From Code Analysis to Fault Localization: A Survey of Graph Neural Network Applications in Software Engineering

Maojie PAN*, Shengxu LIN, Zhenghong XIAO
School of Computer Science, Guangdong Polytechnic Normal University, Guangzhou, Guangdong 510665, China

Abstract—**Graph Neural Networks (GNNs) represent a class of deep machine learning algorithms for analyzing or processing data in graph structure. Most software development activities, such as fault localization, code analysis, and measures of software quality, are inherently graph-like. This survey assesses GNN applications in different subfields of software engineering with special attention to defect identification and other quality assurance processes. A summary of the current state-of-the-art is presented, highlighting important advances in GNN methodologies and their application in software engineering. Further, the factors that limit the current solutions in terms of their use for a wider range of tasks are also considered, including scalability, interpretability, and compatibility with other tools. Some suggestions for future work are presented, including the enhancement of new architectures of GNNs, the enhancement of the interpretability of GNNs, and the design of a large-scale dataset of GNNs. The survey will, therefore, provide detailed insight into how the application of GNNs offers the possibility of enhancing software development processes and the quality of the final product.**

Keywords—Graph neural networks; fault localization; code analysis; software quality

I. INTRODUCTION

A. Context

Graph Neural Networks (GNNs) form branches of neural networks that generate inferences from data in a graph form. These graphs comprise nodes and edges and facilitate comprehension of intricate data dependencies and connections [1]. GNNs have recently proven to be effective in several real-world applications, including social networks, chemistry, and natural language processing [2]. Because of these characteristics, deep graphs can be applied to modeling and learning structures with complex interdependencies between them [3].

In software development, a lot of processes are inherently associated with data that can be naturally modeled using graphs. These include control flow diagrams, dependency diagrams, and an abstract syntax tree where software programs' structure and relationships are analyzed [4]. These representations are rather complex, and the traditional paradigm of machine learning often fails to identify all the necessary features and relationships within the graphs, which leads to poor results in fault localization, code analysis, and software quality evaluation [5]. However, the appearance of GNNs provides a more suitable opportunity to explore the areas of modern software development, as they can make use of the structural information encoded in these diagrams [6].

B. Motivation

Fault location, a key component of software debugging and maintenance, is one area where GNNs have shown considerable promise [7]. Through program graphs, where programs are modeled, GNNs can identify patterns related to faulty code snippets, guiding developers to locate the precise location of the bug more effectively [8]. Similarly, GNNs can be used in code analysis for functions including code synthesis, clone detection, and refactoring by understanding the structural similarities and differences between code segments. These capabilities can significantly reduce the time and effort required to maintain and improve software systems [9].

In addition to fault location and code analysis, software quality assurance also utilizes GNNs to prioritize test cases by identifying the potential effects of each test case on the software, to predict error-prone regions based on historical data, and to enhance the overall reliability of software systems [10]. Developing applications with the help of GNNs is not without challenges, although. Scalability, interpretability, data availability, and integration with existing tools are some of the areas that need to be addressed to leverage the benefits of GNNs in this field fully.

C. Problem Statement

Despite recent growth in the adoption of GNNs in software development, there remains a lack of complete knowledge about their proper usage within various software development processes, such as fault localization, code analysis, and quality assurance. Some of the unanswered questions include the scalability issues of GNNs, the interpretability of the results, and the integration of GNNs with traditional software tools.

D. Research Objectives

The paper provides an exhaustive survey of recent advancements, outlines the various applications of GNNs in software development, and presents challenges and directions for this emerging field. Applications of GNNs to fault localization, code analysis, and quality assurance will be discussed, and the methods and results will be assessed. We will also analyze the practical issues faced by researchers, including those related to scalability, the need for easy-to-understand models, and the integration of GNNs with current software development tools and processes. This survey aims to identify the revolutionary promise of GNNs in software development

and to encourage more research and development in this field. The research attempts to answer the following research questions.

- What are the current applications of GNNs to essential software engineering activities like fault detection and code analysis?
- What are the comparative advantages of GNN-based approaches over traditional static and dynamic analyses?
- What are the limitations and challenges of deploying GNNs at scale in real-world software development processes?
- What are hybrid approaches that integrate classic methods with GNNs?

The paper is organized as follows. In Section II, a general overview of GNNs and their fundamental concepts and essential techniques is presented. In Section III, the targeted applications of GNNs in software development are presented, and their influence on fault localization, code analysis, and quality assurance is highlighted. In Section IV, directions and potential advancements are explored, and contemporary challenges to applying GNNs to software development are discussed. The paper concludes with a general discussion of its findings and contributions in Section V.

II. Graph Neural Networks: An Overview

A. Definition and Background

GNNs are a class of neural networks designed to perform inference on data represented as graphs. A graph G is defined as $G = (V, E)$, where V is a set of vertices (or nodes) and E is a set of edges connecting the nodes. Each node $v \in V$ and edge $e \in E$ can have associated features, which are essential for capturing the properties and relationships within the data.

GNNs are inspired by Convolutional Neural Networks (CNNs). Before delving into GNNs, it is essential to understand why CNNs and Recurrent Neural Networks (RNNs) cannot handle graph data effectively. As depicted in Fig. 1, CNNs operate on data with a grid structure, such as images. As an alternative, RNNs are adapted to sequences, like text.

