Broadening the Scope of Graph Regression: Introducing A Novel Dataset with Multiple Representation Settings

Anonymous authors
Paper under double-blind review

Abstract

Graph regression is a vital task across various domains, however, the majority of publicly available datasets for graph regression are concentrated in the fields of chemistry, drug discovery, and bioinformatics. This narrow focus on dataset availability restricts the development and application of predictive models in other important areas. Here, we introduce a novel graph regression dataset tailored to the domain of software performance prediction, specifically focusing on estimating the execution time of source code. Accurately predicting execution time is crucial for developers, as it provides early insights into the code's complexity. Furthermore, it also facilitates better decision-making in code optimization and refactoring processes. Source code can be represented syntactically as trees and semantically as graphs, capturing the relationships between different code components. In this work, we integrate these two perspectives to create a unified graph representation of source code. We present two versions of the dataset: `RelSC` (Relational Source Code), which incorporates node features, and `Multi-RelSC` (Multi-Relational Source Code), which treats the graphs as multi-relational, allowing nodes to be connected by multiple edges, each representing a distinct semantic relationship. Finally, we apply various Graph Neural Network models to assess their performance in this relatively unexplored task. Our findings demonstrate the potential of these datasets to advance the field of graph regression, particularly in the context of software performance prediction.

1 Introduction

Graph Neural Networks (GNNs)(Scarselli et al., 2008; Micheli, 2009) have demonstrated outstanding performance in processing network data across various real-world applications, ranging from biology to recommendation systems. Their ability to effectively model complex relationships between entities, capture structural dependencies, and incorporate node and edge features has made GNNs an essential tool in a variety of domains. High performance in GNNs is attributed not only to advancements in architectural design Kipf and Welling (2016); Hamilton et al. (2017a); Veličković et al. (2017); Gasteiger et al. (2018); Zhang and Chen (2018); Wu et al. (2019); Zhang et al. (2021a); Lachi et al. (2024); Zaghen et al. (2024) but also to the availability of publicly accessible benchmark datasets Armstrong et al. (2013); Hu et al. (2020a); Morris et al. (2020); Dwivedi et al. (2022); Zhiyao et al. (2024); Huang et al. (2024). These benchmarks have played a crucial role in facilitating research progress by providing standardized datasets and tasks, enabling researchers to evaluate, compare, and improve their models consistently.

However, while the availability of public datasets for node and graph classification has driven rapid advancements across fields such as biology Zhang et al. (2021b); Bongini et al. (2022), mobility Jiang and Luo (2022), social networks Li et al. (2023), and recommendation systems Fan et al. (2019), the same is not true for graph regression tasks. Public datasets for graph regression are predominantly concentrated in specific fields, particularly in Chemistry and Drug Discovery Jiang et al. (2021). These datasets have been instrumental in advancing GNN-based models for applications like molecular property prediction Wieder et al. (2020) and drug-target interaction Zhang et al. (2022). Despite their utility, this narrow focus presents a significant limitation: the exploration of graph regression in other domains remains largely underdeveloped due to the lack of diverse, high-quality datasets.

This scarcity of benchmarks beyond Chemistry and Drug Discovery restricts researchers' ability to fully explore the potential of GNNs in graph regression tasks across other fields. Domains such as finance, transportation, environmental modeling, and even social sciences could greatly benefit from graph regression models, but the absence of appropriate datasets makes it challenging to develop, adapt, and evaluate these models effectively. Addressing this gap is essential for expanding the applicability of GNNs to a broader set of problems, enabling the development of more generalizable models, and pushing the boundaries of graph-based machine learning.

In this paper, we address this issue by introducing two novel graph regression datasets specifically tailored to the domain of software performance prediction, focusing on the estimation of execution time for source code. Accurately predicting execution time is critical for developers, as it provides early insights into code complexity, helping in decision-making processes related to code optimization and refactoring. These new datasets not only broaden the scope of graph regression tasks but also offer a valuable resource for exploring GNN applications in software engineering, contributing to a more diverse set of benchmarks for future research. In the literature, source code has been analyzed using three well-known representations: Abstract Syntax Tree (AST), Control Flow Graph (CFG), and Data Flow Graph (DFG). Each of these representations captures different aspects of the source code. Inspired by Samoaa et al. (2022a), we enhance the AST by incorporating information derived from both the CFG and the DFG. We create two versions of our dataset: `RelSC` and `Multi-RelSC`. The `RelSC` dataset features relational graphs that represent Java source code along with their corresponding execution times. In contrast, the `Multi-RelSC` dataset includes multi-relational graphs, where the same source and destination nodes can be connected by various types of relationships. Additionally, in `Multi-RelSC`, node types are derived from the semantic information of the source code, while still representing Java source code and their execution times.

In summary, our contributions are as follows:

- **Unified Framework:** We present a comprehensive framework for converting source code into graph data by integrating information from ASTs, CFGs, and DFGs. This unified approach enhances the representation of source code, capturing a broader range of structural and semantic features.
- **Publicly Available Datasets:** We introduce two novel, publicly accessible datasets for execution time prediction. The first dataset, `RelSC`, features homogeneous networks representing Java source codes and their associated execution times. The second dataset, `Multi-RelSC`, consists of heterogeneous networks where node types are derived from the semantic information of the source code, also including Java source codes and their execution times.

These contributions not only broaden the scope of graph regression tasks but also provide a principled procedure to convert any Java source code into graph data.

2 Related Work

Several open datasets have been released over the past decades, with a predominant focus on Chemistry and Drug Discovery. For molecular property prediction, datasets such as QM9 Wu et al. (2018) and ZINC Gómez-Bombarelli et al. (2018) are used to predict various properties of small molecules. In the realm of solubility and free energy prediction, datasets like ESOL Li et al. (2022) and Freesolv Mobley and Guthrie (2014) aim to forecast the solubility and free energy of molecules. Similarly, Peptides-struct Dwivedi et al. (2022) is employed to predict aggregated 3D properties of peptides at the graph level. PDBbind Liu et al. (2015) is focused on the study of interactions between proteins and ligands. Toxicity and bioactivity prediction tasks utilize datasets such as ogbg-moltox21 Hu et al. (2020a) and ogbg-moltoxcast Hu et al. (2020a) to assess molecular toxicity and bioactivity. Additionally, datasets like ogbg-mollipo Hu et al. (2020a) are dedicated to lipophilicity prediction, while ogbg-molesol Hu et al. (2020a) is used for solubility prediction. Furthermore, the work by Liu et al. Liu et al. (2022) utilizes monomers as polymer graphs to predict properties such as the glass transition temperature. While significant progress has been made in these domains, there is a growing need for comprehensive benchmarks and datasets in other fields to further advance the state of graph regression tasks across diverse applications.

3 Preliminaries

In this section, we introduce the foundational concepts essential for understanding the core contributions of our work. Specifically, we present three key techniques for representing source code as graphs: the Abstract Syntax Tree (AST), the Control Flow Graph (CFG), and the Data Flow Graph (DFG). These representations form the basis for various program analysis methods and are critical for the discussions that follow.

3.1 Abstract Syntax Trees

ASTs Samoaa et al. (2023a) offer a hierarchical abstraction of source code, focusing on core programming constructs such as variables, operators, and control structures, while ignoring superficial syntactic details like punctuation. Each node in an AST represents a construct from the source code, with edges defining relationships based on the language's syntax rules. The root typically represents the entire program, and the leaves correspond to basic elements like literals or variable names Samoaa et al. (2023b). The process of building an AST involves parsing the source code according to its grammar, creating a structured representation that supports tasks such as code analysis, optimization, and refactoring Samoaa et al. (2022b). ASTs are widely used in applications such as static analysis, bug detection, and even machine learning-based techniques for code summarization and generation.

To gain a deeper understanding of ASTs, in Listing 1 we report a snippet of code and its AST representation is shown in figure 2.

```
public static int factorial(int n) {
  if (n <= 1) {
    return 1;
  } else {
    return n * factorial(n - 1);
  }
}
```

Figure 1: Simple example of Java source code

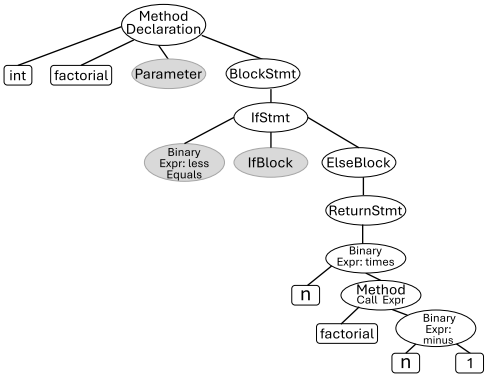

Figure 2: Simplified abstract syntax tree (AST) representing the illustrative example presented in Listing on the left.

3.2 Control Flow Graph (CFG)

To gain insights into the runtime behaviour and the potential paths that may be taken during code execution, Control Flow Graphs (CFG) are essential. The CFG of our Java method, depicted in Figure 3(left), provides a high-level view of all possible execution paths, from the method's entry point to its return statements.

In conclusion, the integration of these different code representations allows for a comprehensive analysis and modelling of the behaviour, structure, and data flow within a software system, which is particularly valuable for machine learning-driven software engineering research.

3.3 Data Flow Graph (DFG)

While the AST provides essential insights into the syntactic structure of code, it falls short in representing the movement and interaction of data within the program. Data Flow Graphs (DFG) address this limitation by illustrating the flow of data between variables and computations. As shown in Figure 3(right), a DFG captures the dependencies between various components of the code, offering a clear view of how data propagates throughout the program.

Figure 3: **Left:** Simplified control flow graph (CFG) representing the illustrative example presented in Listing 1. Package declarations, import statements, as well as the declaration in Line 15 are skipped for brevity. **Right:** Simplified data flow graph (DFG) representing the illustrative example presented in Listing 1. Package declarations, import statements, as well as the declaration in Line 15 are skipped for brevity.

3.4 Graph Neural Network

Graph Neural Networks (GNNs) are a type of neural network architecture specifically designed for analyzing graph-structured data. GNNs utilize a mechanism known as message passing, which allows for localized computation across the graph (Gilmer et al., 2017). In essence, the feature vector of each node is iteratively updated by incorporating information from its neighboring nodes. After l iterations, $\mathbf{x}_v^l$ encodes both the structural and attribute information from the l-hop neighborhood of node v.

More formally, the output of the l-th layer of a GNN is defined as:

$$\mathbf{x}_v^l = \text{COMB}^{(l)}(\mathbf{x}_v^{l-1}, \text{AGG}^{(l)}(\{\mathbf{x}_u^{l-1},\ u \in N[v]\})) \tag{1}$$

Here, $\text{AGG}^{(l)}$ refers to the aggregation function that gathers features from the neighbours $N[v]$ at the $(l-1)$-th iteration, while $\text{COMB}^{(l)}$ combines the features of the node itself with those of its neighbours. For graph-level tasks such as classification or regression, a global readout function is applied to the node embeddings to produce the final output:

$$\mathbf{o} \ = \ \text{READ}(\{\mathbf{x}_v^L,\ v \in V_G\}). \tag{2}$$

The READ function can be implemented as a sum, mean, or max overall node features or through more sophisticated approaches (Bruna et al., 2013; Yuan and Ji, 2020; Khasahmadi et al., 2020).

RGCNs Schlichtkrull et al. (2017) extend GNNs to handle multi-relational graphs by incorporating different relation types in the message passing mechanism (Schlichtkrull et al., 2017). In this framework, each relation type has its own set of parameters, allowing for learning distinct transformations for each relation.

