where deep learning heuristics can be applied. This involves determining the semantic information required to guide these heuristics, specifying the source of training data, and defining the learning task (e.g., classification or regression).
2. **Graph Representation of Formulas:** To leverage GNNs, we encode logical formulas as graphs that capture the necessary syntactic and structural information, thereby reflecting the semantic properties aligned with our learning objectives.
3. **GNN Selection:** We select an appropriate GNN architecture to process these graph representations. The choice of architecture depends on the problem domain and the specific graph representation employed.
4. **Integration into the Solver:** We determine how to integrate the trained GNN model into the solver as a heuristic. This involves specifying how the model's predictions will influence the solver's decision-making process.

As a first experimental instance of this framework, we address program verification problems by encoding them as CHCs and solving them using the CHC solver `Eldarica`. Eldarica supports both Counterexample-Guided Abstraction Refinement (CEGAR) and Symbolic Execution (SymEx) approaches, representing two fundamental categories of symbolic solvers.

For this experimental instance, our approach is detailed as follows:

1. **Modeling and Data Collection:** Eldarica explores the state space of CHCs by constructing an Abstract Reachability Hypergraph (ARG), with solver performance heavily dependent on the order in which clauses are expanded. We train a GNN to predict the likelihood that a clause is part of a Minimal Unsatisfiable Subset (MUS), as identifying a MUS allows the solver to establish unsatisfiability without further expansion. We experiment with different variations of MUS-based training data, including single MUS, intersection of MUSes, and union of MUSes, to determine which variant offers the most effective guidance.
2. **Graph Representation of Formulas:** We adopt the CG and CDHG representations from Paper I, which have demonstrated strong performance

in capturing both the syntactic and semantic characteristics of CHCs, making them well-suited for our learning tasks.
3. **GNN Selection:** We employ the R-HyGNN architecture, which has shown excellent performance in handling the hypergraph structures of CHCs and aligns well with our chosen graph representations.
4. **Integration into the Solver:** We investigate eight distinct strategies for incorporating GNN predictions into the solver. These strategies fall into three broad categories: (i) *GNN-only*, where the solver relies solely on GNN predictions; (ii) *GNN + Existing Heuristics*, which combines GNN predictions with Eldarica's default heuristics to leverage both learned and handcrafted guidance; and (iii) *GNN + Randomness*, which introduces an element of randomness to balance exploration and exploitation, ensuring that the solver does not over-rely on the GNN predictions while maintaining a diverse search strategy.

The experimental results confirm the effectiveness of our framework, showing improvements in both the number of solved problems and the average solving time for the CEGAR and SymEx approaches. This work demonstrates the potential of integrating deep learning-based heuristics into symbolic solvers to enhance their decision-making processes, and our framework is flexible enough to be extended to other symbolic solvers beyond CHCs.

Author Contributions

The theoretical framework presented in this paper was developed collaboratively by Abdulla, Rümmer, and Liang, with equal contributions. The implementation was primarily carried out by Liang, with contributions and consultations from Rümmer. The evaluation of the framework was conducted by Liang. The paper was written by Liang and reviewed and corrected by Rümmer and Abdulla.

4.3 Paper III

Guiding Word Equation Solving Using Graph Neural Networks. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Julie Cailler, Chencheng Liang, Philipp Rümmer. *In Proceedings of 22nd International Symposium on Automated Technology for Verification and Analysis (ATVA)*, 2024.

Summary

In this paper, we demonstrate another application of our framework to symbolic methods, specifically focusing on word equation solving. To the best of our knowledge, no prior work before our paper has applied deep learning techniques to this problem. Our approach indicates that the proposed framework can be generalized to any symbolic method.

First, we introduce a calculus for word equations, which consists of a set of inference rules for branching and simplifying such equations. Next, we present an algorithm that recursively applies these rules until the solution to a word equation becomes trivial. This procedure produces a proof tree that leads to the final solution.

A key challenge arises in deciding which branch of an inference rule to explore first, as it significantly impacts performance. Choosing the wrong branch may increase the length of the word equation rather than simplifying it, potentially causing the algorithm to diverge. To address this issue, we adapt our framework to guide the branching process and improve overall performance:

1. **Modeling and Data Collection:** We formulate the branching decisions as a multi-class classification task. For each branching requirement in the inference rules, we determine which branch is optimal based on the current state of the word equation. We label a branch as “positive” if it leads to a valid solution with a minimal search sub-tree.
2. **Graph Representation of Formulas:** We propose five distinct graph representations for word equations. The simplest one is an abstract syntax tree rooted at the “=” symbol, connecting elements on the left and right sides sequentially. The most comprehensive representation includes additional nodes and edges to capture variables and letters, thereby preserving all necessary information about the word equation.
3. **GNN Selection:** Since our chosen graph representations do not contain hyper-edges, we employ Graph Convolutional Networks (GCNs) to learn relevant features.
4. **Integration into the Solver:** We investigate two primary strategies: using the GNN model alone or combining the GNN model with a random selection process.

Furthermore, we define three different backtracking strategies to manage the tree search. Experimental results show that, apart from the systematic search strategy, our framework remains robust under various search strategies. This outcome indicates that the GNN effectively captures semantic information, enabling it to consistently select good branches under diverse conditions.

We evaluate our framework on both artificially generated and real-world encoded benchmarks. Our experimental results demonstrate that our method outperforms leading solvers, such as Z3 and cvc5, on benchmarks consisting of single word equations while achieving competitive performance on conjunctive word equation benchmarks.

Note that we fixed an error in the calculus by adding an additional inference rule:

$$R_9 \frac{X \cdot u = X \cdot v \wedge \phi}{u = v \wedge \phi}$$

in the extended version [90] of the published work.

Author Contributions

The theoretical framework presented in this paper was developed collaboratively by Abdulla, Rümmer, Atig, and Liang, with equal contributions. The implementation was primarily carried out by Liang, with contributions from Rümmer and consultations from Cailler and Atig. The evaluation of the framework was conducted by Liang. The paper was written by Liang and reviewed and corrected by Rümmer, Cailler, Abdulla, and Atig.

4.4 Paper IV

When GNNs Met a Word Equations Solver: Learning to Rank Equations. Parosh Aziz Abdulla, Mohamed Faouzi Atig, Julie Cailler, Chencheng Liang, Philipp Rümmer. *Under Submission to 30th International Conference on Automated Deduction (CADE)*, 2025.

Summary

In this work, we extend our learning framework to handle ranking tasks. Unlike classification problems, ranking must accommodate variable input sizes and produce an ordered list of items. We instantiate this framework in the word equation solver presented in Paper IV.

The algorithm in Paper IV solves word equations by recursively applying a set of inference rules to branch and simplify the equation until a solution is apparent. We first enhance this algorithm for more efficient handling of conjunctive word equations and then adapt it to select, from a set of equations, the one to which inference rules should be applied. This strategy permits the selection of different starting points; in the context of conjunctive word equations, identifying an unsatisfiable equation early allows the solver to bypass further checks, thereby reducing the overall solving time.

To realize this, we train a GNN model to emulate an oracle that, at each step, predicts the word equation most likely to be unsatisfiable or that will lead quickly to an unsatisfiable branch. The model outputs a score for each word equation, and the equation with the highest score is processed first.

The four key components to adapt the framework are listed below:

1. **Modeling and Data Collection:** We propose three methods to adapt a classification model for ranking, i.e., to output a list of scores for a set of elements. Our training data is designed to enable the model to (i) detect unsatisfiable word equations and (ii) identify the equation that leads to the shortest solution path. Data is collected from two sources: MUSes obtained by solving problems that our algorithm could not resolve using leading solvers such as Z3 and cvc5, allowing the model to learn unsatisfiability patterns; and execution traces from our algorithm, from which we extract rankings corresponding to the shortest solution paths.
2. **Graph Representation of Formulas:** We introduce a novel graph representation for word equations that encodes variable and letter occurrences in conjunctive equations, thereby capturing the global structural information.
3. **GNN Selection:** We employ a GCN as our feature extractor to minimize extraneous influences from other components.
4. **Integration into the Solver:** We explore three integration strategies: using the GNN model exclusively at varying frequencies, combining the GNN model with a random ranking baseline, and controlling the GNN model via run-time parameters.

For satisfiable problems, the order of processing conjuncts does not affect performance since all conjuncts must be checked to verify satisfiability. Nevertheless, the model is designed not only to identify unsatisfiable word equations but also to learn which selection leads to the shortest solution path. Consequently, it proves beneficial even for satisfiable instances.

We evaluate our framework on two benchmark categories: *linear* and *non-linear* word equations. A linear word equation is one in which each variable appears only once, whereas a non-linear word equation features at least one variable appearing multiple times. Our framework outperforms all leading solvers on linear benchmarks and demonstrates competitive performance on non-linear benchmarks. However, in cases of high non-linearity (i.e., when a variable appears many times), the advantages of our algorithm diminish due to inherent limitations of the inference rules that cannot be mitigated by ranking.

Author Contributions

The theory presented in the paper was developed by Abdulla, Atig, Cailler, Liang, and Rümmer, the implementation was done by Liang and Rümmer, and the evaluation was done by Liang.

The theoretical framework presented in this paper was developed collaboratively by Abdulla, Rümmer, Atig, Cailler, and Liang, with equal contributions. The implementation was primarily carried out by Liang, with contributions from Rümmer and consultations from Cailler and Atig. The evaluation of the framework was conducted by Liang. The paper was written by Liang and reviewed and corrected by Rümmer, Cailler, Abdulla, and Atig.

5. Conclusions and Future Work

5.1 Conclusions

This dissertation explored the integration of deep learning with symbolic methods, focusing on leveraging Graph Neural Networks (GNNs) to enhance decision processes in symbolic solvers. The key contributions include:

1. Introducing a general learning-based framework to replace heuristic decision-making in symbolic methods with data-driven deep learning models.
2. Proposing novel graph representations for Constrained Horn Clauses (CHCs) and word equations to capture essential structural and semantic features for GNN models.
3. Investigating different training task formulations, including classification-based and ranking-based tasks, to effectively model decision processes.
4. Experimenting with various GNN architectures, such as Graph Convolutional Networks (GCNs) and Relational Hypergraph Neural Networks (R-HyGNNs), to determine their suitability for different symbolic reasoning tasks.
5. Implementing and evaluating the framework in CHC solvers and word equation solvers, demonstrating significant improvements in solver performance.
6. Exploring multiple strategies to integrate trained models into solvers while minimizing computational overhead, including caching mechanisms, hybrid heuristic approaches, and selective model querying.

The experimental results show that deep learning can effectively complement and, in some cases, surpass traditional heuristics in guiding symbolic methods. The proposed framework generalizes across different problem domains, making it a valuable tool for automating heuristic selection in solvers.

5.2 Future Work

While this dissertation establishes a solid foundation for integrating deep learning into symbolic methods, several open directions remain for future exploration.

For our word equation solver, potential improvements include introducing new inference rules to improve the handling of non-linear word equations and investigating how length constraints and regular expressions can be integrated into the solving process. Currently, branch selection and ranking are handled as separate heuristics. A promising direction is to develop a unified learning-based approach that shares embeddings for both decision processes.

Regarding deep learning models, incorporating attention mechanisms into R-HyGNN could help selectively focus on the most relevant parts of a formula, particularly in tasks where long-range dependencies play a crucial role. Another potential direction is exploring hybrid architectures that combine classical logic-based techniques with deep learning, leveraging the strengths of both approaches. Additionally, hierarchical reinforcement learning techniques could be investigated to coordinate decision-making at different levels.

Current learning-based heuristics rely on training from solver execution traces, meaning they learn heuristics based on past solving history. A novel direction is to develop heuristics that predict solver decisions by anticipating near-future solving trajectories. This could involve using sequence modeling techniques such as Long Short-Term Memory (LSTM) [91] or transformers to predict solver states over multiple steps, developing a lookahead mechanism that considers multiple solver actions before making a decision, or exploring contrastive learning approaches where models learn to differentiate between effective and ineffective solving trajectories, thereby improving heuristic selection.

In summary, this dissertation has demonstrated the potential of deep learning to enhance symbolic reasoning. With further refinements and extensions, learning-based heuristics could become a fundamental component of next-generation symbolic solvers, bridging the gap between deep learning and formal methods.

Summary in Swedish

Symboliska metoder är avgörande för formella resonemangsuppgifter såsom programverifiering, teorembevisning och lösning av villkorsproblem. Dessa metoder använder symboliska uttryck för att kompakt representera stora eller oändliga tillståndsrum, vilket gör dem mer effektiva än explicit uppräkning. Traditionella heuristiker som används i symboliska lösare är dock ofta manuellt utformade och har svårt att generalisera över olika domäner. Denna avhandling föreslår ett djupinlärningsbaserat ramverk för att förbättra heuristiskt beslutsfattande i symboliska lösare. med hjälp av graf-neuronnät (GNNs) möjliggör ramverket adaptiv, datadriven vägledning genom att lära sig strukturella mönster från symboliska uttryck. Beslutsproblem inom symboliska metoder formuleras som klassificerings- eller rangordningsuppgifter, vilket möjliggör ett systematiskt tillvägagångssätt för att integrera maskininlärning i symboliskt resonemang. Ramverket implementeras inom domänerna för lösare av begränsade Horn-klausuler (CHC) och ordekvationer, och visar förbättrad lösningseffektivitet genom inlärningsbaserade heuristiker.

Symboliska metoder arbetar med logiska formler och algebraiska strukturer för att effektivt utforska och manipulera tillståndsrum. Inom programverifiering säkerställer symboliska metoder att ett program är korrekt genom att koda exekveringsvägar symboliskt och verifiera att de följer givna specifikationer. Inom teorembevisning omvandlas logiska påståenden till formella uttryck, där inferensregler tillämpas för att härleda bevis. Lösning av villkorsproblem innebär resonemang över logiska formler för att avgöra satisfierbarhet inom olika matematiska teorier, såsom de som används i SMT-lösare (Satisfiability Modulo Theories) och lösare av ordekvationer.

Trots sina fördelar medför symboliska metoder utmaningar, bland annat hantering av komplexa symboliska representationer, integration av flera logiska teorier och säkerställande av skalbarhet. Många backend-verktyg, såsom SAT/SMT-lösare och CHC-lösare, använder heuristiker för att optimera beslutsprocesser. Denna avhandling undersöker hur djupinlärning kan ersätta eller förbättra dessa heuristiker för att förbättra lösarnas prestanda och anpassningsförmåga.

Ramverket som introduceras i denna avhandling består av flera centrala komponenter. Logiska formler kodas först som grafer som fångar både syntaktiska och semantiska egenskaper, vilket möjliggör effektiv inlärning med graf-neuronnät (GNNs). Nya grafrepresentationer har utvecklats för både CHCs och ordekvationer för att underlätta strukturerad inlärning. Träninguppgifter formuleras antingen som klassificerings- eller rangordningsproblem,

där beslutsprocesser såsom klausulval i CHC-lösare och grenval i ordekvationer hanteras.

Flera GNN-arkitekturer undersöks för att utvärdera deras effektivitet i olika resonemangsuppgifter. Modeller såsom Graph Convolutional Networks (GCNs) och Relational Hypergraph Neural Networks (R-HyGNNs) utvärderas utifrån deras förmåga att fånga strukturell och semantisk information från symboliska uttryck. För att effektivt integrera tränade modeller i lösare implementeras strategier såsom cachelagring, hybrida heuristiker och selektiv modellförfrågning för att minska den beräkningsmässiga belastningen samtidigt som prestandaförbättringar uppnås.

Ramverket implementeras i CHC-lösare och lösare för ordekvationer. I CHC-lösare styr inlärningsbaserade heuristiker klausulvalet genom att förutsäga vilka klausuler som tillhör minimala osatisfierbara delmängder (MUS:er). Flera träningsstrategier undersöks, inklusive användning av enskilda MUS:er, deras unioner och snitt. I lösare för ordekvationer förbättrar ramverket prestandan genom att rangordna ekvationer och välja optimala grenar baserat på strukturell inlärning. En lösare baserad på Nielsen-transformationer förbättras genom GNN-baserad vägledning, vilket visar betydande effektivitetsvinster vid lösning av problem.

De experimentella resultaten bekräftar att inlärningsbaserade heuristiker konsekvent förbättrar lösarnas prestanda genom att minska lösningstiden och öka antalet lösta problem jämfört med traditionella heuristiker. Ramverket generaliserar effektivt över olika symboliska resonemangsuppgifter, vilket gör det till ett värdefullt verktyg för att automatisera heuristiskt urval i lösare.

Avhandlingen öppnar flera möjligheter för vidare forskning. En möjlig riktning är att vidareutveckla lösare för ordekvationer genom att ta fram nya inferensregler och utöka ramverket för att hantera mer komplexa ordekvationer. Optimering av GNN-arkitekturer för symboliskt resonemang är en annan viktig aspekt, där fokus ligger på att utforma mer effektiva modeller som kan fånga djupare strukturell information. Att utöka träningsdatakällor är också en central fråga, eftersom en ökad mångfald och mängd av träningsdata kan förbättra generaliseringen över flera problemområden. Resultaten tyder på att djupinlärning har potential att revolutionera symboliskt resonemang och bana väg för en ännu djupare integration av AI-tekniker i formella metoder.

References

[1] V. D'Silva, D. Kroening, and G. Weissenbacher, "A survey of automated techniques for formal software verification," *IEEE Transactions on Computer-Aided Design of Integrated Circuits and Systems*, vol. 27, no. 7, pp. 1165–1178, 2008.

[2] M. Mukelabai, D. Nešić, S. Maro, T. Berger, and J.-P. Steghöfer, "Tackling combinatorial explosion: a study of industrial needs and practices for analyzing highly configurable systems," in *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, ASE '18, (New York, NY, USA), p. 155–166, Association for Computing Machinery, 2018.

[3] R. Baldoni, E. Coppa, D. C. D'elia, C. Demetrescu, and I. Finocchi, "A survey of symbolic execution techniques," *ACM Comput. Surv.*, vol. 51, May 2018.

[4] A. Robinson and A. Voronkov, eds., *Handbook of automated reasoning*. NLD: Elsevier Science Publishers B. V., 2001.

[5] J. Robinson and A. Voronkov, *Handbook of Automated Reasoning: Volume 1*. Cambridge, MA, USA: MIT Press, 2001.

[6] A. Biere, M. Heule, H. van Maaren, and T. Walsh, eds., *Handbook of Satisfiability*, vol. 185 of *Frontiers in Artificial Intelligence and Applications*. IOS Press, 2009.

[7] C. Barrett, R. Sebastiani, S. Seshia, and C. Tinelli, "Satisfiability modulo theories," in *Handbook of Satisfiability* (A. Biere, M. J. H. Heule, H. van Maaren, and T. Walsh, eds.), vol. 185 of *Frontiers in Artificial Intelligence and Applications*, ch. 26, pp. 825–885, IOS Press, Feb. 2009.

[8] N. Bjørner, A. Gurfinkel, K. McMillan, and A. Rybalchenko, *Horn Clause Solvers for Program Verification*, pp. 24–51. Cham: Springer International Publishing, 2015.

[9] Akers, "Binary decision diagrams," *IEEE Transactions on Computers*, vol. C-27, no. 6, pp. 509–516, 1978.

[10] G. Nelson and D. C. Oppen, "Simplification by cooperating decision procedures," *ACM Trans. Program. Lang. Syst.*, vol. 1, p. 245–257, Oct. 1979.

[11] S. Schulz, "E - a brainiac theorem prover," *AI Commun.*, vol. 15, p. 111–126, Aug. 2002.

[12] S. Schulz, S. Cruanes, and P. Vukmirović, "Faster, higher, stronger: E 2.3," in *Automated Deduction – CADE 27* (P. Fontaine, ed.), (Cham), pp. 495–507, Springer International Publishing, 2019.

[13] L. Kovács and A. Voronkov, "First-order theorem proving and Vampire," in *Computer Aided Verification* (N. Sharygina and H. Veith, eds.), (Berlin, Heidelberg), pp. 1–35, Springer Berlin Heidelberg, 2013.

[14] S. Schulz and M. Möhrmann, "Performance of clause selection heuristics for saturation-based theorem proving," in *Proceedings of the 8th International Joint Conference on Automated Reasoning - Volume 9706*, (Berlin, Heidelberg), p. 330–345, Springer-Verlag, 2016.

[15] Z. Li, J. Sun, L. Murphy, Q. Su, Z. Li, X. Zhang, K. Yang, and X. Si, “A Survey on Deep Learning for Theorem Proving,” *arXiv e-prints*, p. arXiv:2404.09939, Apr. 2024.

[16] M. Suda, “Vampire with a brain is a good ITP hammer,” in *Frontiers of Combining Systems* (B. Konev and G. Reger, eds.), (Cham), pp. 192–209, Springer International Publishing, 2021.

[17] J. Jakubův, K. Chvalovský, M. Olšák, B. Piotrowski, M. Suda, and J. Urban, “ENIGMA anonymous: Symbol-independent inference guiding machine (system description),” 2020.

[18] T. Chen and C. Guestrin, “XGBoost: A scalable tree boosting system,” in *Proceedings of the 22nd ACM SIGKDD International Conference on Knowledge Discovery and Data Mining*, KDD ’16, (New York, NY, USA), p. 785–794, Association for Computing Machinery, 2016.

[19] P. W. Battaglia, J. B. Hamrick, V. Bapst, *et al.*, “Relational inductive biases, deep learning, and graph networks,” *CoRR*, vol. abs/1806.01261, 2018.

[20] N. Eén and N. Sörensson, “An extensible SAT-solver,” in *Theory and Applications of Satisfiability Testing* (E. Giunchiglia and A. Tacchella, eds.), (Berlin, Heidelberg), pp. 502–518, Springer Berlin Heidelberg, 2004.

[21] G. Audemard and L. Simon, “Predicting learnt clauses quality in modern SAT solvers,” in *Proceedings of the 21st International Joint Conference on Artificial Intelligence*, IJCAI’09, (San Francisco, CA, USA), p. 399–404, Morgan Kaufmann Publishers Inc., 2009.

[22] A. Biere, T. Faller, K. Fazekas, M. Fleury, N. Froleyks, and F. Pollitt, “CaDiCaL 2.0,” in *Computer Aided Verification* (A. Gurfinkel and V. Ganesh, eds.), (Cham), pp. 133–152, Springer Nature Switzerland, 2024.

[23] J. P. Marques Silva and K. A. Sakallah, *Grasp—A New Search Algorithm for Satisfiability*, pp. 73–89. Boston, MA: Springer US, 2003.

[24] M. W. Moskewicz, C. F. Madigan, Y. Zhao, L. Zhang, and S. Malik, “Chaff: engineering an efficient SAT solver,” in *Proceedings of the 38th Annual Design Automation Conference*, DAC ’01, (New York, NY, USA), p. 530–535, Association for Computing Machinery, 2001.

[25] A. Biere and A. Fröhlich, “Evaluating CDCL variable scoring schemes,” in *Theory and Applications of Satisfiability Testing – SAT 2015* (M. Heule and S. Weaver, eds.), (Cham), pp. 405–422, Springer International Publishing, 2015.

[26] D. Selsam and N. Bjørner, “Guiding high-performance SAT solvers with unsat-core predictions,” in *Theory and Applications of Satisfiability Testing – SAT 2019* (M. Janota and I. Lynce, eds.), (Cham), pp. 336–353, Springer International Publishing, 2019.

[27] J. H. Liang, C. Oh, M. Mathew, C. Thomas, C. Li, and V. Ganesh, “Machine learning-based restart policy for CDCL SAT solvers,” in *Theory and Applications of Satisfiability Testing – SAT 2018* (O. Beyersdorff and C. M. Wintersteiger, eds.), (Cham), pp. 94–110, Springer International Publishing, 2018.

[28] Y. LeCun, Y. Bengio, and G. Hinton, “Deep learning,” *Nature*, vol. 521, pp. 436–444, May 2015.

[29] H. Hojjat and P. Ruemmer, “The ELDARICA Horn solver,” in *2018 Formal Methods in Computer Aided Design (FMCAD)*, pp. 1–7, 2018.

[30] D. Silver, A. Huang, C. J. Maddison, *et al.*, “Mastering the game of Go with deep

neural networks and tree search," *Nature*, vol. 529, pp. 484–489, Jan 2016.

[31] J. Jumper, R. Evans, A. Pritzel, T. Green, *et al.*, "Highly accurate protein structure prediction with AlphaFold," *Nature*, vol. 596, pp. 583–589, Aug 2021.

[32] OpenAI, J. Achiam, S. Adler, S. Agarwal, L. Ahmad, *et al.*, "GPT-4 technical report," 2024.

[33] G. Cybenko, "Approximation by superpositions of a sigmoidal function," *Mathematics of Control, Signals and Systems*, vol. 2, pp. 303–314, Dec 1989.

[34] C. Cortes and V. Vapnik, "Support-vector networks," *Machine Learning*, vol. 20, pp. 273–297, Sep 1995.

[35] R. L. Hardy, "Multiquadric equations of topography and other irregular surfaces," *Journal of Geophysical Research (1896-1977)*, vol. 76, no. 8, pp. 1905–1915, 1971.

[36] K. Fukushima, "Neocognitron: A self-organizing neural network model for a mechanism of pattern recognition unaffected by shift in position," *Biological Cybernetics*, vol. 36, pp. 193–202, Apr 1980.

[37] Y. LeCun, B. Boser, J. S. Denker, D. Henderson, R. E. Howard, W. Hubbard, and L. D. Jackel, "Backpropagation applied to handwritten zip code recognition," *Neural Computation*, vol. 1, no. 4, pp. 541–551, 1989.

[38] E. Clarke, O. Grumberg, S. Jha, Y. Lu, and H. Veith, "Counterexample-guided abstraction refinement," in *Computer Aided Verification* (E. A. Emerson and A. P. Sistla, eds.), (Berlin, Heidelberg), pp. 154–169, Springer Berlin Heidelberg, 2000.