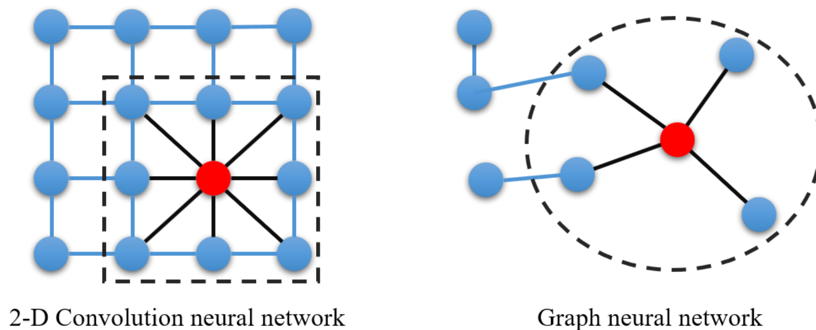

Fig. 1. CNN vs. GNN.

Text data is usually stored within arrays. Similarly, matrices are optimal to store image data. But, as also depicted in Fig. 1, arrays and matrices are incapable of handling graph data. Graphs employ a special process called graph convolution. This technique enables deep neural networks to process graph-structured data and yield a GNN directly. It can be seen that masking techniques and filtering operations are employed to convert images to vectors. However, classical masking techniques are not suitable for graph data input, as depicted in the rightmost image.

In contrast to classical static and dynamic analyses that act based on pre-declared rules or symbolic rationale, GNNs use data-driven learning to extract patterns from graph-structured program representations. This learning feature enables GNNs to generalize across codebases, identify patterns that are not easily specified by human-created rules, and evolve to respond to new domains without requiring human intervention. Some static analyses incorporate feedback loops (e.g., via CEGAR) but fail to include the ongoing, end-to-end learning process that allows GNNs to fine-tune with additional data.

B. GNN Evolution

Significant development has occurred in applying neural networks to graph-structured data over the years. Early approaches, like the use of recursive neural networks, laid the foundation by leveraging the same set of parameters repeatedly over the graph structure [11]. Nonetheless, this was limited by the fact that the approaches were unable to efficiently adopt arbitrary graph structures.

The advent of Graph Convolutional Networks (GCNs) was a groundbreaking development. GCNs generalize the concept of convolution to graph-structured data, rather than grid-like data (like images). By operationalizing convolution over the neighborhood of every node, GCNs are especially useful for node classification and link prediction tasks [12]. Following the success of GCNs, several different classes of GNNs have emerged:

- Graph Attention Networks (GATs): These networks utilize an attention mechanism that weighs the importance of various node features [13]. This will enable the model to emphasize the most significant components of the graph and improve performance in cases where some nodes have greater influence.
- Graph Recurrent Networks (GRNs): Utilize recurrent network architectures to process graph data. Such networks are ideally suited to sequential graph data, where the ordering of the node or edge is considered significant [14].

- Graph Autoencoders (GAEs): Used in unsupervised learning from graph data. GAEs encode graph data into a latent space and reconstruct the graph, finding applications in graph generation and detecting anomalies [15].

C. GNN Training

Training a GNN involves learning parameters to identify patterns and relationships within the graph data. Training can be supervised, unsupervised, or semi-supervised based on the availability of labeled data.

- Supervised learning: Trains the GNN based on labeled graph data, and the labels are associated with the node, edge, or graph level [16]. The model is trained to make predictions of labels based on input features and graph topology.
- Unsupervised learning: The GNN is trained to embed the graph data without labels. Techniques such as graph autoencoders and contrastive learning are typically employed to obtain informative representations of the graph [17].
- Semi-supervised learning: It combines labeled and unlabeled data to improve the learning process [18]. In cases where labeled data is limited, and many real-world applications face this issue, this is especially helpful.

III. GNN Applications in Software Engineering

GNNs have become a universal tool in software development, leveraging the inherent graph-like characteristics of a wide variety of software artifacts. As shown in Table I, various GNN architectures possess distinct strengths and applications, making them suitable for a wide range of software development activities. Varying from code analysis and fault location to software quality assurance, numerous paths can be modeled, analyzed, and optimized using GNNs to enhance software systems. This section focuses on the application of GNNs in software development, with a particular emphasis on the various architectures employed to address complex issues and enhance the effectiveness and efficiency of software development.

TABLE I. SUMMARY OF GNN ARCHITECTURES

GNN architecture	Key features	Strengths	Typical applications in software engineering
GCN	Applies convolution operations to graph data and aggregates information from neighboring nodes.	Efficient in collecting local neighborhood information.	Node classification, fault localization, code analysis.
GAT	Utilizes attention mechanisms to weigh the importance of neighboring nodes' features.	Allows the model to focus on the most relevant parts of the graph.	Code summarization, bug prediction, and test case prioritization.
GRN	Incorporates recurrent neural network architectures for processing graph data over time.	Effective for sequential graph data, capturing temporal dependencies.	Analyzing execution traces, dynamic analysis.
GAE	Encodes graph structures into a latent space and reconstructs the graph for unsupervised learning.	Useful for graph generation and anomaly detection.	Detecting code clones unsupervised code analysis.
Message Passing Neural Network (MPNN)	Generalizes GNNs with a message passing framework where nodes iteratively exchange messages.	Flexible in handling different types of graph structures and tasks.	Program dependency analysis bug prediction.
Spatial-Temporal GNN (ST-GNN)	Models both spatial and temporal aspects of graph data, handling dynamic changes in the graph.	Captures both structural and temporal evolution of graphs.	Real-time monitoring of software systems and dynamic code analysis.