Several architectures have been proposedVeličković et al. (2018); Hamilton et al. (2017b); Xu et al. (2018a); Defferrard et al. (2016), all utilizing the same underlying mechanism but differing in their choice of COMB and AGG functions.

Multi-relational GNNs, such as Relational Graph Convolutional Networks (Schlichtkrull et al., 2017), are specifically designed to handle graphs with multiple types of relations between nodes. In this framework, the message passing mechanism is extended to account for relation types, ensuring that information from different relations is treated distinctively. For a multi-relational graph $G = (V, E, R)$ where R is the set of relation types, the feature update for a node $v \in V$ in the l-th layer is defined as:

$$\mathbf{x}_v^l = \sigma\left(\sum_r^R \sum_u^{N_r(v)} \frac{1}{c_{r,v,u}} \mathbf{W}_r \mathbf{x}_u^{l-1} + \mathbf{W}_0 \mathbf{x}_v^{l-1}\right) \tag{3}$$

where $N_r(v)$ represents the neighbors of node V connected by relation r, $\mathbf{W}_r$ is a learnable weight matrix specific to relation r, $c_{r,v,u}$ is a normalization constant that can account for the degree of nodes, $\mathbf{W}_0$ is a

weight matrix for the self-loop connection, and σ is a non-linear activation function. In this formulation, the feature propagation process aggregates messages from neighbors for each relation type separately, applying distinct transformations before combining them. This mechanism allows the model to learn relation-specific patterns, making it particularly suitable for tasks such as knowledge graph completion and multi-relational node classification. Additionally, a global readout function READ can be applied to obtain graph-level outputs as described in Equation 2. Recent advancements in RGCNs have improved multi-relational data modeling Zhu et al. (2019); Yun et al. (2019); Hu et al. (2020b); Lv et al. (2021); Yu et al. (2021); Mitra et al. (2022); Ferrini et al. (2024a;b), yet diverse benchmarks remain limited. This article introduces a dataset and framework to convert Java source code into relational and multi-relational graphs, capturing structural and semantic aspects. Focused on software performance prediction, it offers a novel benchmark for RGCNs in underexplored domains.

4 Proposed Datasets

The proposed dataset focuses on predicting the execution time of Java source code, providing an early estimate of code complexity. This is particularly valuable when using cloud computing services, where execution time plays a critical role. The dataset consists of Java code files paired with their corresponding execution times. Each file is parsed into an AST, which is then augmented with edges representing control and data flows, offering a comprehensive view of both code structure and behaviour.

4.1 Data Collection

For our experiments, we employed two different real-world datasets of performance measurements to ensure robustness. The first dataset (*OSSBuild*) consists of actual build data sourced from the continuous integration systems of four open-source projects. The second dataset (*HadoopTests*) is a larger collection that we gathered by repeatedly running the unit tests of the Hadoop open-source project in a controlled environment. A summary of both datasets can be found in Table 1. To address the quality, diversity, and representativeness of our datasets, as well as the steps taken to mitigate potential biases in the collected data and collection process, we provide a detailed analysis in Appendix G.

In the following subsections, we provide further details about each dataset used in our experimental studies.

4.1.1 OSSBuild Dataset

This dataset, initially utilized in Samoaa et al. (2022a), contains data on test execution times from production build systems for four open-source projects: systemDS [1], H2 [2], Dubbo [3], and RDF4J [4]. These projects utilize public continuous integration servers, from which we extracted test execution times as a proxy for performance during the summer of 2021. Table 1 (top) presents basic statistics about the projects in this dataset. "Files" indicates the number of unit test files for which we collected execution times, and each file will be represented as one graph, while "Avg.Nodes" relates to the average number of nodes in the resulting graphs. Prior to parsing, code comments were removed to reduce the number of nodes in each graph, as they are considered non-essential.

4.1.2 HadoopTests Dataset

To overcome the limitations of the OSSBuild dataset, particularly the limited number of files (graphs) per project, we compiled a second dataset for this study. We chose the Apache Hadoop framework [5] due to its extensive number of test files (2,895) and its sufficient complexity. Each unit test in the project was executed five times, with the JUnit framework Samoaa and Leitner (2021) recording the execution duration for each test file at millisecond granularity. The data collection was conducted on a dedicated virtual machine within

[1] https://github.com/apache/systemds
[2] https://github.com/h2database/h2database
[3] https://github.com/apache/dubbo
[4] https://github.com/eclipse/rdf4j
[5] https://github.com/apache/hadoop

Table 1: Overview of the OSSBuilds and HadoopTests datasets.

	Project	Description	Files	Avg. Nodes
OSSBuilds	systemDS	Apache Machine Learning system for data science lifecycle	127	871
	H2	Java SQL DB	194	2091
	Dubbo	Apache Remote Procedure Call framework	123	616
	RDF4J	Scalable RDF processing	478	450
	Total		**922**	**875**
HadoopTests	Hadoop	Apache framework for processing large datasets on clusters	**2895**	**1490**

a private cloud environment equipped with two virtual CPUs and 8 GB of RAM. Following best practices in performance engineering, we disabled all non-essential services during the test runs. Statistics for the HadoopTests dataset are provided in Table 1 (bottom).

4.2 AST Construction

To construct the AST, we parse the Java code to extract the tree object directly. The produced AST does not contain purely syntactical elements, such as comments, brackets, or code location information. We make use of the pure Python Java parser javalang[6] to parse each Java file and use the node types, values, and production rules in javalang to describe our ASTs. The Abstract Syntax Tree comprises various node types, each representing different elements of a program (detailed in Appendix C). Upon completing the AST construction, the result is a tree (an acyclic undirected graph) consisting of 72 unique node types.

4.3 From AST to `RelSC`

The obtained AST is an acyclic undirected graph associated to a Java souce code file. To work with it effectively, we first convert the tree into a directed graph by creating directed edges that point from parent nodes to child nodes. Then we augment the graph by adding 11 different types of edges to capture data flow information, control flow information and marking the graph more connected. Bellow we describe the added edges.

- **Next Token** (b): This type of edge connects leaf nodes in sequence, creating a chain of connections between them.
- **Next Sibling** (c): This connects each node to its siblings.
- **Next Use** d): This type of edge connects a node representing a variable to the node where the variable is next used.
- **If Flow** (e): This type of edge connects the predicate (condition) of the if-statement with the code block that is executed if the condition is true.
- **Else Flow** (f): Conversely, this edge type connects the predicate to the (optional) else code block.
- **While Execution Flow** (g): A while loop essentially entails two elements - a condition and a code block that is executed as long as the condition remains true.

[6]https://pypi.org/project/javalang/

- **While Next Flow** (h): This flow edge mainly connects the node from the body of the while statement to the condition node to simulate the execution process of while loops.
- **For Execution Flow** (j): For loops conceptually consists of two nodes: *ForControl* node and a body node that is executed once the condition is activated. This flow edge connects the condition to the body node.
- **For next Flow**(k): The flow edge is similar to the *While Next Flow* edge.
- **Next Statement Flow** (i): In addition to the control flow constructs discussed so far, Java also supports sequential execution of multiple statements in a sequence within a code block. Next Statement Flow edges (i) are used to represent this case. Different from the constructs discussed so far, a code block can contain an arbitrary number of children, and the Next Statement Flow edge is always used to connect each statement to the one directly following it.

In Figure 4 (left), we present the `RelSC` graph generated from the example in Listing 1. For detailed instructions on constructing a `RelSC` graph, please refer to Samoaa et al. (2022a). The resulting `RelSC` graph is a multigraph, meaning multiple edges can connect the same pair of nodes. To simplify this, we consolidate multi-edges by considering only a single edge between each pair of nodes. The features of this consolidated edge are computed by summing the one-hot encodings of the features from all the original multi-edges.

4.4 From `RelSC` to `Multi-RelSC`

Once `RelSC` graphs have been computed, we also provide a multi-relational version of the dataset, referred to as `Multi-RelSC`. This extension introduces an additional layer of semantic information by categorizing nodes based on their roles and meanings within the Abstract Syntax Tree (AST) (see Section 4.2). The decision to split node types into categories stems from the need to capture the diverse and domain-specific relationships that exist in programming constructs. Specifically, we identify seven categories of nodes: **Declarations**, which refer to the definition or declaration of variables, methods, classes, and similar constructs; **Data Types**, representing specific data types or references to types; **Control Flow**, which includes terms

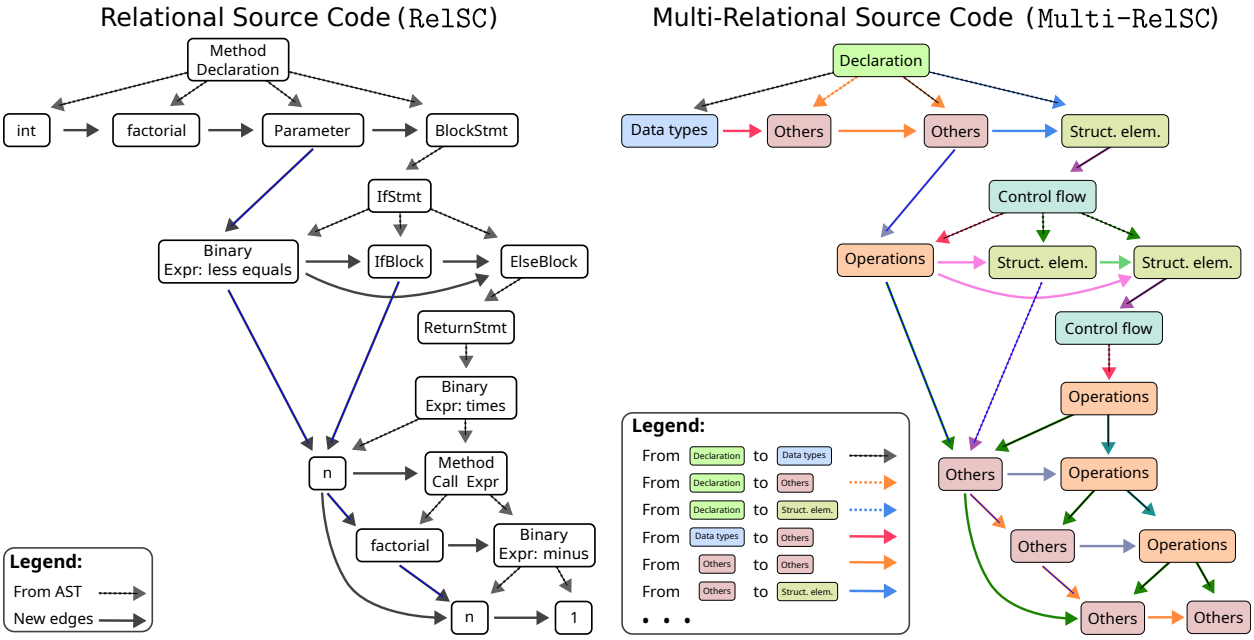

Figure 4: (**Left**) `RelSC` graph for the example presented in Listing 1. (**Right**) `Multi-RelSC` graph for the example presented in Listing 1

associated with constructs that control the program's execution flow; **Operations**, referring to terms that signify operations or expressions; **Structural Elements**, covering structural components of the code such as blocks, compilation units, and packages; **Exceptions and Errors**, relating to exception and error handling mechanisms; and finally, **Others**, for terms that do not fit into any of the previously defined categories. In Appendix C, we provide the categorization of each node type, grouping them into these distinct categories. Additionally, we define a relationship for every possible connection between these categories, resulting in a maximum of 49 possible unique relations (more details in Appendix E). As a result, we generate a multi-relational graph with up to 49 relation types. Each node is represented by a feature vector composed of the one-hot encoding of its node type, concatenated with the sum of its outgoing edge types (see Section 4.3). Figure 4 (right) illustrates the `Multi-RelSC` graph for the example provided in Listing 1.

