GraphPrior: Mutation-based Test Input Prioritization for Graph Neural Networks

XUEQI DANG, University of Luxembourg, Luxembourg
YINGHUA LI*, University of Luxembourg, Luxembourg
MIKE PAPADAKIS, University of Luxembourg, Luxembourg
JACQUES KLEIN, University of Luxembourg, Luxembourg
TEGAWENDÉ F. BISSYANDÉ, University of Luxembourg, Luxembourg
YVES LE TRAON, University of Luxembourg, Luxembourg

Graph Neural Networks (GNNs) have achieved promising performance in a variety of practical applications. Similar to traditional DNNs, GNNs could exhibit incorrect behavior that may lead to severe consequences, and thus testing is necessary and crucial. However, labeling all the test inputs for GNNs can be costly and time-consuming, especially when dealing with large and complex graphs, which seriously affects the efficiency of GNN testing. Existing studies have focused on test prioritization for DNNs, which aims to identify and prioritize fault-revealing tests (i.e., test inputs that are more likely to be misclassified) to detect system bugs earlier in a limited time. Although some DNN prioritization approaches have been demonstrated effective, there is a significant problem when applying them to GNNs: they do not take into account the connections (edges) between GNN test inputs (nodes), which play a significant role in GNN inference. In general, DNN test inputs are independent of each other, while GNN test inputs are usually represented as a graph with complex relationships between each test. In this paper, we propose GraphPrior (**GNN**-oriented Test **Prior**itization), a set of approaches to prioritize test inputs specifically for GNNs via mutation analysis. Inspired by mutation testing in traditional software engineering, in which test suites are evaluated based on the mutants they kill, GraphPrior generates mutated models for GNNs and regards test inputs that kill many mutated models as more likely to be misclassified. Then, GraphPrior leverages the mutation results in two ways, killing-based and feature-based methods. When scoring a test input, the killing-based method considers each mutated model equally important, while feature-based methods learn different importance for each mutated model through ranking models. Finally, GraphPrior ranks all the test inputs based on their scores. We conducted an extensive study based on 604 subjects to evaluate GraphPrior on both natural and adversarial test inputs. The results demonstrate that KMGP, the killing-based GraphPrior approach, outperforms the compared approaches in a majority of cases, with an average improvement of 4.76%~49.60% in terms of APFD. Furthermore, the feature-based GraphPrior approach, RFGP, performs the best among all the GraphPrior approaches. On adversarial test inputs, RFGP outperforms the compared approaches across different adversarial attacks, with the average improvement of 2.95%~46.69%.

CCS Concepts: • **Software and its engineering** → **Software testing and debugging**; • **Computer systems organization** → *Neural networks.*

Additional Key Words and Phrases: Test Input Prioritization, Graph Neural Networks, Mutation, Labelling

*Corresponding author.

Authors' addresses: Xueqi Dang, xueqi.dang@uni.lu, University of Luxembourg, Luxembourg; Yinghua Li, yinghua.li@uni.lu, University of Luxembourg, Luxembourg; Mike PAPADAKIS, michail.papadakis@uni.lu, University of Luxembourg, Luxembourg; Jacques Klein, jacques.klein@uni.lu, University of Luxembourg, Luxembourg; Tegawendé F. Bissyandé, tegawende.bissyande@uni.lu, University of Luxembourg, Luxembourg; Yves LE TRAON, yves.letraon@uni.lu, University of Luxembourg, Luxembourg.

1049-331X/2023/7-ART
https://doi.org/10.1145/3607191

1 INTRODUCTION

In recent years, graph machine learning [27, 38] has been widely adopted for modeling graph-structured data. In this realm, the emergence of graph neural networks (GNNs) [71] has offered promising results in diverse domains, such as recommendation systems [25, 85, 91], social network analysis [47, 84, 93], and drug discovery [4, 73]. GNNs, like typical neural networks [45] [75], are abstractions of the underlying data. Thus, their inference can suffer from faults [28] [53] [58], which can lead to severe prediction failures, especially in security-critical use cases. Testing is considered to be a fundamental practice that is widely adopted to ensure the performance of neural networks, including GNNs. However, like traditional deep neural networks (DNNs), GNN testing also suffers from the lack of automated testing oracles, which necessitates the manual labeling of test inputs. However, this labeling process can require significant human effort, especially for large and complex graphs. Moreover, in certain specialized domains, such as the protein interface prediction [62] of drug discovery, labeling intensively relies on domain-specific knowledge, further increasing its costs.