A. Fault Localization

Fault location is a critical part of software maintenance and debugging, aiming to identify the precise fault locations within a software program [19]. Fault location strategies are typically based on static or dynamic analysis techniques, which can be time-consuming and may not always yield accurate results. The capability of modeling and learning with graph-structured data offers a promising solution for enhancing fault location by leveraging the intrinsic software program structure. As demonstrated in geotechnical engineering, domain-specific modeling in the field of geotechnical engineering [20] shows that adapting models to consider material heterogeneity and structural anisotropy enhances prediction capability. Similarly, task-specific tuning of the GNN architecture may be necessary for code analysis and fault localization.

Trained static analysis approaches, such as data flow analysis, control flow analysis, and abstraction-based analysis, have long supported software fault detection and code understanding. However, they are typically based on predefined rules and cannot handle dynamic software behaviors or loosely formatted source code. By contrast, GNN-based analysis learns to operate directly from the graph structure of code and execution traces. This can guide the model to detect subtle, non-local relations and semantic structures that are not detectable with earlier analyses. Additionally, various program representations, such as Abstract Syntax Trees (ASTs), Program Dependency Graphs (PDGs), and runtime traces, can be combined by GNNs within a single learning framework, providing a richer and more dynamic understanding of software systems.

1) Program dependency graphs: PDGs are a popular format adopted in fault localization [21]. PDGs encode the interdependencies between various components of a program, in the form of data dependencies (which variables depend on) and control dependencies (which statements cause others to execute). In encoding a program in a PDG, GNNs can examine the interrelations between various facets of the code.

The GNNs can be trained with the PDGs to identify patterns related to defective code snippets. For example, a GNN can be trained to identify nodes within the PDG that are likely to have faults by learning from historical fault data. This requires

encoding node and edge features within the PDG and applying a message-passing function to consolidate information passed by the neighboring nodes. The node representations generated can be used to make predictions about the presence of a fault at each node.

2) Abstract syntax trees: ASTs embody the program's syntax structure. Each node within an AST denotes a construct in the source code, e.g., a variable, an operator, or a control-flow statement [22]. ASTs give a hierarchical representation of a program, reflecting the nested relationships between program components.

ASTs are amenable to GNN applications that aid in fault localization by leveraging code semantics and structure. By applying GNNs to processed ASTs, faults can be identified based on syntactic patterns, effectively learning to recognize error-prone code patterns or patterns or combinations of patterns. This can be especially useful for identifying faults that result from intricate relationships between various code components.

GNNs capture and leverage the hierarchies and control flows embedded in code, such as nested loops, conditional statements, and recursion. Such patterns of code are common to frequent bugs, including infinite loops, misbounds in loops (i.e., off-by-one errors), faulty exception handling, and misuse of break and continue statements. Through examination of ASTs and control flow graphs, GNNs can identify recurring structural patterns that relate to such bugs. For instance, GNNs can identify anomalies in nesting within loops that suggest missing base cases or faulty exit conditions within recursive routines, enabling early detection of runtime faults and logical errors.

3) Dynamic analysis with execution traces: Most software defects have their roots in faulty state initialization or incorrect state transitions, and not all of them are explicitly programmed in the static code structure [23]. To overcome this, execution traces, runtime event representations in graph format, can be fed to GNNs to capture variable updates, function call sequences, and conditional jumps. Traces embed time-dependent relationships and implicit transitions between states, enabling GNNs to capture patterns related to erroneous program execution. In cases where more implicit information is lacking, hybrid GNNs can take both static data (e.g., ASTs or PDGs) and dynamic data (e.g., memory dumps, execution traces) to create a more holistic fault detection system.

4) Empirical studies and results: Numerous empirical studies have demonstrated the effectiveness of GNNs in fault localization. The investigations typically involve training the GNNs with known faulty programs and subsequently with unknown programs. Precision, F1-score, and recall are metrics used to measure a model's capability to identify faulty portions of code correctly.

For instance, one experiment may involve training a GNN using a database of Java programs based on their PDGs and ASTs to make predictions about fault locations. The outcome can demonstrate that a GNN outperforms conventional fault localization methods, such as SBFL, by yielding more accurate and precise fault predictions. Such experiments highlight the tremendous potential of GNNs to enhance the efficiency and efficacy of fault localization processes within software development.

Fig. 2 illustrates a novel fault-localization technique that utilizes a graph-based representation of faulty feeders. Fault detection accuracy is improved by integrating data from different data sources, like geographic information system (GIS) databases and supervisory control and data acquisition (SCADA) systems. GIS databases provide important information about network topology, protection device locations, and electrical characteristics. SCADA systems provide real-time operational data such as protection device activations, fault currents and voltage measurements. To further increase the intelligence of the system, data from customer information systems (CIS) and station oscillographs can be integrated.