[39] P. A. Abdulla, M. F. Atig, J. Cailler, C. Liang, and P. Rümmer, "Guiding word equation solving using graph neural networks," in *Automated Technology for Verification and Analysis* (S. Akshay, A. Niemetz, and S. Sankaranarayanan, eds.), (Cham), pp. 279–301, Springer Nature Switzerland, 2025.

[40] J. Nielsen, "Die Isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden," *Mathematische Annalen*, vol. 78, pp. 385–397, 1917.

[41] W. Plandowski, "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.

[42] A. Horn, "On sentences which are true of direct unions of algebras," *Journal of Symbolic Logic*, vol. 16, no. 1, p. 14–21, 1951.

[43] A. Colmerauer and P. Roussel, *The birth of Prolog*, p. 331–367. New York, NY, USA: Association for Computing Machinery, 1996.

[44] C. Barrett, P. Fontaine, and C. Tinelli, "The SMT-LIB Standard: Version 2.7," tech. rep., Department of Computer Science, The University of Iowa, 2025. Available at `www.SMT-LIB.org`.

[45] X. Si, A. Naik, H. Dai, M. Naik, and L. Song, "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*, (Berlin, Heidelberg), p. 151–164, Springer-Verlag, 2020.

[46] Z. Luo and X. Si, "Chronosymbolic learning: Efficient chc solving with symbolic reasoning and inductive learning," in *AI Verification* (G. Avni, M. Giacobbe, T. T. Johnson, G. Katz, A. Lukina, N. Narodytska, and C. Schilling, eds.), (Cham), pp. 1–28, Springer Nature Switzerland, 2024.

[47] N. Le, X. Si, and A. Gurfinkel, "Data-driven optimization of inductive generalization," in *2021 Formal Methods in Computer Aided Design (FMCAD)*,

pp. 86–95, 2021.

[48] A. Komuravelli, A. Gurfinkel, and S. Chaki, “SMT-based model checking for recursive programs,” in *Computer Aided Verification* (A. Biere and R. Bloem, eds.), (Cham), pp. 17–34, Springer International Publishing, 2014.

[49] A. Thue, *Über die gegenseitige Lage gleicher Teile gewisser Zeichenreihen*. Skrifter udgivne af Videnskabsselskabet i Christiania. 1,Math.Nat.wiss.Kl.1912,1, Jacob Dybwad, 1912.

[50] *Combinatorics on Words*. Cambridge Mathematical Library, Cambridge University Press, 2 ed., 1997.

[51] G. S. Makanin, “The problem of solvability of equations in a free semigroup,” *Math. Sb. (N.S.)*, vol. 103(145), no. 2(6), pp. 147–236, 1977.

[52] R. Amadini, “A survey on string constraint solving,” *CoRR*, vol. abs/2002.02376, 2020.

[53] A. Jez, “Recompression: a simple and powerful technique for word equations,” *CoRR*, vol. abs/1203.3705, 2012.

[54] J. D. Day, T. Ehlers, M. Kulczynski, F. Manea, D. Nowotka, and D. B. Poulsen, “On solving word equations using SAT,” in *Reachability Problems* (E. Filiot, R. Jungers, and I. Potapov, eds.), (Cham), pp. 93–106, Springer International Publishing, 2019.

[55] A. W. Lin and R. Majumdar, “Quadratic word equations with length constraints, counter systems, and presburger arithmetic with divisibility,” *CoRR*, vol. abs/2007.15478, 2020.

[56] L. De Moura and N. Bjørner, “Z3: an efficient SMT solver,” in *Proceedings of the Theory and Practice of Software, 14th International Conference on Tools and Algorithms for the Construction and Analysis of Systems*, TACAS’08/ETAPS’08, (Berlin, Heidelberg), p. 337–340, Springer-Verlag, 2008.

[57] H. Barbosa, C. Barrett, M. Brain, G. Kremer, H. Lachnitt, M. Mann, A. Mohamed, M. Mohamed, A. Niemetz, A. Nötzli, A. Ozdemir, M. Preiner, A. Reynolds, Y. Sheng, C. Tinelli, and Y. Zohar, “cvc5: A versatile and industrial-strength SMT solver,” in *Tools and Algorithms for the Construction and Analysis of Systems* (D. Fisman and G. Rosu, eds.), (Cham), pp. 415–442, Springer International Publishing, 2022.

[58] J. Piepenbrock, M. Janota, J. Urban, and J. Jakubův, “First experiments with neural cvc5,” in *EPiC Series in Computing*, vol. 100, p. 264–249, EasyChair.

[59] M. H. Liffiton and K. A. Sakallah, “Algorithms for computing minimal unsatisfiable subsets of constraints,” *Journal of Automated Reasoning*, vol. 40, pp. 1–33, Jan 2008.

[60] D. E. Rumelhart, G. E. Hinton, and R. J. Williams, “Learning representations by back-propagating errors,” *Nature*, vol. 323, pp. 533–536, Oct 1986.

[61] V. Nair and G. E. Hinton, “Rectified linear units improve restricted boltzmann machines,” in *Proceedings of the 27th International Conference on International Conference on Machine Learning*, ICML’10, (Madison, WI, USA), p. 807–814, Omnipress, 2010.

[62] C. M. Bishop, *Pattern Recognition and Machine Learning (Information Science and Statistics)*. Berlin, Heidelberg: Springer-Verlag, 2006.

[63] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, and G. E. Dahl, “Neural message passing for quantum chemistry,” *CoRR*, vol. abs/1704.01212, 2017.

[64] T. N. Kipf and M. Welling, “Semi-supervised classification with graph convolutional networks,” *CoRR*, vol. abs/1609.02907, 2016.

[65] V. Kurin, S. Godil, S. Whiteson, and B. Catanzaro, “Improving SAT solver heuristics with graph networks and reinforcement learning,” *CoRR*, vol. abs/1909.11830, 2019.

[66] A. Paliwal, S. Loos, M. Rabe, K. Bansal, and C. Szegedy, “Graph representations for higher-order logic and theorem proving,” 2019.

[67] K. Bansal, S. M. Loos, M. N. Rabe, C. Szegedy, and S. Wilcox, “HOList: An environment for machine learning of higher-order theorem proving (extended version),” *CoRR*, vol. abs/1904.03241, 2019.

[68] J. Hůla, D. Mojžíšek, and M. Janota, “Graph neural networks for scheduling of SMT solvers,” in *2021 IEEE 33rd International Conference on Tools with Artificial Intelligence (ICTAI)*, pp. 447–451, 2021.

[69] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, and D. L. Dill, “Learning a SAT solver from single-bit supervision,” *CoRR*, vol. abs/1802.03685, 2018.

[70] Z. Zhang, D. Chetelat, J. Cotnareanu, A. Ghose, W. Xiao, H.-L. Zhen, Y. Zhang, J. Hao, M. Coates, and M. Yuan, “GraSS: Combining graph neural networks with expert knowledge for SAT solver selection,” 2024.

[71] A. A. Alemi, F. Chollet, G. Irving, C. Szegedy, and J. Urban, “DeepMath - deep sequence models for premise selection,” *CoRR*, vol. abs/1606.04442, 2016.

[72] M. Wang, Y. Tang, J. Wang, and J. Deng, “Premise selection for theorem proving by deep graph embedding,” *CoRR*, vol. abs/1709.09994, 2017.

[73] M. Crouse, S. Whitehead, I. Abdelaziz, B. Makni, C. Cornelio, P. Kapanipathi, E. Pell, K. Srinivas, V. Thost, M. Witbrock, and A. Fokoue, “A deep reinforcement learning based approach to learning transferable proof guidance strategies,” *CoRR*, vol. abs/1911.02065, 2019.

[74] A. Loreggia, Y. Malitsky, H. Samulowitz, and V. Saraswat, “Deep learning for algorithm portfolios,” in *Proceedings of the Thirtieth AAAI Conference on Artificial Intelligence*, AAAI’16, p. 1280–1286, AAAI Press, 2016.

[75] Y. Wang, F. Gao, A. Alipour, L. Wang, X. Li, and Z. Su, “CNNSAT: Fast, accurate boolean satisfiability using convolutional neural networks,” 2019.

[76] C. Grozea and M. Popescu, “Can machine learning learn a decision oracle for NP problems? a test on SAT,” *Fundam. Inf.*, vol. 131, p. 441–450, July 2014.

[77] M. Davis, G. Logemann, and D. Loveland, “A machine program for theorem-proving,” *Commun. ACM*, vol. 5, p. 394–397, July 1962.

[78] S. A. Cook, “The complexity of theorem-proving procedures,” in *Proceedings of the Third Annual ACM Symposium on Theory of Computing*, STOC ’71, (New York, NY, USA), p. 151–158, Association for Computing Machinery, 1971.

[79] K. Xu, W. Hu, J. Leskovec, and S. Jegelka, “How powerful are graph neural networks?,” *CoRR*, vol. abs/1810.00826, 2018.

[80] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio, “Graph attention networks,” 2018.

[81] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, and M. Welling, “Modeling relational data with graph convolutional networks,” 2017.

[82] Z. Li, J. Guo, and X. Si, “G4SATBench: Benchmarking and advancing SAT solving with graph neural networks,” 2024.

[83] G. Audemard and L. Simon, “On the Glucose SAT solver,” *International Journal*

on Artificial Intelligence Tools, vol. 27, no. 01, p. 1840001, 2018.

[84] O. Miroslav, K. Cezary, and U. Josef, *Property Invariant Embedding for Automated Reasoning*. IOS Press, 2020.

[85] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, and I. Polosukhin, “Attention is all you need,” *CoRR*, vol. abs/1706.03762, 2017.

[86] W. Wang, Y. Hu, M. Tiwari, S. Khurshid, K. McMillan, and R. Miikkulainen, “NeuroBack: Improving CDCL SAT solving using graph neural networks,” 2024.

[87] K. Chvalovský, K. Korovin, J. Piepenbrock, and J. Urban, “Guiding an instantiation prover with graph neural networks,” in *Proceedings of 24th International Conference on Logic for Programming, Artificial Intelligence and Reasoning* (R. Piskac and A. Voronkov, eds.), vol. 94 of *EPiC Series in Computing*, pp. 112–123, EasyChair, 2023.

[88] G. Hinton, O. Vinyals, and J. Dean, “Distilling the knowledge in a neural network,” 2015.

[89] Y. Kuang, J. Wang, H. Liu, F. Zhu, X. Li, J. Zeng, J. Hao, B. Li, and F. Wu, “Rethinking branching on exact combinatorial optimization solver: The first deep symbolic discovery framework,” in *The Twelfth International Conference on Learning Representations, ICLR 2024, Vienna, Austria, May 7-11, 2024*, OpenReview.net, 2024.

[90] P. A. Abdulla, M. F. Atig, J. Cailler, C. Liang, and P. Rümmer, “Guiding word equation solving using graph neural networks (extended technical report),” 2024.

[91] S. Hochreiter and J. Schmidhuber, “Long short-term memory,” *Neural Comput.*, vol. 9, no. 8, pp. 1735–1780, 1997.

Distribution: publications.uu.se
urn:nbn:se:uu:diva-551906

ACTA UNIVERSITATIS
UPSALIENSIS
2025

Paper I

Exploring Representation of Horn Clauses using GNNs

Chencheng Liang[1], Philipp Rümmer[1,2] and Marc Brockschmidt[3]

[1] *Uppsala University, Department of Information Technology, Uppsala, Sweden*

[2] *University of Regensburg, Regensburg, Germany*

[3] *Microsoft Research*

Abstract

In recent years, the application of machine learning in program verification, and the embedding of programs to capture semantic information, has been recognised as an important tool by many research groups. Learning program semantics from raw source code is challenging due to the complexity of real-world programming language syntax and due to the difficulty of reconstructing long-distance relational information implicitly represented in programs using identifiers. Addressing the first point, we consider Constrained Horn Clauses (CHCs) as a standard representation of program verification problems, providing a simple and programming language-independent syntax. For the second challenge, we explore graph representations of CHCs, and propose a new Relational Hypergraph Neural Network (R-HyGNN) architecture to learn program features.

We introduce two different graph representations of CHCs. One is called *constraint graph* (CG), and emphasizes syntactic information of CHCs by translating the symbols and their relations in CHCs as typed nodes and binary edges, respectively, and constructing the constraints as abstract syntax trees. The second one is called *control- and data-flow hypergraph* (CDHG), and emphasizes semantic information of CHCs by representing the control and data flow through ternary hyperedges.

We then propose a new GNN architecture, *R-HyGNN,* extending Relational Graph Convolutional Networks, to handle hypergraphs. To evaluate the ability of R-HyGNN to extract semantic information from programs, we use R-HyGNNs to train models on the two graph representations, and on five proxy tasks with increasing difficulty, using benchmarks from CHC-COMP 2021 as training data. The most difficult proxy task requires the model to predict the occurrence of clauses in counter-examples, which subsumes satisfiability of CHCs. CDHG achieves 90.59% accuracy in this task. Furthermore, R-HyGNN has perfect predictions on one of the graphs consisting of more than 290 clauses. Overall, our experiments indicate that R-HyGNN can capture intricate program features for guiding verification problems.

Keywords

Constraint Horn clauses, Graph Neural Networks, Automatic program verification

PAAR'22: 8th Workshop on Practical Aspects of Automated Reasoning, August 11–12, 2022, Haifa, Israel

EMAIL: chencheng.liang@it.uu.se (C. Liang); philipp.ruemmer@it.uu.se (P. Rümmer); marc@marcbrockschmidt.de (M. Brockschmidt)
URL: https://github.com/ChenchengLiang/ (C. Liang); https://github.com/pruemmer/ (P. Rümmer); https://github.com/mmjb/ (M. Brockschmidt)
ORCID: 0000-0002-4926-8089 (C. Liang); 0000-0002-2733-7098 (P. Rümmer); 0000-0001-6277-2768 (M. Brockschmidt)

1. Introduction

Automatic program verification is challenging because of the complexity of industrially relevant programs. In practice, constructing domain-specific heuristics from program features (e.g., information from loops, control flow, or data flow) is essential for solving verification problems. For instance, [1] and [2] extract semantic information by performing systematical static analysis to refine abstractions for the counterexample-guided abstraction refinement (CEGAR) [3] based system. However, manually designed heuristics usually aim at a specific domain and are hard to transfer to other problems. Along with the rapid development of deep learning in recent years, learning-based methods have evolved quickly and attracted more attention. For example, the program features are explicitly given in [4, 5] to decide which algorithm is potentially the best for verifying the programs. Later in [6, 7], program features are learned in the end-to-end pipeline. Moreover, some generative models [8, 9] are also introduced to produce essential information for solving verification problems. For instance, Code2inv [10] embeds the programs by graph neural networks (GNNs) [11] and learns to construct loop invariants by a deep neural reinforcement framework.

For deep learning-based methods, no matter how the learning pipeline is designed and the neural network structure is constructed, learning to represent semantic program features is essential and challenging (a) because the syntax of the source code varies depending on the programming languages, conventions, regulations, and even syntax sugar and (b) because it requires capturing intricate semantics from long-distance relational information based on re-occurring identifiers. For the first challenge, since the source code is not the only way to represent a program, learning from other formats is a promising direction. For example, inst2vec [12] learns control and data flow from LLVM intermediate representation [13] by recursive neural networks (RNNs) [14]. Constrained Horn Clauses (CHCs) [15], as an intermediate verification language, consist of logic implications and constraints and can alleviate the difficulty since they can naturally encode program logic with simple syntax. For the second challenge, we use graphs to represent CHCs and learn the program features by GNNs since they can learn from the structural information within the node's N-hop neighbourhood by recursive neighbourhood aggregation (i.e., neural message passing) procedure.

In this work, we explore how to learn program features from CHCs by answering two questions: (1) What kind of graph representation is suitable for CHCs? (2) Which kind of GNN is suitable to learn from the graph representation?

For the first point, we introduce two graph representations for CHCs: the constraint graph (CG) and control- and data-flow hypergraph (CDHG). The constraint graph encodes the CHCs into three abstract layers (predicate, clause, and constraint layers) to preserve as much structural information as possible (i.e., it emphasizes program syntax). On the other hand, the Control- and data-flow hypergraph uses ternary hyperedges to capture the flow of control and data in CHCs to emphasize program semantics. To better express control and data flow in CDHG, we construct it from normalized CHCs. The normalization changes the format of the original CHC but retains logical meaning. We assume that different graph representations of CHCs capture different aspects of semantics. The two

Table 1
Proxy tasks used to evaluate suitability of different graph representations.

Task	Task type	Description
1. Argument identification	Node binary classification	For each element in CHCs, predict if it is an argument of relation symbols.
2. Count occurrence of relation symbols in all clauses	Regression task on node	For each relation symbol, predict how many times it occurs in all clauses.
3. Relation symbol occurrence in SCCs	Node binary classification	For each relation symbol, predict if a cycle exists from the node to itself (membership in strongly connected component, SCC).
4. Existence of argument bounds	Node binary classification	For each argument of a relation symbol, predict if it has a lower or upper bound.
5. Clause occurrence in counter-examples	Node binary classification	For each CHC, predict if it appears in counter-examples.

graph representations can be used as a baseline to construct new graph representations of CHC to represent different semantics. In addition, similar to the idea in [16], our graph representations are invariant to the concrete symbol names in the CHCs since we map them to typed nodes.

For the second point, we propose a Relational Hypergraph Neural Network (R-HyGNN), an extension of Relational Graph Convolutional Networks (R-GCN) [17]. Similar to the GNNs used in LambdaNet [18], R-HyGNN can handle hypergraphs by concatenating the node representations involved in a hyperedge and passing the messages to all nodes connected by the hyperedge.

Finally, we evaluate our framework (two graph representations of CHCs and R-HyGNN) by five proxy tasks (see details in Table 1) with increasing difficulties. Task 1 requires the framework to learn to classify syntactic information of CHCs, which is explicitly encoded in the two graph representations. Task 2 requires the R-HyGNN to predict a syntactic counting task. Task 3 needs the R-HyGNN to approximate the Tarjan's algorithm [19], which solves a general graph theoretical problem. Task 4 is much harder than the last three tasks since the existence of argument bounds is undecidable. Task 5 is harder than solving CHCs since it predicts the trace of counter-examples (CEs). Note that Task 1 to 3 can be easily solved by specific, dedicated standard algorithms. We include them to systematically study the representational power of graph neural networks applied to different graph construction methods. However, we speculate that using these tasks as pre-training objectives for neural networks that are later fine-tuned to specific (data-poor) tasks may be a successful strategy which we plan to study in future work.

Our benchmark data is extracted from the 8705 linear and 8425 non-linear Linear Integer Arithmetic (LIA) problems in the CHC-COMP repository[1] (see Table 1 in the competition report [20]). The verification problems come from various sources (e.g., higher-order program verification benchmark[2] and benchmarks generated with JayHorn[3]),

[1]https://chc-comp.github.io/
[2]https://github.com/chc-comp/hopv
[3]https://github.com/chc-comp/jayhorn-benchmarks

therefore cover programs with different size and complexity. We collect and form the train, valid, and test data using the predicate abstraction-based model checker Eldarica [21]. We implement R-HyGNNs[4] based on the framework tf2_gnn[5]. Our code is available in a Github repository[6]. For both graph representations, even if the predicted accuracy decreases along with the increasing difficulty of tasks, for undecidable problems in Task 4, R-HyGNN still achieves high accuracy, i.e., 91% and 94% for constraint graph and CDHG, respectively. Moreover, in Task 5, despite the high accuracy (96%) achieved by CDHG, R-HyGNN has a perfect prediction on one of the graphs consisting of more than 290 clauses, which is impossible to achieve by learning simple patterns (e.g., predict the clause including *false* as positive). Overall, our experiments show that our framework learns not only the explicit syntax but also intricate semantics.

Contributions of the paper. (i) We encode CHCs into two graph representations, emphasising abstract program syntactic and semantic information, respectively. (ii) We extend a message passing-based GNN, R-GCN, to R-HyGNN to handle hypergraphs. (iii) We introduce five proxy supervised learning tasks to explore the capacity of R-HyGNN to learn semantic information from the two graph representations. (iv) We evaluate our framework on the CHC-COMP benchmark and show that this framework can learn intricate semantic information from CHCs and has the potential to produce good heuristics for program verification.

2. Background

2.1. From Program Verification to Horn clauses

Constrained Horn Clauses are logical implications involving unknown predicates. They can be used to encode many formalisms, such as transition systems, concurrent systems, and interactive systems. The connections between program logic and CHCs can be bridged by Floyd-Hoare logic [22, 23], allowing to encode program verification problems into the CHC satisfiability problems [24]. In this setting, a program is guaranteed to satisfy a specification if the encoded CHCs are satisfiable, and vice versa.

We write CHCs in the form $H \leftarrow B_1 \wedge \cdots \wedge B_n \wedge \varphi$, where (i) B_i is an application $q_i(\bar{t_i})$ of the relation symbol q_i to a list of first-order terms $\bar{t_i}$; (ii) H is either an application $q(\bar{t})$, or *false*; (iii) φ is a first-order constraint. Here, H and $B_1 \wedge \cdots \wedge B_n \wedge \varphi$ in the left and right hand side of implication $\leftarrow$ are called “head" and “body", respectively.

An example in Figure 1 explains how to encode a verification problem into CHCs. In Figure 1a, we have a verification problem, i.e., a C program with specifications. It has an external input n, and we can initially assume that $x == n, y == n,$ and, $n >= 0$. While x is not equal to 0, x and y are decreased by 1. The assertion is that finally, $y == 0$. This program can be encoded to the CHC shown in Figure 1b. The variables x and y

[4]https://github.com/ChenchengLiang/tf2-gnn
[5]https://github.com/microsoft/tf2-gnn
[6]https://github.com/ChenchengLiang/Systematic-Predicate-Abstraction-using-Machine-Learning

```
0  extern int n;
1  void main(){
2      int x,y;
3      assume(x==n && y==n && n>=0);
4      while(x!=0){
5          x--;
6          y--;
7      }
8      assert(y==0);
9  }
```

(a) An verification problem written in C.

$$
\begin{array}{rll}
L_0(n) \leftarrow \mathit{true} & & \text{line 0} \\
L_1(n) \leftarrow L_0(n) & & \text{line 1} \\
L_2(x,y,n) \leftarrow L_1(n) & & \text{line 2} \\
L_3(x,y,n) \leftarrow L_2(x,y,n) \wedge n \geq 0 & & \\
\wedge\, x = n \wedge y = n & & \text{line 3} \\
L_8(x,y,n) \leftarrow L_3(x,y,n) \wedge x = 0 & & \text{line 4} \\
L_4(x,y,n) \leftarrow L_3(x,y,n) \wedge x \neq 0 & & \text{line 4} \\
L_5(x,y,n) \leftarrow L_4(x',y,n) \wedge x = x' - 1 & & \text{line 5} \\
L_6(x,y,n) \leftarrow L_5(x,y',n) \wedge y = y' - 1 & & \text{line 6} \\
L_3(x,y,n) \leftarrow L_6(x,y,n) & & \text{line 6} \\
\mathit{false} \leftarrow L_8(x,y,n) \wedge y \neq 0 & & \text{line 8}
\end{array}
$$

(b) CHCs encoded from C program in Figure 1a.

$$
\begin{array}{rl}
L(x,y,n) \leftarrow n \geq 0 \wedge x = n \wedge y = n & \text{line 3} \\
L(x,y,n) \leftarrow L(x',y',n') \wedge x' \neq 0 \wedge x = x' - 1 \wedge y = y' - 1 \wedge n = n' & \text{line 4-7} \\
\mathit{false} \leftarrow L(x,y,n) \wedge x = 0 \wedge y \neq 0 & \text{line 8}
\end{array}
$$

(c) Simplified CHCs from Figure 1b.

Figure 1: An example to show how to encode a verification problem written in C to CHCs. For the C program, the left-hand side numbers indicate the line number. The line numbers in Figure 1b and 1c correspond to the line in Figure 1a. For example, the line $L_0(n) \leftarrow \mathit{true}$ in Figure 1b is transformed from line 1 "extern int n ;" in Figure 1a.

are quantified universally. We can further simplify the CHCs in Figure 1b to the CHCs shown in Figure 1c without changing the satisfiability by some preprocessing steps (e.g., inlining and slicing) [25]. For example, the first CHC encodes line 3, i.e., the assume statement, the second clause encodes lines 4-7, i.e., the while loop, and the third clause encodes line 8, i.e., the assert statement in Figure 1a. Solving the CHCs is equivalent to answering the verification problem. In this example, with a given n, if the CHCs are satisfiable for all x and y, then the program is guaranteed to satisfy the specifications.

2.2. Graph Neural Networks

Let $G = (V, R, E, X, \ell)$ denote a graph in which $v \in V$ is a set of nodes, $r \in R$ is a set of edge types, $E \in V \times V \times R$ is a set of typed edges, $x \in X$ is a set of node types, and $\ell : v \to x$ is a labelling map from nodes to their type. A tuple $e = (u, v, r) \in E$ denotes an edge from node u to v with edge type r.

Message passing-based GNNs use a neighbourhood aggregation strategy, where at timestep t, each node updates its representation h_v^t by aggregating representations of its neighbours and then combining its own representation. The initial node representation h_v^0 is usually derived from its type or label $\ell(v)$. The common assumption of this architecture is that after T iterations, the node representation h_v^T captures local information within

t-hop neighbourhoods. Most GNN architectures [26, 27] can be characterized by their used "aggregation" function ρ and "update" function ϕ. The node representation of the t-th layer of such a GNN is then computed by $h_v^t = \phi(\rho(\{h_u^{t-1} \mid u \in N_v^r, r \in R\}), h_v^{t-1})$, where R is a set of edge type and N_v^r is the set of nodes that are the neighbors of v in edge type r.