Prior works [6, 26, 44, 81] have focused on *test prioritization* to relieve the labeling-cost problem for DNNs. Test prioritization approaches aim to prioritize test inputs that are more likely to be misclassified (i.e., fault-revealing test inputs) so that such inputs can be identified earlier to reveal system bugs. Existing approaches are mainly divided into two categories: coverage-based and confidence-based test prioritization approaches. Coverage-based approaches prioritize test inputs based on neuron coverage through adapting coverage-based prioritization methods from traditional software testing [51, 92]. Confidence-based approaches assume that test inputs for which the model is less confident are more likely to be misclassified and thus should be prioritized higher. Feng *et al.* [26] proposed the state-of-the-art confidence-based approach DeepGini, which considers that a test input is more likely to be misclassified by a DNN model if the model outputs similar prediction probabilities for each class. More recently, Wang *et al.* [81] proposed PRIMA, which leveraged mutation analysis and learning-to-rank methods to prioritize test inputs for DNNs. However, despite its effectiveness in DNN test prioritization, PRIMA cannot be directly applied to GNNs since their mutation operators are not adapted to graph-structured data and GNN models.

Furthermore, existing studies [36] have focused on metrics for data selection (e.g., margin and least confidence), which can also be used to detect possibly-misclassified test data. Although the aforementioned approaches have been demonstrated to be effective for DNN models in some cases, they have the following limitations when applied to GNN models:

- First, to the best of our knowledge, current coverage-based approaches do not provide interfaces for GNN models and thus cannot be directly applied. Moreover, existing research [26] has demonstrated that coverage-based approaches are not effective compared to confidence-based approaches.
- Second, despite the effectiveness of confidence-based approaches on traditional DNNs, they do not take into account the interdependencies between test inputs of GNNs, which are particularly crucial for GNN inference. In other words, GNN test inputs are typically represented as graph-structured data consisting of nodes and edges, while confidence-based prioritization approaches usually deal with test sets in which each test is independent and has no connections with others.
- Third, the effectiveness of uncertainty-based metrics can be limited when facing some specific adversarial attacks. If the aim of an attack is to generate test inputs that maximize the probability of incorrect classification, then the utility of uncertainty metrics can be limited. This is because the underlying assumption of uncertainty-based metrics is that: if a model is more uncertain about classifying a test, this test is more likely to be misclassified. However, in such scenarios, even if a model is confident on a test, this test can still have a high probability of being misclassified.

To overcome the aforementioned problems, in this paper, we propose GraphPrior (**GNN**-oriented Test **Prior**itization), a set of test prioritization approaches specifically for GNNs. GraphPrior identifies and prioritizes possibly-misclassified test inputs via mutation analysis. Given a test set for a GNN model, GraphPrior regards a test input that kills more mutated models (i.e., variants of the original GNN model that is slightly changed) of the original GNN model as more likely to be misclassified. Here, killing means the prediction result to the test input via the GNN model and the mutated model is different. To this end, we design a set of mutation rules to generate mutated models specifically for GNNs by slightly changing the training parameters of the original model. After obtaining the mutation results of each test input, GraphPrior introduces several ranking models (ML/DL models) [5, 42, 83] to rank the test set. The working principle of GraphPrior is inspired by mutation testing research as this has been realized for both model-based [1, 18, 63] and code-based [2, 17, 64] testing. The key underlying principle in all cases is that test cases that distinguish the behavior of mutants from that of the original artifact are useful and more likely to detect other underlying faults [1, 9, 63].

While both the GraphPrior and PRIMA (i.e., the state-of-the-art DNN test prioritization approach) use mutation analysis, GraphPrior differs from PRIMA in terms of its mutation rules, feature generation, and ranking models: 1) GraphPrior's mutation rules can directly or indirectly affect the message passing between nodes in graph data. In contrast, the mutation rules of PRIMA are designed for traditional DNNs, where the test inputs are independent, and therefore, the mutation rules do not affect the relationships between tests; 2) GraphPrior generates a mutation feature vector for each test input based on its mutation results, where the i_{th} element in the vector denotes whether the i_{th} mutated model is killed by this input. This feature generation strategy is intuitive and reproducible. In addition to this, the generation method exhibits several other advantages. First, by using binary indicators (1 or 0) as elements of the mutation feature vector, the information is transformed into a concise vector representation. Second, the fine-grained nature of the mutation feature vector allows for a detailed analysis of the effects of individual mutations. In particular, further analysis can be conducted to assess the contributions of each mutated model to GraphPrior. By tracing back to the corresponding mutation rules for the top critical mutated models, we can gain insights into which mutation rules made higher contributions to GraphPrior. The experimental results demonstrate its effectiveness; 3) GraphPrior employs five ranking models and compares their effectiveness in utilizing mutation features for test prioritization, while PRIMA only uses a single ranking model. By comparing multiple ranking models, GraphPrior can identify the optimal ranking model for learning mutation features in test prioritization.

GraphPrior has broad applicability across a wide range of contexts, including software development, scientific research, and financial systems. For instance, GraphPrior can be employed to gain insights into the vulnerabilities of GNN models used in financial transaction fraud detection. In this specific context, where nodes represent accounts and edges represent transaction transfers, the first step is to utilize the GNN model under test to identify a group of potentially fraudulent accounts. Subsequently, these identified accounts serve as test inputs for GraphPrior. By prioritizing accounts that are more likely to be misclassified by the model (i.e., accounts falsely classified as fraudulent), GraphPrior places them at the top of the recommendation list. Consequently, by labelling and analyzing these bug-revealing tests earlier, the fraud analysis team can unveil the bugs and vulnerabilities of the GNN model more efficiently.