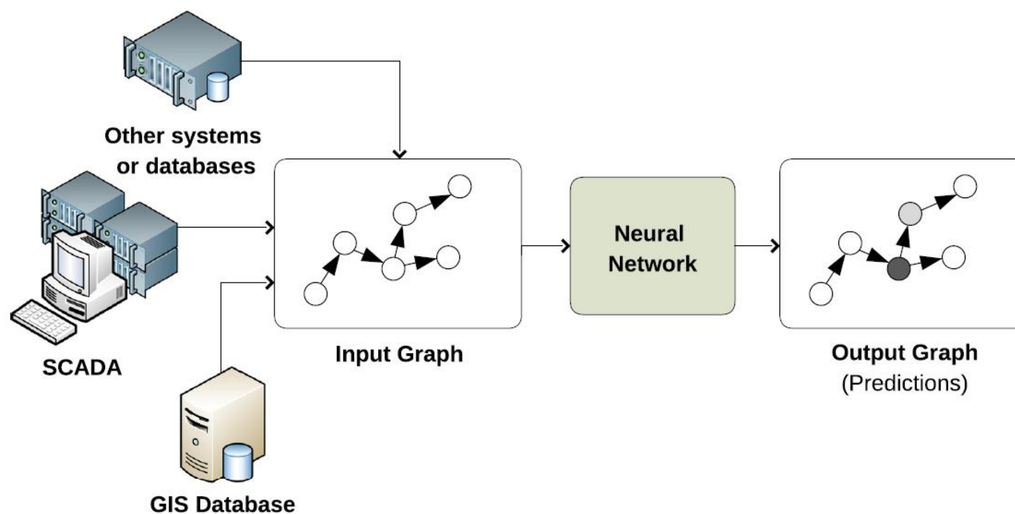

Fig. 2. A novel fault localization technique using graph-based representation.

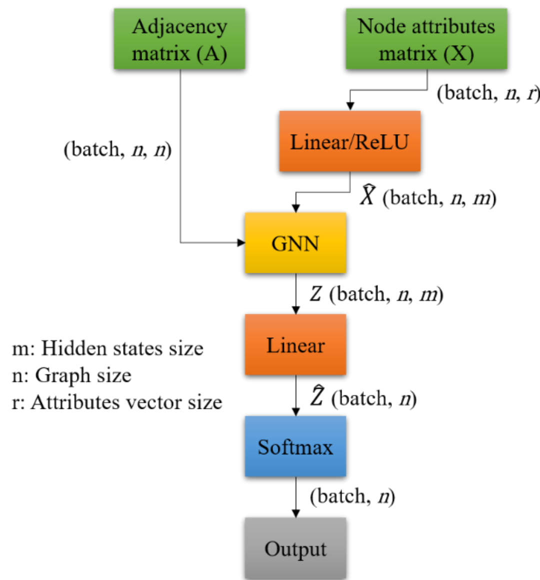

Fig. 3. Neural network model processing graph.

The graph-based representation is then fed to a neural network model, as shown in Fig. 3. The input to this model is the adjacency matrix A and the attribute matrix X of the graph. The first stage employs a linear combination with a rectified linear unit (ReLU) activation function, which projects the input features into a new space of representations. The size of the hidden states and the number of input attributes are hyperparameters that influence the model's capacity and generalization capability.

$$\hat{X} = ReLU(W_{in}X + b_{in}) \quad (1)$$

In the subsequent layers, a GNN is employed to extract the dense relationships between the nodes in the graph. A GNN propagates information over the graph sequentially, allowing the model to capture relationships between non-immediately adjacent nodes. The number of propagation steps is a hyperparameter that controls how much the model supports long-term dependencies. But more steps of propagation correspond to increased computational cost and memory demand.

The network computes a score for every node via a linear combination and passes this to a softmax function, where the scores are normalized to a probability distribution. This is to calculate the probability that a node is the location of the fault.

B. *Code Analysis*

Code analysis involves a set of activities to comprehend and enhance the quality of software [24]. Some of the activities involved include code summarization, clone detection, and refactoring, among others. Given that GNNs can capture the structural and relational aspects of code, they provide substantial benefits in accomplishing the above activities by generating more precise and informative findings compared to the conventional methods.

1) Code summarization: Code summarization necessitates the creation of compact, natural language descriptions of code functionality [25]. This is a fundamental requirement of documentation and codebase browsing over large and intricate codebases. GNNs can leverage the structural information contained in ASTs and other code graphs to enhance code summarization.

By representing code in a graph format, GNNs can capture the structural and relational information that is crucial to the functionality of code snippets. As a case in point, a GNN can be trained to embed the AST of a code snippet and create a summary by translating the learned representation back into natural language. This enables the model to comprehend the relationships and contexts within the code, providing more accurate and relevant summaries.

2) Clone detection: Code clone detection aims to identify similar or duplicated code sequences within a codebase. Clones are a source of maintenance issues and potential errors and are therefore especially essential to detect for software quality [26]. Clone detection can be greatly aided by the use of GNNs that focus on structural similarities in the graph representations of code.

Clone detection can be performed by representing code snippets as graphs (e.g., PDGs, ASTs) and applying GNNs to extract their structural representations. A comparison of the structural representations will enable the identification of similar code fragments, despite their syntactic differences. This feature is especially helpful in Type-3 clone detection, where the code snippets are syntactically different but semantically the same.