A closed GNN architecture to the R-HyGNN is R-GCN [17]. They update the node representation by

$$h_v^t = \sigma(\sum_{r\in R}\sum_{u\in N_v^r}\frac{1}{c_{v,r}}W_r^t h_u^{t-1} + W_0 h_v^{t-1}), \tag{1}$$

where W_r and W_0 are edge-type-dependent trainable weights, $c_{v,r}$ is a learnable or fixed normalisation factor, and σ is a activation function.

3. Graph Representations for CHCs

Graphs as a representation format support arbitrary relational structure and thus can naturally encode information with rich structures like CHCs. We define two graph representations for CHCs that emphasize the program syntax and semantics, respectively. We map all symbols in CHCs to typed nodes and use typed edges to represent their relations. In this section, we give concrete examples to illustrate how to construct the two graph representations from a single CHC modified from Figure 1c. In the examples, we first give the intuition of the graph design and then describe how to construct the graph step-wise. To better visualize how to construct the two graph representations in Figures 2 and 3, the concrete symbol names for the typed nodes are shown in the blue boxes. R-HyGNN is not using these names (which, as a result of repeated transformations, usually do not carry any meaning anyway) and only consumes the node types. The formal definitions of the graph representations and the algorithms to construct them from multiple CHCs are in the full version of this paper [28]. Note that the two graph representations in this study are designed empirically. They can be used as a baseline to create variations of the graphs to fit different purposes.

3.1. Constraint Graph (CG)

Our Constraint graph is a directed graph with binary edges designed to emphasize syntactic information in CHCs. One concrete example of constructing the constraint graph for a single CHC $L(x, y, n) \leftarrow L(x', y', n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1$ modified from Figure 1c is shown in Figure 2. The corresponding node and edge types are described in Tables 2 and 3 in the full version of this paper [28].

We construct the constraint graph by parsing the CHCs in three different aspects (relation symbol, clause structure, and constraint) and building relations for them. In other words, a constraint graph consists of three layers: the predicate layer depicts the relation between relation symbols and their arguments; the clause layer describes the abstract syntax of head and body items in the CHC; the constraint layer represents the constraint by abstract syntax trees (ASTs).

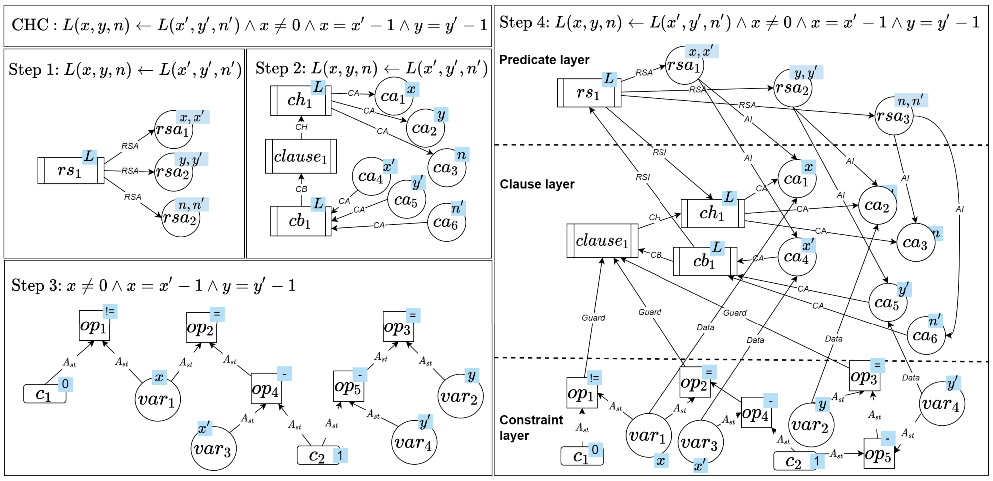

CHC : $L(x, y, n) \leftarrow L(x', y', n') \wedge x \neq 0 \wedge x = x' - 1 \wedge y = y' - 1$
Step 1: $L(x, y, n) \leftarrow L(x', y', n')$
Step 2: $L(x, y, n) \leftarrow L(x', y', n')$
Step 3: $x \neq 0 \wedge x = x' - 1 \wedge y = y' - 1$
Step 4: $L(x, y, n) \leftarrow L(x', y', n') \wedge x \neq 0 \wedge x = x' - 1 \wedge y = y' - 1$
Predicate layer
Clause layer
Constraint layer
RSA
RSI
AI
CH
CB
CA
Guard
Data
Ast

Figure 2: Construct constraint graph from the CHC $L(x, y, n) \leftarrow L(x', y', n') \wedge x \neq 0 \wedge x = x' - 1 \wedge y = y' - 1$. Note that some nodes have multiple concrete symbol names (e.g., node rsa_1 has two concrete names, x and x') since one relation symbol may bind with different arguments.

Constructing a constraint graph. Now we give a concrete example to describe how to construct a constraint graph for a single CHC $L(x, y, n) \leftarrow L(x', y', n') \wedge x \neq 0 \wedge x = x' - 1 \wedge y = y' - 1$ step-wise. All steps correspond to the steps in Figure 2. In the first step, we draw relation symbols and their arguments as typed nodes and build the connection between them. In the second step, we construct the clause layer by drawing clauses, the relation symbols in the head and body, and their arguments as typed nodes and build the relation between them. In the third step, we construct the constraint layer by drawing ASTs for the sub-expressions of the constraint. In the fourth step, we add connections between three layers. The predicate and clause layer are connected by the relation symbol instance (*RSI*) and argument instance (*AI*) edges, which means the elements in the predicate layer are instances of the clause layer. The clause and constraint layers are connected by the *GUARD* and *DATA* edges since the constraint is the guard of the clause implication, and the constraint and clause layer share some elements.

3.2. Control- and Data-flow Hypergraph (CDHG)

In contrast to the constraint graph, the CDHG representation emphasizes the semantic information (control and data flow) in the CHCs by hyperedges which can join any number of vertices. To represent control and data flow in CDHG, first, we preprocess every CHC and then split the constraint into control and data flow sub-formulas.

Normalization. We normalize every CHC by applying the following rewriting steps: (i) We ensure that every relation symbol occurs at most once in every clause. For instance, the CHC $q(a) \leftarrow q(b) \wedge q(c)$ has multiple occurrences of the relation symbol q, and we

Table 2
Control- and data-flow sub-formula in constraints for the normalized CHCs from Figure 1c

Normalized CHCs	Control-flow sub-formula	Data-flow sub-formula
$L(x,y,n) \leftarrow n \geq 0 \land x = n \land y = n$	$n \geq 0$	$x = n, y = n$
$L(x,y,n) \leftarrow L'(x',y',n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1$	$x \neq 0$	$x = x' - 1, y = y' - 1$
$L'(x',y',n') \leftarrow L(x,y,n) \land x' = x \land y' = y \land n' = n$	empty	$x' = x, y' = y, n' = n$
false $\leftarrow L(x,y,n) \land x = 0 \land y \neq 0$	$y \neq 0$	$x = 0$

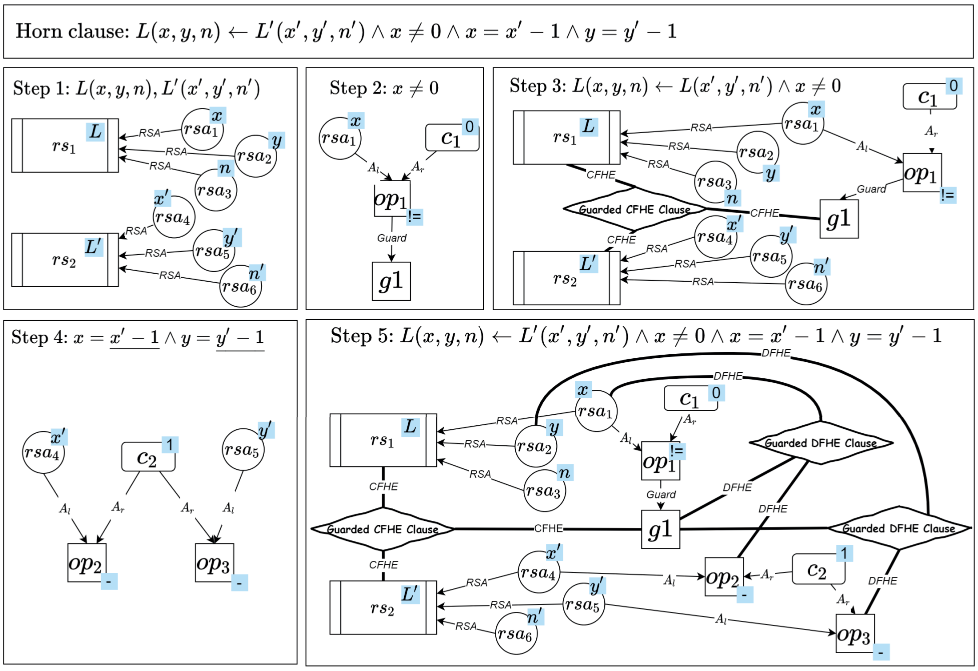
normalize it to equi-satisfiable CHCs $q(a) \leftarrow q'(b) \land q''(c), q'(b) \leftarrow q(b') \land b = b'$ and $q''(c) \leftarrow q(c') \land c = c'$. (ii) We associate each relation symbol q with a unique vector of pair-wise distinct argument variables $\bar{x}_q$, and rewrite every occurrence of q to the form $q(\bar{x}_q)$. In addition, all the argument vectors $\bar{x}_q$ are kept disjoint. The normalized CHCs from Figure 1c are shown in Table 2.

Splitting constraints into control- and data-flow formulas. We can rewrite the constraint φ to a conjunction $\varphi = \varphi_1 \land \cdots \land \varphi_k,\ k \in \mathbb{N}$. The sub-formula φ_i is called a "data-flow sub-formula" if and only if it can be rewritten to the form $x = t(\bar{y})$ such that (i) x is one of the arguments in head $q(\bar{x}_q)$; (ii) $t(\bar{y})$ is a term over variables $\bar{y}$, where each element of $\bar{y}$ is an argument of some body literal $q'(\bar{x}_{q'})$. We call all other φ_j "control-flow sub-formulas". A constraint φ can then be represented by $\varphi = g_1 \land \cdots \land g_m \land d_1 \land \cdots \land d_n$, where $m, n \in \mathbb{N}$ and g_i and d_j are the control- and data-flow sub-formulas, respectively. The control and data flow sub-formulas for the normalized CHCs of our running example are shown in Table 2.

Constructing a CDHG. The CDHG represents program control- and data-flow by guarded control-flow hyperedges *CFHE* and data-flow hyperedges *DFHE*. A *CFHE* edge denotes the flow of control from the body to head literals of the CHC. A *DFHE* edge denotes how data flows from the body to the head. Both control- and data-flow are guarded by the control flow sub-formula.

Constructing the CDHG for a normalized CHC $L(x,y,n) \leftarrow L'(x',y',n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1$ is shown in Figure 3. The corresponding node and edge types are described in Tables 5 and 6 in the full version of this paper [28].

In the first step, we draw relation symbols and their arguments and build the relation between them. In the second step, we add a *guard* node and draw ASTs for the control flow sub-formulas. In the third step, we construct guarded control-flow edges by connecting the relation symbols in the head and body and the *guard* node, which connects the root of control flow sub-formulas. In the fourth step, we construct the ASTs for the right-hand side of every data flow sub-formula. In the fifth step, we construct the guarded data-flow edges by connecting the left- and right-hand sides of the data flow sub-formulas and the *guard* node. Note that the diamond shapes in Figure 3 are not nodes in the graph but are used to visualize our (ternary) hyperedges of types *CFHE* and *DFHE*. We show it in



Horn clause: $L(x,y,n) \leftarrow L'(x',y',n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1$
Step 1: $L(x,y,n), L'(x',y',n')$
Step 2: $x \neq 0$
Step 3: $L(x,y,n) \leftarrow L(x',y',n') \land x \neq 0$
Step 4: $x = x' - 1 \land y = y' - 1$
Step 5: $L(x,y,n) \leftarrow L'(x',y',n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1$
RSA
CFHE
DFHE
Guard
Guarded CFHE Clause
Guarded DFHE Clause
A_l
A_r

Figure 3: Construct the CDHG from the CHC $L(x,y,n) \leftarrow L'(x',y',n') \land x \neq 0 \land x = x' - 1 \land y = y' - 1$.

this way to visualize CDHG better.

4. Relational Hypergraph Neural Network

Different from regular graphs, which connect nodes by binary edges, CDHG includes hyperedges which connect arbitrary number of nodes. Therefore, we extend R-GCN to R-HyGNN to handle hypergraphs. The updating rule for node representation at time step t in R-HyGNN is

$$h_v^t = \mathrm{ReLU}(\sum_{r\in R}\sum_{p\in P_r}\sum_{e\in E_v^{r,p}} W_{r,p}^t \cdot \|[h_u^{t-1} \mid u \in e]), \tag{2}$$

where $\|\{\cdot\}$ denotes concatenation of all elements in a set, $r \in R = \{r_i \mid i \in \mathbb{N}\}$ is the set of edge types (relations), $p \in P_r = \{p_j \mid j \in \mathbb{N}\}$ is the set of node positions under edge type r, $W_{r,p}^t$ denotes learnable parameters when the node is in the pth position with edge type r, and $e \in E_v^{r,p}$ is the set of hyperedges of type r in the graph in which node v appears in position p, where e is a list of nodes. The updating process for the representation of node v at time step 1 is illustrated in Figure 4.

Note that different edge types may have the same number of connected nodes. For instance, in CDHG, *CFHE* and *DFHE* are both ternary edges.

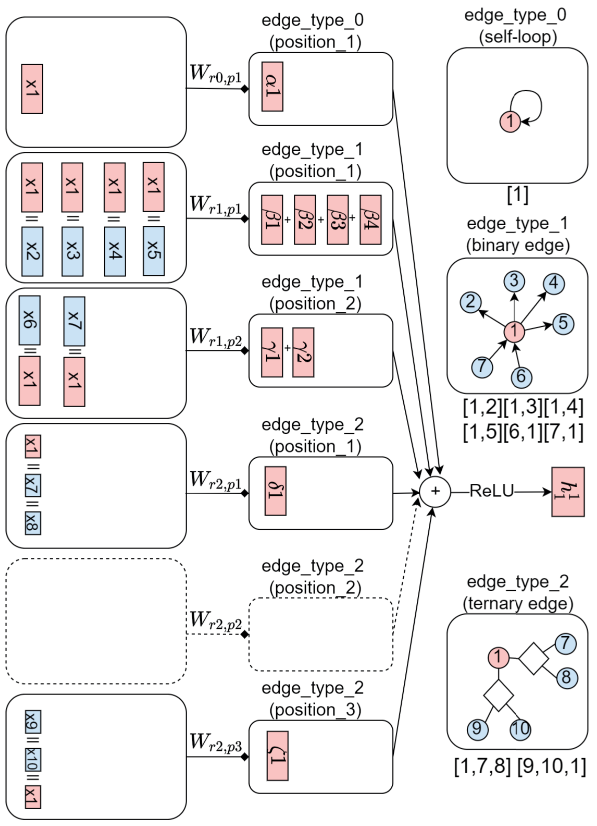


Figure 4: An example to illustrate how to update node representation for R-HyGNN using (2). At the right-hand side, there are three types of edges connected with node 1. We compute the updated representation h_1^1 for node 1 at the time step 1. $\|$ means concatenation. x_i is the initial feature vector of node i. The red blocks are the trace of the updating for node 1. The edge type 1 is a unary edge and is a self-loop. It has one set of learnable parameters as the update function i.e., $W_{r0,p1}$. The edge type 2 is binary edge, it has two update functions i.e., $W_{r1,p1}$ and $W_{r1,p2}$. Node 1 is in the first position in edge [1,2], [1,3], [1,4], and [1,5], so the concatenated node representation will be updated by $W_{r1,p1}$. On the other hand, for the other two edges [6,1] and [7,1], node 1 is in the second position, so the concatenated node representation will be updated by $W_{r1,p2}$. For edge type 3, the same rule applies, i.e., depending on node 1's position in the edge, the concatenated node representation will go through different parameter sets. Since there is no edge that node 1 is in the second position, we use a dashed box and arrow to represent it. The aggregation is to add all updated representations from different edge types.

Overall, our definition of R-HyGNN is a generalization of R-GCN. Concretely, it can directly be applied to the special-case of binary graphs, and in that setting is slightly more powerful as each message between nodes is computed using the representations of both source and target nodes, whereas R-GCN only uses the source node representation.

4.1. Training Model

The end-to-end model consists of three components: the first component is an embedding layer, which learns to map the node's type (encoded by integers) to the initial feature vectors; the second component is R-HyGNN, which learns program features; the third component is a set of fully connected neural networks, which learns to solve a specific task by using gathered node representations from R-HyGNNs. All parameters in these three components are trained together. Note that different tasks may gather different node representations. For example, Task 1 gathers all node representations, while Task 2 only gathers node representations with type rs.

We set all vector lengths to 64, i.e., the embedding layer output size, the middle layers' neuron size in R-HyGNNs, and the layer sizes in fully connected neural networks are all 64. The maximum training epoch is 500, and the patient is 100. The number of message passing steps is 8 (i.e., (2) is applied eight times). For the rest of the parameters (e.g., learning rate, optimizer, dropout rate, etc.), we use the default setting in the tf2_gnn framework. We set these parameters empirically according to the graph size and the structure. We apply these fixed parameter settings for all tasks and two graph representations without fine-tuning.

5. Proxy Tasks

We propose five proxy tasks with increasing difficulty to systematically evaluate the R-HyGNN on the two graph representations. Tasks 1 to 3 evaluate if R-HyGNN can solve general problems in graphs. In contrast, Tasks 4 and 5 evaluate if combining our graph representations and R-HyGNN can learn program features to solve the encoded program verification problems. We first describe the learning target for every task and then explain how to produce training labels and discuss the learning difficulty.

Task 1: Argument identification. For both graph representations, the R-HyGNN model performs binary classification on all nodes to predict if the node type is a relation symbol argument (rsa) and the metric is accuracy. The binary training label is obtained by reading explicit node types. This task evaluates if R-HyGNN can differentiate explicit node types. This task is easy because the graph explicitly includes the node type information in both typed nodes and edges.

Task 2: Count occurrence of relation symbols in all clauses. For both graph representations, the R-HyGNN model performs regression on nodes with type rs to predict how many times the relation symbols occur in all clauses. The metric is mean square error. The training label is obtained by counting the occurrence of every relation symbol in all clauses. This task is designed to see if R-HyGNN can correctly perform a counting task. For example, the relation symbol L occurs four times in all CHCs in Figure 2, so the training label for node L is value 4. This task is harder than Task 1 since it needs to count the connected binary edges or hyperedges for a particular node.

Task 3: Relation symbol occurrence in SCCs. For both graph representations, the R-HyGNN model performs binary classification on nodes with type rs to predict if this node is an SCC (i.e., in a cycle) and the metric is accuracy. The binary training label is obtained using Tarjan's algorithm [19]. For example, in Figure 2, L is an SCC because L and L' construct a cycle by $L \leftarrow L'$ and $L' \leftarrow L$. This task is designed to evaluate if R-HyGNN can recognize general graph structures such as cycles. This task requires the model to classify a graph-theoretic object (SCC), which is harder than the previous two tasks since it needs to approximate a concrete algorithm rather than classifying or counting explicit graphical elements.

Task 4: Existence of argument bounds. For both graph representations, we train two independent R-HyGNN models which perform binary classification on nodes with type rsa to predict if individual arguments have (a) lower and (b) upper bounds in the least solution of a set of CHCs, and the metric is accuracy. To obtain the training label, we apply the Horn solver Eldarica to check the correctness of guessed (and successively increased) lower and upper arguments bounds; arguments for which no bounds can be shown are assumed to be unbounded. We use a timeout of 3 s for the lower and upper bound of a single argument, respectively. The overall timeout for extracting labels from one program is 3 hours. For example, consider the CHCs in Fig. 1c. The CHCs contain a single relation symbol L; all three arguments of L are bounded from below but not from above. This task is (significantly) harder than the previous three tasks, as boundedness of arguments is an undecidable property that can, in practice, be approximated using static analysis methods.

Task 5: Clause occurrence in counter-examples This task consists of two binary classification tasks on nodes with type *guard* (for CDHG), and with type *clause* (for constraint graph) to predict if a clause occurs in the counter-examples. Those kinds of nodes are unique representatives of the individual clauses of a problem. The task focuses on unsatisfiable sets of CHCs. Every unsatisfiable clause set gives rise to a set of minimal unsatisfiable subsets (MUSes); MUSes correspond to the minimal CEs of the clause set. Two models are trained independently to predict whether a clause belongs to (a) the intersection or (b) the union of the MUSes of a clause set. The metric for this task is accuracy. We obtain training data by applying the Horn solver Eldarica [25], in combination with an optimization library that provides an algorithm to compute MUSes[7]. This task is hard, as it attempts the prediction of an uncomputable binary labelling of the graph.

6. Evaluation

We first describe the dataset we use for the training and evaluation and then analyse the experiment results for the five proxy tasks.

[7]https://github.com/uuverifiers/lattice-optimiser/

Table 3
The number of labeled graph representations extracted from a collection of CHC-COMP benchmark [20]. For each SMT-LIB file, the graph representations for Task 1,2,3,4 are extracted together using the timeout of 3 hours, and for task 5 is extracted using 20 minutes timeout. Here, T. denotes Task.

	SMT-LIB files		Constraint graphs			CDHGs		
	Total	Unsat	T. 1-4	T. 5 (a)	T. 5 (b)	T. 1-4	T. 5 (a)	T. 5 (b)
Linear LIA	8705	1659	2337	881	857	3029	880	883
Non-linear LIA	8425	3601	3376	1141	1138	4343	1497	1500
Aligned	17130	5260	5602	1927	1914	5602	1927	1914

6.1. Benchmarks and Dataset

Table 3 shows the number of labelled graph representations from a collection of CHC-COMP benchmarks [20]. All graphs were constructed by first running the preprocessor of Eldarica [25] on the clauses, then building the graphs as described in Section 3, and computing training data. For instance, in the first four tasks we constructed 2337 constraint graphs with labels from 8705 benchmarks in the CHC-COMP LIA-Lin track (linear Horn clauses over linear integer arithmetic). The remaining 6368 benchmarks are not included in the learning dataset because when we construct the graphs, (1) the data generation process timed out, or (2) the graphs were too big (more than 10,000 nodes), or (3) there was no clause left after simplification. In Task 5, since the label is mined from CEs, we first need to identify unsat benchmarks using a Horn solver (1-hour timeout), and then construct graph representations. We obtain 881 and 857 constraint graphs when we form the labels for Task 5 (a) and (b), respectively, in LIA-Lin.

Finally, to compare the performance of the two graph representations, we align the dataset for both two graph representations to have 5602 labelled graphs for the first four tasks. For Task 5 (a) and (b), we have 1927 and 1914 labelled graphs, respectively. We divide them to train, valid, and test sets with ratios of 60%, 20%, and 20%. We include all corresponding files for the dataset in a Github repository [8].

6.2. Experimental Results for Five Proxy Tasks

From Table 4, we can see that for all binary classification tasks, the accuracy for both graph representations is higher than the ratios of the dominant labels. For the regression task, the scattered points are close to the diagonal line. These results show that R-HyGNN can learn the syntactic and semantic information for the tasks rather than performing simple strategies (e.g., fill all likelihood by 0 or 1). Next, we analyse the experimental results for every task.

Task 1: Argument identification. When the task is performed in the constraint graph, the accuracy of prediction is 100%, which means R-HyGNN can perfectly differentiate if a node is a relation symbol argument rsa node. When the task is performed in CDHG, the accuracy is close to 100% because, unlike in the constraint graph, the number of incoming

[8]https://github.com/ChenchengLiang/Horn-graph-dataset

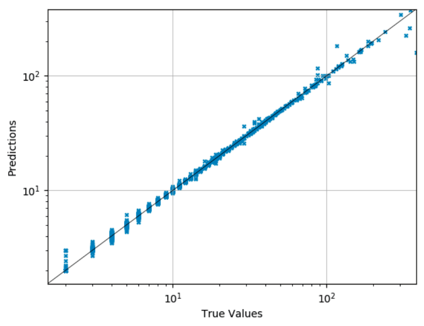
and outgoing edges are fixed (i.e., RSA and AI), the rsa nodes in CDHG may connect with a various number of edges (including RSA, AST_1, AST_2, and $DFHE$) which makes R-HyGNN hard to predict the label precisely.

Besides, the data distribution looks very different between the two graph representations because the normalization of CHCs introduces new clauses and arguments. For example, in the simplified CHCs in Figure 1c, there are three arguments for the relation symbol L, while in the normalized clauses in Figure 2, there are six arguments for two relation symbols L and L'. If the relation symbols have a large number of arguments, the difference in data distribution between the two graph representations becomes larger. Even though the predicted label in this task cannot directly help solve the CHC-encoded problem, it is important to study the message flows in the R-HyGNNs.

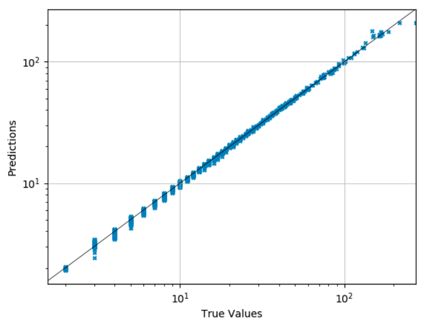
Task 2: Count occurrence of relation symbols in all clauses. In the scattered plots in Figure 5, the x- and y-axis denote true and the predicted values in the logarithm scale, respectively. The closer scattered points are to the diagonal line, the better performance of predicting the number of relation symbol occurrences in CHCs. Both CDHG and constraint graph show good performance (i.e., most of the scattered points are near the diagonal lines). This syntactic information can be obtained by counting the *CFHE* and *RSI* edges for CDHG and constraint graph, respectively. When the number of nodes is large, the predicted values are less accurate. We believe this is because graphs with a large number of nodes have a more complex structure, and there is less training data. Moreover, the mean square error for the CDHG is larger than the constraint graph because normalization increases the number of nodes and the maximum counting of relation symbols for CDHG, and the larger range of the value is, the more difficult for regression task. Notice that the number of test data (1115) for this task is less than the data in the test set (1121) shown in Table 3 because the remaining six graphs do not have a rs node.