5 Datasets Statistics

In this section, we provide a detailed analysis of the `RelSC` and `Multi-RelSC` datasets, highlighting their key structural characteristics and diversity. By examining node and edge statistics, as well as node type distributions, we demonstrate the complexity and variability of the datasets. These insights establish the suitability of `RelSC` and `Multi-RelSC` as robust benchmarks for evaluating graph-based models in diverse scenarios and application domains.

`RelSC`: Table 2 summarizes the key characteristics of the homogeneous graphs in our `RelSC` dataset, offering insights into their diversity and complexity. The average node and edge counts vary notably across datasets, with Hadoop having the highest averages, indicating greater complexity, while Dubbo represents a more compact framework, highlighting the dataset's versatility in covering both large-scale and smaller graphs. Variability, as shown by SDT values, is significant in H2 and Hadoop, pointing to diverse structural complexities. For instance, Hadoop ranges from 23 to 32,592 nodes and 80 to 127,822 edges, illustrating the presence of both simple and highly complex graphs. RDF4J and SystemDS also show broad ranges, reflecting the dataset's overall diversity. These statistics demonstrate the `RelSC` dataset's suitability as a strong benchmark for evaluating graph-based models, ensuring that GNNs can be tested across different scenarios. The variety of graphs presents challenges and opportunities for developing more sophisticated algorithms that generalize across multiple domains and software systems.

`Multi-RelSC`: Table 2 summarizes the `Multi-RelSC` dataset, featuring multi-relational graphs. Compared to `RelSC`, `Multi-RelSC` shows increased complexity, with higher average edge counts, such as Hadoop's 11,764.1 edges, reflecting the intricacies of multi-relational connections. H2 in OssBuilds has the highest mean node and edge counts, highlighting its structural complexity. The dataset also exhibits significant variability, with Hadoop ranging from 23 nodes and 176 edges to 32,592 nodes and 259,820 edges, indicating diverse graph structures. These statistics establish `Multi-RelSC` as a robust benchmark for evaluating graph-based models, challenging them to handle multi-relational data. Its diversity makes it a valuable resource for testing GNNs across different scenarios and domains. In summary, `Multi-RelSC` offers a rich collection of graphs, fostering the development of advanced algorithms to address complex software systems.

	Hadoop		**OssBuilds**									
			H2		Dubbo		rdf		SystemDS		**Tot**	
	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$	$\|V\|$	$\|E\|$
mean	1490.3	5731.1	2091.3	8019.6	616.1	2354.2	449.9	1740	871.3	3321	875.5	3361
std	2283.4	8817.9	2631.1	10133.8	998.9	3818.5	726.2	2826.1	629.9	2410.9	1524.7	5869.7
min	23	80	130	500	7	20	22	76	22	78	7	20
max	32592	127822	15947	61758	6374	24540	5918	23146	3396	13208	15947	61758
mean	1490.3	11764.1	2091.3	16517.8	616.1	4811.6	449.9	3573.6	871.3	6804.5	875.5	6907.4
std	2283.4	18052.4	2631.1	20828.4	998.9	7800.6	726.2	5783.4	629.9	4946.3	1524.7	12060.3
min	23	176	130	1020	7	40	22	156	22	156	7	40
max	32592	259820	15947	127032	6374	50672	5918	47284	3396	27740	15947	127032

Table 2: Statistics for `RelSC` datasets (upper) and for `Multi-RelSC` (lower)

5.1 Distribution of Node Types

Figure 5 shows the node category distributions for OssBuilds (left) and Hadoop (right) datasets. Most nodes fall into *"Operation"* and *"Others"*, indicating a high occurrence of expressions, operations, literals, and constants. The standard error (black arrows) is especially large for these categories, particularly in Hadoop, showing high variability across samples. Categories like *"Control Flow"* and *"Data Types"* have lower counts and variability, reflecting the diverse complexity of the graphs. More node distributions are in Appendix D.

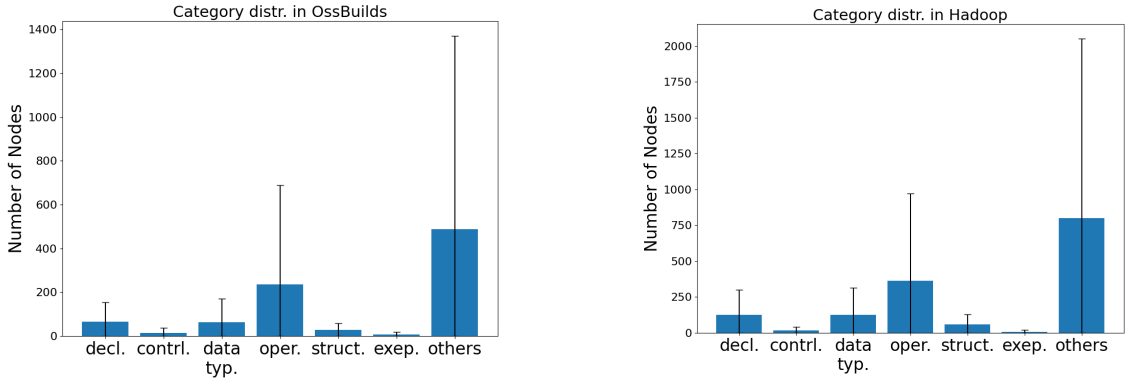

Figure 5: Node Category Distribution for OssBuilds (left), and Hadoop (right)

6 Experiments

In this section, we present the performance of basic GNN and HeteroGNN models on the `RelSC` and `Multi-RelSC` datasets. It is important to note that the main objective of our work is to introduce a novel dataset, not to propose a new architecture.

6.1 Implementation Details and Evaluation

We used GCN Kipf and Welling (2017), ChebConv Defferrard et al. (2017), GIN Xu et al. (2018b), and GraphSAGE Hamilton et al. (2017c) for `RelSC` graphs, and GraphSAGE and GAT Veličković et al. (2017) for `Multi-RelSC` datasets. It is important to note that for the models trained on the `Multi-RelSC` datasets, we rely on heterogeneous message passing[7], where separate parameters are used for each relation type. All models have two convolutional layers (hidden dimension of 30) and two fully connected layers. We applied mean and max global pooling for graph prediction, with batch normalization and dropout for regularization. Models were implemented using PyTorch-Geometric. Each model was trained for 100 epochs with early stopping (patience 15), repeated five times with different seeds, a learning rate of 0.01, and batch size of 32. Experiments were conducted on a machine with four NVIDIA Tesla A100 GPUs (48GB), two Xeon Gold 6338 CPUs, and 256GB DDR4 RAM.

The proposed datasets for the graph regression task exhibit a notable imbalance in target values. For example, in the Hadoop dataset, approximately 50% of the target values fall within the range of $[0, 0.22]$, indicating a significant concentration of samples in this lower range. This imbalance in the targets makes evaluation more challenging. Therefore, we report the Mean Absolute Error (MAE) as our primary metric. However, since MAE does not account for relative errors, we include additional metrics in Appendix B, specifically Root Mean Squared Error (RMSE), Mean Absolute Percentage Error (MAPE), Spearman's rank correlation coefficient, and the Maximum Relative Error (MRE).

[7] `https://pytorch-geometric.readthedocs.io/en/latest/generated/torch_geometric.nn.conv.HeteroConv.html`

6.2 Results

`RelSC`: Table 3 presents the performance of GNN-based models on the `RelSC` datasets, evaluated using MAE with standard deviation across five initialization seeds. For the Hadoop dataset, ChebConv achieved the lowest test MAE of **0.11**, slightly outperforming GConv and GIN, which had MAEs of 0.12, and GraphSAGE with 0.13. In OssBuilds, all models performed similarly, except GraphSAGE for Dubbo, which achieved the lowest test MAE of 0.12. When combined with OssBuilds, ChebConv had the highest MAE (0.15), while others reported 0.14. Overall, the models performed comparably across multiple datasets, with minor variations, suggesting that the choice of GNN architecture has a modest impact on performance for `RelSC` datasets.

`Multi-RelSC`: Table 3 shows that HeteroGAT consistently outperforms HeteroSAGE across all datasets, achieving the lowest MAE values in both validation and test sets. This is due to HeteroGAT's ability to capture complex multi-relational connections in the `Multi-RelSC` datasets, providing a richer context for predictions. MAE variation across datasets is notable. Hadoop, with its larger node and edge counts, has lower MAE compared to smaller datasets like SystemDS and H2, where MAE values are higher, especially for HeteroSAGE. Hadoop's complexity helps HeteroGAT generalize better, while simpler datasets offer fewer relational patterns, leading to higher MAEs. Datasets with higher variability, like SystemDS and H2, also show greater MAE fluctuations, reflecting the challenge of adapting to diverse graph structures. In summary, HeteroGAT performs best overall, but differences in graph size, complexity, and variability affect MAE. The multi-relational nature of the `Multi-RelSC` datasets allows HeteroGAT to leverage these complexities for better prediction accuracy.

Table 3: Test and validation MAE (lower the better) for `RelSC` and `Multi-RelSC` datasets

			`RelSC`				`Multi-RelSC`	
			GConv	Cheb	GIN	Sage	HeteroSage	HeteroGAT
	Hadoop	val	0.11(±0.00)	0.12(±0.00)	0.11(±0.00)	0.11(±0.00)	0.17(±0.04)	0.12(±0.00)
		test	0.12(±0.00)	0.11(±0.00)	0.12(±0.01)	0.13(±0.00)	0.27(±0.11)	0.14(±0.02)
OssBuilds	RDF4J	val	0.13(±0.01)	0.13(±0.01)	0.12(±0.00)	0.13(±0.01)	0.16(±0.01)	0.15(±0.00)
		test	0.13(±0.00)	0.12(±0.01)	0.12(±0.00)	0.13(±0.01)	0.20(±0.05)	0.15(±0.01)
	SystemDS	val	0.06(±0.02)	0.09(±0.03)	0.07(±0.04)	0.07(±0.03)	0.73(±0.41)	0.14(±0.01)
		test	0.07(±0.02)	0.08(±0.04)	0.08(±0.05)	0.07(±0.03)	5.82(±5.45)	0.31(±0.11)
	H2	val	0.13(±0.00)	0.15(±0.00)	0.14(±0.01)	0.15(±0.00)	0.89(±0.58)	0.24(±0.07)
		test	0.18(±0.01)	0.18(±0.01)	0.20(±0.01)	0.19(±0.01)	4.35(±3.51)	0.69(±0.54)
	Dubbo	val	0.09(±0.01)	0.09(±0.01)	0.08(±0.01)	0.08(±0.00)	0.38(±0.34)	0.08(±0.03)
		test	0.14(±0.02)	0.13(±0.00)	0.14(±0.01)	0.12(±0.01)	3.65(±5.60)	0.19(±0.09)
	OssBuilds	val	0.14(±0.00)	0.14(±0.00)	0.14(±0.00)	0.14(±0.00)	0.47(±0.24)	0.19(±0.05)
		test	0.14(±0.01)	0.15(±0.01)	0.14(±0.01)	0.14(±0.01)	0.58(±0.31)	0.18(±0.02)

The results emphasize the challenges posed by the proposed datasets. For `RelSC` datasets, traditional GNNs such as GraphConv, ChebConv, and GINConv showed similar performance, with minor variations; ChebConv achieved the best test MAE of 0.11 on Hadoop. In contrast, for `Multi-RelSC` datasets, HeteroGAT outperformed HeteroSAGE but struggled on smaller datasets like SystemDS and H2, where test MAEs reached 0.31 and 0.69, respectively. This highlights the limitations of current multi-relational models in capturing the complexity of diverse graph structures. These observations underscore the significance of the proposed datasets as a challenging benchmark for evaluating GNN models. *While this work does not propose an ad hoc model, the datasets provide a robust testbed for developing and validating more advanced multi-relational architectures that can better exploit the nuanced relationships and complexities present in real-world graphs.*

6.3 Ablation Study

Abstract Syntax Trees represent source code syntax but lack semantic details like control and data flow. To address this, we augment ASTs with edges from Control Flow Graphs (CFGs) and Data Flow Graphs

(DFGs), creating Flow-Augmented ASTs (FA-ASTs). An ablation study on the OssBuilds dataset (Table 4) shows that adding these edges significantly improves performance compared to plain ASTs (Table 3).