3) Code refactoring: Code refactoring rearranges previously written code without altering its external functionality, making the code more readable, maintainable, and efficient [27]. Identifying refactoring areas and recommending suitable transformations are the two fundamental challenges of refactoring. GNNs can help refactoring by inspecting the code structure and extracting patterns that suggest that refactoring is warranted.

The GNNs are trained over refactoring histories and can detect code smells and anti-patterns that are usually amenable to refactoring. By encoding code graphs and applying a GNN to operate over them, the models can suggest refactoring opportunities based on the detected patterns. A GNN, for instance, will detect duplicated code, long sequences of methods, or highly coupled classes that are amenable to refactoring. The employment of GNNs in code refactoring has made refactoring proposals more accurate and beneficial. Developers employ such models to automate refactoring suggestions and provide optimal recommendations.

4) Empirical studies and results: Empirical studies on applying GNNs to code analysis issues have demonstrated their effectiveness and superiority over traditional alternatives. For example, studies on code summarization using GNN-based methods have demonstrated improved performance in generating accurate and concise descriptions of code. Likewise, studies on clone detection have demonstrated that clones can be

effectively identified by GNNs, yielding increased precision and recall compared to traditional methods.

Empirical studies of code refactoring have demonstrated the capability of GNNs to identify sophisticated code smells and provide useful refactoring suggestions. The studies are carried out with benchmark data sets and actual codebases and yield evidence of the utility of GNNs in code analysis.

Fig. 4 shows a hybrid GNN framework for code analysis. This framework integrates both static and dynamic graph representations to improve code summary learning. It consists of four main components: (1) Retrieval-Augmented Static Graph Construction, which augments the original code graph with retrieved code summary pairs to improve feature learning; (2) Attention-based dynamic graph construction, where a global attention mechanism enables message propagation between arbitrary pairs of nodes, enabling more flexible relationships; (3) Hybrid GNN (HGNN), which combines information from static and dynamic graphs through hybrid messaging to enrich node representations; and (4) Decoder, which uses an attention-based LSTM model to generate a code summary from the learned representations. This framework effectively leverages both structural and dynamic aspects of code to improve the quality of code analysis and summarization tasks.

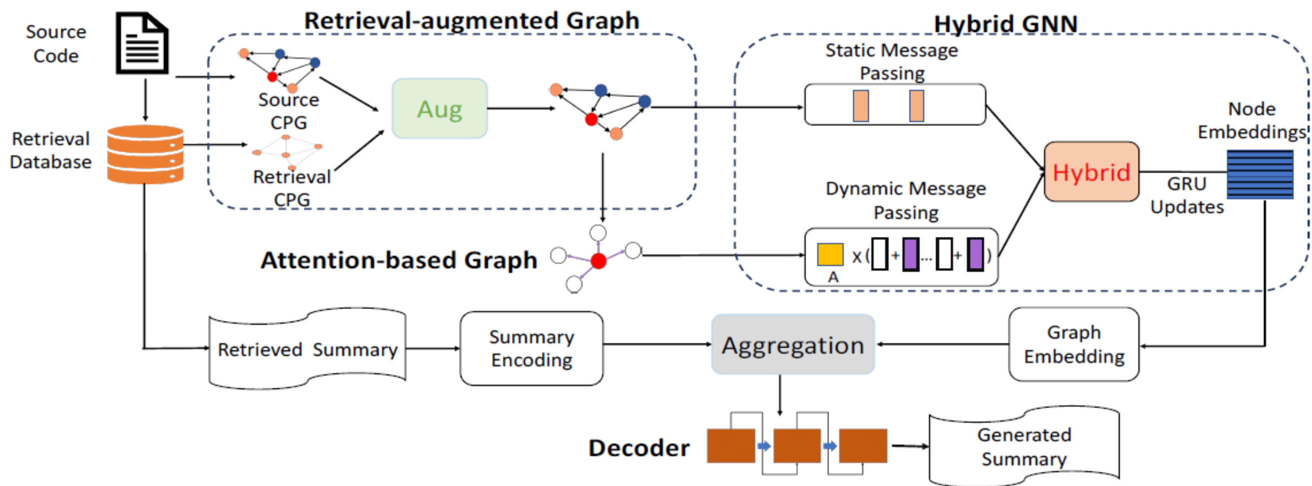

Fig. 4. Hybrid GNN framework for code analysis.

C. Software Quality Assurance

Software Quality Assurance (SQA) is an essential software engineering process that ensures software artifacts meet the quality standards expected of them [28]. This includes activities that are related to testing, verification, validation, and bug prediction. GNNs have demonstrated tremendous potential to improve many aspects of SQA by extracting structural information embedded in software artefacts to make more accurate predictions and provide better insights.

1) Test case prioritization: Test case prioritization involves sequencing test cases in a way that prioritized test cases are run first [29]. This becomes more significant with regression testing, where a full test suite must be rerun, and for that, there are time and cost factors. Sorting test cases can be facilitated with the help of GNNs by identifying the relationships and dependencies within the software.

By representing the software and test cases as a graph, where code components serve as nodes and dependencies or interactions are represented as edges, the areas of the software most likely to be affected by recent updates can be identified using GNNs. This helps the model concentrate only on test cases that correspond to the key areas. Experimental studies have demonstrated that test case prioritization with the aid of GNNs can facilitate fault detection much earlier, thereby enhancing the efficiency and effectiveness of the test process.