Task 3: Relation symbol occurrence in SCCs. The predicted high accuracy for both graph representations shows that our framework can approximate Tarjan's algorithm [19]. In contrast to Task 2, even if the CDHG has more nodes than the constraint graph on average, the CDHG has better performance than the constraint graph , which means the control and data flow in CDHG can help R-HyGNN to learn graph structures better. For the same reason as task 2, the number of test data (1115) for this task is less than the data in the test set (1121).

Task 4: Existence of argument bounds. For both graph representations, the accuracy is much higher than the ratio of the dominant label. Our framework can predict the answer for undecidable problems with high accuracy, which shows the potential for guiding CHC solvers. The CDHG has better performance than the constraint graph, which might be because predicting argument bounds relies on semantic information. The number of test data (1028) for this task is less than the data in the test set (1121) because the remaining 93 graphs do not have a rsa node.



(a) Scatter plot for CDHG. The total rs node number is 16858. The mean square error is 4.22.



(b) Scatter plot for constraint graph. The total rs node number is 11131. The mean square error is 1.04.

Figure 5: Scatter plot for Task 2. The x- and y-axis are in logarithmic scales.

Task 5: Clause occurrence in counter-examples. For Task (a) and (b), the overall accuracy for two graph representations is high. We manually analysed some predicted results by visualizing the (small) graphs[9]. We identify some simple patterns that are learned by R-HyGNNs. For instance, the predicted likelihoods are always high for the rs nodes connected to the *false* nodes. One promising result is that the model can predict all labels perfectly for some big graphs[10] that contain more than 290 clauses, which confirms that the R-HyGNN is learning certain intricate patterns rather than simple patterns. In addition, the CDHG has better performance than the constraint graph , possibly because semantic information is more important for solving this task.

7. Related Work

Learning to represent programs. Contextual embedding methods (e.g. transformer [29], BERT [30], GPT [31], etc.) achieved impressive results in understanding natural languages. Some methods are adapted to explore source code understanding in text format (e.g. CodeBERT [32], cuBERT [33], etc.). But, the programming language usually contains rich, explicit, and complicated structural information, and the problem sets (learning targets) of it [34, 35] are different from natural languages. Therefore, the way of representing the programs and learning models should adapt to the programs' characteristics. Recently, the frameworks consisting of structural program representations (graph or AST) and graph or tree-based deep learning models made good progress in solving program language-related problems. For example, [36] represents the program by a sequence of code subtokens

[9] https://github.com/ChenchengLiang/Horn-graph-dataset/tree/main/example-analysis/task5-small-graphs

[10] https://github.com/ChenchengLiang/Horn-graph-dataset/tree/main/example-analysis/task5-big-graphs

Table 4
Experiment results for Tasks 1,3,4,5. Both the fourth and fifth tasks consist of two independent binary classification tasks. Here, $+$ and $-$ stands for the positive and negative label. The T and P represent the true and predicted labels. The Acc. is the accuracy of binary classifications. The Dom. is dominant label ratio. Notice that even if the two graph representations originate from the same original CHCs, the label distributions are different since the CDHG is constructed from normalized CHCs.

			Constraint graph				CDHG			
Task	Files	T \ P	+	-	Acc.	Dom.	+	-	Acc.	Dom.
1	1121	+	93863	0	100%	95.1%	142598	0	99.9%	72.8%
		-	0	1835971			10	381445		
3	1115	+	3204	133	96.1%	70.1%	8262	58	99.6%	50.7%
		-	301	7493			15	8523		
4 (a)	1028	+	13685	5264	91.2%	79.7%	30845	4557	94.3%	75.2%
		-	2928	71986			3566	103630		
4 (b)		+	18888	4792	91.4%	74.8%	41539	4360	94.3%	67.8%
		-	3291	66892			3715	92984		
5 (a)	386	+	1048	281	95.0%	84.7%	1230	206	96.9%	86.4%
		-	154	7163			121	9036		
5 (b)	383	+	3030	558	84.6%	53.1%	3383	481	90.6%	54.8%
		-	622	3428			323	4361		

and predicts source code snippets summarization by a novel convolutional attention network. Code2vec [37] learns the program from the paths in its AST and predicts semantic properties for the program using a path-based attention model. [38] use AST to represent the program and classify programs according to functionality using the tree-based convolutional neural network (TBCNN). Some studies focus on defining efficient program representations, others focus on introducing novel learning structures, while we do both of them (i.e. represent the CHC-encoded programs by two graph representations and propose a novel GNN structure to learn the graph representations).

Deep learning for logic formulas. Since deep learning is introduced to learn the features from logic formulas, an increasing number of studies have begun to explore graph representations for logic formulas and corresponding learning frameworks because logic formulas are also highly structured like program languages. For instance, DeepMath [39] had an early attempt to use text-level learning on logic formulas to guide the formal method's search process, in which neural sequence models are used to select premises for automated theorem prover (ATP). Later on, FormulaNet [40] used an edge-order-preserving embedding method to capture the structural information of higher-order logic (HOL) formulas represented in a graph format. As an extension of FormulaNet, [41] construct syntax trees of HOL formulas as structural inputs and use message-passing GNNs to learn features of HOL to guide theorem proving by predicting tactics and tactic arguments at every step of the proof. LERNA [42] uses convolutional neural networks (CNNs) [43] to learn previous proof search attempts (logic formulas) represented by graphs to guide the current proof search for ATP. NeuroSAT [44, 45] reads SAT queries (logic formulas) as graphs and learns the features using different graph embedding strategies (e.g. message passing GNNs) [46, 47, 26]) to directly predict the satisfiability or guide the SAT solver. Following this trend, we introduce R-HyGNN to learn the program features from the graph representation of CHCs.

Graph neural networks. Message passing GNNs [26], such as graph convolutional network (GCN) [17], graph attention network (GAT) [48], and gated graph neural network (GGNN) [47] have been applied in several domains ranging from predicting molecule properties to learning logic formula representations. However, these frameworks only apply to graphs with binary edges. Some spectral methods have been proposed to deal with the hypergraph [49, 50]. For instance, the hypergraph neural network (HGNN) [51] extends GCN proposed by [52] to handle hyperedges. The authors in [53] integrate graph attention mechanism [48] to hypergraph convolution [52] to further improve the performance. But, they cannot be directly applied to the spatial domain. Similar to the fixed arity predicates strategy described in LambdaNet [18], R-HyGNN concatenates node representations connected by the hyperedge and updates the representation depending on the node's position in the hyperedge.

8. Conclusion and Future Work

In this work, we systematically explore learning program features from CHCs using R-HyGNN, using two tailor-made graph representations of CHCs. We use five proxy tasks to evaluate the framework. The experimental results indicate that our framework has the potential to guide CHC solvers analysing Horn clauses. In future work, among others we plan to use this framework to filter predicates in Horn solvers applying the CEGAR model checking algorithm.

References

[1] J. Leroux, P. Rümmer, P. Subotić, Guiding Craig interpolation with domain-specific abstractions, Acta Informatica 53 (2016) 387–424. URL: https://doi.org/10.1007/s00236-015-0236-z. doi:10.1007/s00236-015-0236-z.

[2] Y. Demyanova, P. Rümmer, F. Zuleger, Systematic predicate abstraction using variable roles, in: C. Barrett, M. Davies, T. Kahsai (Eds.), NASA Formal Methods, Springer International Publishing, Cham, 2017, pp. 265–281.

[3] E. Clarke, O. Grumberg, S. Jha, Y. Lu, H. Veith, Counterexample-guided abstraction refinement, in: E. A. Emerson, A. P. Sistla (Eds.), Computer Aided Verification, Springer Berlin Heidelberg, Berlin, Heidelberg, 2000, pp. 154–169.

[4] V. Tulsian, A. Kanade, R. Kumar, A. Lal, A. V. Nori, MUX: Algorithm selection for software model checkers, in: Proceedings of the 11th Working Conference on Mining Software Repositories, MSR 2014, Association for Computing Machinery, New

York, NY, USA, 2014, p. 132–141. URL: https://doi.org/10.1145/2597073.2597080. doi:10.1145/2597073.2597080.

[5] Y. Demyanova, T. Pani, H. Veith, F. Zuleger, Empirical software metrics for benchmarking of verification tools, in: J. Knoop, U. Zdun (Eds.), Software Engineering 2016, Gesellschaft für Informatik e.V., Bonn, 2016, pp. 67–68.

[6] C. Richter, E. Hüllermeier, M.-C. Jakobs, H. Wehrheim, Algorithm selection for software validation based on graph kernels, Automated Software Engineering 27 (2020) 153–186.

[7] C. Richter, H. Wehrheim, Attend and represent: A novel view on algorithm selection for software verification, in: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020, pp. 1016–1028.

[8] D. P. Kingma, M. Welling, Auto-encoding variational Bayes, 2013. URL: https://arxiv.org/abs/1312.6114. doi:10.48550/ARXIV.1312.6114.

[9] H. Dai, Y. Tian, B. Dai, S. Skiena, L. Song, Syntax-directed variational autoencoder for structured data, 2018. URL: https://arxiv.org/abs/1802.08786. doi:10.48550/ARXIV.1802.08786.

[10] X. Si, A. Naik, H. Dai, M. Naik, L. Song, 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, Springer-Verlag, Berlin, Heidelberg, 2020, p. 151–164. URL: https://doi.org/10.1007/978-3-030-53291-8_9. doi:10.1007/978-3-030-53291-8_9.

[11] P. W. Battaglia, J. B. Hamrick, V. Bapst, A. Sanchez-Gonzalez, V. F. Zambaldi, M. Malinowski, A. Tacchetti, D. Raposo, A. Santoro, R. Faulkner, Ç. Gülçehre, H. F. Song, A. J. Ballard, J. Gilmer, G. E. Dahl, A. Vaswani, K. R. Allen, C. Nash, V. Langston, C. Dyer, N. Heess, D. Wierstra, P. Kohli, M. Botvinick, O. Vinyals, Y. Li, R. Pascanu, Relational inductive biases, deep learning, and graph networks, CoRR abs/1806.01261 (2018). URL: http://arxiv.org/abs/1806.01261. arXiv:1806.01261.

[12] T. Ben-Nun, A. S. Jakobovits, T. Hoefler, Neural code comprehension: A learnable representation of code semantics, CoRR abs/1806.07336 (2018). URL: http://arxiv.org/abs/1806.07336. arXiv:1806.07336.

[13] C. Lattner, V. Adve, LLVM: a compilation framework for lifelong program analysis amp; transformation, in: International Symposium on Code Generation and Optimization, 2004. CGO 2004., 2004, pp. 75–86. doi:10.1109/CGO.2004.1281665.

[14] T. Mikolov, M. Karafiát, L. Burget, J. H. Cernocký, S. Khudanpur, Recurrent neural network based language model, in: INTERSPEECH, 2010.

[15] A. Horn, On sentences which are true of direct unions of algebras, The Journal of Symbolic Logic 16 (1951) 14–21. URL: http://www.jstor.org/stable/2268661.

[16] M. Olšák, C. Kaliszyk, J. Urban, Property invariant embedding for automated reasoning, CoRR abs/1911.12073 (2019). URL: http://arxiv.org/abs/1911.12073. arXiv:1911.12073.

[17] M. Schlichtkrull, T. N. Kipf, P. Bloem, R. v. d. Berg, I. Titov, M. Welling, Modeling relational data with graph convolutional networks, 2017. URL: https://arxiv.org/abs/1703.06103. doi:10.48550/ARXIV.1703.06103.

[18] J. Wei, M. Goyal, G. Durrett, I. Dillig, LambdaNet: Probabilistic type inference

using graph neural networks, 2020. arXiv:2005.02161.

[19] R. E. Tarjan, Depth-first search and linear graph algorithms, SIAM J. Comput. 1 (1972) 146–160.

[20] G. Fedyukovich, P. Rümmer, Competition report: CHC-COMP-21, 2021. URL: https://chc-comp.github.io/2021/report.pdf.

[21] P. Rümmer, H. Hojjat, V. Kuncak, Disjunctive interpolants for Horn-clause verification (extended technical report), 2013. arXiv:1301.4973.

[22] R. W. Floyd, Assigning meanings to programs, Proceedings of Symposium on Applied Mathematics 19 (1967) 19–32. URL: http://laser.cs.umass.edu/courses/cs521-621.Spr06/papers/Floyd.pdf.

[23] C. A. R. Hoare, An axiomatic basis for computer programming, Commun. ACM 12 (1969) 576–580. URL: https://doi.org/10.1145/363235.363259. doi:10.1145/363235.363259.

[24] N. Bjørner, A. Gurfinkel, K. McMillan, A. Rybalchenko, Horn clause solvers for program verification, 2015, pp. 24–51. doi:10.1007/978-3-319-23534-9_2.

[25] H. Hojjat, P. Ruemmer, The ELDARICA Horn solver, in: 2018 Formal Methods in Computer Aided Design (FMCAD), 2018, pp. 1–7. doi:10.23919/FMCAD.2018.8603013.

[26] J. Gilmer, S. S. Schoenholz, P. F. Riley, O. Vinyals, G. E. Dahl, Neural message passing for quantum chemistry, CoRR abs/1704.01212 (2017). URL: http://arxiv.org/abs/1704.01212. arXiv:1704.01212.

[27] K. Xu, W. Hu, J. Leskovec, S. Jegelka, How powerful are graph neural networks?, CoRR abs/1810.00826 (2018). URL: http://arxiv.org/abs/1810.00826. arXiv:1810.00826.

[28] C. Liang, P. Rümmer, M. Brockschmidt, Exploring representation of horn clauses using gnns, 2022. URL: https://arxiv.org/abs/2206.06986. doi:10.48550/ARXIV.2206.06986.

[29] A. Vaswani, N. Shazeer, N. Parmar, J. Uszkoreit, L. Jones, A. N. Gomez, L. Kaiser, I. Polosukhin, Attention is all you need, 2017. arXiv:1706.03762.

[30] J. Devlin, M. Chang, K. Lee, K. Toutanova, BERT: pre-training of deep bidirectional transformers for language understanding, CoRR abs/1810.04805 (2018). URL: http://arxiv.org/abs/1810.04805. arXiv:1810.04805.

[31] A. Radford, K. Narasimhan, Improving language understanding by generative pre-training, 2018.

[32] Z. Feng, D. Guo, D. Tang, N. Duan, X. Feng, M. Gong, L. Shou, B. Qin, T. Liu, D. Jiang, M. Zhou, CodeBERT: A pre-trained model for programming and natural languages, CoRR abs/2002.08155 (2020). URL: https://arxiv.org/abs/2002.08155. arXiv:2002.08155.

[33] A. Kanade, P. Maniatis, G. Balakrishnan, K. Shi, Pre-trained contextual embedding of source code, CoRR abs/2001.00059 (2020). URL: http://arxiv.org/abs/2001.00059. arXiv:2001.00059.

[34] H. Husain, H. Wu, T. Gazit, M. Allamanis, M. Brockschmidt, CodeSearchNet challenge: Evaluating the state of semantic code search, CoRR abs/1909.09436 (2019). URL: http://arxiv.org/abs/1909.09436. arXiv:1909.09436.

[35] S. Lu, D. Guo, S. Ren, J. Huang, A. Svyatkovskiy, A. Blanco, C. B. Clement, D. Drain, D. Jiang, D. Tang, G. Li, L. Zhou, L. Shou, L. Zhou, M. Tufano, M. Gong, M. Zhou, N. Duan, N. Sundaresan, S. K. Deng, S. Fu, S. Liu, CodeXGLUE: A machine learning benchmark dataset for code understanding and generation, CoRR abs/2102.04664 (2021). URL: https://arxiv.org/abs/2102.04664. arXiv:2102.04664.

[36] M. Allamanis, H. Peng, C. Sutton, A convolutional attention network for extreme summarization of source code, CoRR abs/1602.03001 (2016). URL: http://arxiv.org/abs/1602.03001. arXiv:1602.03001.

[37] U. Alon, M. Zilberstein, O. Levy, E. Yahav, Code2vec: Learning distributed representations of code, Proc. ACM Program. Lang. 3 (2019) 40:1–40:29. URL: http://doi.acm.org/10.1145/3290353. doi:10.1145/3290353.

[38] L. Mou, G. Li, Z. Jin, L. Zhang, T. Wang, TBCNN: A tree-based convolutional neural network for programming language processing, CoRR abs/1409.5718 (2014). URL: http://arxiv.org/abs/1409.5718. arXiv:1409.5718.

[39] G. Irving, C. Szegedy, A. A. Alemi, N. Een, F. Chollet, J. Urban, DeepMath - deep sequence models for premise selection, in: D. D. Lee, M. Sugiyama, U. V. Luxburg, I. Guyon, R. Garnett (Eds.), Advances in Neural Information Processing Systems 29, Curran Associates, Inc., 2016, pp. 2235–2243. URL: http://papers.nips.cc/paper/6280-deepmath-deep-sequence-models-for-premise-selection.pdf.

[40] M. Wang, Y. Tang, J. Wang, J. Deng, Premise selection for theorem proving by deep graph embedding, in: I. Guyon, U. V. Luxburg, S. Bengio, H. Wallach, R. Fergus, S. Vishwanathan, R. Garnett (Eds.), Advances in Neural Information Processing Systems 30, Curran Associates, Inc., 2017, pp. 2786–2796. URL: http://papers.nips.cc/paper/6871-premise-selection-for-theorem-proving-by-deep-graph-embedding.pdf.

[41] A. Paliwal, S. M. Loos, M. N. Rabe, K. Bansal, C. Szegedy, Graph representations for higher-order logic and theorem proving, CoRR abs/1905.10006 (2019). URL: http://arxiv.org/abs/1905.10006. arXiv:1905.10006.

[42] M. Rawson, G. Reger, A neurally-guided, parallel theorem prover, in: A. Herzig, A. Popescu (Eds.), Frontiers of Combining Systems, Springer International Publishing, Cham, 2019, pp. 40–56.

[43] A. Krizhevsky, I. Sutskever, G. E. Hinton, ImageNet classification with deep convolutional neural networks, in: F. Pereira, C. J. C. Burges, L. Bottou, K. Q. Weinberger (Eds.), Advances in Neural Information Processing Systems, volume 25, Curran Associates, Inc., 2012. URL: https://proceedings.neurips.cc/paper/2012/file/c399862d3b9d6b76c8436e924a68c45b-Paper.pdf.

[44] D. Selsam, N. Bjørner, Neurocore: Guiding high-performance SAT solvers with unsat-core predictions, CoRR abs/1903.04671 (2019). URL: http://arxiv.org/abs/1903.04671. arXiv:1903.04671.

[45] D. Selsam, M. Lamm, B. Bünz, P. Liang, L. de Moura, D. L. Dill, Learning a SAT solver from single-bit supervision, CoRR abs/1802.03685 (2018). URL: http://arxiv.org/abs/1802.03685. arXiv:1802.03685.

[46] F. Scarselli, M. Gori, A. C. Tsoi, M. Hagenbuchner, G. Monfardini, The graph neural network model, IEEE Transactions on Neural Networks 20 (2009) 61–80. doi:10.1109/TNN.2008.2005605.

[47] Y. Li, D. Tarlow, M. Brockschmidt, R. Zemel, Gated graph sequence neural networks, 2015. URL: https://arxiv.org/abs/1511.05493. doi:10.48550/ARXIV.1511.05493.

[48] P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, Y. Bengio, Graph attention networks, 2017. URL: https://arxiv.org/abs/1710.10903. doi:10.48550/ARXIV.1710.10903.

[49] H. Gui, J. Liu, F. Tao, M. Jiang, B. Norick, J. Han, Large-scale embedding learning in heterogeneous event data, 2016, pp. 907–912. doi:10.1109/ICDM.2016.0111.

[50] K. Tu, P. Cui, X. Wang, F. Wang, W. Zhu, Structural deep embedding for hyper-networks, CoRR abs/1711.10146 (2017). URL: http://arxiv.org/abs/1711.10146. arXiv:1711.10146.

[51] Y. Feng, H. You, Z. Zhang, R. Ji, Y. Gao, Hypergraph neural networks, CoRR abs/1809.09401 (2018). URL: http://arxiv.org/abs/1809.09401. arXiv:1809.09401.

[52] T. N. Kipf, M. Welling, Semi-supervised classification with graph convolutional networks, CoRR abs/1609.02907 (2016). URL: http://arxiv.org/abs/1609.02907. arXiv:1609.02907.

[53] S. Bai, F. Zhang, P. H. S. Torr, Hypergraph convolution and hypergraph attention, 2020. arXiv:1901.08150.

Paper II

Boosting Constrained Horn Solving by Unsat Core Learning

Parosh Aziz Abdulla[1(✉)], Chencheng Liang[1], and Philipp Rümmer[1,2]

[1] Uppsala University, Uppsala, Sweden
{parosh.abdulla,chencheng.liang}@it.uu.se, philipp.ruemmer@ur.de
[2] University of Regensburg, Regensburg, Germany

Abstract. The Relational Hyper-Graph Neural Network (R-HyGNN) was introduced in [1] to learn domain-specific knowledge from program verification problems encoded in Constrained Horn Clauses (CHCs). It exhibits high accuracy in predicting the occurrence of CHCs in counterexamples. In this research, we present an R-HyGNN-based framework called MUSHyperNet. The goal is to predict the Minimal Unsatisfiable Subsets (MUSes) (i.e., unsat core) of a set of CHCs to guide an abstract symbolic model checking algorithm. In MUSHyperNet, we can predict the MUSes once and use them in different instances of the abstract symbolic model checking algorithm. We demonstrate the efficacy of MUSHyperNet using two instances of the abstract symbolic model-checking algorithm: Counter-Example Guided Abstraction Refinement (CEGAR) and symbolic model-checking-based (SymEx) algorithms. Our framework enhances performance on a uniform selection of benchmarks across all categories from CHC-COMP, solving more problems (6.1% increase for SymEx, 4.1% for CEGAR) and reducing average solving time (13.3% for SymEx, 7.1% for CEGAR).

Keywords: Automatic program verification · Constrained Horn clauses · Graph Neural Networks

1 Introduction

Constrained Horn Clauses (CHCs) [2] are logical formulas that can describe program behaviors and specifications. Encoding program verification problems in CHCs and solving them (checking CHCs' satisfiability) has been an active research area for a number of years [3,4]. If the encoded CHCs are satisfiable, the corresponding program verification problem is safe; if not, it is unsafe.

Solving a set of CHCs means that we either find an interpretation for the predicate (relation) symbols and variables that satisfies all the clauses, or prove that no such interpretation exists. Various techniques, such as Counterexample

Author names in alphabetical order. The theory presented in the paper was developed by Abdulla, Liang, and Rümmer, the implementation is by Liang and Rümmer, evaluation was done by Liang.

Guided Abstraction Refinement (CEGAR) [5] and IC3 [6], have been utilized for this purpose. However, due to the undecidability of solving CHCs, we need carefully designed or tuned heuristics for specific instances.

In this paper, we consider Minimal Unsatisfiable Subsets (MUSes) [7] of sets of CHCs to support the solving process. Given an unsatisfiable set of CHCs, each MUS is a subset that is again unsatisfiable, but removing any CHC from an MUS makes it satisfiable. Understanding MUSes of a set of CHCs can guide solvers to focus on error-prone clauses [8, Section 3.1]: for an unsatisfiable set of CHCs, evaluating the satisfiability of MUSes first can quickly identify unsatisfiable CHCs, eliminating the need for a comprehensive check. In a satisfiable set of CHCs, examining MUSes first can provide a better starting point for refining potentially problematic constraints. This guidance can help the solver converge towards a solution more efficiently. Manually designed heuristics to find MUSes involves summarizing and generalizing the features of a set of CHCs from examples, which can be replaced by data-driven methods.

Various studies that apply deep learning to formal methods for verification have been published in the recent past, e.g., [9–12]. Primarily, deep learning serves as a feature extractor, which can automatically summarize and generalize program features from examples, alleviating the need for manual crafting and tuning of various heuristics. In addition, the idea of representing logic formulas as graphs and using Graph Neural Networks (GNNs) [13] to guide solvers has been employed in many successful studies [14–17].

However, we are not aware of any study that applies GNNs to guide CHC-based symbolic model checking techniques. The main contribution of this paper is to train a GNN model to predict MUSes of a set of CHCs. We train a GNN using the CHC-R-HyGNN framework [1] to predict values between 0 and 1, representing the probabilities of CHCs being elements of MUSes. For example, we assume that a set $\mathcal{C} = \{c_1, c_2, c_3\}$ of CHCs has one MUS $\{c_1, c_2\}$. The predicted probabilities for c_1, c_2, and c_3 being in the MUS could be 0.9, 0.8, and 0.1, respectively. We propose several strategies that use the predicted probability to guide an abstract symbolic model checking algorithm. The strategies can be instantiated for different algorithms on CHCs, such as CEGAR- and symbolic execution-based satisfiability checkers.

Figure 1 depicts an overview of our framework MUSHyperNet. Firstly, we encode the program verification problem into a set of CHCs. Secondly, we use the graph encoder in the CHC-R-HyGNN framework [1] to convert the set of CHCs into a graph format. Then, we train a GNN model named Relational Hypergraph Neural Network (R-HyGNN) to predict the probability of each CHC occurring in MUSes. Finally, we employ the predicted probabilities to guide the abstract symbolic model checking algorithm by determining the sequence for processing each CHC in the set of CHCs.