Table 4: Test MAE (Mean ± Std) on OssBuilds using ASTs without Flow Augmentation

Model	**Test MAE**
GraphConv	$0.22_{(\pm 0.02)}$
ChebConv	$0.23_{(\pm 0.01)}$
GINConv	$0.21_{(\pm 0.01)}$
GraphSAGE	$0.22_{(\pm 0.01)}$

The inclusion of flow edges significantly enhances the performance of all models, reducing the test MAE by approximately 0.07 to 0.09. For instance, the MAE for GraphConv improved from $0.22_{(\pm 0.02)}$ to $0.14_{(\pm 0.01)}$, ChebConv from $0.23_{(\pm 0.01)}$ to $0.15_{(\pm 0.01)}$, GINConv from $0.21_{(\pm 0.01)}$ to $0.14_{(\pm 0.01)}$, and GraphSAGE from $0.22_{(\pm 0.01)}$ to $0.14_{(\pm 0.01)}$. These results underscore the critical role of semantic augmentation, as the incorporation of control and data flow information enables GNN models to learn richer representations that better capture execution pathways and dependencies within the code, ultimately leading to significant improvements in prediction accuracy. This demonstrates the importance of flow augmentation for constructing informative graph representations in software performance prediction tasks.

7 Real-World Applications

Accurately predicting source code execution time is essential for optimizing software performance, improving development workflows, and enhancing user experience. The proposed datasets, `RelSC` and `Multi-RelSC`, can be leveraged in several impactful ways:

- *Code Optimization and Refactoring:* Execution time predictions help developers identify performance bottlenecks early, allowing targeted optimizations to improve efficiency in large-scale software systems.
- *Continuous Integration and Deployment (CI/CD):* By integrating GNN models trained on these datasets into CI/CD pipelines, teams can detect performance regressions during development, ensuring new code changes maintain or improve performance before reaching production.
- *Performance-Aware Scheduling:* In cloud environments, predicting execution times aids in resource allocation, enabling efficient scheduling that meets performance requirements while reducing operational costs.

These applications demonstrate the value of our datasets in driving performance-focused decision-making in software engineering, with potential for future integration into automated performance tuning, debugging, and energy-efficient coding tools.

8 Data Release

To support further research, we release the raw data and PyTorch Geometric graph objects on Zenodo and the code on GitHub[8]. The repository includes model implementations, instructions for constructing graphs, a tutorial for loading the dataset and training models, and dataset statistics. The code is well-documented for ease of use by researchers and practitioners.

9 Conclusion

In this work, we have addressed the critical gap in publicly available benchmarks for graph regression tasks by introducing two novel datasets specifically tailored to software performance prediction. Our proposed

[8] `https://anonymous.4open.science/r/graph_regression_datasets-407E/`

datasets, `RelSC` and `Multi-RelSC`, represent Java source code and their corresponding execution times, providing valuable resources for the exploration of GNN models in a new domain—software engineering. These contributions extend the scope of GNN applications beyond the traditionally explored domains of Chemistry and Drug Discovery, enabling researchers to investigate graph regression in software performance and related fields. With our datasets being publicly accessible, we aim to foster further research, providing a standardized benchmark that can drive the development, evaluation, and comparison of GNN models in software engineering and other underexplored areas.

References

Franco Scarselli, Marco Gori, Ah Chung Tsoi, Markus Hagenbuchner, and Gabriele Monfardini. The graph neural network model. *IEEE transactions on neural networks*, 20(1):61–80, 2008.

Alessio Micheli. Neural network for graphs: A contextual constructive approach. *IEEE Transactions on Neural Networks*, 20(3):498–511, 2009.

Thomas N Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907*, 2016.

W. Hamilton, Z. Ying, and J. Leskovec. Inductive representation learning on large graphs. *NeurIPS*, 30, 2017a.

P. Veličković, G. Cucurull, A. Casanova, A. Romero, P. Liò, and Y. Bengio. Graph attention networks. *arXiv preprint arXiv:1710.10903*, 2017.

Johannes Gasteiger, Aleksandar Bojchevski, and Stephan Günnemann. Predict then propagate: Graph neural networks meet personalized pagerank. *arXiv preprint arXiv:1810.05997*, 2018.

Muhan Zhang and Yixin Chen. Link prediction based on graph neural networks. *Advances in neural information processing systems*, 31, 2018.

Felix Wu, Amauri Souza, Tianyi Zhang, Christopher Fifty, Tao Yu, and Kilian Weinberger. Simplifying graph convolutional networks. In *International conference on machine learning*, pages 6861–6871. PMLR, 2019.

Muhan Zhang, Pan Li, Yinglong Xia, Kai Wang, and Long Jin. Labeling trick: A theory of using graph neural networks for multi-node representation learning. *Advances in Neural Information Processing Systems*, 34: 9061–9073, 2021a.

Veronica Lachi, Francesco Ferrini, Antonio Longa, Bruno Lepri, and Andrea Passerini. A simple and expressive graph neural network based method for structural link representation. In *ICML 2024 Workshop on Geometry-grounded Representation Learning and Generative Modeling*, 2024. URL `https://openreview.net/forum?id=EGGSCLyVrz`.

Olga Zaghen, Antonio Longa, Steve Azzolin, Lev Telyatnikov, Andrea Passerini, and Pietro Lio. Sheaf diffusion goes nonlinear: Enhancing GNNs with adaptive sheaf laplacians. In *ICML 2024 Workshop on Geometry-grounded Representation Learning and Generative Modeling*, 2024. URL `https://openreview.net/forum?id=MGQtGV5gPO`.

Timothy G Armstrong, Vamsi Ponnekanti, Dhruba Borthakur, and Mark Callaghan. Linkbench: a database benchmark based on the facebook social graph. In *Proceedings of the 2013 ACM SIGMOD International Conference on Management of Data*, pages 1185–1196, 2013.

Weihua Hu, Matthias Fey, Marinka Zitnik, Yuxiao Dong, Hongyu Ren, Bowen Liu, Michele Catasta, and Jure Leskovec. Open graph benchmark: Datasets for machine learning on graphs. *Advances in neural information processing systems*, 33:22118–22133, 2020a.

Christopher Morris, Nils M Kriege, Franka Bause, Kristian Kersting, Petra Mutzel, and Marion Neumann. Tudataset: A collection of benchmark datasets for learning with graphs. *arXiv preprint arXiv:2007.08663*, 2020.

Vijay Prakash Dwivedi, Ladislav Rampášek, Michael Galkin, Ali Parviz, Guy Wolf, Anh Tuan Luu, and Dominique Beaini. Long range graph benchmark. *Advances in Neural Information Processing Systems*, 35:22326–22340, 2022.

Zhou Zhiyao, Sheng Zhou, Bochao Mao, Xuanyi Zhou, Jiawei Chen, Qiaoyu Tan, Daochen Zha, Yan Feng, Chun Chen, and Can Wang. Opengsl: A comprehensive benchmark for graph structure learning. *Advances in Neural Information Processing Systems*, 36, 2024.

Shenyang Huang, Farimah Poursafaei, Jacob Danovitch, Matthias Fey, Weihua Hu, Emanuele Rossi, Jure Leskovec, Michael Bronstein, Guillaume Rabusseau, and Reihaneh Rabbany. Temporal graph benchmark for machine learning on temporal graphs. *Advances in Neural Information Processing Systems*, 36, 2024.

Xiao-Meng Zhang, Li Liang, Lin Liu, and Ming-Jing Tang. Graph neural networks and their current applications in bioinformatics. *Frontiers in genetics*, 12:690049, 2021b.

Pietro Bongini, Niccolò Pancino, Franco Scarselli, and Monica Bianchini. Biognn: how graph neural networks can solve biological problems. In *Artificial Intelligence and Machine Learning for Healthcare: Vol. 1: Image and Data Analytics*, pages 211–231. Springer, 2022.

Weiwei Jiang and Jiayun Luo. Graph neural network for traffic forecasting: A survey. *Expert Systems with Applications*, 207:117921, 2022.

Xiao Li, Li Sun, Mengjie Ling, and Yan Peng. A survey of graph neural network based recommendation in social networks. *Neurocomputing*, 549:126441, 2023.

Wenqi Fan, Yao Ma, Qing Li, Yuan He, Eric Zhao, Jiliang Tang, and Dawei Yin. Graph neural networks for social recommendation. In *The world wide web conference*, pages 417–426, 2019.

Dejun Jiang, Zhenxing Wu, Chang-Yu Hsieh, Guangyong Chen, Ben Liao, Zhe Wang, Chao Shen, Dongsheng Cao, Jian Wu, and Tingjun Hou. Could graph neural networks learn better molecular representation for drug discovery? a comparison study of descriptor-based and graph-based models. *Journal of Cheminformatics*, 13(1):12, Feb 2021. ISSN 1758-2946. doi: 10.1186/s13321-020-00479-8.

Oliver Wieder, Stefan Kohlbacher, Mélaine Kuenemann, Arthur Garon, Pierre Ducrot, Thomas Seidel, and Thierry Langer. A compact review of molecular property prediction with graph neural networks. *Drug Discovery Today: Technologies*, 37:1–12, 2020.

Zehong Zhang, Lifan Chen, Feisheng Zhong, Dingyan Wang, Jiaxin Jiang, Sulin Zhang, Hualiang Jiang, Mingyue Zheng, and Xutong Li. Graph neural network approaches for drug-target interactions. *Current Opinion in Structural Biology*, 73:102327, 2022.

Peter Samoaa, Antonio Longa, Mazen Mohamad, Morteza Haghir Chehreghani, and Philipp Leitner. Tepgnn: Accurate execution time prediction of functional tests using graph neural networks. In Davide Taibi, Marco Kuhrmann, Tommi Mikkonen, Jil Klünder, and Pekka Abrahamsson, editors, *Product-Focused Software Process Improvement*, pages 464–479, Cham, 2022a. Springer International Publishing. ISBN 978-3-031-21388-5.

Zhenqin Wu, Bharath Ramsundar, Evan N Feinberg, Joseph Gomes, Caleb Geniesse, Aneesh S Pappu, Karl Leswing, and Vijay Pande. Moleculenet: a benchmark for molecular machine learning. *Chemical science*, 9(2):513–530, 2018.

Rafael Gómez-Bombarelli, Jennifer N Wei, David Duvenaud, José Miguel Hernández-Lobato, Benjamín Sánchez-Lengeling, Dennis Sheberla, Jorge Aguilera-Iparraguirre, Timothy D Hirzel, Ryan P Adams, and Alán Aspuru-Guzik. Automatic chemical design using a data-driven continuous representation of molecules. *ACS central science*, 4(2):268–276, 2018.

Yuquan Li, Chang-Yu Hsieh, Ruiqiang Lu, Xiaoqing Gong, Xiaorui Wang, Pengyong Li, Shuo Liu, Yanan Tian, Dejun Jiang, Jiaxian Yan, et al. An adaptive graph learning method for automated molecular interactions and properties predictions. *nature machine intelligence*, 4(7):645–651, 2022.