2) Bug prediction: Bug prediction entails predicting where and when defects are likely to occur in various areas of the software. Proper bug prediction can be useful in better allocating resources and targeting quality assurance activities to the areas of the code that are at the highest risk of defects [30]. Bug prediction can be significantly improved by utilizing GNNs that analyze the structural characteristics of software and learn from bug data over time.

Software can be modeled using different types of graphs, such as dependency graphs or co-change graphs, where nodes and dependencies, or co-change relations, represent software components, and edges represent these relationships. GNNs can process such graphs to identify patterns that predict bug-prone locations. For instance, a GNN can be trained using past data to make predictions about the likelihood of defects in various components based on their structural characteristics and change history. Experiments have established that bug prediction models based upon GNNs are more precise and detailed compared to conventional statistical and machine-learning-based models.

3) Code review assistance: Code reviews are an essential aspect of the software development life cycle, ensuring improvement in code quality through peer review. GNNs can be leveraged to assist with code reviews by suggesting and detecting potential issues, as well as recommending enhancements [31]. Based on analyzing the code structure and

the relationships between various code components, GNNs can identify problematic areas.

For example, code smells, security issues, or compliance with the coding standard can be detected by GNNs. By treating the code and its dependencies as a graph data type and learning patterns typical of high-quality code, GNNs can provide developers with real-time feedback during code reviews. This not only accelerates the review process but also helps ensure a superior level of code quality.

4) Empirical studies and results: Empirical research on applying the use of GNNs in software quality assurance has produced promising evidence. In test case prioritization, research has shown that GNNs are capable of achieving better fault detection at earlier stages of the test process than other prioritization techniques. In bug prediction, research has shown that predictions made by GNNs are more accurate, enabling teams to address issues proactively.

While helping with code review, we have observed that systems utilizing GNNs enhance review efficiency and effectiveness by identifying a higher percentage of issues without manual examination. Such research involves real-world data sets and compares them with standard practices to validate the benefits gained from applying GNNs.

IV. Future Directions

The application of GNNs in software engineering is still in its early stages, with numerous areas to explore and develop in the future. With the improvement and development of GNNs, their capability to revolutionize various facets of software engineering, including fault localization, code analysis, and software quality assurance, becomes increasingly evident. Table II outlines some of the key areas to explore and refine in the future, providing a systematic overview of the essential directions that will drive the continued improvement and deployment of GNNs within this discipline.

TABLE II. Key Areas for Future Research and Development in GNNs for Software Engineering

Future direction	Description	Expected outcomes
Advanced GNN architectures	Development of more specialized and scalable GNN architectures to handle large-scale, complex software systems.	Improved efficiency and effectiveness in handling vast data and intricate relationships.
Explainable AI for GNNs	Creation of methods to enhance the interpretability and transparency of GNN models.	Increased trust and adoption of GNNs through clearer, more understandable predictions.
Real-world applications	Conducting empirical studies and applying GNNs in real-world software projects.	Validation of GNN effectiveness, identification of strengths and weaknesses, and wider industry adoption.
Integration with development tools	Seamless integration of GNNs into IDEs and CI/CD pipelines.	Enhanced real-time analysis, automated testing, and proactive bug detection.
Large-scale and high-quality datasets	Creation of comprehensive and publicly available datasets for GNN training and evaluation.	Improved performance of GNN models through access to diverse, well-annotated datasets.
Cross-disciplinary research	Encouraging collaboration across computer science, network science, cognitive science, and other disciplines.	Innovative solutions, improved scalability, and interpretability of GNNs in software engineering.

A. Advanced GNN Architectures

To fully realize the potential of GNNs in software development, more advanced and specialized architectures of GNNs need to be designed. Currently, architectures such as GCNs and GATs show promise but also face limitations when dealing with large and complex software systems. In their next steps, researchers should strive to develop scalable architectures of GNNs that can meaningfully interact with the vast amounts of data and intricate relationships found in large software projects.

Moreover, hybrid approaches that integrate GNNs with additional machine learning methods or domain-specific knowledge could further enhance their effectiveness. For example, integrating NLP methods with GNNs could enhance code documentation and abstraction. Additionally, integrating standard static and dynamic analysis tools with GNNs could result in more accurate fault localization and bug prediction.

B. Explainable AI for GNNs

One of the biggest hindrances to the large-scale adoption of GNNs in software development is the interpretability of their outputs. Developers and stakeholders should be able to know the rationale behind the predictions and suggestions made by the GNN models. As a result, there is a vital need to develop explainable AI methods for GNNs.

Research in this area should aim to develop methods that yield transparent and comprehensible explanations of the predictions made by a GNN. Mechanisms such as attention, feature importance analysis, and visualization tools can be designed to ensure that GNNs are more transparent and their output is more interpretable. As the explainability of GNN models improves, developers are more likely to have confidence in and efficiently utilize them in their development process.

C. Real-world Applications

To demonstrate the utility of GNNs in software development, it is necessary to conduct extensive empirical research and evaluate GNN models against real-world software projects. Such research should be conducted using different datasets and programming languages, as well as various development platforms and software fields. By comparing GNN models with conventional methods and measuring their efficiency in actual cases, the strengths and limitations can be identified, allowing for targeted areas for improvement.