It is important to note that, GraphPrior is specifically designed for GNNs, and its impact on DNNs has not been evaluated. This is because in graph datasets, nodes are interconnected, and the mutation rules of GraphPrior can directly or indirectly affect the message passing between nodes in the prediction process. In contrast, in traditional DNNs, each sample in a dataset is typically independent, and as a result, such mutation rules are unlikely to affect the transmission of information between tests. Therefore, the effectiveness of GraphPrior's mutation rules for DNNs remains uncertain, as no related experiments have been conducted to evaluate it.

We conducted an extensive study to evaluate the performance of GraphPrior based on 604 subjects. Here, a subject refers to a pair of graph dataset and GNN model. We compare GraphPrior with 6 uncertainty-based metrics [82] [26] [80] that can be used to prioritize possibly-misclassified test inputs and adopt random selection as the baseline method. Our experimental results demonstrate that GraphPrior performs well across all subjects and outperforms the compared approaches on average.

As mentioned before, one essential problem of confidence-based approaches is that adversarial attacks may lead to a model being more confident in the incorrect prediction, resulting in the failure of the approach. Therefore, we also evaluate GraphPrior on test inputs generated from graph adversarial attacks of existing studies [3, 48, 86, 100]. Furthermore, since the effectiveness of test prioritization methods may vary depending on the degree of the adversarial attack, we set different attack levels to generate adversarial data and compared GraphPrior with the compared approaches. In addition to the evaluation of GraphPrior, we compare the effectiveness of different mutation rules in generating top contributing mutated models, aiming to identify which mutated rules contribute more to each GNN model. In the last step, we investigate whether GraphPrior and the uncertainty-based metrics can select informative retraining tests to improve a GNN model. Our experimental results demonstrate that GraphPrior achieved better effectiveness compared with the uncertainty-based test prioritization methods. We publish our dataset, results, and tools to the community on Github[1].

Our work has the following major contributions:

- **Approach.** We propose GraphPrior, a set of mutation-based test prioritization approaches for GNNs. To this end, we design a set of mutation rules that mutate GNN models by slightly changing their training parameters. We carefully select ranking models to analyze the mutation results for effective test prioritization.
- **Study** We conduct an extensive study based on 604 GNN subjects involving natural and adversarial test sets. We compare GraphPrior with existing DNN approaches that could detect possibly misclassified test inputs. Our experimental results demonstrate the effectiveness of GraphPrior.
- **Mutation rule analysis** We compare the effectiveness of the GNN mutation rules in generating top contributing mutated models, observing that the mutation rule HC (i.e., mutating Hidden Channels) makes top contributions to most GNN models in test input prioritization.

2 BACKGROUND

In this section, we introduce the key domain concepts for our work, including Graph Neural Networks and Test Input Prioritization for DNNs.

2.1 Graph Neural Networks

Graph neural networks (GNNs) have achieved great success in handling machine learning problems on graph-structured data [98] [25] [76]. Unlike traditional neural networks running on fixed-sized vectors, GNNs deal with graphs of varying sizes and structures. Therefore, GNNs can capture complex relationships between data points and make more accurate predictions. GNNs have been used in a wide range of tasks, including recommendation system [25, 85, 90], protein-protein interaction (PPI) prediction [40, 62, 97] and traffic forecasting [10, 41, 95].
Graphs A graph is a data structure consisting of two components: nodes (vertices) and edges. A graph H can be defined as $H = (V, E)$, where V is the set of nodes, and E are the edges between them. In a graph, nodes can represent entities (e.g., persons, places, or things), while the edges define the relationships between nodes. The edges can be either directed or undirected based on the directional dependencies that exist between nodes. Graphs can be utilized to model complex systems such as social media networks, molecular structures, and citation networks. For example, in the context of citation networks, publications can be represented as nodes, and the citations between them can be represented as edges. Graph datasets are collections of graph data that can be used to train and evaluate GNNs. Some benchmark graph datasets [79] include Cora, CiteSeer, and PubMed.

[1] https://github.com/yinghuali/GraphPrior

In this paper, we evaluated GraphPrior and the compared approaches on several graph datasets obtained from existing studies [88] [70].

Graph Embeddings Graph embedding [7] is an approach used to transform nodes, edges, and their associated features into lower dimensional representation while maximally preserving the graph structural information and graph properties. Graph analytics methods usually suffer from high computational and storage costs, limiting their applicability in real-world scenarios. The use of graph embedding has shown promising results as an efficient and effective way to address the graph analytics problem.

Message Passing Scheme In GNNs, the message-passing scheme is commonly employed [29], whereby nodes aggregate and transform the information from their neighbors in each layer. Through stacking multiple GNN layers, this mechanism facilitates the propagation of information across the entire graph structure, allowing for the effective embedding of nodes into low-dimensional representations. These node representations may subsequently be leveraged by a differentiable prediction layer, thereby enabling end-to-end training of the complete model.