We utilize the same benchmarks as in [1], comprising 17 130 Linear Integer Arithmetic (LIA) problems from the CHC-COMP 2021 repository [18] for training and evaluating our approach. Further details about the benchmark can be found in [19, Table 1]. The problems for evaluation are uniformly selected

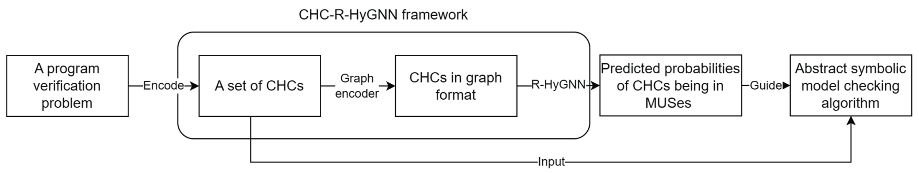

CHC-R-HyGNN framework
A program verification problem
Encode
A set of CHCs
Graph encoder
CHCs in graph format
R-HyGNN
Predicted probabilities of CHCs being in MUSes
Guide
Abstract symbolic model checking algorithm
Input

Fig. 1. The CHC-R-HyGNN framework [1] (represented by the round box) comprises a CHC graph encoder and a GNN known as the Relational Hypergraph Neural Network (R-HyGNN), which is capable of handling hypergraphs. In our previous work, we introduced the CHC-R-HyGNN framework that employs various proxy tasks to generalize valuable information for constructing heuristics for CHC-encoded program verification problems.

from this benchmark and can be found in a public repository [20]. The experimental results show an improvement of up to 4.1% and 6.1% in the number of solved problems for the CEGAR and SymEx algorithms, respectively. Additionally, the average solving time demonstrates enhancements of 7.1% and 13.3% for the CEGAR and SymEx algorithms, respectively. In other words, MUSHyperNet can increase the number of solved problems and decrease the solving time for problems similar to those in the CHC-COMP benchmark. To the best of our knowledge, this is the first time unsat core learning has been used successfully in the context of CHCs.

In summary, our contributions are as follows:

– We develop a GNN-based framework named MUSHyperNet, which trains a GNN to predict the MUSes of CHCs and utilizes the predicted probabilities to guide an abstract symbolic model checking algorithm.
– We explore GNN models trained on different datasets and methods for applying predicted MUSes to guide two instances of the model checking algorithm.
– We evaluate MUSHyperNet on 383 linear and 488 non-linear LIA problems, uniformly sampled from the CHC-COMP benchmark [19]. The improvements in the number of solved problems and average solving time are up to 6.1% and 13.3% for the SymEx, and 4.1% and 7.1% for the CEGAR.

2 Preliminaries

We first introduce required notation for multi-sorted first-order logic, and define Constrained Horn Clauses (CHCs) and the encoding of a program verification problem as CHCs. Finally, we explain basic concepts of Graph Neural Network (GNN) and introduce Relational Hyper Graph Neural Network (R-HyGNN).

2.1 Notations

We assume familiarity with standard multi-sorted first-order logic (e.g., see [21]). A first-order language $\mathcal{L}$ is defined by a signature $\Sigma = (\mathcal{S}, \mathcal{R}, \mathcal{F}, \mathcal{X})$, where $\mathcal{S}$ is a non-empty set of sorts; $\mathcal{R}$ is a set of fixed-arity predicate (relation) symbols, each of which is associated with a list of argument sorts; $\mathcal{F}$ is a set of fixed-arity function symbols, each of which is associated with a list of argument sorts and a result sort; and $\mathcal{X} = \bigcup_{s\in\mathcal{S}} \mathcal{X}_s$ is a set of sorted variables, where $\mathcal{X}_s$ are the variables of sort s. A *term* t is a variable from $\mathcal{X}$, or an n-ary function symbol $f \in \mathcal{F}$ applied to terms $t_1, \ldots, t_n$ of the right sorts. An *atomic formula* (*atom*) of $\mathcal{L}$ is of the form $p(t_1, \ldots, t_n)$, where $p \in \mathcal{R}$ is an n-ary predicate symbol and $t_1, \ldots, t_n$ are terms of the right sorts. A *formula* is a Boolean literal *true*, *false*, an atom, or obtained by applying logical connectives $\neg$, $\wedge$, $\vee$, $\rightarrow$ and quantifiers $\forall$, $\exists$ to formulas. We write implications both left-to-right ($\varphi \rightarrow \psi$) and right-to-left ($\psi \leftarrow \varphi$). A formula is *closed* if all variables occurring in the formula are bound by quantifiers.

A *multi-sorted structure* $\mathcal{M} = (\mathcal{U}, I)$ for $\mathcal{L}$ consists of a set $\mathcal{U} = \bigcup_{s\in\mathcal{S}} \mathcal{U}_s$, being the union of non-empty domains $\mathcal{U}_s$ of each sort $s \in \mathcal{S}$, and an interpretation I such that $I(s) = \mathcal{U}_s$ for every sort $s \in \mathcal{S}$; for each n-ary predicate symbol $p \in \mathcal{R}$ with argument sorts $s_1, \ldots, s_n$, $I(p) \subseteq U_{s_1} \times \cdots \times U_{s_n}$; and for each n-ary function symbol $f \in \mathcal{F}$ with argument sorts $s_1, \ldots, s_n$ and result sort s, $I(f) \in U_{s_1} \times \cdots \times U_{s_n} \rightarrow U_s$. A *variable assignment* β for the structure $\mathcal{M} = (\mathcal{U}, I)$ is a function $\mathcal{X} \rightarrow \mathcal{U}$ that maps each variable $x \in \mathcal{X}_s$ to an element of the corresponding domain $\mathcal{U}_s$. Given $\mathcal{L}$, a structure $\mathcal{M} = (\mathcal{U}, I)$, and a variable assignment β, the *evaluation* of a term or formula is performed by the function $val_{\mathcal{M},\beta}$, defined by $val_{\mathcal{M},\beta}(x) = \beta(x)$ for a variable $x \in \mathcal{X}$; $val_{\mathcal{M},\beta}(f(t_1, \ldots, t_n)) = I(f)[val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_n)]$ for a function $f \in \mathcal{F}$; and $val_{\mathcal{M},\beta}(p(t_1, \ldots, t_n)) = true$ iff $(val_{\mathcal{M},\beta}(t_1), \ldots, val_{\mathcal{M},\beta}(t_n)) \in I(p)$. The evaluation of compound formulas is defined as is common. When $\mathcal{M}$ is clear from the context, we also write val_β instead of $val_{\mathcal{M},\beta}$.

We say that a formula φ is *satisfied* in $\mathcal{M}, \beta$ if $val_{\mathcal{M},\beta}(\varphi) = true$, and that it is *satisfiable* (SAT) if it is satisfied by some $\mathcal{M}, \beta$. We say a set Γ of formulae *entails* a formula φ, denoted $\Gamma \models \varphi$, if φ is satisfied whenever all formulas in Γ are satisfied.

Example 1. We assume a language $\mathcal{L}$ consisting of a sort s, two constants a and b, a unary function symbol f with argument and result of sort s; a binary predicate symbol p with two arguments of sort s. A structure $\mathcal{M} = (U_s, I)$ can be defined by $U_s = \mathbb{Z}$, $I(a) = 1$, $I(b) = 2$, $I(f)[x] = x+2$, and $I(p)[x, y] = x \leq y$. The formula $\varphi = p(a, b) \rightarrow p(x, f(a))$ is satisfied in $\mathcal{M}$ by a variable assignment $\beta(x) = 1$ since $val_\beta(\varphi) = I(p)[val_\beta(a), val_\beta(b)] \rightarrow I(p)[val_\beta(x), I(f)[val_\beta(x)]] = \neg(1 \leq 2) \vee 1 \leq (1 + 2)$ is true. The formula φ is satisfiable since $val_\beta(\varphi) = true$ for $\mathcal{M}$ and a variable assignment $\beta(x) = 1$.

2.2 Constraint Horn Clauses

To introduce the notion of constrained Horn clauses, we assume a fixed base signature $\Sigma = (\mathcal{S}, \mathcal{R}, \mathcal{F}, \mathcal{X})$, as well as a unique structure $\mathcal{M}$ over this signature, forming the background theory. In this paper, we mainly consider the background theory of Linear Integer Arithmetic (LIA), following the SMT-LIB standard [22]. We further assume a set $\mathcal{R}_C$ of additional relation symbols that is disjoint from $\mathcal{R}$, which will be used to formulate the head and body of clauses.

Definition 1 (Constrained Horn clause). *Given signature Σ and the set $\mathcal{R}_C$, a* Constrained Horn Clause *(CHC) is a closed formula in the form*

$$\forall \bar{x}.\ H \leftarrow p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \wedge \varphi, \tag{1}$$

where $\bar{x}$ is a vector of variables; H is either false or an atom $p(t_1, \ldots, t_n)$ with $p \in \mathcal{R}_C$; the relation symbols $p_1, \ldots, p_n$ are elements of $\mathcal{R}_C$; and $\bar{t}_1, \ldots, \bar{t}_n$ and φ are vectors of terms and a formula over Σ, respectively. We call H the head *and $p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \wedge \varphi$ the* body *of the clause, respectively. We call the formula φ in the body the* constraint *of the clause.*

For convenience, in many places we leave out the quantifiers $\forall \bar{x}$ when writing clauses. A CHC without atoms in its body (the case $n = 0$) is called a *fact*. If the body of a CHC contains zero or one atom, the CHC is called *linear*. Otherwise, it is called *non-linear*.

Solving CHCs boils down to searching interpretations of the relation symbols $\mathcal{R}_C$ that satisfy the CHCs, assuming that all background symbols from Σ are interpreted by the fixed structure $\mathcal{M}$:

Definition 2 (Satisfiability of a CHC). *A CHC h is* satisfiable *if there is a structure $\mathcal{M}_C = (\mathcal{U}, I_C)$ for the extended signature $\Sigma_C = (\mathcal{S}, \mathcal{R} \cup \mathcal{R}_C, \mathcal{F}, \mathcal{X})$ such that (i) I_C coincides with I on Σ, and (ii) $\mathcal{M}_C$ satisfies h. A set $\mathcal{C}$ of CHCs is* satisfiable *if there is an extended structure $\mathcal{M}_C$ simultaneously satisfying all clauses in $\mathcal{C}$.*

Encoding Program Verification Problems using CHCs. A program verification problem involves checking whether a program adheres to its specified behavior. One approach to verification is to transform the problem into determining the satisfiability of a set of CHCs. This can be done, for instance, by encoding the partial correctness of a procedural imperative program into a negated Existential positive Least Fixed-point Logic (E+LFP) formula [4] using the weakest precondition calculus. Generally, encodings are designed such that the set of CHCs is satisfiable if and only if a program is safe. Various encoding schemes for different programming languages have been introduced in the literature, e.g., [4].

2.3 Graph Neural Networks

A Graph Neural Network (GNN) [13] is a type of neural network that consists of Multi-Layer Perceptrons (MLPs) [23]. GNNs operate on graph-structured data with nodes and edges, making them suitable for logic formulas which can naturally be represented as graphs. A GNN can take a set of typed nodes and edges as input and output a set of feature representations (vectors of real numbers) associated with the properties of the nodes. We refer to them as *node representations* in the rest of the sections.

The Message-Passing based GNN (MP-GNN) [24] is a type of GNN model. It utilizes an iterative message-passing algorithm in which each node in the graph aggregates messages from its neighboring nodes to update its own node representation. This mechanism assists in identifying the inner connections within substructures, such as terms and atoms, in graph represented logic formulae.

Formally, let $G = (V, E)$ be a graph, where V is the set of nodes and E is the set of edges. Let x_v be the initial node representation (a vector of random real numbers) for node v in the graph, and let N_v be the set of neighbors of node v. An MP-GNN consists of a series of T message-passing steps. At each step t, every node v in the graph updates its node representation as follows:

$$h_v^t = \phi_t(\rho_t(\{h_u^{t-1} \mid u \in N_v\}), h_v^{t-1}), \tag{2}$$

where $h_v^t \in \mathbb{R}^n$ is the updated node representation for node v after t iterations. The initial node representation, h_v^0, is usually derived from the node type and given by 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. $\rho_t : (\mathbb{R}^n)^{|N_v|} \rightarrow \mathbb{R}^n$ is a aggregation function with trainable parameters (e.g., a MLP followed by sum, min, or max) that aggregates the node representations of v's neighboring nodes at the t-th iteration. $\phi_t : \mathbb{R}^n \rightarrow \mathbb{R}^n$ is a function with trainable parameters (e.g., a MLP) that takes the aggregated node representation from ρ_t and the node representation of v in previous iteration as input, and outputs the updated node representation of v at the t-th iteration. MP-GNN assumes a node can capture local structural information from t-hop's neighbors by updating the node representation using aggregated representations of the neighbor nodes.

The final output of the MP-GNN could be the set of updated node representations for all nodes in the graph after T iterations. These node representations can be used for a variety of downstream tasks, such as node classification or graph classification.

Relational Hyper-Graph Neural Network (R-HyGNN) [1] is an extension of one MP-GNN called Relational Graph Convolutional Networks (R-GCN) [25], and it is specifically designed to handle labeled hypergraphs.

A labeled (typed) hypergraph is a hypergraph where each vertex (node) and hyperedge is assigned a type from a predefined set of types. Formally, a labeled hypergraph LHG is defined as a tuple $LHG = (V, E, \lambda_V, L_V, L_E)$, where V is a set of elements called vertices (or nodes), L_E is a set of pair consisting of a label (type) r and the number of nodes k under the label r, $E \subseteq V^* \times L_E$ is a set of hyperedges in which each hyperedge consists of a non-empty subsets of V and a pair $(r, k) \in L_E$. Here, $\lambda_V : V \rightarrow L_V$ is a labeling function that assigns a type from the set L_V to each vertex in V. The L_V is a set of possible types (labels) for the vertices V.

The node representation updating rule of R-HyGNN for one node v at timestep t is

$$h_v^t = \mathrm{ReLU}\Big(\sum_{\substack{(w_1,\ldots,w_k,(r,k))\\ \in E}} \; \sum_{\substack{i\in\{1,\ldots,k\},\\ w_i=v}} W_{r,i}^t \cdot \|(h_{w_1}^{t-1}, \ldots, h_{w_{i-1}}^{t-1}, h_{w_{i+1}}^{t-1}, \ldots, h_{w_k}^{t-1})\Big), \tag{3}$$

where the pair $(r, k) \in L_E$ is the edge type (relation) and the number of node for a edge $(w_1, \ldots, w_k, (r, k)) \in E$, $W^t_{r,i}$ is a matrix of learnable parameters in time step t for node $v = w_i$ in the edge with type r. There are $|L_E| \times \sum_{(r,k)\in L_E}(k) \times t$ matrices of learnable parameters in total. Here, $\|(h^{t-1}_{w_1}, \ldots, h^{t-1}_{w_{i-1}}, h^{t-1}_{w_{i+1}}, \ldots, h^{t-1}_{w_k})$ means concatenate v' neighbour node representations in time step $t-1$. The initial node representation h^0_v is derived from the node types L_V.

Intuitively, to update the representation of a node, R-HyGNN first concatenates the neighbor representations of node v for each edge from the previous time step $t-1$. It then multiplies the concatenated neighbor representations by the corresponding matrix of trained parameters (i.e., $W^t_{r,i}$) to derive a local representation of v. Next, it aggregates the local node representations (e.g., by addition). In other words, $\sum_{\substack{(w_1,\ldots,w_k,(r,k))\\ \in E}} \sum_{\substack{i\in\{1,\ldots,k\},\\ w_i=v}} W^t_{r,i} \cdot \|(h^{t-1}_{w_1}, \ldots, h^{t-1}_{w_{i-1}}, h^{t-1}_{w_{i+1}}, \ldots, h^{t-1}_{w_k})$ can be abstract to $\rho_t(\|(h^{t-1}_{w_1}, \ldots, h^{t-1}_{w_{i-1}}, h^{t-1}_{w_{i+1}}, \ldots, h^{t-1}_{w_k}))$ where ρ_t is an aggregation function with trainable parameters in $W^t_{r,i}$. Finally, it applies a ReLU function [26] as the update function ϕ_t. This function takes the aggregated local node representations as input and produces the final node representation h^t_v. This updating process for a single node recurs t times.

3 Abstract Symbolic Model Checking for CHCs

The goal of our work is to utilize GNNs to obtain improved state space exploration methods for CHCs. To this end, in this section we introduce an abstract formulation of CHC state space exploration, covering both the classical CEGAR approach and exploration in the style of symbolic execution.

3.1 Satisfiability Checking for CHCs

Our Algorithm 1 checks the satisfiability of a given a set $\mathcal{C}$ of CHCs by constructing an abstract reachability hyper-graph (ARG). An ARG is an over-approximation of the facts $p(\bar{a})$ that are logically entailed by $\mathcal{C}$; by demonstrating that the atom *false* does not follow from $\mathcal{C}$, it can be shown that $\mathcal{C}$ is satisfiable. Since each node of an ARG can represent a whole set of facts, a finite ARG can be a representation even of infinite models of a set $\mathcal{C}$.

We first give an abstract definition of ARGs that does not mandate any particular symbolic representation of sets of $p(\bar{a})$. We later introduce two instances of this abstract framework.

Like in Sect. 2.2, we denote the set of relation symbols used in CHCs by $\mathcal{R}_C$. For a k-ary relation symbol $p \in \mathcal{R}_C$ with argument sorts $s_1, \ldots, s_k$, we write $\mathcal{R}_p = \mathcal{P}(U_{s_1} \times \cdots \times U_{s_k})$ for the set of possible relations represented by p.

Definition 3. *An* abstract reachability graph *for a set $\mathcal{C}$ of CHCs is a hypergraph (V, E), where*

- *the set V of nodes is a set of pairs (p, R), with p being a relation symbol and $R \in \mathcal{R}_p$;*

- *$E \subseteq V^* \times \mathcal{C} \times V$ is a set of hyper-edges labelled with clauses. For every edge $(v_1, \ldots, v_n, h, v_0) \in E$, it is the case that:*
  - *the head of a clause h is not false, i.e., h is of the form $\forall \bar{x}.\ p_0(\bar{t}_0) \leftarrow p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \wedge \varphi$;*
  - *nodes $v_0, \ldots, v_n$ correspond to the head and the body of h, i.e., for $i \in \{0, \ldots, n\}$ it is the case that $v_i = (p_i, R_i)$ for some $R_i \in \mathcal{R}_{p_i}$;*
  - *R_0 over-approximates the facts implied by the clause h:*

$$R_0 \supseteq \left\{ val_\beta(\bar{t}_0) \mid \begin{array}{l} val_\beta(\varphi) = true \text{ and } val_\beta(\bar{t}_i) \in R_i \text{ for } i = \{1, \ldots, n\} \\ \text{for some variable assignment } \beta \end{array} \right\} . \tag{4}$$

The algorithm starts with an empty ARG, and then adds nodes and edges to it until the ARG is *complete,* which intuitively means that all possible non-trivial edges are present in the graph. To define the notion of a complete ARG, we first need to characterize what it means for a clause to correspond to a feasible edge of the ARG:

Definition 4. *A clause $h \in \mathcal{C}$ is* feasible *for nodes $v_1, \ldots, v_n \in V$ of an ARG (V, E) if*

- *h is of the form $\forall \bar{x}.\ H \leftarrow p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \wedge \varphi$;*
- *nodes $v_1, \ldots, v_n$ correspond to the body of h, i.e., for $i \in \{1, \ldots, n\}$ it is the case that $v_i = (p_i, R_i)$ for some $R_i \in \mathcal{R}_{p_i}$;*
- *the constraints imposed by φ and $v_1, \ldots, v_n$ are not contradictory, i.e., there is a variable assignment β such that $val_\beta(\varphi) = true$ and $val_\beta(\bar{t}_i) \in R_i$ for $i = \{1, \ldots, n\}$.*

Definition 5. *An ARG (V, E) is complete for a set $\mathcal{C}$ of CHCs if*

- *for every CHC $h \in \mathcal{C}$ that has a head different from false, and that is feasible for $v_1, \ldots, v_n \in V$, there is some edge $\langle (v_1, \ldots, v_n), h, v_0 \rangle \in E$;*
- *there is no CHC $h \in \mathcal{C}$ with head false that is feasible for any $v_1, \ldots, v_n \in V$.*

We can finally observe that complete ARGs correspond to models of the clause set $\mathcal{C}$.

Lemma 1. *A set $\mathcal{C}$ of CHCs has a complete ARG iff $\mathcal{C}$ is satisfiable.*

Algorithm 1 describes how ARGs can be constructed for a given clause set $\mathcal{C}$. The algorithm starts with an empty ARG (V, E), and maintains a queue $Q \subseteq \mathcal{C} \times V^*$ of feasible edges to be added to the graph next. The queue is initialized with the clauses with empty body, representing the initial states of a program. Once the queue runs empty, the constructed ARG is complete and the set $\mathcal{C}$ has been shown to be satisfiable.

In each iteration, in lines 5–6 an element $(h, \bar{v})$ is picked and removed from the queue Q. If the head of h is *false* (line 7), the edge to be added might be part of a witness of unsatisfiability of $\mathcal{C}$. In this case, it has to be checked whether the nodes $\bar{v}$ are over-approximate (line 8); this can happen when the

```
   **Input**: A set $\mathcal{C}$ of CHCs
   **Output**: Satisfiability of $\mathcal{C}$
   **Initialise**: $V := \emptyset, E := \emptyset, Q := \{(h, ()) \mid h \in \mathcal{C}, h = \forall \bar{x}.\ H \leftarrow \varphi\}$
1  while true do
2      if Q is empty then
3          return satisfiable
4      else
5          Pick $(h, \bar{v}) \in Q$ to be considered next (guided by GNNs)
6          $Q := Q \setminus \{(h, \bar{v})\}$
7          if the head in h is false then
8              if derivation of false is genuine then
9                  return unsatisfiable
10             else
11                 Refine over-approximations
12                 Delete all affected nodes in $(V, E)$
13                 Regenerate elements in $Q$
14             end
15         else
16             Assume $h = \forall \bar{x}.\ p_0(\bar{t}_0) \leftarrow p_1(\bar{t}_1) \wedge \cdots \wedge p_n(\bar{t}_n) \wedge \varphi$
17             Compute new node $u = (p_0, R_0)$ for $(h, \bar{v})$
18             if $u \notin V$ then
19                 $V := V \cup \{u\}$
20                 $Q := Q \cup \left\{ (d, (w_1, \ldots, w_m)) \;\middle|\; \begin{array}{l} d \in \mathcal{C} \text{ is feasible for } w_1, \ldots, w_m, \\ u \in \{w_1, \ldots, w_m\} \subseteq V \end{array} \right\}$
21             end
22             $E := E \cup \{(\bar{v}, h, u)\}$
23         end
24     end
25 end
```

Algorithm 1: Abstract symbolic model checking algorithm for checking satisfiability of CHCs

containment in (4) is sometimes strict in the constructed ARG, and occurs in particular when instantiating the abstract algorithm as CEGAR (see Sect. 3.2). The over-approximation can then be refined in lines 11–13.

If the head of h is not *false*, a further edge is added to the graph by computing in line 17 some set R_0 satisfying (4). If the resulting node u is new in the graph, the queue Q is updated by adding possible outgoing edges for u (line 19–20).

In this paper, we apply the GNN-based guidance in line 5. Specifically, we presume that the GNN can predict the probability of h being in MUSes for each $q = (h, \bar{v}) \in Q$. We then combine this probability with certain features of q, such as the number of iterations $q \in Q$ has been waiting, to calculate a priority of q. When selecting an element q from Q, we consult this priority. We explain more details in Sect. 4.

3.2 CEGAR- And Symbolic Execution-Style Exploration

We now discuss two concrete instantiations of Algorithm 1. The first one, in this paper called SymEx, resembles the symbolic execution [27] of a program, and represents the relation R in an ARG node (p, R), for a k-ary relation symbol p, as a formula over free variables $z_1, \ldots, z_k$.

To obtain SymEx, in line 17 of Algorithm 1 the relation R_0 is derived from the nodes $\bar{v}$ by simple symbolic manipulation. Assuming that $v_i = (p_i, R_i)$ for $i \in \{1, \ldots, n\}$, and the relations R_i are all represented as formulas, we can define:

$$R_0[z_1, \ldots, z_k] \;=\; \exists \bar{x}.\ \bar{z} = \bar{t}_0 \wedge R_1[\bar{t}_1] \wedge \cdots \wedge R_n[\bar{t}_n] \wedge \varphi$$

where the notation $R_i[\bar{t}_i]$ expresses that the terms $\bar{t}_i$ are substituted for the free variables $\bar{z}$. In our implementation on top of the CHC solver Eldarica [28], the formula $R_0[z_1, \ldots, z_k]$ is afterwards simplified by eliminating the quantifiers, albeit only in a best-effort way by running the built-in formula simplifier of Eldarica. In SymEx, since no over-approximation is applied, the test in line 8 always succeeds, and lines 11–13 are never executed.

Our second instantiation, called CEGAR, is designed following counter example-guided abstraction refinement [5,29] with Cartesian predicate abstraction [30]. In this version of the algorithm, we assume that a finite pre-defined set Π_p of predicates is available for every relation symbol p. If p is k-ary, then the elements of Π_p are formulas over free variables $z_1, \ldots, z_k$. The relation R in a node (p, R) is now represented as a subset of Π_p.

In line 17, the set R_0 is computed by determining the elements of Π_{p_0} that are entailed by the body of clause h:

$$R_0 \;=\; \{\phi \in \Pi_{p_0} \mid \bar{z} = \bar{t}_0 \wedge \bigwedge R_1[\bar{t}_1] \wedge \cdots \wedge \bigwedge R_n[\bar{t}_n] \wedge \varphi \models \phi\}$$

The notation $\bigwedge R_i[\bar{t}_i]$ denotes the conjunction of the elements of R_i, with $\bar{t}_i$ substituted for the free variables $\bar{z}$.