Collaborating with industrial partners to implement GNNs in real-world applications can yield valuable insights and feedback. Industry case studies that demonstrate successful GNN implementation also have the potential to present practical applications and promote broader adoption.

D. Integration with Development Tools

For GNNs to be successfully employed in software development, they must be integrated seamlessly into existing development tools and processes. This includes developing usable interfaces, plugins, and APIs that enable developers to integrate GNN-based recommendations and analysis into their everyday workflows.

Future efforts should be directed toward building Integrated Development Environments (IDEs) and Continuous Integration/Continuous Deployment (CI/CD) pipelines that leverage the power of GNNs. This integration would enable real-time analysis, automated testing, and early bug detection, resulting in a more efficient and higher-quality software development process.

Outside of single-use cases, GNNs have the potential to extend to pre-existing analyses by delivering rich semantic outputs, such as code representations from summarization or similarity measures in clone detection. Those representations can be incorporated into symbolic or data-flow analysis to enhance inference procedures. Semantic embedding, for instance, can be used as a feature input in path prioritization during symbolic execution. Code clone clusters can be utilized to facilitate property propagation in verification. This inter-model synergy presents a hybridized strategy that blends the accuracy of conventional tools with the adaptability and abstraction power of deep learning algorithms.

E. Large-Scale and High-Quality Datasets

The effectiveness of GNN models is highly dependent upon having large, high-quality datasets to train and test them. Software engineering makes the development of such datasets problematic due to the heterogeneity of software projects and the necessity of accurate annotations. Future research should be directed toward developing well-rounded and public datasets that span a large gamut of software engineering activities.

Joint initiatives between academia, industry, and open-source projects can curate and pool valuable datasets. Such datasets should comprise different representations of graphs, such as program dependency graphs, execution traces, abstract syntax trees, and labeled data to perform activities like bug prediction, code summarization, and fault localization.

F. Cross-Disciplinary Research

Software engineering is a multidisciplinary field that combines components of computer science, mathematics, and engineering. Increased cross-disciplinary research will be encouraged in the future to harness the power of GNNs in software engineering. Concepts and methods borrowed from network science, data mining, and cognitive science can offer new insights and approaches to enhance GNN applications.

Joint research endeavors have the potential to provide innovative solutions to intricate problems in software development. For example, concepts borrowed from cognitive science enhance the usability and interpretation of GNN models, while innovations in network science facilitate the design of more scalable and efficient GNN architectures.

V. Conclusion

This research highlighted the significant potential of GNNs for revolutionizing software engineering, particularly fault localization, code analysis, and software quality assurance. With the capability to tap into the graph-structured information of software data, GNNs provide better insights and more precise predictions than classical alternatives. Our survey identified the current applications of GNNs to software issues in areas of interest, outlined key research and development directions, and suggested areas to address these challenges. Some of these include improving GNN architectures, enhancing model transparency, and integrating GNNs into development tools. By overcoming such challenges and increasing interdisciplinary research and development, the research highlights the potential of GNNs to enhance software development efficiency, accuracy, and reliability significantly. As technology improves in GNNs, the application of this technology to software engineering has the potential to yield better-quality software products, marking a groundbreaking improvement in the field.

Funding

This work was supported by 2022 Guangdong Province undergraduate Teaching Quality and teaching reform construction project: “Exploration and Practice of Teaching Reform in the Course of Software Testing Technology Based on CDIO Engineering Education Model” (No. 991700189).

References

[1] Y. Gan and Z. Hu, "Fusion Privacy Protection of Graph Neural Network Points of Interest Recommendation," International Journal of Advanced Computer Science and Applications, vol. 14, no. 4, 2023.

[2] G. Corso, H. Stark, S. Jegelka, T. Jaakkola, and R. Barzilay, "Graph neural networks," Nature Reviews Methods Primers, vol. 4, no. 1, p. 17, 2024.

[3] P. Veličković, "Everything is connected: Graph neural networks," Current Opinion in Structural Biology, vol. 79, p. 102538, 2023.

[4] X. Cheng, H. Wang, J. Hua, G. Xu, and Y. Sui, "Deepwukong: Statically detecting software vulnerabilities using deep graph neural network," ACM Transactions on Software Engineering and Methodology (TOSEM), vol. 30, no. 3, pp. 1-33, 2021.

[5] M. B. Bagherabad, E. Rivandi, and M. J. Mehr, "Machine Learning for Analyzing Effects of Various Factors on Business Economic," Authorea Preprints, 2025, doi: https://doi.org/10.36227/techrxiv.174429010.09842200/v1.

[6] S. Liu, "A unified framework to learn program semantics with graph neural networks," in Proceedings of the 35th IEEE/ACM International Conference on Automated Software Engineering, 2020, pp. 1364-1366.

[7] M. N. Rafi, D. J. Kim, A. R. Chen, T.-H. Chen, and S. Wang, "Towards Better Graph Neural Network-Based Fault Localization through Enhanced Code Representation," Proceedings of the ACM on Software Engineering, vol. 1, no. FSE, pp. 1937-1959, 2024.

[8] A. A. Kulkarni, D. G. Niranjan, N. Saju, P. R. Shenoy, and A. Arya, "Graph-Based Fault Localization in Python Projects with Class-Imbalanced Learning," in International Conference on Engineering Applications of Neural Networks, 2024: Springer, pp. 354-368.