GNN models A graph neural network (GNN) model is a type of neural network designed to operate on graph data structures. Typically, a GNN model contains two crucial parts: a graph convolution layer [45] to capture the relationship between nodes in the graph and a classifier [87] to make predictions based on the captured relationship. In general, a GNN model takes graph-structured data as inputs and produces outputs based on its corresponding task. For example, the output for a GNN model that deals with node-level tasks (i.e., GNN tasks that are concerned with predicting the identity or role of each node within a graph) is typically a prediction for nodes in the input graph. In this paper, we evaluated our proposed test prioritization approach, GraphPrior, and the compared approaches on various GNN models [21, 30, 45, 79] that deal with node classification tasks.

Graph Adversarial Attacks Graph adversarial attacks [3] [16] [77] [99] involve the manipulation of graph structure or node features to generate graph adversarial perturbations that can fool GNN models. This vulnerability of GNNs has raised serious concerns regarding their reliability and safety, particularly in safety-critical applications such as financial systems and risk management. For instance, in a credit scoring system, attackers can exploit the vulnerability of GNNs to create fake connections with high-credit customers to evade fraud detection models. In this paper, we applied eight graph adversarial attacks from existing studies [3, 48, 86, 100] to generate adversarial inputs for the evaluation of GraphPrior.

2.2 Test Input Prioritization for DNNs

In DNN testing, test input prioritization aims to prioritize tests that are more likely to be misclassified (i.e., bug-revealing test inputs) by the DNN model. In this way, more important test inputs can be labeled earlier in a limited time, which can improve the efficiency of DNN testing. In the literature, several prioritization approaches have been proposed to deal with the labeling-cost issues [6, 26, 81, 94].

The majority of approaches for prioritizing tests in Deep Neural Networks (DNNs) can be classified into two categories, coverage-based and confidence-based [81]. Confidence-based approaches, such as DeepGini [26], prioritize test inputs based on the model's confidence. Specifically, these methods identify inputs that are likely to be incorrectly predicted by the DNN model, given that the model outputs similar probabilities for each class. In contrast, coverage-based approaches, such as CTM [92], simply extend traditional software system testing methods to DNN testing, and have been shown to underperform compared to confidence-based approaches [26]. Weiss *et al.* [82] conducted a comprehensive investigation of the capabilities of various DNN test input prioritization techniques, including some notable uncertainty-based metrics such as Vanilla Softmax, Prediction-Confidence Score (PCS), and Entropy. The Vanilla Softmax metric is calculated as the highest activation in the output softmax layer for a classification problem, subtracted from 1. PCS, on the other hand, is defined as the difference in softmax likelihood between the predicted class and the second runner-up class. Additionally, Entropy is considered as an alternative metric in the softmax layer proposed by the authors of DeepGini. These metrics have been demonstrated to be effective in identifying possibly-misclassified test inputs, and can aid in guiding test prioritization efforts.

The aforementioned uncertainty-based test prioritization can be adapted for test input prioritization for GNNs. GraphPrior differs from these approaches in that GraphPrior leverages mutation analysis to perform test prioritization. The mutation analysis of GraphPrior exploits the specific properties of GNNs. Specifically, GraphPrior's mutation rules can directly or indirectly affect the message passing between nodes in a graph. In contrast, uncertainty-based approaches rely on the prediction uncertainty of the DNN model to prioritize test inputs without accounting for the interdependence between nodes.

Currently, the state-of-the-art technique for DNN test prioritization is PRIMA, which prioritizes fault-revealing test inputs based on mutation analysis. However, PRIMA is not suitable for GNN test prioritization because: 1) its input mutation rules are specifically designed for DNN testing datasets where each sample is independent of each other. In contrast, graph datasets have complex interdependence between nodes, making PRIMA unsuitable for test prioritization in this context; 2) GNNs employ graph operations and message passing mechanisms to aggregate and update information from neighboring nodes, thereby facilitating improved representation and learning within graph structures. The model mutation rules employed in PRIMA are not suitable for accommodating the graph operation mechanisms intrinsic to GNNs.

In addition to the aforementioned test prioritization techniques, several active learning [80] methods can also be adapted to prioritize DNN tests, such as Least Confidence and Margin. Active learning aim to selects the most informative samples to be labeled by a human expert. When applied to test prioritization, active learning can be used to identify the most critical and informative test cases that can reveal bugs in the system.

3 APPROACH

3.1 Overview

In this paper, we propose GraphPrior, a set of test prioritization approaches for GNNs to prioritize test inputs. GraphPrior consists of six mutation-based test prioritization approaches: KMGP, LRGP, RFGP, LGGP, DNGP and XGGP. These approaches are discussed later in this Section. We present the overview of GraphPrior in Figure 1, in which the input of GraphPrior is a GNN test set, and the output is the test set that has been prioritized. Given a test set T for a GNN model G, the implementation process of GraphPrior is presented as follows.