Over-approximation in CEGAR stems from the fact that a chosen set of predicates Π_p will oftentimes not be able to exactly represent a relation; the constructed ARG might then include facts $p(\bar{a})$ that are not actually entailed by $\mathcal{C}$. In line 8, the algorithm therefore has to verify that discovered derivations of *false* are genuine. This is done by collecting the clauses that were used to derive the nodes $\bar{v}$ in the ARG and constructing a counterexample tree. If the counterexample turns out to be *spurious,* further predicates are added to the sets Π_p, for instance using tree interpolation [31], in lines 11–13.

4 Guiding CHC Solvers Using MUSes

In this section, we begin by defining the notion of Minimal Unsatisfiable Sets (MUSes), then we detail the process of collecting three types of training labels using the MUSes. Following that, we explain the various strategies of employing the predicted probability of a CHC being in MUSes to guide Algorithm 1.

4.1 Minimal Unsatisfiable Sets

Throughout the section, assume that $\mathcal{C}$ is an unsatisfiable set of CHCs.

Definition 6 (Minimal Unsatisfiable Set). *A subset $U \subseteq \mathcal{C}$ is a* Minimal Unsatisfiable Set *(MUS) if U is unsatisfiable and for all CHCs $h \in U$ it is the case that $U \backslash \{h\}$ is satisfiable.*

Intuitively, MUSes of a set of CHCs encoding a program correspond to minimal counterexamples (i.e., a subset of program statements) witnessing the incorrectness of the program. MUSes are therefore good candidates for guiding CHC solvers towards the critical clauses, and we aim at predicting MUSes using GNNs.

The number of MUSes can, however, be exponential in the number of CHCs in $\mathcal{C}$. We therefore consider the *union*, *intersection*, and a particular *single* MUS for $\mathcal{C}$. Denoting the set of all MUSes of $\mathcal{C}$ by MUS($\mathcal{C}$), those are:

$$\mathcal{C}^{\text{union}}_{\text{MUSes}} = \bigcup \text{MUS}(\mathcal{C}), \qquad \mathcal{C}^{\text{intersection}}_{\text{MUSes}} = \bigcap \text{MUS}(\mathcal{C}),$$
$$\mathcal{C}^{\text{single}}_{\text{MUSes}} = \underset{U \in \text{MUS}(\mathcal{C})}{\text{argmax}} \ \text{numAtom}(U),$$

where *numAtom*(U) is the total number of atoms of the CHCs in U, and $\mathcal{C}^{\text{single}}_{\text{MUSes}}$ is some MUS that maximizes the total number of atoms. The three clause sets can be computed using the OptiRica extension of the Eldarica Horn solver [32].

Intuitively, $\mathcal{C}^{\text{union}}_{\text{MUSes}}$ includes all information about possible MUSes and encourages the algorithm to go through all possible error-prone areas. In contrast, $\mathcal{C}^{\text{intersection}}_{\text{MUSes}}$ only takes the intersection of all MUSes which can guide the algorithm to only focus on the most suspicious areas. $\mathcal{C}^{\text{single}}_{\text{MUSes}}$ is one of MUSes and corresponds to a long path in the ARG, given that a high number of atoms is associated with a large number of nodes. We believe a long path contains intricate information, which is challenging for human to parse, but easier for a deep-learning-based model to find.

We form three types of Boolean clause labelings by using $\mathcal{C}^{\text{union}}_{\text{MUSes}}$, $\mathcal{C}^{\text{intersection}}_{\text{MUSes}}$, and $\mathcal{C}^{\text{single}}_{\text{MUSes}}$, respectively. For instance, for $\mathcal{C}^{\text{union}}_{\text{MUSes}}$, we obtain the labels

$$l^{\text{union}}(c) = \begin{cases} 1 & \textit{if } h \in \mathcal{C}^{\text{union}}_{\text{MUSes}} \\ 0 & \textit{if } h \in \mathcal{C} \setminus \mathcal{C}^{\text{union}}_{\text{MUSes}} \end{cases}$$

4.2 MUS-Based Guidance for CHC Solvers

Eldarica Heuristics The implementations of CEGAR and SymEx in the Horn solver Eldarica [28] by default use fixed, hand-written selection heuristics in line 5 of Algorithm 1. Such heuristics are defined by a ranking function $r : Q \to \mathbb{Z}$ that maps every element in the queue to an integer; in line 5, the element of Q is picked that minimizes r. The standard implementation of r used for CEGAR is defined by

$$\textit{EldCEGARRank}(q) = \textit{numPredicate}(q) + \textit{birthTime}(q) + \textit{falseClause}(q) \ ,$$

where $\mathit{numPredicate}((h, \bar{v})) = \sum_{(p,R) \in \bar{v}} |R|$ is the total number of predicates occurring in the considered nodes of the ARG; $\mathit{birthTime}(q)$ is the iteration (as an integer) in which q was added to Q;[1] and $\mathit{falseClause}((h, \bar{v}))$ is 0 if the head of h is not *false*, and some big integer otherwise. The rationale behind the ranking function is that nodes with few predicates tend to subsume nodes with many predicates, every clause should be picked eventually, and clauses with head *false* will trigger either termination of the algorithm or abstraction refinement.

In SymEx, for a node (p, R), we define *numConstraint(R)* to be the number of conjuncts of R. For instance, for $R = (x > 1 \wedge y < 0)$ we would get $\mathit{numConstraint}(R) = 2$. The default Eldarica ranking function for SymEx is defined by

$$\mathit{EldSymExRank}((h, \bar{v})) = \sum_{(p,R) \in \bar{v}} \mathit{numConstraint(R)}\ .$$

Similar to *numPredicate*, the intuition behind *EldSymExRank* is that nodes with larger formulas (more restrictions) tend to be subsumed by nodes with smaller formulas (fewer restrictions).

MUS-Guided Heuristics We now introduce several new ranking functions defined with the help of MUSes. For this, suppose that $\mathcal{C}$ is the set of CHCs that a Horn solver is applied on. Under the assumption that $\mathcal{C}$ is unsatisfiable, we obtain three labeling functions l^{union}, $l^{\text{intersection}}$, l^{single} that are able to point out clauses to be prioritized in Algorithm 1.

In practice, of course, the status of $\mathcal{C}$ will initially be unknown. We therefore use three GNNs to *predict* labels $l^{\text{union}}(h)$, $l^{\text{intersection}}(h)$, $l^{\text{single}}(h)$, respectively, given just the set $\mathcal{C}$ and some clause $h \in \mathcal{C}$ as input. We interpret the prediction of a GNN as a *probability* $P(h)$ of a clause h to be in the union, intersection, or the single MUS set, and use those probabilities to define ranking functions. Table 1 lists several candidate ranking functions in terms of this membership probability P, where P stands for one of P^{union}, P^{single}, or $P^{\text{intersection}}$.

We consider two ways to convert probabilities to integers that can be used in the ranking function. In $\mathit{rank}_P(q)$, the elements $q = (h, \bar{v})$ of the queue Q are first sorted in descending order of $P(h)$; the number $\mathit{rank}_P(q)$ is the position of q in this sequence. This means that $\mathit{rank}_P(q)$ ranges from 0 to $|Q| - 1$, and elements with large probability $P(h)$ will be assigned small rank.

In $\mathit{norm}_P(q)$, we assume that $\min_P = \min\{P(h) \mid (h, \bar{v}) \in Q\}$ and $\max_P = \max\{P(h) \mid (h, \bar{v}) \in Q\}$ denote the minimum and maximum probability, respectively, among elements of Q. The normalized value $\mathit{norm}_P(q) \in [0, 1]$ is defined by

$$\mathit{norm}_P((h, \bar{v})) = \frac{P(h) - \min_P}{\max_P - \min_P}$$

In the ranking functions, we multiple $\mathit{norm}_P(q)$ with a negative coefficient *coef* and round the result to the nearest integer.

[1] Strictly speaking, this information cannot be computed from q, it is in practice stored as an additional field of the queue elements.

Table 1. Ranking function for queue elements $q \in Q$ in Algorithm 1, where $P \in \{P^{\text{union}}, P^{\text{single}}, P^{\text{intersection}}\}$.

Algorithm	Name	Ranking function
CEGAR	Fixed	$\mathit{EldCEGARRank}(q)$
	Random	$\mathit{RandomRank}$
	Score	$\mathit{coef} \cdot \mathit{norm}_P(q)$
	Rank	$\mathit{rank}_P(q)$
	R-Plus	$\mathit{rank}_P(q) + \mathit{EldCEGARRank}(q)$
	S-Plus	$\mathit{coef} \cdot \mathit{norm}_P(q) + \mathit{EldCEGARRank}(q)$
	R-Minus	$\mathit{rank}_P(q) - \mathit{EldCEGARRank}(q)$
	S-Minus	$\mathit{coef} \cdot \mathit{norm}_P(q) - \mathit{EldCEGARRank}(q)$
SymEx	Fixed	$\mathit{EldSymExRank}(q)$
	Random	$\mathit{RandomRank}$
	Score	$\mathit{coef} \cdot \mathit{norm}_P(q)$
	Rank	$\mathit{rank}_P(q)$
	R-Plus	$\mathit{rank}_P(q) + \mathit{EldSymExRank}(q) + \mathit{birthTime}(q)$
	S-Plus	$\mathit{coef} \cdot \mathit{norm}_P(q) + \mathit{EldSymExRank}(q) + \mathit{birthTime}(q)$
	R-Minus	$\mathit{rank}_P(q) - \mathit{EldSymExRank}(q) - \mathit{birthTime}(q)$
	S-Minus	$\mathit{coef} \cdot \mathit{norm}_P(q) - \mathit{EldSymExRank}(q) - \mathit{birthTime}(q)$
	Two-queue	$\begin{cases} \text{R-Minus, 80\% } \mathit{probability} \\ \text{Random, 20\% } \mathit{probability} \end{cases}$

The *RandomRank* function ensures that each CHC has an equal opportunity to be selected in each iteration. The Two-queue case denotes a setup with two queues, Q_1 and Q_2, each used with a certain probability in line 5 of Algorithm 1. We list just one such combination, which alternates between queues with the R-Minus and Random functions: there's an 80% chance we use the R-Minus queue and a 20% chance we use the Random queue.

5 Design of Model

As shown in Fig. 1, within the CHC-R-HyGNN framework, we first encode a set of CHCs into a graph format, and then we build a GNN model consisting of an encoder, GNN layers, and a decoder.

5.1 Encode CHCs to Graph Representation

We apply the two (hyper-)graph representations of CHCs defined in [1]. We will briefly describe their main features here.

The first graph representation is called a *constraint graph* (CG). This graph encapsulates syntactic information by using nodes to represent each symbol and connecting them with binary edges. Each CHC and each predicate symbol is represented by a unique node. Terms and formulas are represented using their abstract syntax trees (AST). CHC nodes are connected to the constituent atoms and constraints by binary edges. Within one CHC, common sub-expressions are represented by the same nodes. Different hyper-graph node and edges types are used to distinguish the various encoded operators (see Sect. 2.3).

The second graph representation is the *control- and data-flow hypergraph* (CDHG). This graph is designed to capture both control- and data-flow within CHCs using hyperedges, and therefore captures the semantics of CHCs more directly than GCs. Similar to the CG, in the CDHG, each CHC is represented by a unique node, and the atoms are rendered in the same way as in CG. Unlike the CG, the CDHG uses *control-flow hyperedges* (CFHEs) to describe the control-flow from the body to the head in each CHC, guarded by the constraint of the CHC. Furthermore, the CDHG uses *data-flow hyperedges* (DFHEs) to represent data-flow from the terms in the body to the terms in the head. These data-flows are also guarded by the constraint.

5.2 Model Structure

The R-HyGNN model consists of three sub-components: (i) encoder, (ii) R-HyGNN [1] (a GNN), and (iii) decoder:

$$\text{(i) } \mathcal{H}_0 = \text{encoder}(V, \lambda_V, L_V), \qquad \text{(ii) } \mathcal{H}_t = \text{R-HyGNN}(\mathcal{H}_0, E, L_E),$$
$$\text{(iii) } \hat{\mathcal{L}} = \text{decoder}(\mathcal{H}_t)\ .$$

The encoder in (i) first maps each node in V to an integer according to the node's type determined by λ_V and L_V. Then, it passes the encoded integers to a single-layer neural network (embedding layer) to compute initial node representations $\mathcal{H}_0$. The R-HyGNN in (ii) is a GNN with its node representation updating rule defined in (3). It takes the initial node representations ($\mathcal{H}_0$), edges (E), and edge types (L_E) as input and outputs the updated node representations $\mathcal{H}_t$. The decoder in (iii) first identifies the node that denote the CHC instead of the variables, atoms, or other elements in the CHC, then we collect the representations of these CHC nodes $\mathcal{H}_t^{\text{CHCs}}$ from all node representations $\mathcal{H}_t$. Finally, we pass $\mathcal{H}_t^{\text{CHCs}}$ to a set of fully connected neural networks to compute the probability of each CHC being in the MUSes, and $\hat{\mathcal{L}}$ is a set of probabilities.

The parameters of neural networks in (i), (ii), and (iii) are optimized together by minimizing the binary cross-entropy loss [33] between $\hat{\mathcal{L}}$ and the true labels $\mathcal{L}$, i.e.,

$$loss = -\frac{1}{N}\sum_{i=1}^{N} \mathcal{L}_i log(\hat{\mathcal{L}}_i) + (1 - \mathcal{L}_i) log(1 - \hat{\mathcal{L}}_i). \tag{5}$$

6 Evaluation

We first describe how the benchmarks are split for training and evaluation, then list some important parameters. Finally, we show and explain the experimental results. This work can be reproduced by following the instructions in [34].

Table 2. Distribution of the number of problems for both training and evaluation. T/O, N/A, and Evail. denote timeout, not available, and evaluation, respectively.

Linear LIA problems							Non-linear LIA problems						
8705							8425						
Benchmarks for training					Holdout set		Benchmarks for training					Holdout set	
7834 (90%)					871 (10%)		7579 (90%)					846 (10%)	
UNSAT			SAT	T/O	Eval.	N/A	UNSAT			SAT	T/O	Eval.	N/A
1585			4004	2245	383	488	3315			4010	254	488	358
Train	Valid	N/A					Train	Valid	N/A				
782	87	716					1617	180	1518				

6.1 Benchmark

The training and evaluation data are specified in Table 2, and available in [20]. There are 8705 linear and 8425 non-linear LIA problems, taken from CHC-COMP 2021 [19]. We first uniformly reserved 10% of the benchmarks as hold-out set for the final evaluation. We ran the CEGAR in Eldarica using a 3-hour timeout to solve the remaining 90% of benchmarks, leading to three groups of benchmarks: SAT, UNSAT, and timeout. For UNSAT problems, we also computed the MUS sets $\mathcal{C}^{\text{union}}_{\text{MUSes}}$, $\mathcal{C}^{\text{intersection}}_{\text{MUSes}}$, and $\mathcal{C}^{\text{single}}_{\text{MUSes}}$. Some problems in UNSAT were eliminated in this process (N/A, for both training and evaluation) because the problems were trivial (already solved by the Eldarica preprocessor), the process of extracting MUSes timed out (3 h), or a timeout occurred when encoding CHCs as graphs. The remaining problems are divided into training (90%) and validation (10%) datasets.

6.2 Parameters

We select the hyper-parameters for the GNN empirically, and according to the experimental results from [1]. We set the vector size of the initial node representation and the number of neurons in all intermediate neural network layers to 32; we also set the number of message-passing steps to 8 (i.e., applying (3) 8 times). The constant *coef* in Table 1 is -1000. Other parameters and the instructions to reproduce these results can be found in [20].

6.3 Experimental Results

In our experiment, we measure the number of solved problems and the average solving time for the holdout evaluation set. This included 383 and 488 problems in the linear and non-linear LIA datasets, respectively. The timeout for evaluating each problem is 1200 s. The additional overhead for reading and applying the GNN predicted results in each iteration is included in the solving time. The numerical results are shown in Tables 3 and 4. We also visualize some numerical results by scatter plots in Fig. 2.

Table 3. Overview of the best ranking function and improvement in number of solved problems compared to the Eldarica. A ranking function marked with * (e.g., S-Plus*) denotes that there are multiple ranking functions with the same performance.

Benchmark Algorithm	MUS data set (best count)	**Best ranking function (improvement in %)**						
		Number of Solved Problems			Average Time			
		Total	SAT	UNSAT	All	Common	SAT	UNSAT
Linear CEGAR	Union (0)	R-Plus (1.4%)	R-Plus (2.4%)	R-Minus (1.0%)	R-Plus (1.3%)	S-Plus (19.1%)	S-Minus (46.5%)	Rank (31.1%)
	Single (3)	Rank (3.6%)	**R-Plus (4.0%)**	Rank (8.2%)	R-Plus (1.9%)	S-Plus (26.6%)	**R-Minus (57.9%)**	**Rank (36.3%)**
	Intersection (4)	**R-Plus (4.1%)**	S-Plus (0.8%)	**R-Plus (9.3%)**	**R-Plus (3.1%)**	**S-Plus (27.6%)**	R-Minus (45.0%)	S-Plus (0.0%)
Linear SymEx	Union (4)	**Two-Q (1.0%)**	**S-Plus* (0.0%)**	**Random (2.0%)**	Two-Q (0.9%)	R-Minus (12.7%)	**R-Minus (30.2%)**	S-Plus (26.5%)
	Single (3)	S-Minus* (0.5%)	**S-Plus* (0.0%)**	**Random (2.0%)**	Random (0.8%)	**S-Plus (12.9%)**	Random (28.4%)	S-Plus (17.6%)
	Intersection (5)	**S-Plus* (1.0%)**	**S-Plus* (0.0%)**	**S-Plus* (2.0%)**	**S-Plus (1.3%)**	Score (9.5%)	Random (28.4%)	**R-Plus (35.8%)**
Non-Linear CEGAR	Union (7)	**S-Plus (0.5%)**	**S-Plus (0.8%)**	**S-Plus* (0.0%)**	**S-Plus** B	**R-Minus (20.8%)**	**Rank (53.5%)**	**S-Plus (19.4%)**
	Single (1)	R-Plus (0.2%)	R-Plus (0.4%)	**R-Plus* (0.0%)**	R-Plus (6.6%)	S-Plus (18.4%)	R-Minus (52.8%)	R-Minus (14.2%)
	Intersection (1)	R-Plus* (0.0%)	S-Plus (0.5%)	**S-Plus* (0.0%)**	R-Plus (5.9%)	R-Plus (20.3%)	Rank (45.8%)	S-Plus (16.8%)
Non-Linear SymEx	Union (6)	**Two-Q (6.1%)**	**S-Minus* (1.6%)**	Random (12.3%)	**Two-Q (13.3%)**	**R-Minus (7.3%)**	**Score (5.1%)**	**R-Plus (19.9%)**
	Single (3)	**Two-Q (6.1%)**	**Score (1.6%)**	**Two-Q (12.9%)**	Two-Q (12.4%)	Rank (-2.2%)	R-Minus (0.2%)	Two-Q (11.2%)
	Intersection (3)	**Two-Q (6.1%)**	**S-Plus (1.6%)**	**Two-Q (12.9%)**	Two-Q (12.7%)	S-Minus (0.6%)	Two-Q (1.7%)	S-Plus (5.4%)

In Tables 3 and 4, under the Number of Solved Problems column, the Total, and SAT, UNSAT columns denote the number of totals solved, solved SAT, and solved UNSAT problems, respectively. Under the Average Time column, the All column denotes the average solving time for all problems, including those that timed out; the Common column means the average solving time for problems that were commonly solved using one of the ranking functions in Table 1, and the default Eldarica ranking function; the SAT and UNSAT columns are the average solving times for SAT and UNSAT problems, respectively. In certain cells, the percentage in brackets represents the improvement compared to the corresponding default ranking function. The bold text highlights the best performance in a Benchmark Algorithm block for each measurement.

Table 3 displays the best ranking function and its improvement over the default Eldarica ranking function in different measurements for various combinations of benchmarks (linear and non-linear LIA), algorithms (CEGAR and SymEx), and MUS datasets (union, single, and intersection of MUSes). In the MUS dataset column, the numbers in brackets represent the count of bold text cells in the row, indicating the number of best performances achieved by that type of MUS dataset. For instance, in the last row (i.e., the Intersection row of the Non-Linear SymEx block), the number in the bracket is counted from the bold text highlighted cells in columns Total, SAT, and UNSAT under the Number of Solved Problems. This suggests that using the intersection of the MUSes

Table 4. Evaluation on holdout problems using union dataset. The time unit is second.

		Number of Solved Problems (improvement %)			**Average Time(improvement %)**			
Benchmark Algorithm	Ranking Function	Total	SAT	UNSAT	All	Common	SAT	UNSAT
Linear CEGAR	Default	222	125	97	519.38	25.77	38.97	8.77
	Random	221 (−0.5%)	124 (−0.8%)	97 (0.0%)	523.58 (−0.8%)	27.49 (−29.5%)	37.05 (4.9%)	15.85 (−80.7%)
	R-Plus	**225 (1.4%)**	**128 (2.4%)**	97 (0.0%)	**512.41 (1.3%)**	21.65 (16.0%)	42.89 (−10.1%)	11.99 (−36.7%)
	R-Minus	220 (−0.9%)	122 (−2.4%)	**98 (1.0%)**	526.08 (−1.3%)	18.02 (−24.4%)	30.93 (20.6%)	21.60 (−146.3%)
	S-Plus	222 (0.0%)	125 (0.0%)	97 (0.0%)	517.43 (0.4%)	**20.92 (19.1%)**	34.13 (12.4%)	**7.32 (16.5%)**
	S-Minus	219 (−1.4%)	122 (−2.4%)	97 (0.0%)	522.97 (−0.7%)	12.56 (2.4%)	**20.86 (46.5%)**	9.81 (−11.9%)
	Portfolio	229 (3.2%)	130 (4.0%)	99 (2.1%)	503.16 (3.1%)	18.28 (29.1%)	45.67 (−17.2%)	19.94 (−127.4%)
Linear SymEx	Default	200	101	99	590.68	33.16	55.42	10.44
	Random	201 (0.5%)	100 (−1.0%)	101 (2.0%)	586.12 (0.8%)	30.08 (−8.5%)	39.69 (28.4%)	20.95 (−100.7%)
	R-Plus	192 (−4.0%)	**101 (0.0%)**	91 (−8.1%)	617.60 (−4.6%)	38.59 (−10.9%)	52.87 (4.6%)	21.99 (−110.6%)
	R-Minus	200 (0.0%)	100 (−1.0%)	100 (1.0%)	586.24 (0.8%)	**24.67 (12.7%)**	**38.69 (30.2%)**	10.60 (−1.5%)
	S-Plus	198 (−1.0%)	**101 (0.0%)**	97 (−2.0%)	595.02 (−0.7%)	30.22 (11.6%)	50.97 (8.0%)	**7.67 (26.5%)**
	S-Minus	201 (0.5%)	**101 (0.0%)**	100 (1.0%)	586.35 (0.7%)	30.64 (7.8%)	50.57 (8.8%)	10.65 (−2.0%)
	Two-queue	**202 (1.0%)**	**101 (0.0%)**	**101 (2.0%)**	**585.58 (0.9%)**	35.11 (−5.9%)	49.94 (9.9%)	20.14 (−92.9%)
	Portfolio	206 (3%)	101 (0.0%)	105 (6.1%)	569.1 (3.7%)	25.79 (22.2%)	44.58 (19.6%)	10.16 (2.6%)
Non Linear CEGAR	Default	432	250	182	131.12	42.05	43.34	40.28
	Random	425 (−1.6%)	243 (−2.8%)	**182 (0.0%)**	143.42 (−9.4%)	34.27 (−11.1%)	34.84 (19.6%)	38.75 (3.8%)
	R-Plus	432 (0.0%)	250 (0.0%)	**182 (0.0%)**	122.29 (6.7%)	31.74 (17.8%)	28.59 (34.0%)	37.82 (6.1%)
	R-Minus	417 (−3.5%)	240 (−4.0%)	177 (−2.7%)	154.07 (−17.5%)	**26.20 (20.8%)**	**21.46 (50.5%)**	**32.51 (19.3%)**
	S-Plus	**434 (0.5%)**	**252 (0.8%)**	**82 (0.0%)**	**121.75 (7.1%)**	34.64 (13.1%)	35.97 (17.0%)	39.10 (2.9%)
	S-Minus	421 (−2.5%)	242 (−3.2%)	179 (−1.6%)	149.02 (−13.7%)	31.76 (−2.0%)	26.33 (39.2%)	38.95 (3.3%)
	Portfolio	435 (0.7%)	253 (1.2%)	182 (0.0%)	113.49 (13.4%)	28.24 (29.1%)	30.57 (29.5%)	31.75 (21.2%)
Non Linear SymEx	Default	342	187	155	343.82	28.39	29.05	27.59
	Random	362 (5.8%)	188 (0.5%)	**174 (12.3%)**	301.90 (12.2%)	32.67 (−15.1%)	**36.24 (−24.8%)**	41.83 (−51.6%)
	R-Plus	339 (−0.9%)	**190 (1.6%)**	149 (−3.9%)	357.18 (−3.9%)	27.88 (0.3%)	47.71 (−64.2%)	**22.10 (19.9%)**
	R-Minus	361 (5.6%)	189 (1.1%)	172 (11.0%)	299.86 (12.8%)	**26.35 (7.3%)**	37.68 (−29.7%)	27.98 (−1.4%)
	S-Plus	340 (−0.6%)	189 (1.1%)	151 (−2.6%)	352.84 (−2.6%)	29.04 (−0.3%)	41.41 (−42.5%)	24.54 (11.1%)
	S-Minus	362 (5.8%)	**190 (1.6%)**	172 (11.0%)	303.65 (11.7%)	28.62 (−0.4%)	44.11 (−51.8%)	37.95 (−37.5%)
	Two-queue	**363 (6.1%)**	189 (1.1%)	**174 (12.3%)**	**297.93 (13.3%)**	30.15 (−6.2%)	41.14 (−41.6%)	32.51 (−17.8%)
	Portfolio	366 (7.0%)	191 (2.1%)	175 (12.9%)	288.85 (16.0%)	22.29 (21.4%)	42.42 (−46.0%)	26.75 (3.0%)

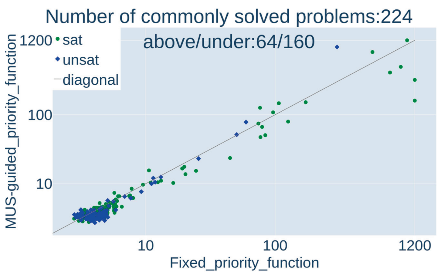
dataset achieves the best performance when the evaluation set is non-linear and the algorithm is SymEx. Across the entire table, there are 17, 10, and 13 bold text counts for the union, single, and intersection MUS datasets, respectively. This indicates that the union is the most effective MUS dataset for better performance across different benchmarks and algorithms. Consequently, we provide further numerical details for the union MUS dataset in Table 4. Evaluation results for both the single and intersection MUS datasets can be found in [20].