David L Mobley and J Peter Guthrie. Freesolv: a database of experimental and calculated hydration free energies, with input files. *Journal of computer-aided molecular design*, 28:711–720, 2014.

Zhihai Liu, Yan Li, Li Han, Jie Li, Jie Liu, Zhixiong Zhao, Wei Nie, Yuchen Liu, and Renxiao Wang. Pdb-wide collection of binding data: current status of the pdbbind database. *Bioinformatics*, 31(3):405–412, 2015.

Gang Liu, Tong Zhao, Jiaxin Xu, Tengfei Luo, and Meng Jiang. Graph rationalization with environment-based augmentations. In *Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining*, pages 1069–1078, 2022.

Peter Samoaa, Linus Aronsson, Antonio Longa, Philipp Leitner, and Morteza Haghir Chehreghani. A unified active learning framework for annotating graph data with application to software source code performance prediction, 2023a.

Peter Samoaa, Linus Aronsson, Philipp Leitner, and Morteza Haghir Chehreghani. Batch mode deep active learning for regression on graph data. In *2023 IEEE International Conference on Big Data (BigData)*, pages 5904–5913, 2023b. doi: 10.1109/BigData59044.2023.10386685.

Peter Samoaa, Firas Bayram, Pasquale Salza, and Philipp Leitner. A systematic mapping study of source code representation for deep learning in software engineering. *IET Software*, 16(4):351–385, 2022b. doi: https://doi.org/10.1049/sfw2.12064. URL `https://ietresearch.onlinelibrary.wiley.com/doi/abs/10.1049/sfw2.12064`.

Justin Gilmer, Samuel S Schoenholz, Patrick F Riley, Oriol Vinyals, and George E Dahl. Neural message passing for quantum chemistry. In *International conference on machine learning*, pages 1263–1272. PMLR, 2017.

Joan Bruna, Wojciech Zaremba, Arthur Szlam, and Yann LeCun. Spectral networks and locally connected networks on graphs. *arXiv preprint arXiv:1312.6203*, 2013.

Hao Yuan and Shuiwang Ji. Structpool: Structured graph pooling via conditional random fields. In *Proceedings of the 8th International Conference on Learning Representations*, 2020.

Amir Hosein Khasahmadi, Kaveh Hassani, Parsa Moradi, Leo Lee, and Quaid Morris. Memory-based graph networks. In *International Conference on Learning Representations*, 2020.

Michael Schlichtkrull, Thomas N. Kipf, Peter Bloem, Rianne van den Berg, Ivan Titov, and Max Welling. Modeling relational data with graph convolutional networks, 2017.

Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Liò, and Yoshua Bengio. Graph Attention Networks. In *ICLR*, 2018.

William L. Hamilton, Zhitao Ying, and Jure Leskovec. Inductive Representation Learning on Large Graphs. In *NIPS*, pages 1024–1034, 2017b.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? *arXiv preprint arXiv:1810.00826*, 2018a.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering. *Advances in neural information processing systems*, 29, 2016.

Shichao Zhu, Chuan Zhou, Shirui Pan, Xingquan Zhu, and Bin Wang. Relation structure-aware heterogeneous graph neural network. In *2019 IEEE International Conference on Data Mining (ICDM)*, pages 1534–1539, 2019. doi: 10.1109/ICDM.2019.00203.

Seongjun Yun, Minbyul Jeong, Raehyun Kim, Jaewoo Kang, and Hyunwoo J Kim. Graph transformer networks. In H. Wallach, H. Larochelle, A. Beygelzimer, F. d'Alché-Buc, E. Fox, and R. Garnett, editors, *Advances in Neural Information Processing Systems*, volume 32. Curran Associates, Inc., 2019. URL `https://proceedings.neurips.cc/paper_files/paper/2019/file/9d63484abb477c97640154d40595a3bb-Paper.pdf`.

Ziniu Hu, Yuxiao Dong, Kuansan Wang, and Yizhou Sun. Heterogeneous graph transformer. In *Proceedings of The Web Conference 2020*, WWW '20, page 2704–2710, New York, NY, USA, 2020b. Association for Computing Machinery. ISBN 9781450370233. doi: 10.1145/3366423.3380027. URL https://doi.org/10.1145/3366423.3380027.

Qingsong Lv, Ming Ding, Qiang Liu, Yuxiang Chen, Wenzheng Feng, Siming He, Chang Zhou, Jianguo Jiang, Yuxiao Dong, and Jie Tang. Are we really making much progress? revisiting, benchmarking and refining heterogeneous graph neural networks. In *Proceedings of the 27th ACM SIGKDD Conference on Knowledge Discovery & Data Mining*, KDD '21, page 1150–1160, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450383325. doi: 10.1145/3447548.3467350. URL https://doi.org/10.1145/3447548.3467350.

Le Yu, Leilei Sun, Bowen Du, Chuanren Liu, Weifeng Lv, and Hui Xiong. Heterogeneous graph representation learning with relation awareness. *CoRR*, abs/2105.11122, 2021. URL https://arxiv.org/abs/2105.11122.

Anasua Mitra, Priyesh Vijayan, Sanasam Ranbir Singh, Diganta Goswami, Srinivas Parthasarathy, and Balaraman Ravindran. Revisiting link prediction on heterogeneous graphs with a multi-view perspective. *2022 IEEE International Conference on Data Mining (ICDM)*, pages 358–367, 2022. URL https://api.semanticscholar.org/CorpusID:256463320.

Francesco Ferrini, Antonio Longa, Andrea Passerini, and Manfred Jaeger. Meta-path learning for multi-relational graph neural networks. In *Learning on Graphs Conference*, pages 2–1. PMLR, 2024a.

Francesco Ferrini, Antonio Longa, Andrea Passerini, and Manfred Jaeger. A self-explainable heterogeneous gnn for relational deep learning. *arXiv preprint arXiv:2412.00521*, 2024b.

Peter Samoaa and Philipp Leitner. An exploratory study of the impact of parameterization on jmh measurement results in open-source projects. In *Proceedings of the ACM/SPEC International Conference on Performance Engineering*, ICPE '21, page 213–224, New York, NY, USA, 2021. Association for Computing Machinery. ISBN 9781450381949. doi: 10.1145/3427921.3450243. URL https://doi.org/10.1145/3427921.3450243.

Thomas N. Kipf and Max Welling. Semi-supervised classification with graph convolutional networks. In *International Conference on Learning Representations*, 2017. URL https://openreview.net/forum?id=SJU4ayYgl.

Michaël Defferrard, Xavier Bresson, and Pierre Vandergheynst. Convolutional neural networks on graphs with fast localized spectral filtering, 2017. URL https://arxiv.org/abs/1606.09375.

Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. How powerful are graph neural networks? In *International Conference on Learning Representations*, 2018b.

Will Hamilton, Zhitao Ying, and Jure Leskovec. Inductive representation learning on large graphs. *Advances in neural information processing systems*, 30, 2017c.

A Licensing and Ethical Statement

Licensing: To construct our dataset, we rely on source code available on GitHub, distributed under the following licenses:

- Hadoop: Apache License, Version 2.0
- H2: MPL 2.0 (Mozilla Public License, Version 2.0) or EPL 1.0 (Eclipse Public License)
- Dubbo: Apache License, Version 2.0
- rdf: BSD-3-Clause License
- SystemDS: Apache License, Version 2.0

We executed the source code and recorded the execution times, as described in Sections 4.1.1 and 4.1.2. The resulting graphs, along with their execution times, are being released under the CC-BY license.

Ethical Statement: This dataset is designed to address challenges in graph representation learning, with a particular emphasis on graph regression tasks. While it is not intended for this purpose, there is a possibility that it could be used to enhance models for harmful applications. However, to the best of our knowledge, our work does not directly pose any threat to individuals or society.

B Additional Metrics

In this section we evaluate standard GNN techniques on the proposed datasets. In particular, table 5 reports the Root Mean Squared Error (RMSE), table 6 reports the Mean Absolute Percentage Error (MAPE), table 7 shows the Spearman's Rank Correlation Coefficient (ρ), and finally table 8 shows the Maximum Relative Error (MRE).

The MAPE is defined as

$$MAPE = \frac{1}{n} \sum_{i=1}^{n} \frac{y_i - \bar{y}_i}{y_i} \tag{4}$$

where n is the number of observations, y_i is the actual value, and $\bar{y}_i$ is the predicted value.

While the Spearman's Rank Correlation Coefficient is a non-parametric measure of rank correlation and it is defined as:

$$\rho = 1 - \frac{6 \sum d_i^2}{n(n^2 - 1)} \tag{5}$$

where n is the number of observations, d_i is the difference between the ranks of each pair of observations. Note that ρ ranges from -1 to 1, where $\rho = 1$ indicates perfect positive correlation, $\rho = -1$ indicates perfect negative correlation, and $\rho = 0$ indicates no correlation.

C Node Types

In table 9, we report the definition of each node type with their associated category.

Table 5: Test and validation RMSE for `RelSC` and `Multi-RelSC` datasets

			`RelSC`				`Multi-RelSC`	
			GConv	Cheb	GIN	Sage	HeteroSage	HeteroGAT
	Hadoop	val	0.17(±0.00)	0.17(±0.00)	0.16(±0.00)	0.17(±0.00)	0.35(±0.21)	0.18(±0.00)
		test	0.16(±0.00)	0.15(±0.00)	0.16(±0.01)	0.17(±0.01)	0.68(±0.54)	0.21(±0.04)
OssBuilds	RDF4J	val	0.17(±0.00)	0.17(±0.00)	0.17(±0.00)	0.18(±0.00)	0.19(±0.01)	0.19(±0.00)
		test	0.15(±0.01)	0.15(±0.01)	0.15(±0.00)	0.16(±0.01)	0.27(±0.11)	0.18(±0.02)
	SystemDS	val	0.10(±0.02)	0.11(±0.03)	0.11(±0.03)	0.10(±0.02)	0.93(±0.51)	0.18(±0.02)
		test	0.08(±0.02)	0.09(±0.05)	0.09(±0.05)	0.09(±0.02)	8.71(±8.88)	0.43(±0.17)
	H2	val	0.17(±0.01)	0.19(±0.01)	0.18(±0.01)	0.19(±0.01)	1.43(±0.99)	0.32(±0.12)
		test	0.21(±0.00)	0.21(±0.01)	0.23(±0.01)	0.22(±0.01)	6.08(±4.53)	0.97(±0.73)
	Dubbo	val	0.11(±0.01)	0.13(±0.01)	0.11(±0.02)	0.13(±0.00)	0.55(±0.49)	0.10(±0.03)
		test	0.17(±0.01)	0.17(±0.01)	0.17(±0.01)	0.17(±0.01)	7.82(±12.22)	0.32(±0.22)
	OssBuilds	val	0.17(±0.00)	0.17(±0.01)	0.17(±0.01)	0.17(±0.00)	0.98(±0.68)	0.27(±0.12)
		test	0.18(±0.01)	0.19(±0.01)	0.18(±0.01)	0.18(±0.01)	1.89(±1.99)	0.24(±0.04)

Table 6: Test and validation MAPE for `RelSC` and `Multi-RelSC` datasets. We report "-" to indicate that the value diverged.