Generating mutants for the GNN model G First, GraphPrior generates mutated models (i.e., mutants) for the GNN model G based on carefully designed mutation rules (cf. Section 3.2).

Obtaining mutation results through killing mutants For each test input, GraphPrior identifies which mutated models it kills. Here, a mutated model is killed by a test input if the prediction results of this input via the mutated model and the original model G are different. In this way, GraphPrior obtains the mutation result of each test input.

Generating feature vectors from the mutation results For each test input, GraphPrior generates a mutation feature vector for it based on its mutation results. The i_{th} element of this feature vector denotes whether this input kills the i_{th} mutated model. More specifically, given a test input $t \in T$, if t kills a mutated model M_i, then the i_{th} element of t's mutation feature vector is set to 1. Otherwise, the i_{th} element is set to 0.

Ranking test input based on mutation feature vectors via ranking models GraphPrior utilizes ranking models [5, 42, 83] to calculate a misclassification score for each test input based on its feature vector. This score can indicate how likely a test input will be misclassified by the GNN model. Finally, GraphPrior ranks them based on their misclassification scores in descending order and outputs the prioritized test set T'.

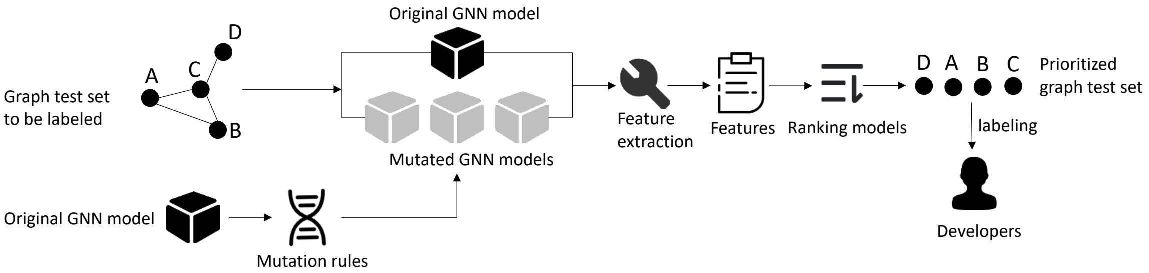


Fig. 1. Overview of GraphPrior

3.2 Mutation Rules

In GraphPrior, mutation rules are employed to generate mutated models of a GNN model by making slight changes to its training parameters. We select the following parameters because they can impact the message passing in the GNN prediction process. More specifically, in the mutated GNN model, the manner in which nodes acquire information from their neighboring nodes is slightly different from that of the original GNN model. Although variations of GNNs can be obtained even without changing training parameters, the resulting model mutants cannot produce meaningful differences in the GNN model's behavior. By changing the selected training parameters to generate mutants, we can intentionally introduce meaningful modifications to the model's behavior in terms of the interdependencies between nodes during the prediction process. We present all the mutation rules of GraphPrior as follows.

- **Self Loops (SL)** [45, 79] SL is a Boolean parameter, which controls whether to add self-loops to the input graph. When the SL parameter is set to True, self-loops are introduced to each node in the graph. By incorporating self-loops, the inherent information of nodes can be effectively aggregated into their representation vectors, leading to a change in the weighting of their neighboring nodes, and thus affecting the interdependence of nodes in the prediction process.
- **Bias (BIA)** [30, 45, 79] BIA is a Boolean parameter, which determines whether to introduce a predetermined offset to the representation vectors of nodes. When the BIA parameter is enabled (set to True), each node will be assigned a corresponding bias parameter to its representation vector, allowing the GNN model to better capture the inherent properties of the graph and improve the interdependence between nodes in the prediction process.
- **Cached (CA)** [45] CA is a Boolean parameter that controls whether to cache the computation of node embeddings during the forward pass. When the CA parameter is set to True, the node embeddings are cached and reused during the backward pass to save computation time. Caching the computation of node embeddings can affect the interdependence between nodes by altering the order and efficiency of message passing.
- **Improved (IMP)** [45] IMP is a Boolean parameter that controls whether to use the improved message passing strategy, thus affecting the interdependence between nodes in the prediction process.
- **Normalize (NOR)** [21, 30] NOR is a Boolean parameter, which determines whether to normalize the messages passed between nodes in the prediction process. When this parameter is set to "True," the messages are normalized by the number of neighbors that a node has before being passed to the next layer. This normalization can impact the contribution of each neighbor to the node's final representation, thus affecting the message passing between nodes in the prediction process.
- **Concat (CON)** [79] CON is a Boolean parameter, which controls how the representations of neighboring nodes are combined during message passing. When it is set to True, the representations of neighboring nodes are concatenated before being passed, resulting in a more expressive representation of the nodes, enabling the GNN to capture more nuanced interdependencies between them.