Table 4 illustrates the evaluation results using MUS-guided ranking functions (see Table 1), compared to the default and random ranking functions. In terms of the total number of solved problems, the improvement for the Linear dataset is at most 1.4%, achieved by the CEGAR algorithm with the R-Plus ranking function. Meanwhile, for the Non-linear dataset, the improvement is 6.1%, achieved by the SymEx algorithm with the two-queue ranking function. This is consistent with the average solving time for all benchmarks. In each Benchmark Algorithm block, we also show the results obtained by a virtual portfolio that selects the best ranking function for each benchmark.

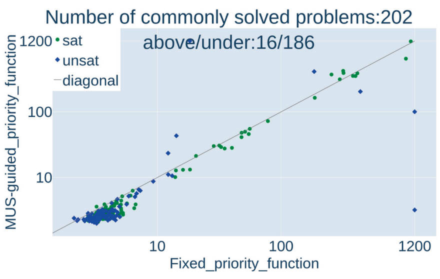
The biggest improvement in Average Time for SAT and UNSAT problems are 50.5% and 26.5%, achieved by the CEGAR with R-Minus and the SymEx with S-Plus ranking function, respectively. When combined with the corresponding numbers of total solved problems (i.e., −3.5% and −1.0%), it suggests that these ranking functions either solve the problems quickly or not at all. The Average Times in the Common column often differ significantly from the times in the All column. This suggests that the number of newly solved problems has a greater impact on the improvement in the average solving time for all problems than the commonly solved problems.

Figure 2 shows the solving time scatter plots for the problems from the best configurations in each Benchmark Algorithm block in Table 4. Notably, a majority of the dots lie below the diagonal lines in each scatter plot, indicating the solving time is improved by the MUS-guided ranking function for more than half of the problems. This is consistent with the numerical results. The plots also show that the MUS-guided ranking functions achieve speedups in CEGAR in particular for long-running problems that are SAT, while MUS-guidance in SymEx makes it possible to solve a significant number of UNSAT problems on which the default configuration times out.

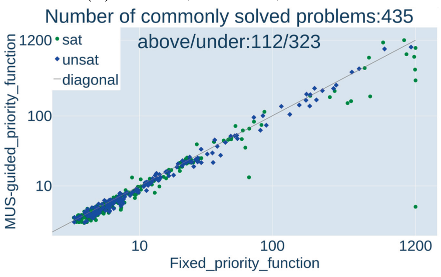
In summary, for both algorithms in different datasets, there is always at least one of the MUS-guided ranking functions that achieves the best result in terms of all aspects of measurements. Using the predicted probabilities alone (i.e., using the Rank and Score ranking function) performs weaker than other MUS-guided ranking functions that combine the predicted probability and the default heuristics. Currently, the MUS-guided ranking functions in Table 1 are designed by simply varying the relation symbols "+" and "−" between different elements (e.g., ranking functions S-Plus and S-Minus for both CEGAR and SymEx) or by setting the restart point randomly (i.e., ranking function Two-queue in SymEx). We believe MUSHyperNet has more potential if the ranking functions are designed carefully or learned from some good tasks.



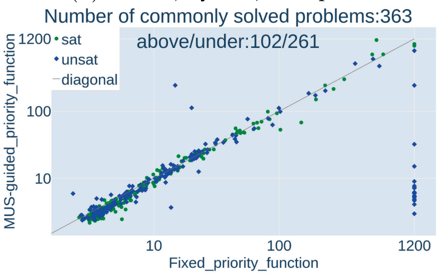
(a) Linear, CEGAR, R-Plus



(b) Linear, SymEx, two-queue



(c) Non-Linear, CEGAR, S-Plus



(d) Non-Linear, SymEx, two-queue

Fig. 2. All benchmark average solving time scatter plots for best ranking functions in different dataset and algorithms. "above/under" means the number of dots above and under the diagonal line.

7 Related Work

Machine learning techniques have been adapted in various ways to assist in formal verification. For example, the study in [35] employs Support Vector Machines (SVM) [36] and Gaussian processes [37] to select heuristics for theorem proving. Similarly, [38] introduces the use of a Recurrent Neural Network Based Language Model (RNNLM) to derive finite-state automaton-based specifications from execution traces. In the domain of selecting algorithms for program verification, [39] apply the Transformer architecture [40], while [41] uses kernel-based methods [42]. With the thriving of deep learning techniques, an increasing number of works are utilizing GNNs to learn the features from programs and logic formulae. This trend is attributed to the inherent structure of these languages, which can be naturally represented as graphs and subsequently learned by GNNs. For instance, studies like [14–16,43], and [44] use GNNs [24,45,46] to learn features from graph-represented logic formulas and programs, aiding in tasks such as theorem proving, SAT solving, and loop invariant reasoning.

One closed idea is NeuroSAT [16,17], which trains a GNN to predict the probability of variables appearing in unsat cores. This prediction can guide the variable branching decisions for Conflict-Driven Clause Learning (CDCL) [47]-based SAT solvers. In a similar vein, our study trains a GNN to predict the probability of a CHC appearing in MUSes. This aids in determining the processing sequence of CHCs in Algorithm 1 used for solving a set of CHCs.

8 Conclusion and Future Works

In this study, we train a GNN model to predict the probability of each CHC in a set of CHCs being in the MUSes. We then utilize these predicted probabilities to guide the abstract symbolic model-checking algorithm in selecting a CHC during each iteration. Extensive experiments demonstrate improvements in both the number of solved problems and average solving time when using the MUS-guided ranking functions, compared to the default ranking function. This was observed in two instances of the abstract symbolic model checking algorithm: CEGAR and SymEx. We believe that this approach can be extended to other algorithms, as many could benefit from understanding more about the MUSes of a set of CHCs.

There are several ways to further enhance the performance of MUSHyperNet. One of our future work is to integrate the work of manually designing the ranking functions in Table 1 to the learning process. Regarding the GNN model, we believe that incorporating an attention mechanism could bolster its performance, subsequently refining the quality of the predicted probabilities. Another avenue to explore involves integrating the GNN with the solver in a more interactive manner. Instead of predicting something at once and then using them in each iteration, we could query the GNN model to predict something in real-time based on the current context during each iteration.

Acknowledgement. We thank Zafer Esen for providing assistance in using the symbolic execution engine of Eldarica, and Marc Brockschmidt for various discussions on this work. The computations and data handling were enabled by resources provided by the Swedish National Infrastructure for Computing (SNIC) at UPPMAX and C3SE. The research was partially funded by the Swedish Research Council through grant agreement no. 2018–05973, 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. Liang, C., Rümmer, P., Brockschmidt, M.: Exploring representation of Horn clauses using GNNs (extended technique report). CoRR, abs/2206.06986 (2022)
2. Horn, A.: On sentences which are true of direct unions of algebras. J. Symbol. Logic **16**(1), 14–21 (1951)
3. De Angelis, E., Fioravanti, F., Gallagher, J.P., Hermenegildo, M.V., Pettorossi, A., Proietti, M.: Analysis and transformation of constrained Horn clauses for program verification. CoRR, abs/2108.00739 (2021)
4. Bjørner, N., Gurfinkel, A., McMillan, K., Rybalchenko, A.: Horn clause solvers for program verification. In: Beklemishev, L.D., Blass, A., Dershowitz, N., Finkbeiner, B., Schulte, W. (eds.) Fields of Logic and Computation II. LNCS, vol. 9300, pp. 24–51. Springer, Cham (2015). https://doi.org/10.1007/978-3-319-23534-9_2

5. Clarke, E.M.: SAT-based counterexample guided abstraction refinement in model checking. In: Baader, F. (ed.) CADE 2003. LNCS (LNAI), vol. 2741, pp. 1–1. Springer, Heidelberg (2003). https://doi.org/10.1007/978-3-540-45085-6_1
6. Bradley, A.R.: SAT-based model checking without unrolling. In: Jhala, R., Schmidt, D. (eds.) VMCAI 2011. LNCS, vol. 6538, pp. 70–87. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-18275-4_7
7. Liffiton, M.H., Sakallah, K.A.: Algorithms for computing minimal unsatisfiable subsets of constraints. J. Autom. Reason. **40**(1), 1–33 (2008)
8. Gurfinkel, A.: Program verification with constrained horn clauses (invited paper). In: Shoham, S., Vizel, Y. (eds.) Computer Aided Verification. CAV 2022. LNCS, vol. 13371, pp. 19–29. Springer, Cham (2022). https://doi.org/10.1007/978-3-031-13185-1_2
9. Chvalovský, K., Jakubův, J., Suda, M., Urban, J.: ENIGMA-NG: efficient neural and gradient-boosted inference guidance for E. In: Fontaine, P. (ed.) CADE 2019. LNCS (LNAI), vol. 11716, pp. 197–215. Springer, Cham (2019). https://doi.org/10.1007/978-3-030-29436-6_12
10. Jakubův, J., Urban, J.: ENIGMA: efficient learning-based inference guiding machine. In: Geuvers, H., England, M., Hasan, O., Rabe, F., Teschke, O. (eds.) CICM 2017. LNCS (LNAI), vol. 10383, pp. 292–302. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-62075-6_20
11. 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.) IJCAR 2020. LNCS (LNAI), vol. 12167, pp. 448–463. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-51054-1_29
12. Suda, M.: Vampire with a brain is a good ITP hammer. In: Konev, B., Reger, G. (eds.) FroCoS 2021. LNCS (LNAI), vol. 12941, pp. 192–209. Springer, Cham (2021). https://doi.org/10.1007/978-3-030-86205-3_11
13. Battaglia, P.W., et al.: Relational inductive biases, deep learning, and graph networks. CoRR, abs/1806.01261 (2018)
14. Wang, M., Tang, Y., Wang, J., Deng, J.: Premise selection for theorem proving by deep graph embedding. In: Guyon, I., et al. (eds.), Advances in Neural Information Processing Systems, vol. 30, pp. 2786–2796. Curran Associates Inc. (2017)
15. 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)
16. Selsam, D., Bjørner, N.: Neurocore: guiding high-performance SAT solvers with unsat-core predictions. CoRR, abs/1903.04671 (2019)
17. 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, 6–9 May 2019. OpenReview.net (2019)
18. CHC-COMP benchmarks. https://chc-comp.github.io/. Accessed 07 Sept 2023
19. Fedyukovich, G., Rümmer, P.: Competition report: CHC-COMP-21. In: Hojjat, H., Kafle, B., (eds.), Proceedings 8th Workshop on Horn Clauses for Verification and Synthesis, HCVS@ETAPS 2021, Virtual, 28th March 2021, vol. 344. EPTCS, pp. 91–108 (2021)
20. Repository for the training and evaluation dataset in this work. https://github.com/ChenchengLiang/Horn-graph-dataset. Accessed 07 Sept 2023
21. Harrison, J.: Handbook of Practical Logic and Automated Reasoning. Cambridge University Press, Cambridge (2009)

22. Barrett, C., Fontaine, P., Tinelli, C.: The SMT-LIB Standard: Version 2.6. Technical report, Department of Computer Science, The University of Iowa (2017). www.SMT-LIB.org
23. Goodfellow, I.J., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge (2016). http://www.deeplearningbook.org
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)
25. Schlichtkrull, M., Kipf, T.N., Bloem, P., van den Berg, R., Titov, I., Welling, M.: Modeling relational data with graph convolutional networks. In: Gangemi, A., et al. (eds.) ESWC 2018. LNCS, vol. 10843, pp. 593–607. Springer, Cham (2018). https://doi.org/10.1007/978-3-319-93417-4_38
26. Agarap, A.F.: Deep Learning using Rectified Linear Units (ReLU). arXiv e-prints: arXiv:1803.08375, March 2018
27. King, J.C.: Symbolic execution and program testing. Commun. ACM **19**(7), 385–394 (1976)
28. Hojjat, H., Rümmer, P.: The ELDARICA Horn solver. In: 2018 Formal Methods in Computer Aided Design (FMCAD), pp. 1–7 (2018)
29. Graf, S., Saidi, H.: Construction of abstract state graphs with PVS. In: Grumberg, O. (ed.) CAV 1997. LNCS, vol. 1254, pp. 72–83. Springer, Heidelberg (1997). https://doi.org/10.1007/3-540-63166-6_10
30. Ball, T., Podelski, A., Rajamani, S.K.: Boolean and Cartesian abstraction for model checking C programs. Int. J. Softw. Tools Technol. Transf. **5**(1), 49–58 (2003). https://doi.org/10.1007/s10009-002-0095-0
31. Rümmer, P., Hojjat, H., Kuncak, V.: Disjunctive interpolants for horn-clause verification. In: Sharygina, N., Veith, H. (eds.) CAV 2013. LNCS, vol. 8044, pp. 347–363. Springer, Heidelberg (2013). https://doi.org/10.1007/978-3-642-39799-8_24
32. Hojjat, H., Rümmer, P.: OptiRica: towards an efficient optimizing Horn solver. In: Hamilton, G.W., Kahsai, T., Proietti, M., (eds.), Proceedings 9th Workshop on Horn Clauses for Verification and Synthesis and 10th International Workshop on Verification and Program Transformation, HCVS/VPT@ETAPS 2022, and 10th International Workshop on Verification and Program TransformationMunich, Germany, 3rd April 2022, vol. 373. EPTCS, pp. 35–43 (2022)
33. David R. Cox. The regression analysis of binary sequences. J. R. Statist. Soc. Ser. B (Methodol.) **20**(2), 215–242 (1958)
34. Code repository for reproduce this work. https://github.com/ChenchengLiang/Relational-Hypergraph-Neural-Network-PyG. Accessed 07 Sept 2023
35. Bridge, J., Holden, S., Paulson, L.: Machine learning for first-order theorem proving. J. Autom. Reason. **53**, 08 (2014)
36. Cortes, C., Vapnik, V.: Support-vector networks. Mach. Learn. **20**(3), 273–297 (1995)
37. Rasmussen, C.E., Williams, C.K.I.: Gaussian Processes for Machine Learning (Adaptive Computation and Machine Learning). The MIT Press, Cambridge (2005)
38. Le, T.-D.B., Lo, D.: Deep specification mining. In: Proceedings of the 27th ACM SIGSOFT International Symposium on Software Testing and Analysis, ISSTA 2018, pp. 106–117. Association for Computing Machinery, New York, NY, USA (2018)
39. Richter, C., Wehrheim, H.: Attend and represent: a novel view on algorithm selection for software verification. In: 2020 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), pp. 1016–1028 (2020)

40. Vaswani, A., et al.: Attention is all you need. In: Guyon, I., et al. (eds.), Advances in Neural Information Processing Systems, vol. 30. Curran Associates Inc. (2017)
41. Richter, C., Hüllermeier, E., Jakobs, M.-C., Wehrheim, H.: Algorithm selection for software validation based on graph kernels. Autom. Softw. Eng. **27**(1), 153–186 (2020)
42. Scholkopf, B., Smola, A.J.: Learning with Kernels: Support Vector Machines, Regularization, Optimization, and Beyond. MIT Press, Cambridge (2001)
43. Selsam, D., Lamm, M., Bünz, B., Liang, P., de Moura, L., Dill, D.L.: Learning a SAT solver from single-bit supervision. CoRR, abs/1802.03685 (2018)
44. 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)
45. Scarselli, F., Gori, M., Tsoi, A.C., Hagenbuchner, M., Monfardini, G.: The graph neural network model. IEEE Trans. Neural Netw. **20**(1), 61–80 (2009)
46. Li, Y., Tarlow, D., Brockschmidt, M., Zemel, R.S.: Gated graph sequence neural networks. In: Bengio, Y., LeCun, Y., (eds.), 4th International Conference on Learning Representations, ICLR 2016, San Juan, Puerto Rico, 2–4 May 2016, Conference Track Proceedings (2016)
47. Marques-Silva, J.P., Sakallah, K.A.: GRASP: a search algorithm for propositional satisfiability. IEEE Trans. Comput. **48**(5), 506–521 (1999)

Paper III

Guiding Word Equation Solving Using Graph Neural Networks

Parosh Aziz Abdulla[1], Mohamed Faouzi Atig[1], Julie Cailler[2], Chencheng Liang[1](✉), and Philipp Rümmer[1,2](✉)

[1] Uppsala University, Uppsala, Sweden
{parosh.abdulla,mohamed_faouzi.atig,chencheng.liang}@it.uu.se
[2] University of Regensburg, Regensburg, Germany
{philipp.ruemmer,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 [9,38], and logical formulae [18,29,41,50]. 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 generated 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-

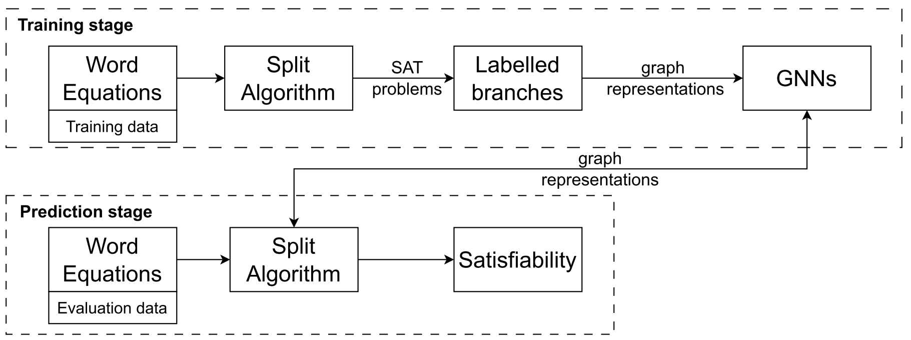


Fig. 1. The workflow diagram for the training and prediction stage

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:

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

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 $\pi : \Gamma \rightarrow \Sigma^*$ 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, $\rho_t : (\mathbb{R}^n)^{|N_v|} \rightarrow \mathbb{R}^n$ 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, $\eta_t : (\mathbb{R}^n)^2 \rightarrow \mathbb{R}^n$ 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 Fig. 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:

$$\mathit{Name}\ \frac{P}{\begin{array}{c|c|c} [\mathit{cond_1}] & & [\mathit{cond_n}] \\ C_1 & \cdots & C_n \end{array}}$$

Here, *Name* is the name of the rule, P is the premise, and C_is are the conclusions. Each $\mathit{cond_i}$ 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 (Fig. 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 (Fig. 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 (Fig. 2c) manages cases where one side begins with a letter and the other one with a 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 (Fig. 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 $\epsilon = X \wedge \phi$.

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

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

(a) Simplification rules

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

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

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 Fig. 2 are sound and locally complete.*

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 Fig. 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;
- $\alpha : N \to \mathit{For} \cup \{\text{SAT}, \text{UNSAT}\}$ 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 Fig. 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.

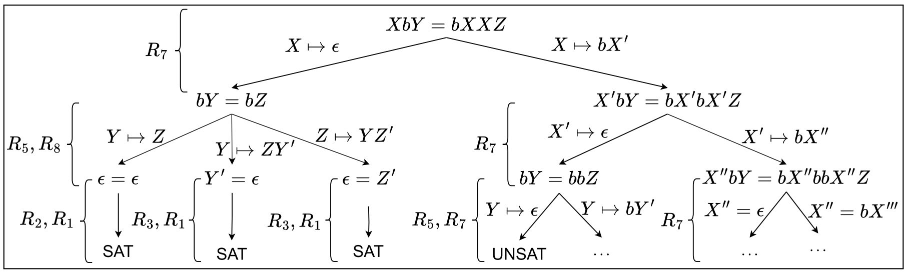


Fig. 3. Proof tree resulting from the word equation $XbY = bXXZ$

3.3 GNN-Guided Split Algorithm

We use the proof rules in Fig. 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 Fig. 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 $l^{step}_{BT_2}$ (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 Sect. 7.

The correctness of Algorithm 1 directly follows from the soundness and local completeness of the proof rules in Fig. 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.*

```
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 $l_{BT_2}, l^{step}_{BT_2}, l_{BT_3}$.
Output: The status of $\phi$: SAT, UNSAT, or UNKNOWN
1  begin
2      $res \leftarrow$ UNKNOWN
3      if $BT \in \{BT_1, BT_2\}$ then
4          $res \leftarrow solveEqsRec(\phi, 0, BT, \Sigma, \Gamma)$
5      if $BT = BT_3$ then
6          do
7              $res \leftarrow solveEqsRec(\phi, 0, BT_3, \Sigma, \Gamma)$
8              $l_{BT_3} \leftarrow l_{BT_3} + 1$
9          while $res \neq$ UNKNOWN
10     return $res$
```

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

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 $BT \in \{BT_1, BT_2, BT_3\}$;
       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
1  begin
2      if $BT = BT_2 \wedge currentDepth \geq l_{BT_2}$ then
3          $l_{BT_2} \leftarrow l_{BT_2} + l_{BT_2}^{step}$
4          return UNKNOWN
5      if $BT = BT_3 \wedge currentDepth \geq l_{BT_3}$ then
6          return UNKNOWN
7      $branches \leftarrow applyRules(\phi, \Sigma, \Gamma)$
8      $branches \leftarrow orderBranches(branches)$
9      $unknownFlag \leftarrow false$
10     foreach $child$ in $branches$ do
11         $res \leftarrow solveEqsRec(child, currentDepth + 1, BT, \Sigma, \Gamma)$
12         if $res$ = SAT then
13             return SAT
14         if $res$ = UNKNOWN then
15             $unknownFlag \leftarrow true$
16     if $unknownFlag$ then
17         return UNKNOWN
18     else
19         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:

$$w_1^l \# w_2^l \# ... \# w_n^l = w_1^r \# w_2^r \# ... \# w_n^r, \quad (3)$$

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 $\{a, b, c\} \subseteq \Sigma$.

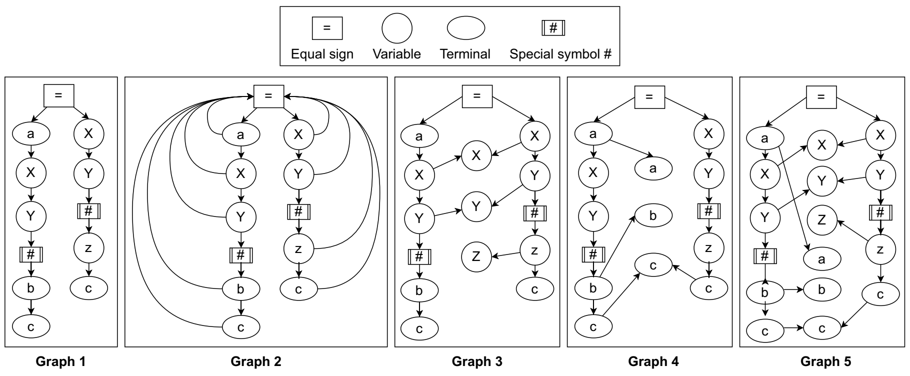


Fig. 4. The five graph representations for the word equation $aXY\#bc = XY\#Zc$ where X, Y, Z are variables and a, b, c are terminals

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 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 Sect. 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:

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

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

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

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

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

$$S = \{\Delta(v_i) \mid \sigma(v_i) = \text{SAT}\}$$

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 Fig. 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.

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:

$$X_n a X_n b X_{n-1} b X_{n-2} \cdots b X_1 = a X_n X_{n-1} X_{n-1} b X_{n-2} X_{n-2} b \cdots b X_1 X_1 baa \quad (6)$$

  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. 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

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

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 Sect. 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 Sect. 4.1), three strategies to apply GNN back to the algorithm (S1, S2, S3 in Sect. 4.4), and three backtracking strategies (BT_1, BT_2, BT_3 in Sect. 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

networks, and used a two message-passing layer (i.e. $t = 2$ in Eq. 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 s 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 Sect. 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 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 $l_{BT_2}^{step}$, 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 [18,29,41] exploring this path, this trend quickly expands to related fields. For instance, for SAT solvers [33,52], 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, 27,35]. They provide the empirical foundations for designing the learning task in Sect. 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 25 April 2024. https://smtlib.cs.uiowa.edu/benchmarks.shtml
2. DragonLi github repository (2024). Accessed 28 June 2024. https://github.com/ChenchengLiang/boosting-string-equation-solving-by-GNNs
3. Zenodo record of DragonLi (2024). Accessed 21 Aug 2024. https://zenodo.org/records/13354774
4. Abdulla, P.A., et al.: String constraints for verification. In: Biere, A., Bloem, R. (eds.) Computer Aided Verification, pp. 150–166. Springer International Publishing, Cham (2014)
5. Kroening, D., Păsăreanu, C.S. (eds.): Norn: an SMT solver for string constraints. In: Computer Aided Verification, pp. 462–469. Springer International Publishing, Cham (2015)
6. Abdulla, P.A., et al.: 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 arxvi preprint arxiv: abs/1711.00740 (2017). http://arxiv.org/abs/1711.00740
10. Barbosa, H., et al.: 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., et al.: CVC4. In: Gopalakrishnan, G., Qadeer, S. (eds.) CAV 2011. LNCS, vol. 6806, pp. 171–177. Springer, Heidelberg (2011). https://doi.org/10.1007/978-3-642-22110-1_14
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., et al.: 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., et al.: 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
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

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, pp. 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
22. Fan, W., et al.: 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. Artif. Intell. **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 arxiv preprint arxiv: abs/1905.10006 (2019)
42. Paszke, A., et al.: 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 arxiv preprint arxiv: 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: Lahiri, S.K., Wang, C. (eds.) CAV 2020. LNCS, vol. 12225, pp. 151–164. Springer, Cham (2020). https://doi.org/10.1007/978-3-030-53291-8_9
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., et al.: 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

Paper IV

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 [26]; while the precise complexity of the problem is still open, it is know to be NP-hard and in PSPACE [31].