			`RelSC`				`Multi-RelSC`	
			GConv	Cheb	GIN	Sage	HeteroSage	HeteroGAT
	Hadoop	val	1.26(±0.12)	1.44(±0.17)	1.19(±0.08)	1.32(±0.06)	1.85(±0.53)	1.40(±0.15)
		test	0.54(±0.02)	0.58(±0.08)	0.51(±0.01)	0.59(±0.02)	1.11(±0.25)	0.67(±0.09)
OssBuilds	RDF4J	val	0.63(±0.07)	0.61(±0.04)	0.55(±0.02)	0.67(±0.06)	0.84(±0.05)	0.79(±0.05)
		test	0.78(±0.08)	0.68(±0.04)	0.64(±0.04)	0.81(±0.08)	1.18(±0.24)	0.94(±0.03)
	SystemDS	val	0.10(±0.03)	0.13(±0.04)	0.11(±0.04)	0.11(±0.03)	1.09(±0.60)	0.21(±0.01)
		test	0.09(±0.02)	0.10(±0.05)	0.11(±0.06)	0.10(±0.03)	7.71(±7.24)	0.41(±0.14)
	H2	val	-	-	-	-	-	-
		test	0.55(±0.07)	0.60(±0.07)	0.60(±0.07)	0.65(±0.03)	10.59(±8.28)	1.73(±1.12)
	Dubbo	val	0.54(±0.09)	0.55(±0.09)	0.51(±0.07)	0.45(±0.01)	1.89(±1.72)	0.55(±0.11)
		test	0.73(±0.21)	0.64(±0.11)	0.70(±0.10)	0.55(±0.03)	12.59(±18.93)	0.73(±0.23)
	OssBuilds	val	-	-	-	-	-	-
		test	0.68(±0.05)	0.84(±0.06)	0.80(±0.08)	0.67(±0.05)	2.03(±0.90)	0.93(±0.06)

D Node Category of the Datasets

In this section, we report the average number of nodes in each category for the remaining datasets: H2, Dubbo, RDF4J, and SystemDS, as shown in Figures 6 to 9. We previously discussed the node distributions for Hadoop and OssBuilds in Section 5.1.

Across these datasets, there is a noticeable consistency in the dominance of the *"Others"* and *"Operation"* categories, which account for a significant portion of the nodes in each dataset. This trend is indicative of the complex and diverse operations and structural elements within these software systems.

While *"Others"* and *"Operation"* categories consistently lead, the distribution among other categories, such as *"DataTypes"* and *"StructuralElements"*, varies between datasets. For instance, SystemDS and RDF4J show a relatively balanced distribution across these additional categories, whereas H2 and Dubbo exhibit higher variability, as reflected by their broader STD bars. This variability suggests that the graphs within each dataset have distinct structural characteristics, further emphasizing the challenges in graph-based model learning.

Overall, these figures highlight the variability and complexity inherent in each dataset, reinforcing the need for flexible and robust models capable of handling diverse graph structures.

Table 7: Test and validation Spearman's Rank Correlation Coefficient (ρ) (higher the better) for `RelSC` and `Multi-RelSC` datasets

			`RelSC`				`Multi-RelSC`	
			GConv	Cheb	GIN	Sage	HeteroSage	HeteroGAT
	Hadoop	val	0.59(±0.03)	0.58(±0.03)	0.61(±0.02)	0.50(±0.02)	0.31(±0.09)	0.50(±0.05)
		test	0.61(±0.03)	0.64(±0.04)	0.64(±0.03)	0.57(±0.02)	0.21(±0.21)	0.50(±0.11)
OssBuilds	RDF4J	val	0.54(±0.02)	0.52(±0.06)	0.54(±0.02)	0.46(±0.04)	0.26(±0.05)	0.33(±0.05)
		test	0.52(±0.03)	0.50(±0.05)	0.53(±0.02)	0.38(±0.05)	0.20(±0.07)	0.32(±0.07)
	SystemDS	val	0.60(±0.14)	0.51(±0.17)	0.55(±0.18)	0.66(±0.06)	−0.10(±0.38)	0.14(±0.23)
		test	0.67(±0.04)	0.74(±0.17)	0.67(±0.08)	0.77(±0.06)	−0.34(±0.08)	0.24(±0.31)
	H2	val	0.52(±0.06)	nan	0.30(±0.34)	nan	0.21(±0.12)	0.30(±0.08)
		test	0.28(±0.09)	nan	0.23(±0.09)	nan	0.02(±0.31)	0.22(±0.27)
	Dubbo	val	0.29(±0.03)	0.28(±0.03)	0.18(±0.05)	0.26(±0.04)	0.07(±0.09)	0.26(±0.11)
		test	0.32(±0.32)	0.49(±0.04)	0.23(±0.35)	0.41(±0.08)	0.13(±0.47)	0.41(±0.17)
	OssBuilds	val	0.50(±0.03)	0.48(±0.01)	0.49(±0.02)	0.48(±0.03)	0.19(±0.07)	0.42(±0.05)
		test	0.59(±0.03)	0.52(±0.03)	0.55(±0.05)	0.56(±0.04)	0.24(±0.18)	0.40(±0.04)

Table 8: Test and validation MRE (lower the better) for `RelSC` and `Multi-RelSC` datasets

			`RelSC`				`Multi-RelSC`	
			GConv	Cheb	GIN	Sage	HeteroSage	HeteroGAT
	Hadoop	val	107.50(±32.46)	118.48(±11.76)	79.51(±22.64)	87.24(±13.31)	74.11(±25.84)	64.66(±14.30)
		test	19.12(±1.88)	22.10(±2.69)	17.89(±2.47)	14.64(±2.71)	23.17(±8.15)	13.11(±1.50)
OssBuilds	RDF4J	val	5.01(±0.30)	5.29(±0.16)	5.50(±0.25)	5.68(±0.21)	4.08(±1.21)	3.47(±0.24)
		test	3.51(±0.42)	3.60(±0.22)	2.91(±0.19)	4.08(±0.95)	5.51(±1.74)	3.47(±0.26)
	SystemDS	val	0.77(±0.03)	0.58(±0.15)	0.67(±0.15)	0.61(±0.15)	3.30(±1.86)	0.64(±0.15)
		test	0.25(±0.03)	0.23(±0.06)	0.25(±0.06)	0.28(±0.05)	33.73(±37.24)	1.47(±0.71)
	H2	val	3.12(±0.51)	3.78(±0.30)	3.41(±0.61)	4.04(±0.10)	13.64(±10.67)	2.77(±0.70)
		test	2.75(±0.72)	3.66(±0.88)	3.17(±1.00)	4.26(±0.26)	42.95(±26.19)	7.55(±3.93)
	Dubbo	val	2.26(±0.57)	2.46(±0.46)	2.11(±0.40)	1.94(±0.11)	5.42(±3.74)	1.82(±0.18)
		test	2.07(±0.96)	1.71(±0.42)	2.19(±0.50)	1.47(±0.14)	73.19(±112.62)	2.56(±1.94)
	OssBuilds	val	8.84(±0.57)	9.02(±1.79)	9.99(±1.53)	6.66(±0.89)	28.70(±11.96)	13.87(±2.50)
		test	4.73(±0.32)	5.26(±0.60)	5.86(±0.34)	4.60(±0.42)	30.15(±27.64)	4.89(±0.39)

E Relations on the Datasets

In these figures, we report the average number of relations between different node types for each dataset. Each figure presents a heatmap where the rows and columns correspond to various categories of nodes (defined in Section 4.4), such as "Declarations," "Control Flow," "Data Types", "Operations", and "Others".

A common pattern across all datasets is the significant number of relations involving the *"Operation"* and *"Others"* categories. These categories consistently show higher interaction counts, indicating their central role in the overall structure of the software systems. Notably, the *"Others"* category frequently interacts with *"Operation"* nodes, underscoring the complexity and interdependence of various node types within the graphs.

The *"Declarations"* and *"Data Types"* categories also show considerable relations, particularly in datasets like H2 and SystemDS (Figures 11 and 15), where they interact heavily with *"Operation"* nodes. This suggests that these systems have a more intricate structure with a higher degree of dependencies between different code elements.

Differences across datasets are most evident in the intensity of specific relations. For example, H2 and Hadoop (Figures 11 and 12) exhibit a higher number of relations between *"Operation"* and *"Others"* compared to Dubbo and RDF4J (Figures 10 and 14), indicating that the former systems have more complex and interconnected codebases.

Node type	Description	Category
AnnotationMethod	Defines a method used in annotations, often to specify default values for elements	declarations
InferredFormalParameter	A formal parameter whose type is inferred by the compiler, often in lambda expressions	declarations
LocalVariableDeclaration	Declares a variable within a method, constructor, or block, with local scope	declarations
SuperConstructorInvocation	Calls the constructor of the superclass from a subclass constructor	expressions_and_operations
Import	Imports classes or entire packages to make them available for use in a Java file	code_structure
ArraySelector	Used to select an element from an array using its index	types_and_references
BreakStatement	Terminates the nearest enclosing loop or switch statement	control_flow
FieldDeclaration	Declares a variable at the class level, which can be accessed by methods of the class	declarations
EnumDeclaration	Declares an enumeration, a special Java type used to define collections of constants	declarations
ConstructorDeclaration	Declares a constructor, a special method to create and initialize objects of a class	declarations
Annotation	A form of metadata that provides data about a program	code_structure
ReferenceType	Specifies a type that refers to objects, such as classes, arrays, or interfaces	types_and_references
EnhancedForControl	Control structure used to iterate over collections or arrays in a simplified way	control_flow
TypeParameter	Represents a generic parameter in a class, interface, or method	declarations
Statement	A single unit of execution within a Java program, such as a declaration or expression	control_flow
CompilationUnit	Represents an entire Java source file, including package, imports, and class	code_structure
EnumConstantDeclaration	Declares constants within an enum type	literals_and_constants
IfStatement	A conditional statement that executes code based on a true or false condition	control_flow
ClassCreator	Creates an instance of a class, possibly an inner or anonymous class	code_structure
SwitchStatement	Selects one of many code blocks to execute based on the value of an expression	control_flow
EnumBody	Defines the body of an enum, including constants and other fields or methods	code_structure
PackageDeclaration	Declares the package that a Java class or interface belongs to	code_structure
Cast	Converts an object or value from one type to another	types_and_references
VariableDeclaration	Declares a variable, specifying its type and optional initial value	declarations
ArrayCreator	Creates a new array with a specified size and type	types_and_references
This	Refers to the current instance of a class	types_and_references
MethodReference	Refers to a method by name without executing it, often used in lambda expressions	expressions_and_operations
InnerClassCreator	Creates an instance of an inner class	code_structure
InterfaceDeclaration	Declares an interface, which can contain method signatures and constants	declarations
FormalParameter	Declares a parameter in a method or constructor	declarations
CatchClauseParameter	A parameter used in the catch block to represent an exception	exceptions
SynchronizedStatement	Ensures that a block of code is executed by only one thread at a time	control_flow
VoidClassReference	Refers to the special 'void' type, representing the absence of a return value	types_and_references
TypeArgument	An actual type passed as a parameter to a generic type	types_and_references
DoStatement	Executes a block of code at least once, then repeatedly based on a condition	control_flow
Assignment	Assigns a value to a variable	expressions_and_operations
ContinueStatement	Skips the current iteration of a loop and proceeds to the next iteration	control_flow
AssertStatement	Tests an assertion about the program, throwing an error if the assertion fails	exceptions
ExplicitConstructorInvocation	Explicitly calls another constructor in the same class or a superclass	declarations
AnnotationDeclaration	Declares an annotation type, used to create custom annotations	declarations
StringLiteralExpr	Represents a literal string value in the code	literals_and_constants
PrimitiveType	Represents a primitive data type such as int, char, or boolean	types_and_references
TryStatement	Defines a block of code that attempts execution and handles exceptions	control_flow
ElementArrayValue	Represents an array of values in an annotation element	code_structure
BlockStatement	Groups multiple statements together in a block enclosed by braces	code_structure
ClassReference	Refers to a class, often using its fully qualified name	types_and_references
ReturnStatement	Terminates a method and optionally returns a value	control_flow
IntegerLiteralExpr	Represents a literal integer value in the code	literals_and_constants
TernaryExpression	A shorthand conditional expression	expressions_and_operations
VariableDeclarator	Declares a variable and its initial value in one statement	declarations
BinaryOperation	Represents an operation involving two operands, such as addition or comparison	expressions_and_operations
ClassDeclaration	Declares a class, including its name, superclass, and body	declarations
TryResource	Represents a resource in a try-with-resources statement that is automatically closed	exceptions
MemberReference	Refers to a member of a class, such as a field or method	expressions_and_operations
SuperMemberReference	Refers to a member in the superclass of the current class	expressions_and_operations
Literal	Represents a literal value, such as a number, character, or boolean	literals_and_constants
CatchClause	Handles exceptions thrown in a try block	exceptions
WhileStatement	Executes a block of code repeatedly based on a condition	control_flow
ElementValuePair	Represents a key-value pair in an annotation	code_structure
ForStatement	Defines a traditional for loop with initialization, condition, and iteration	control_flow
StatementExpression	Represents an expression that can stand as a statement	expressions_and_operations
ConstantDeclaration	Declares a constant, which is a variable whose value cannot be changed	declarations
ArrayInitializer	Specifies the initial values for an array	types_and_references
MethodInvocation	Invokes a method on an object or class	expressions_and_operations
Modifier	Defines modifiers for classes, methods, or fields, such as public, private, or static	declarations
ThrowStatement	Throws an exception, signaling an error or abnormal condition	control_flow
LambdaExpression	Represents an anonymous function	expressions_and_operations
SwitchStatementCase	Represents a case label in a switch statement, matching specific values	code_structure
MethodDeclaration	Declares a method, including its return type, name, and parameters	declarations
BasicType	Represents a basic data type such as int, float, or char	types_and_references
SuperMethodInvocation	Invokes a method from the superclass of the current class	expressions_and_operations
ForControl	Specifies the initialization, condition, and update parts of a for loop	control_flow
CompilationUnit	Represents the top-level node in AST produced by the parser as the root of the tree	declarations