- **Heads (HDS)** [79] HDS is an integer parameter that determines the number of attention heads used in multi-head attention. Increasing the number of heads allows the model to capture more complex interdependence among nodes in the graph. Each attention head can focus on a different aspect of the node neighborhood, enabling the model to learn different representations of the graph.
- **Epoch (EP)** [21, 30, 79] EP is an integer parameter that controls the number of times a GNN model iterates over the training dataset. By increasing the number of epochs, a GNN model can better capture the interdependence between nodes for model inference.
- **Hidden Channel (HC)** [21, 30, 45, 79] HC is an integer parameter, which controls the dimensionality of the hidden representation in each layer of the GNN. Therefore, changing this parameter can impact the interdependence between nodes in a graph by enabling the GNN to learn more expressive node embeddings.
- **Negative Slope (NS)** [79] NP is a float parameter, which controls the slope of the negative part of the activation function used in the Gated Linear Unit (GLU) operation. GLU is a common non-linear function used in GNNs for message passing. Specifically, the GLU operation is used to combine the node features with the weighted sum of their neighboring nodes' features, which is the message passed between nodes in the graph. The negative slope parameter determines the slope of the activation function for negative input values in the GLU operation, thus impacting the message passing between nodes.

Based on the above mutation rules, for a given test set and a GNN model, GraphPrior generates N mutated models of the original model. We consider that a test input kills a mutated model if the predictions for this input via the mutated models and the original GNN model are different. Based on it, GraphPrior obtains the mutation results of all the test inputs.

Considering that the primary objective of generating mutated models is to obtain informative features for test prioritization, a statistical analysis is employed to validate their effectiveness. To achieve this, a series of repeated experiments are conducted, as outlined in Section 5. The results of these experiments demonstrate that GraphPrior's effectiveness is statistically significant, thereby confirming the statistical validity of the generated mutated models for the purpose of test prioritization.

3.3 Killing-based GraphPrior

This section presents the workflow of KMGP, the **K**illing **M**utants-based **G**NN Test **P**rioritization approach. Notably, KMGP operates on a "killing-based" principle, where test inputs that can kill more mutated models are considered as more likely to be misclassified and will be prioritized higher. It is worth noting that KMGP assigns equal importance to each mutated model in the process of test prioritization, a distinct feature that distinguishes it from feature-based approaches, which will be elaborated upon in subsequent sections. Given a GNN model G, and a test input set $T = \{t_1, t_2, \ldots, t_n\}$, the detailed execution of KMGP can be divided into three key stages: mutation generation, killing-based mutation analysis, and test prioritization.

Mutation generation In the mutation generation stage, a group of mutated models $\{G'_1, G'_2, \ldots, G'_N\}$ are generated for the original GNN model G.

Killing-based mutation analysis This stage involves obtaining the mutation results of each test input $t \in T$ using the process outlined in Section 3.2. Subsequently, KMGP counts the number of mutants killed by each test input based on their mutation results.

Test prioritization In the third stage, KMGP prioritizes all the test inputs in T based on the number of mutated models they killed, with those that kill more mutants being prioritized higher in the test sequence.

3.4 Feature-based GraphPrior

In comparison to the killing-based GraphPrior approach, the feature-based approaches are characterized by automatic mutation feature analysis. This process involves the generation of mutated feature vectors based on the execution of mutated models, followed by the use of ranking models (ML/DL models), which assign different importance to each mutated model for test prioritization.

Overall, the feature-based approaches' workflow entails three key stages: mutated model generation, mutation feature generation, and learning-to-rank.

❶ **Mutated model generation** Given a GNN model G and a test set T, during the first stage, the feature-based approaches generate a group of mutated models (denoted as $\{G_1', G_2', \ldots, G_N'\}$) of the GNN model G based on the mutation rules specified in Section 3.2.

❷ **Mutation feature generation** Subsequently, the feature-based approaches associate a feature vector V_t of size N with each test input t, where N represents the number of mutated models, and $v_k (= V_t[k])$ maps to the execution output for the mutated model G_k'. If t kills the mutated model G_k' (i.e., the prediction results for t via the mutated models G_k' and the original model G are different), v_k is set to 1. Otherwise, it is set to 0.

❸ **Learning-to-rank** In the final stage, the feature-based approaches input the mutation features of each test input to the ranking model (ML/DL models) [5] [15] [42] [78] [83]. The ranking models can automatically learn different importance for each mutation feature to output misclassification scores. Here, each mutation feature corresponds to the execution result of a mutated model so that we can consider that the ranking models learn the importance of each mutated model for test prioritization. Finally, the feature-based approaches rank all the test inputs based on their misclassification scores in descending order.

In our study, we propose five feature-based GraphPrior approaches, which follow the similar workflow described above, but leverage different ranking models. These five approaches are XGGP (**XG**Boost-based **G**NN Test **P**rioritization), LRGP (**L**ogistic **R**egression-based **G**NN Test **P**rioritization), LGGP (**L**ight**G**BM-based **G**NN Test **P**rioritization), RFGP (**R**andom **F**orest-based **G**NN Test **P**rioritization) and DNGP (**DN**N-based **G**NN Test **P**rioritization). We briefly introduce the basic principle of the ranking models of these approaches as follows.