Abdulla et al. recently proposed a Nielsen transformation-based algorithm for solving word equation problems [6,29]. 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 Automated Theorem Provers (ATPs) [37,11,5]. 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) [10] 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 [6]. We refer to the selection step as the *ranking process*. For this, we enhanced the existing algorithm [6] 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 [6] 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 [28] and cvc5 [9] 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 [14]. 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 [13], cvc5, Ostrich [12], and Woorpje [14]. 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:

- We adapt the Nielsen transformation-based algorithm [6] to allow control over the ordering of word equations at each iteration.
- 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.
- 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:

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

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

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 e is called* linear *if each variable occurs at most once. Otherwise, it is* non-linear.

Graph Neural Networks. *Message Passing-based GNNs* (MP-GNNs) [17] 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. 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.

In this study, we employ *Graph Convolutional Networks* (GCNs) [22] 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) [8] 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 the inference rules from [6]. The inference rules are provided in Appendix E for convenient reference.

The algorithm begins by simplifying ϕ, eliminating word equations with identical sides and word equations in solved form $x = w$ (Line 2), and checking the satisfiability of the conjunctive formula. 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 3), 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 branchesalternative prospective solving paths for the same equation. This step is called the *branching process.*

```
   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     $(res, \psi) \leftarrow$ SimplifyAndCheckFormula($\phi$)
 3     if $res \neq UKN$ then return res, $\psi$ else
 4         $\psi_s$ = RankEqs($\psi$)                          // Ranking process
 5         Branches = applyRules($\psi_s$)                    // Branching process
 6         $uknFlag \leftarrow 0$
 7         for b in Branches do
 8             $res_b, \psi_b$=splitEquations($b$)
 9             if $res_b = SAT$ then return $SAT$, $\psi_b$
10             if $res_b = UKN$ then $uknFlag \leftarrow 1$
11         if $uknFlag = 1$ then return $UKN$, $\psi$ else return $UNSAT$, $\psi$
```

Algorithm 1: SplitEquations algorithm.

Notably, rules R_7 and R_8 generate two and three branches, respectively, while all other rules do not cause any branching.

Next, the splitEquations call (Line 8) recursively checks the satisfiability of each branch. Let $\{b_1, \ldots, b_n\}$ be the set of branches. ϕ has status SAT if at least one branch b_i is satisfiable, UNSAT if all branches are unsatisfiable, and UKN otherwise.

Since the inference rules apply to the leftmost equation, the performance and termination of the algorithm are strongly influenced by both the order in which branches are processed (Line 7) and the ordering of equations in ϕ (Line 4). While the impact of branch ordering has been studied in [6], 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. 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 [6]:

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 $\phi = (Xb = bXX \land \epsilon = \epsilon \land X = a)$, 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 last two equations, as the 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.

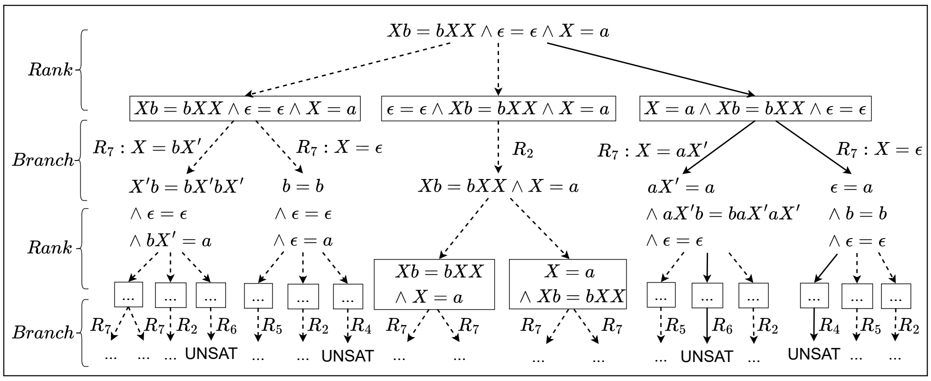

$Xb = bXX \land \epsilon = \epsilon \land X = a$
Rank
$Xb = bXX \land \epsilon = \epsilon \land X = a$
$\epsilon = \epsilon \land Xb = bXX \land X = a$
$X = a \land Xb = bXX \land \epsilon = \epsilon$
Branch
$R_7 : X = bX'$
$R_7 : X = \epsilon$
R_2
$R_7 : X = aX'$
$R_7 : X = \epsilon$
$X'b = bX'bX' \land \epsilon = \epsilon \land bX' = a$
$b = b \land \epsilon = \epsilon \land \epsilon = a$
$Xb = bXX \land X = a$
$aX' = a \land aX'b = baX'aX' \land \epsilon = \epsilon$
$\epsilon = a \land b = b \land \epsilon = \epsilon$
Rank
$Xb = bXX \land X = a$
$X = a \land Xb = bXX$
Branch
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 $Xb = bXX \land \epsilon = \epsilon \land X = a$. 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 the training of classification models to rank a set of word equations for ranking with varying lengths. 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:

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

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 $\mathcal{E} = \{e_1, \dots, e_n\}$ of a conjunctive word equation, the corresponding labels $Y_n = (y_1, \dots, y_n)$ is computed by

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

For both sources, when $\sum_{i=1}^{n} y_i > 1$, we discard the equation with label 1 until $\sum_{i=1}^{n} y_i = 1$ to ensure that the label for each ranking process is consistent. 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 [6]. 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 $G = (V, E, v_=, V_\mathrm{T}, V_\mathrm{Var}, V_\mathrm{T}^0, V_\mathrm{T}^1, V_\mathrm{Var}^0, V_\mathrm{Var}^1)$, 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_T^0 and V_T^1 are special nodes representing letter occurrences and V_Var^0 and V_Var^1 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, 101, and 10, respectively. We encode these as sequentially connected nodes: (V_{Var}^1, V_{Var}^1) for X, (V_{Var}^1, V_{Var}^0) for Y, and (V_T^1, V_T^0, V_T^1) for a. Finally, we connect the roots of these nodes to their corresponding variable and letter nodes.

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

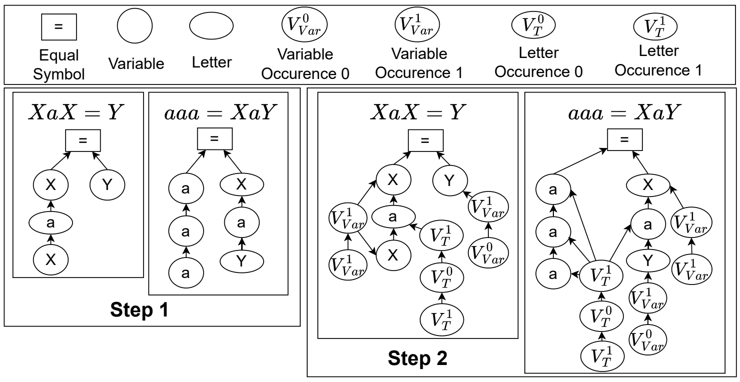


Fig. 2: The steps for constructing graph representation for the conjunctive word equations $XaX = Y \wedge aaa = XaY$ where X, Y are variables and a is a letter.

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 indicating 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 are passed to a trainable embedding function $\mathrm{MLP}_0 : \mathbb{Z} \to \mathbb{R}^m$ to compute the all 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 $H_{G_i} = \sum_i H^T_i$.

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 $\mathbf{z} = \mathrm{MLP}_1(H_{G_i}) = (z_1, z_2)$. The score for graph i is then computed as $y_i = \mathit{softmax}(\mathbf{z})_1$ for $y_i \in \hat{Y}_n$ where $\mathit{softmax}(\mathbf{z}) = (\frac{e^{z_1}}{\sum_{j=1}^n e^{z_j}}, \ldots, \frac{e^{z_n}}{\sum_{j=1}^n e^{z_j}})$ 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 $H_G = \sum(H_{G_1}, \ldots, H_{G_n})$. Then, we compute the score by of each graph by $y_i = \mathit{softmax}(\mathrm{MLP}_2(H_{G_i} || H_G))_1$ for $y_i \in \hat{Y}_n$ where $\mathrm{MLP}_2 : \mathbb{R}^{2m} \to \mathbb{R}^2$ 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 $\hat{Y}_n = \text{MLP}_3(H_{G_1}, \ldots, H_{G_n})$ where $\text{MLP}_3 : \mathbb{R}^{nm} \to \mathbb{R}^n$ 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 $loss = -\frac{1}{n}\sum_1^n y_i \log(\hat{y}_i)$ 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 random to the baseline.
- **RE3 — GNN:** Equation 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 equation early in the process has a greater impact on performance than do 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 is detailed in Appendix F.

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 [1], as well as those from the Zaligvinder benchmark suite [2], 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 [14,6]. 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 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 each benchmark is shown in Table 3 in the Appendix A.

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

5.2 Training Data Collection

Table 1 outlines the training data collection process. 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 Appendix C. 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 heuristics 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.

5.3 Comparison with Other Solvers

The experimental settings, including hardware, GNN hyperparameters, and parameter configurations, are detailed in Appendix D.

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 [28], Z3-Noodler [13], cvc5 [9], Ostrich [12], and Woopje [14].

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

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., [23]), which are present in other 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 [4], 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 through 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 in Appendix B. All benchmarks, evaluation results, and implementation details, including hyperparameters, are available in our GitHub repository [3].

6 Related Work

Axel Thue [32] 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 [26], but the complexity is non-elementary. Plandowski [31] designed an algorithm that reduces the complexity to P-SPACE by using a form of run-length encoding to represent strings and variables more compactly during the solving process. Artur Je [21] proposed a nondeterministic algorithm 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 [30] 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. [16] explored graph representations and graph traversal methods to optimize the solving process for word equations, while Day et al. [14] 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. [15] proposed a transformation system that extends the Nielsen transformation [24] to work with linear length constraints.

Deep learning [18] has been integrated with various formal verification techniques, such as scheduling SMT solvers [19], loop invariant reasoning [35,36], and guiding premise selection for Automated Theorem Provers (ATPs) [38]. Closely related work in learning from Minimal Unsatisfiable Subsets (MUSes) includes NeuroSAT [33,34], which utilizes GNNs to predict the probability of variables appearing in unsat cores, guiding variable branching decisions for Conflict-Driven Clause Learning (CDCL) [27]. Additionally, some recent works [7,25] explore learning MUSes to guide CHC [20] solvers.

7 Conclusion and Future Work

In this work, we extend a Nielsen transformation based algorithm [6] 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.

References

1. The satisfiability modulo theories library (SMT-LIB), accessed: 2024-04-25, https://smtlib.cs.uiowa.edu/benchmarks.shtml
2. Zaligvinder: A string solving benchmark framework, accessed: 2024-09-24, https://zaligvinder.github.io/
3. DragonLi github repository branch:rank (2025), accessed: 2025-01-24. https://github.com/ChenchengLiang/boosting-string-equation-solving-by-GNNs/tree/rank
4. wordeq_solver (2025), accessed: 2025-01-24. https://github.com/tage64/wordeq_solver/
5. 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
6. 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)
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. 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)
10. 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
11. Brtek, 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
12. 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
13. 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)
14. 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)

15. 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. 8797. FormaliSE '20, Association for Computing Machinery, New York, NY, USA (2020). https://doi.org/10.1145/3372020.3391556
16. 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)
17. 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
18. Goodfellow, I.J., Bengio, Y., Courville, A.: Deep Learning. MIT Press, Cambridge, MA, USA (2016), http://www.deeplearningbook.org
19. Hla, J., Mojek, 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
20. Horn, A.: On sentences which are true of direct unions of algebras. Journal of Symbolic Logic **16**(1), 1421 (1951). https://doi.org/10.2307/2268661
21. Je, A.: Recompression: a simple and powerful technique for word equations (2014), https://arxiv.org/abs/1203.3705
22. 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
23. 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
24. Levi, F.W.: On semigroups. Bull. Calcutta Math. Soc **36**(141-146), 82 (1944)
25. 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
26. Makanin, G.S.: The problem of solvability of equations in a free semigroup. Math. Sb. (N.S.) **103(145)**(2(6)), 147–236 (1977)
27. 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
28. de Moura, L., Bjrner, 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)
29. Nielsen, J.: Die Isomorphismen der allgemeinen, unendlichen Gruppe mit zwei Erzeugenden. Mathematische Annalen **78**, 385–397 (1917), https://api.semanticscholar.org/CorpusID:119726936

30. Pin, J.E., Perrin, D.: Infinite Words: Automata, Semigroups, Logic and Games. Elsevier (2004), https://hal.science/hal-00112831
31. 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
32. Power, J.F.: Thue's 1914 paper: a translation. CoRR **abs/1308.5858** (2013), http://arxiv.org/abs/1308.5858
33. Selsam, D., Bjørner, N.: Neurocore: Guiding high-performance SAT solvers with unsat-core predictions. CoRR **abs/1903.04671** (2019)
34. 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
35. 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)
36. 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 2124, 2020, Proceedings, Part II. p. 151164. Springer-Verlag, Berlin, Heidelberg (2020)
37. 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)
38. 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)

A Statistics of Evaluation Data

Table 3 presents the statistics of the evaluation data for each benchmark. For each benchmark, we generate a total of 1000 problems for evaluation. The total variable and letter occurrence ratios are computed by counting the total occurrences of variables and letters across all problems in a benchmark and dividing by the total number of terms. The column "Single equation" reports the length, number of variables, number of letters, and the maximum occurrences of a single variable for individual equations across all problems in the benchmarks. The column "Problem level" measures the number of equations, number of variables, number of letters, and maximum occurrences of a single variable for each conjunctive word equation in the benchmarks. For these columns, the metrics reported are the minimum, maximum, average, and standard deviation values.

For our DragonLi and Z3, the difficulties of these benchmarks are proportional to the total variable occurrence ratio and the length of the equations, as an increase in these values leads to a decrease in the number of solved problems, as shown in Table 2.

The number of equations in problems for Benchmark B is set to 50, differing from other benchmarks. This is because its average maximum single variable occurrence is high (53.5), meaning that, on average, a single variable appears 53.5 times across all equations in a problem. Intuitively, this can be interpreted as a formula with more constraints. Consequently, when the number of equations is set to 100, nearly all problems become UNSAT. When training a data-driven model, the model tends to predict UNSAT to achieve optimal performance. To mitigate this, the number of equations is reduced to 50.

B Ablation Study

Table 4 presents the number of solved SAT and UNSAT problems across different training tasks and GNN options for implementing RankEqs. Benchmark C is excluded from evaluation because the ranking process has no significant impact on performance when non-linearity is high, resulting in insufficient data to train the GNN models.

The differences in the number of solved SAT problems across various configurations are relatively small. This can be attributed to two primary reasons. First, if each conjunct in a conjunctive formula is independent, the ordering would not impact the outcome for SAT problems, as all conjuncts must independently satisfy the formula. Second, conjuncts in conjunctive word equations are usually not fully independent, as they often share variables. Consequently, the sequence in which the conjuncts are processed influences the solving time for SAT problems.

Conversely, for UNSAT problems, the sequence of the conjuncts affects performance more regardless of the independence of individual conjuncts.

Therefore, the following discussion primarily focuses on the UNSAT problems.

Table 3: Statistics of benchmarks in evaluation data. Min, max, avg, and std denote minimum, maximum, average, and standard deviation value respectively.

Bench	A1	A2	B	C
Total	1000	1000	1000	1000
Total variable and letter occurrence ratio				
Varriable	0.135	0.130	0.375	0.416
Letter	0.864	0.869	0.625	0.583

Bench	A1	A2	B	C	A1	A2	B	C
	Single equation				Problem level			
	Length				Number of equations			
min	2	74	2	13	1	1	2	1
max	121	120	110	372	100	100	50	100
avg	38.7	103.2	43.4	155.0	51.4	50.9	26.8	51.8
std	24.3	7.2	20.6	82.0	28.8	28.8	14.0	28.6
	Number of variables				Number of variables			
min	0	1	0	6	1	1	2	1
max	21	29	42	155	100	100	50	100
avg	5.2	13.4	16.3	64.5	51.4	50.9	26.8	51.8
std	3.0	4.6	9.7	34.1	28.8	28.8	14.0	28.6
	Number of terminals				Number of terminals			
min	0	52	0	4	1	69	7	15
max	120	119	100	264	3734	9234	1796	10059
avg	33.4	89.8	27.2	90.5	1720.4	4575.1	729.0	4689.9
std	25.1	11.3	23.3	50.8	975.3	2593.4	401.2	2623.3
	Max single variable occurrences				Max single variable occurrences			
min	0	1	0	3	1	1	0	4
max	1	1	10	14	34	29	109	14
avg	0.97	1	3.6	6.2	15.9	13.1	53.5	10.2
std	0.16	0	2.1	1.9	7.9	6.0	26.3	1.3

In terms of the training tasks, the computational overhead follows the order Task 2 > Task 1 > Task 3. The reasons are outlined as follows:

– **Task 2**: This task first computes the graph representation for each word equation, H_{G_i}, and then aggregates these into a global feature representation using $H_G = \sum(H_{G_1}, \ldots, H_{G_n})$. To compute the ranking score of each conjunct in the conjunctive word equations, forward propagation must be performed n times with the concatenated input $H_{G_i}||H_G$ for each GNN layer.
– **Task 1**: This task requires forward propagation n times using the individual graph representations H_{G_i}, but it does not involve computing or concatenating the global feature representation H_G.
– **Task 3**: This task only requires a single forward propagation step using the set of individual graph representations $(H_{G_1}, \ldots, H_{G_n})$.

The numbers of solved UNSAT problems in columns **SE3** and **SE4** for Benchmarks A2 and B provide supporting evidence. For Benchmark A1, it is not sensitive to the overhead of GNN calls because the overall number of iterations required to solve the problems is low.

In terms of GNN options, **SE3** involves the most of calls to the GNN model. It is generally outperformed by other options across all tasks and benchmarks due to its overhead.

SE4 involves a random process. Its performance is below average for Benchmarks A1 and A2 because it disrupts the predictions of GNNs. For Benchmark B, it achieves the best performance for Tasks 1 and 3. This is because Benchmark B is non-linear; repeatedly applying inference rules to the same non-linear word equation leads to an increase in length and potentially non-termination. The random process helps to escape such situations. However, for Task 3, the overhead of calling the GNN diminishes its advantage.

SE5 has the least overhead from model calls. For Benchmarks A1 and A2, it outperforms most other options. For Benchmark B, it achieves average performance.

SE6 and **SE7** maintain high performance across all benchmarks but do not consistently outperform **SE5**.

C Statistics of MUSes

Table 5 presents the statistics of equations in the MUS for each benchmark, corresponding to the row “Have MUS” in Table 1. Each conjunctive word equation is associated with one MUS.

D 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

Table 4: Number of solved problems for different GNN options and training tasks. The bold numbers, along with their corresponding GNN options and tasks, are referenced in the rows labeled "GNN" in Table 2. The columns "3" ,"4", ..., "7" corresponds to **RE3**, **RE4**, ..., **RE7** in Section 4.4 respectively.

Bench		A1					A2					B				
		RankEqs GNN options														
Tasks		3	4	5	6	7	3	4	5	6	7	3	4	5	6	7
1	SAT	23	23	24	24	24	34	42	59	59	59	10	10	11	11	11
	UNSAT	945	938	954	955	955	404	521	837	829	824	776	876	815	805	833
2	SAT	24	23	**24**	24	24	49	48	**59**	59	59	9	10	**11**	9	11
	UNSAT	946	945	**961**	955	954	300	468	**836**	833	826	78	256	**821**	767	829
3	SAT	24	23	24	24	24	48	38	59	59	59	9	10	11	11	11
	UNSAT	936	940	951	955	953	825	779	816	832	825	884	892	792	805	829

Table 5: Statistics of MUSes.

Bench	A1		A2		B		C	
Total	909		1024		2996		15875	
Total variable and letter occurrence ratio								
Varriable	0.124		0.139		0.489		0.408	
Letter	0.875		0.861		0.511		0.592	
Bench	A1	A2	B	C	A1	A2	B	C
	Single equation				Problem level			
	Length				Number of equations			
min	2	76	11	12	2	1	1	1
max	117	120	114	354	8	8	8	7
avg	40.6	101.7	40.1	66.2	2.2	3.3	2.6	2.0
std	23.5	7.4	17.1	60.8	0.7	1.0	1.3	0.5
	Number of variables				Number of variables			
min	1	2	10	6	2	10	10	7
max	16	26	41	148	49	117	173	525
avg	5.1	14.2	19.6	27.0	11.3	46.3	50.4	54.7
std	2.6	4.6	6.4	25.5	5.4	17.3	26.1	56.3
	Number of terminals				Number of terminals			
min	1	57	1	4	4	69	1	4
max	111	118	99	248	318	710	362	758
avg	35.6	87.5	20.5	39.2	79.2	258.6	52.6	79.3
std	23.8	11.4	17.5	36.8	39.0	90.2	40.1	78.7
	Max single variable occurrences				Max single variable occurrences			
min	1	1	1	3	1	1	2	3
max	1	1	10	14	1	1	10	14
avg	1	1	4.4	4.3	1	1	5.4	4.6
std	0	0	1.5	1.5	0	0	1.4	1.6

fixed the inference rule to the prefix version, meaning it always simplifies the word equation starting from the leftmost term. 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 2) 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 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.

E Calculus for Word Equations [6]

The calculus for word equations proposed by [6] comprises a set of inference rules. Each inference rule is expressed in the following form:

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

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 conjunctive word equation and each C_i is either a conjunctive word equation or a final state, SAT or UNSAT.

To introduce the inference 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 3a) define how to conclude SAT, 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 3b) refer to cases in which each word starts with a letter. The rules simplify the leftmost 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 3c) manages cases where one side begins with a letter and the other one with a 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 and R_9 (Figure 3d) 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.

F Workflow

The workflow of our framework is illustrated in Figure 4. For a benchmark, we begin by randomly splitting the dataset into training and evaluation subsets. During the training phase, we first input the word equation problems into the split algorithm ranking option **RE1** to obtain their satisfiabilities and separate

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

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

(a) Simplification Rule

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

with a, b two different Terminals from Σ.

(b) Terminal-Terminal Rule

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

with X' a *fresh* element of Γ.

(c) Variable-Terminal Rules

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

$$R_9\,\frac{X \cdot u = X \cdot v \wedge \phi}{u = v \wedge \phi}$$

with $X \neq Y$ and X', Y' *fresh* elements of Γ.

(d) Variable-Variable Rule

Fig. 3: Inference rules of the proof system for word equations

them into SAT, UNSAT, and UNKNOWN sets. For the SAT and UNSAT sets, we discard them since our algorithm already know how to solve them. For the UNKNOWN set, the problems are passed to other solvers, such as z3 and cvc5. If the solver concludes UNSAT, we systematically identify Minimal Unsatisfiable Subsets (MUSes) of conjunctive word equations by exhaustively checking the satisfiability of subsets, starting with individual equations and stopping upon finding the first MUS. We use the MUSes to rank and sort conjunctive word equations unsolvable by the split algorithm, then reprocess the sorted equations with it. This allows the split algorithm to solve some problems and construct proof trees. Then, we can extract the labeled data from both and-or tree and MUSes. The way of label them can be found in Section 4.

Next, we convert the labeled conjunctive word equations from textual to graph format, enabling the model with GNN layers to process them. The model

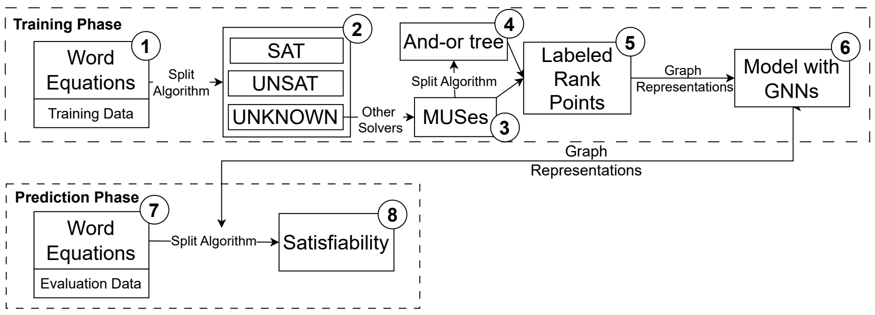


Fig. 4: The workflow diagram for the training and prediction phase.

takes a rank point (i.e., a conjunction of word equations) as input and outputs corresponding scores that indicate the priority of the conjuncts.

In the prediction phase, during the step where inference rules are applied, the trained model ranks and sorts the conjuncts. The equation with the highest score is then selected for apply the branching inference rules.