[9] N. Mehrotra, A. Sharma, A. Jindal, and R. Purandare, "Improving cross-language code clone detection via code representation learning and graph neural networks," IEEE Transactions on Software Engineering, 2023.

[10] Z. Li et al., "Fault localization based on knowledge graph in software-defined optical networks," Journal of Lightwave Technology, vol. 39, no. 13, pp. 4236-4246, 2021.

[11] V. La Gatta, V. Moscato, M. Postiglione, and G. Sperli, "An epidemiological neural network exploiting dynamic graph structured data applied to the COVID-19 outbreak," IEEE Transactions on Big Data, vol. 7, no. 1, pp. 45-55, 2020.

[12] H. Ren et al., "Graph convolutional networks in language and vision: A survey," Knowledge-Based Systems, vol. 251, p. 109250, 2022.

[13] Q. Li, W. Lin, Z. Liu, and A. Prorok, "Message-aware graph attention networks for large-scale multi-robot path planning," IEEE Robotics and Automation Letters, vol. 6, no. 3, pp. 5533-5540, 2021.

[14] L. Ruiz, F. Gama, and A. Ribeiro, "Gated graph recurrent neural networks," IEEE Transactions on Signal Processing, vol. 68, pp. 6303-6318, 2020.

[15] Z. Hou et al., "Graphmae: Self-supervised masked graph autoencoders," in Proceedings of the 28th ACM SIGKDD Conference on Knowledge Discovery and Data Mining, 2022, pp. 594-604.

[16] T. Chen, X. Zhang, M. You, G. Zheng, and S. Lambotharan, "A GNN-based supervised learning framework for resource allocation in wireless IoT networks," IEEE Internet of Things Journal, vol. 9, no. 3, pp. 1712-1724, 2021.

[17] Y.-M. Shin, C. Tran, W.-Y. Shin, and X. Cao, "Edgeless-GNN: Unsupervised Representation Learning for Edgeless Nodes," IEEE Transactions on Emerging Topics in Computing, 2023.

[18] P. Qin, W. Chen, M. Zhang, D. Li, and G. Feng, "CC-GNN: A clustering contrastive learning network for graph semi-supervised learning," IEEE Access, 2024.

[19] M. K. Thota, F. H. Shajin, and P. Rajesh, "Survey on software defect prediction techniques," International Journal of Applied Science and Engineering, vol. 17, no. 4, pp. 331-344, 2020.

[20] A. Azadi and M. Momayez, "Simulating a Weak Rock Mass by a Constitutive Model," Mining, vol. 5, no. 2, p. 23, 2025, doi: https://doi.org/10.3390/mining5020023.

[21] K. Noda, H. Yokoyama, and S. Kikuchi, "Sirius: Static program repair with dependence graph-based systematic edit patterns," in 2021 IEEE International Conference on Software Maintenance and Evolution (ICSME), 2021: IEEE, pp. 437-447.

[22] K. Wang, M. Yan, H. Zhang, and H. Hu, "Unified abstract syntax tree representation learning for cross-language program classification," in Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension, 2022, pp. 390-400.

[23] D. Prestat, N. Moha, R. Villemaire, and F. Avellaneda, "DynAMICS: A tool-based method for the specification and dynamic detection of Android behavioural code smells," IEEE Transactions on Software Engineering, 2024.

[24] A. K. Turzo and A. Bosu, "What makes a code review useful to opendev developers? an empirical investigation," Empirical Software Engineering, vol. 29, no. 1, p. 6, 2024.

[25] A. Bansal, Z. Eberhart, Z. Karas, Y. Huang, and C. McMillan, "Function call graph context encoding for neural source code summarization," IEEE Transactions on Software Engineering, vol. 49, no. 9, pp. 4268-4281, 2023.

[26] M. Zakeri-Nasrabadi, S. Parsa, M. Ramezani, C. Roy, and M. Ekhtiarzadeh, "A systematic literature review on source code similarity measurement and clone detection: Techniques, applications, and challenges," Journal of Systems and Software, p. 111796, 2023.

[27] K. DePalma, I. Miminoshvili, C. Henselder, K. Moss, and E. A. AlOmar, "Exploring ChatGPT’s code refactoring capabilities: An empirical study," Expert Systems with Applications, vol. 249, p. 123602, 2024.

[28] A. Al MohamadSaleh and S. Alzahrani, "Development of a maturity model for software quality assurance practices," Systems, vol. 11, no. 9, p. 464, 2023.

[29] C. Birchler, S. Khatiri, P. Derakhshanfar, S. Panichella, and A. Panichella, "Single and multi-objective test cases prioritization for self-driving cars in virtual environments," ACM Transactions on Software Engineering and Methodology, vol. 32, no. 2, pp. 1-30, 2023.

[30] T. Sharma, A. Jatain, S. Bhaskar, and K. Pabreja, "Ensemble machine learning paradigms in software defect prediction," Procedia Computer Science, vol. 218, pp. 199-209, 2023.

[31] O. B. Sghaier and H. Sahraoui, "A multi-step learning approach to assist code review," in 2023 IEEE International Conference on Software Analysis, Evolution and Reengineering (SANER), 2023: IEEE, pp. 450-460.