1) XGGP leverages the XGBoost algorithm [15] as the ranking model. XGBoost is a highly effective gradient boosting algorithm that combines decision trees to enhance the accuracy of predictions. XGGP utilizes the XGBoost algorithm to predict the misclassification score for a given test input based on its mutation features. This score reflects the likelihood that the input will be misclassified by a GNN model.

2) LRGP leverages the Logistic Regression algorithm [83] as the ranking model. Logistic regression leverages a logistic function to model the association between a categorical dependent variable and one or more independent variables.

3) LGGP leverages the LightGBM algorithm [42] as the ranking model. LightGBM is a gradient boosting framework that employs tree-based learning algorithms. The fundamental principle of LightGBM is similar to XGBoost, which employs decision trees based on learning algorithms. However, LightGBM introduces a novel optimization in the framework, with a primary focus on enhancing the speed of model training.

4) RFGP leverages the random forest algorithm [5] as the ranking model. Random Forest is an ensemble learning algorithm that constructs multiple decision trees using random subsets of the training data and input features. The predictions from individual trees are combined to produce the final prediction using averaging or voting.

5) DNGP leverages a DNN model [78] as the ranking model. The DNN model can learn to rank test inputs based on their mutation features. After training, the DNN model can generate a score that reflects their misclassification probability. This score can then be used to rank test inputs in a test set.

Compared to the mutation features of PRIMA, the distinctive aspect of GraphPrior's mutation features lies in their utilized mutation rules, which are specifically designed for GNNs. These mutation rules have the potential to directly or indirectly impact the message passing mechanism between nodes in graph data. Our experiment results in Section 5 demonstrate the effectiveness of the feature-based GraphPrior approaches. The observed effectiveness can be attributed, in part, to the selection of mutation rules and ranking models. Specifically, our mutation rules have been designed to generate informative mutation features by changing the massage passing between nodes in the GNN prediction process. Furthermore, our ranking models are able to utilize these mutation features for test prioritization effectively. After sufficient training, ranking models can output a misclassification score that indicates how likely a sample would be misclassified based on its mutation features. A score closer to 1 indicates a higher probability of misclassification. By sorting the misclassification scores of test inputs in descending order, the feature-based GraphPrior approaches can effectively prioritize tests that are more likely to be misclassified.

3.5 Usage of GraphPrior

By utilizing ranking models, GraphPrior predicts a misclassification score for each test input within a given test set. These predicted scores are then utilized for test prioritization, whereby test inputs with higher scores are prioritized higher. Particularly, the ranking models are pre-trained before the execution of GraphPrior. The training process is standardized across all the different ranking models and follows a consistent set of procedures, which are presented in detail below.

❶ **Splitting datasets** Given a GNN model G with dataset T. First, we split the dataset T into two partitions: the training set R and the test set, in a 7:3 ratio [61]. The test set remains untouched for the purpose of evaluating GraphPrior.

❷ **Constructing the training set for ranking models** Based on the training set R, we aim to build a training set R' for training the ranking models. First, we generate a group of mutated models for each input $r_i \in R$. Then, we obtain the mutation feature vector V_i of r_i (i.e., a one-dimensional vector in which the i_{th} element denotes whether the i_{th} mutated model is killed by this input). The mutation feature vector of r_i is used to build the training set R' (i.e., the training set of the ranking models). Second, we let the original GNN model G classify each input $r_i \in R$ and compare it with the ground truth of r_i. In this way, we can identify whether r_i is misclassified by the GNN model G. If r_i is misclassified by G, we label it as 1. Otherwise, we label it as 0. In this way, we have built the ranking model training set R'.

❸ **Training ranking models** Based on R', we train the ranking models. Upon the completion of the training process, the ranking model is capable of receiving the mutation feature vector of a test input as an input and producing a misclassification score as an output. This score serves as an indicator of the probability of the test input being incorrectly classified by the GNN model.

It is worth noting that the original labels of the training set R' are binary (i.e., 1 or 0), but the ranking models that are well trained can output values (i.e., the misclassification scores). To achieve this, we make some adaptations to implement the adopted ranking algorithms (e.g., random forest and XGBoost). First, although the ranking algorithms we adopted initially deal with classification tasks, an intermediate value is calculated for the classifications. For example, if the intermediate value exceeds 0.5 (default value which can be adjusted), input will be classified into the first category; otherwise, the other category. Here, after training, we let the ranking models directly output the intermediate value, as this value can indicate the likelihood of a test input being misclassified by the GNN model, where a higher value implies a greater likelihood of misclassification. We call this intermediate value "misclassification scores" and leverage the scores of test inputs to rank them.

4 STUDY DESIGN

4.1 Research Questions

Our experimental evaluation answers the research questions below.