Table 9: Conversion table from NodeType to Category

Overall, these heatmaps illustrate the relational complexity within each dataset, highlighting the critical role of *"Operation"* and *"Others"* categories in maintaining the structural integrity of the codebase. This complexity presents challenges for graph-based models, which must effectively capture these dense interdependencies to make accurate predictions.

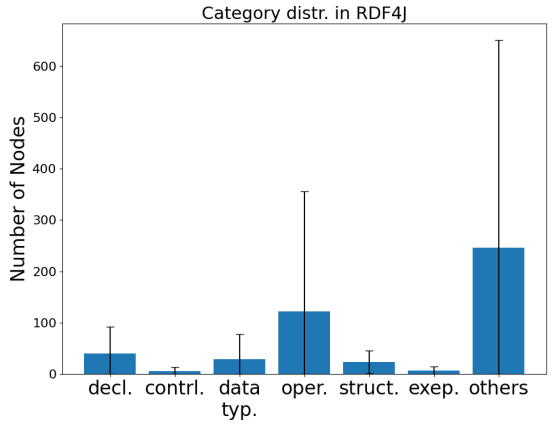

Figure 6: Node Category Distribution for RDF4J dataset

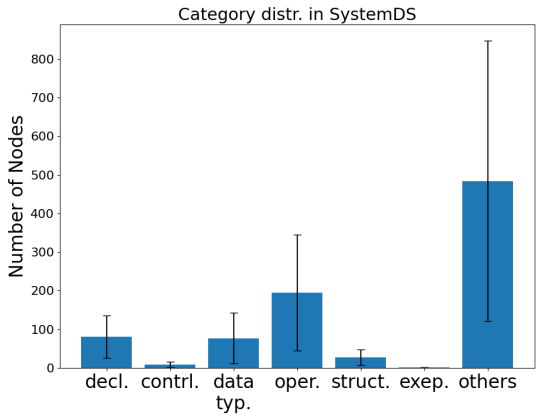

Figure 7: Node Category Distribution for SystemDS dataset

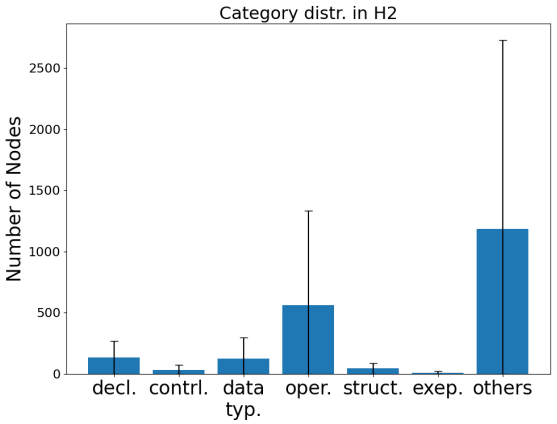

Figure 8: Node Category Distribution for H2 dataset

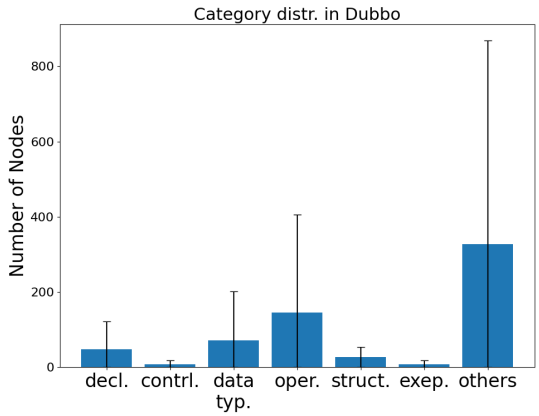

Figure 9: Node Category Distribution for Dubbo dataset

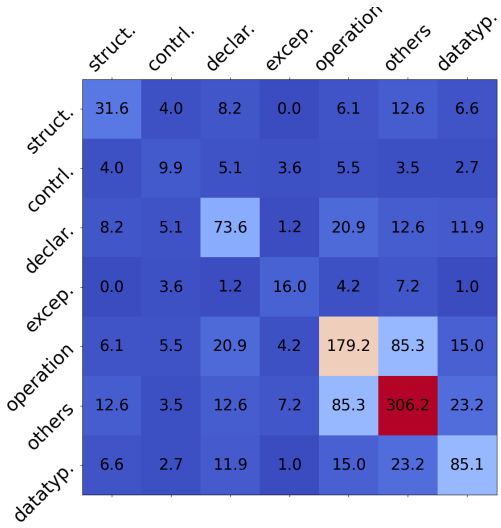

Figure 10: Average number of relations for dataset Dubbo

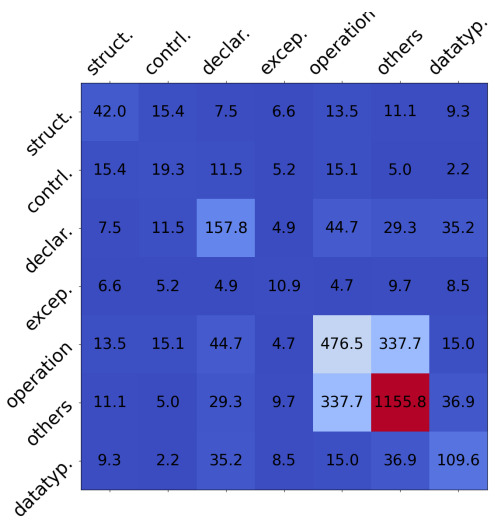

Figure 11: Average number of relations for dataset H2

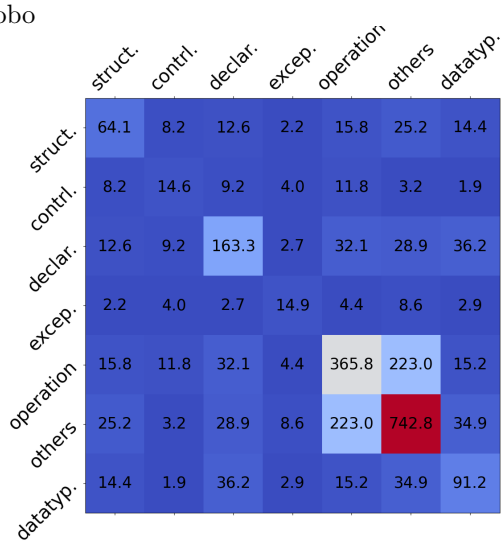

Figure 12: Average number of relations for dataset Hadoop

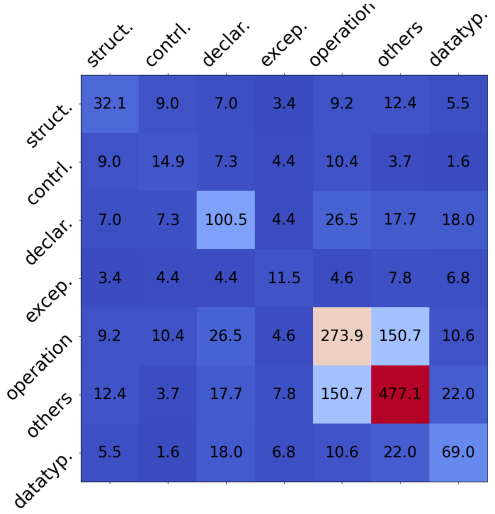

Figure 13: Average number of relations for dataset OssBuilds

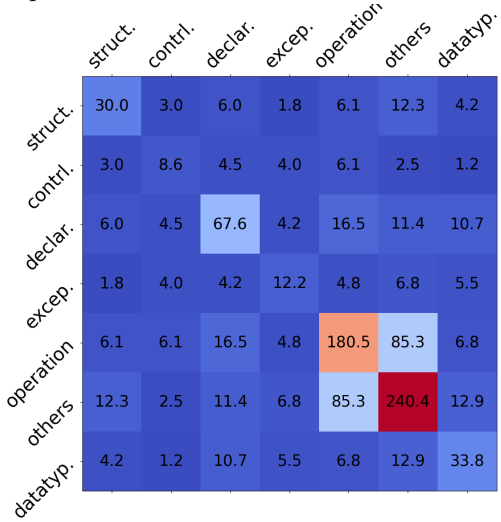

Figure 14: Average number of relations for dataset RDF4J

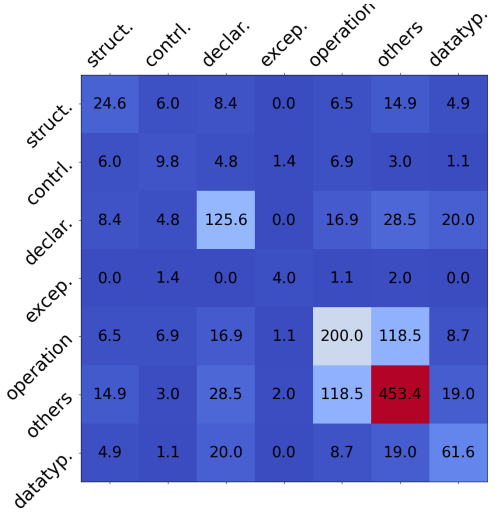

Figure 15: Average number of relations for dataset SystemDS

F Additional Graph Statistics

This section provides additional statistics for an overview of the proposed datasets. Figures 16 and 17 show two `RelSC` and two `Multi-RelSC` networks for Hadoop and OssBuilds, respectively.

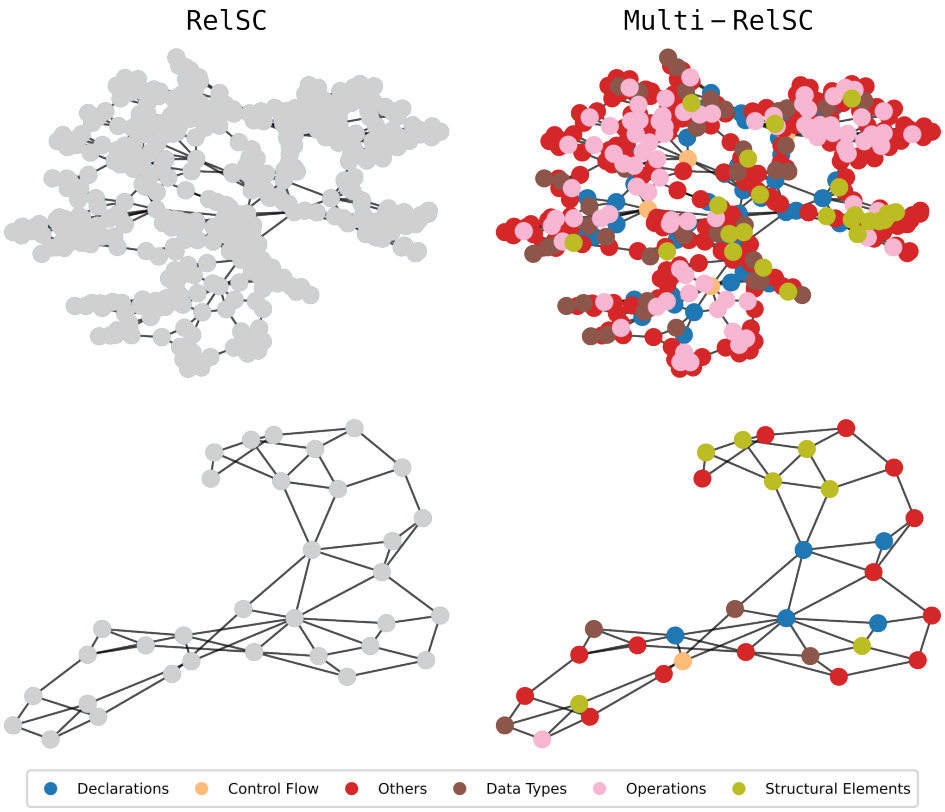

Figure 16: Example of `RelSC` and `Multi-RelSC` graphs from Hadoop

In Table 10, we present the means and standard deviations of several key graph metrics calculated for the proposed datasets. Specifically, we report the average density, indicating the proportion of actual connections to possible connections within each graph. We also include the average degree, reflecting the mean number of connections per node, and the average clustering coefficient, which describes the tendency of nodes to form tightly connected groups. Additionally, we provide the average diameter, representing the longest shortest path between any two nodes, and the average path length, capturing the mean shortest path across all node pairs. Lastly, we report the degree assortativity, which measures the correlation in degree between connected nodes.

Dataset	Density	Degree	Clustering	Diameter	Path Length	Assortativity
SystemDS	0.010 (± 0.023)	3.80 (± 0.06)	0.29 (± 0.02)	18.3 (± 4.5)	7.6 (± 1.3)	0.12 (± 0.06)
Dubbo	0.026 (± 0.047)	3.80 (± 0.12)	0.31 (± 0.04)	13.9 (± 3.7)	6.7 (± 1.4)	0.15(± 0.09)
RDF	0.041 (± 0.046)	3.78 (± 0.14)	0.30 (± 0.03)	12.7 (± 5.7)	5.9 (± 2.1)	0.17(± 0.08)
H2	0.005 (± 0.005)	3.82 (± 0.05)	0.33 (± 0.02)	22.1 (± 9.1)	8.6 (± 1.9)	0.11 (± 0.08)
OSSBuilds	0.027 (± 0.041)	3.79 (± 0.12)	0.31 (± 0.03)	15.6 (± 7.3)	6.8 (± 2.2)	0.15 (± 0.08)
Hadoop	0.011 (± 0.018)	3.82 (± 0.06)	0.30 (± 0.02)	17.3 (± 11.7)	7.5 (± 3.1)	0.12 (± 0.07)

Table 10: Dataset Statistics: Mean Values with Standard Deviations

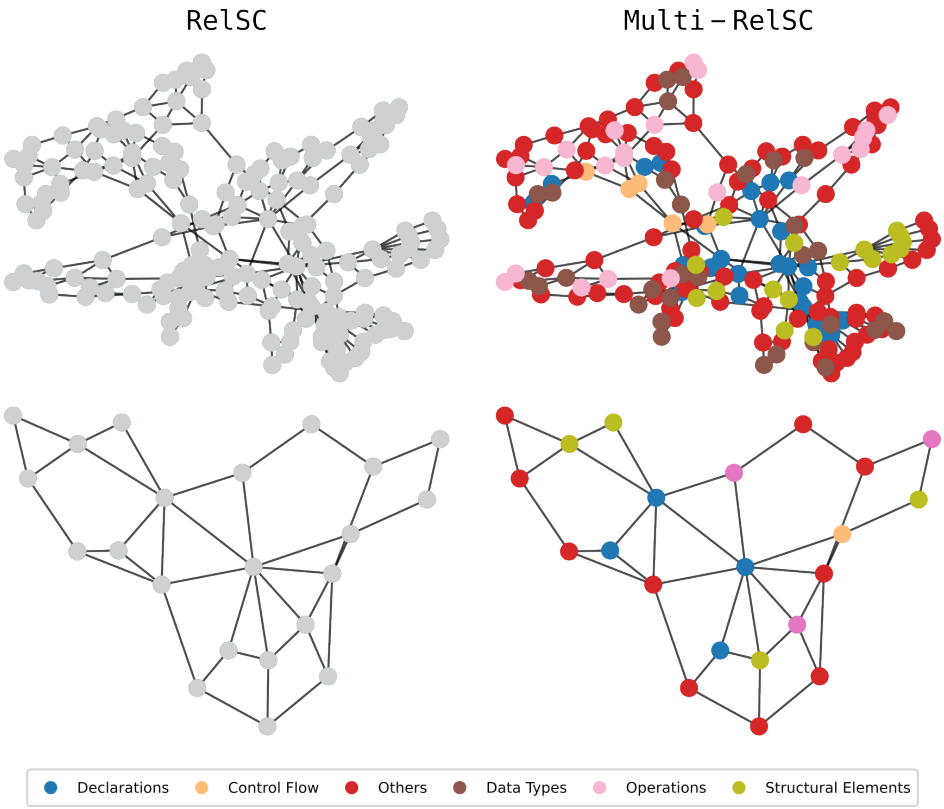

Figure 17: Example of `RelSC` and `Multi-RelSC` graphs from OssBuilds

F.1 Metric Distributions

Figure 18 presents the degree distributions of the OssBuilds and Hadoop datasets. To enhance clarity and make patterns in the distributions more visible, the y-axis is displayed on a logarithmic scale. This adjustment highlights the spread of node degrees across a wide range, helping to capture variations that may be less noticeable on a linear scale.

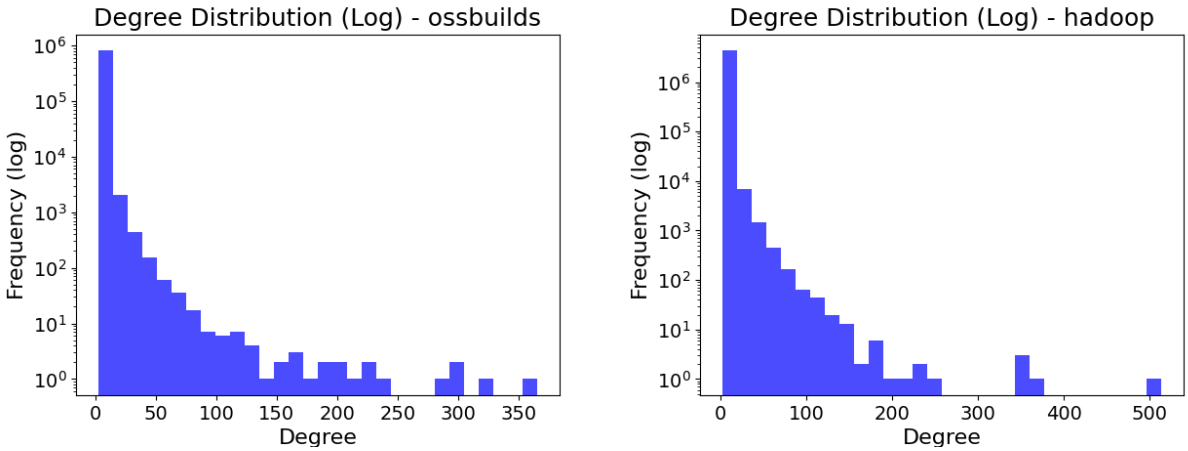

Figure 18: Degree distributions of OssBuilds (left) and Hadoop (right)

G Dataset Diversity and Bias Mitigation

To address concerns about the quality and representativeness of our dataset, we provide a detailed analysis of the diversity of code samples and the steps taken to mitigate potential biases in the data collection process. Our dataset comprises code from five distinct open-source projects collected through two different sources and methods, ensuring a broad coverage of code patterns and complexities relevant to software performance prediction tasks.

G.1 Diversity of Code Samples

Our dataset includes code from the following projects:

- **OSSBuilds Dataset:** This dataset encompasses four open-source projects, each contributing unique code patterns due to their different functionalities:
  - **SystemDS:** An Apache machine learning system for the data science lifecycle.
  - **H2:** A Java SQL database engine.
  - **Dubbo:** An Apache remote procedure call (RPC) framework.
  - **RDF4J:** A framework for scalable RDF data processing.

  These projects introduce a variety of code patterns, including database management, machine learning algorithms, RPC mechanisms, and data processing workflows. The diversity is reflected in the structural variations of the code and the resulting graphs.
- **HadoopTests Dataset:** Derived from the Apache Hadoop framework, this dataset includes 2,895 test files. Hadoop is renowned for processing large datasets across distributed computing environments, contributing complex code structures and control flows to our dataset.

Table 1 illustrates that the average number of nodes in the HadoopTests dataset is almost double that of the OSSBuilds dataset (1,490 vs. 875 nodes), indicating higher complexity in the Hadoop code samples. This indicates that our dataset has two main characteristics: the diversity of the code patterns and the complexity.

G.2 Mitigation of Potential Biases

To minimize biases in our data collection process, we employed two different methods and environments:

- **OSSBuilds Data Collection:** Execution times were collected from the continuous integration (CI) systems of the respective projects using GitHub's shared runners. This approach leverages a standardized environment provided by the CI infrastructure, reducing variability due to hardware differences.
- **HadoopTests Data Collection:** We conducted multiple executions of Hadoop's unit tests on dedicated virtual machines within our private cloud. Each VM was configured with two virtual CPUs and 8 GB of RAM, and all non-essential services were disabled to ensure consistent performance measurements.

By diversifying our data sources and controlling the execution environments, we mitigated potential biases related to hardware configurations, workload fluctuations, and environmental inconsistencies.

G.3 Representativeness and Generalization

The inclusion of diverse projects with varying functionalities enhances the representativeness of our dataset. The code samples encompass different structures, control flow statements, and data dependencies, which are critical for modelling software performance. The resulting graphs are generalized to various coding

patterns, excluding interface files that primarily contain function declarations without executable code. We intentionally did not include call graphs in the augmentation of ASTs to focus on the executable aspects of the code, which are more indicative of performance characteristics.