- **RQ1: How does the killing-based GraphPrior approach perform in prioritizing test inputs for GNNs?** In terms of test prioritization for GNNs, existing prioritization approaches usually do not take into account the interdependencies between nodes (tests) in a graph (test set). To fill the gap, we propose GraphPrior, which contains six GNN-oriented test prioritization approaches. Among them, KMGP is a killing-based approach, which regards a test input that kills more mutants as more likely to be misclassified. In this research question, we evaluate the effectiveness of the killing-based KMGP by comparing it with existing approaches that have been demonstrated as effective in detecting possibly-misclassified test inputs.

- **RQ2: How do the feature-based GraphPrior approaches perform in GNN test prioritization?** In addition to the killing-based KMGP, GraphPrior involves five feature-based approaches. The core difference is that, the killing-based approach regards the importance of each mutated model as equal, while the feature-based approaches learn different importance for each mutated model for test prioritization. More specifically, feature-based approaches extract features from mutation results and adopt ranking models [5, 42, 83] to utilize the mutation features for test prioritization. In this research question, we compare the effectiveness of killing-based and feature-based approaches to investigate the effect of ranking models in leveraging mutation results.
- **RQ3: How does GraphPrior perform on test inputs generated from graph adversarial attacks?** When faced with graph adversarial attacks, confidence-based test prioritization approaches may be fooled, thus becoming more confident in incorrect predictions. Therefore, we evaluate to what extent the effectiveness of GraphPrior is affected by graph adversarial attacks. We compare GraphPrior and confidence-based approaches [26, 36] on test inputs generated from graph adversarial attacks of existing studies [3, 48, 86, 100] to demonstrate its effectiveness.
- **RQ4: How does GraphPrior perform against different levels of graph adversarial attacks?** In this research question, we investigate the effectiveness of GraphPrior against different levels of graph adversarial attacks. To answer this research question, we set different levels of attacks to generate test inputs and compare GraphPrior with existing approaches to demonstrate its effectiveness.
- **RQ5: Which mutation rules generate more top contributing GNN mutants?** We investigate the contributions of each mutation rule in generating effective mutants of GNNs. For each GNN model, we select the top contributing mutation features to it through the XGBoost ranking algorithm [15], which is an optimized ML algorithm for ranking tasks based on the implementation of gradient boosting. We match each selected feature with the corresponding GNN mutant and identify the mutation rule that generates it. In this way, we obtain which mutation rules generate more top contributing mutants for test prioritization.
- **RQ6: Can GraphPrior and the uncertainty-based metrics be used in active learning scenarios to improve a GNN model by retraining?** In the face of a large number of unlabeled inputs and a limited time budget, it is not feasible to manually label all the inputs and use them to retrain a GNN. One established solution to reduce data labeling costs is active learning [67], which involves selecting informative subsets of training samples to improve the model performance. In this research question, we investigate the effectiveness of GraphPrior and the uncertainty-based metrics in selecting informative retraining inputs to improve the quality of a GNN model.

4.2 GNN models and Datasets

In our study, we totally adopt 604 subjects to evaluate the effectiveness of GraphPrior and the compared approaches [26, 36]. Table 1 exhibits their basic information. Among the 604 subjects considered in this study, 16 subjects were utilized in the experiments of RQ1, 16 subjects in RQ2, 108 subjects in RQ3, 432 subjects in RQ4, 16 subjects in RQ5 and 16 subjects in RQ6. It is worth noting that, among these subjects, a total of 64 subjects (which were utilized in RQ1, RQ5, and RQ6) were associated with clean datasets, while the remaining 540 subjects (which were utilized in RQ3 and RQ4) were associated with adversarial datasets.

Our study involves four GNN models: GCN (Graph Convolutional Networks) [45], GAT (Graph Attention Networks) [79], GraphSAGE (Graph SAmple and aggreGatE) [30] and TAGCN (Topology Adaptive Graph Convolutional Network) [21], tested by four datasets, namely the Cora [88], CiteSeer [88], PubMed [88] and LastFM [70]. We present their descriptions as follows.

Table 1. GNN models and datasets

ID	Dataset	#Nodes	#Edges	Model	Type
1	CiteSeer	3327	4732	GCN	Original, DICE, MMA, PGD, RAA, RAF, RAR
2	CiteSeer	3327	4732	GAT	Original, DICE, MMA, PGD, RAA, RAF, RAR
3	CiteSeer	3327	4732	TAGCN	Original, DICE, MMA, PGD, RAA, RAF, RAR
4	CiteSeer	3327	4732	GraphSAGE	Original, DICE, MMA, PGD, RAA, RAF, RAR
5	Cora	2708	5429	GCN	Original, DICE, MMA, PGD, RAA, RAF, RAR, NEAR, NEAA
6	Cora	2708	5429	GAT	Original, DICE, MMA, PGD, RAA, RAF, RAR, NEAR, NEAA
7	Cora	2708	5429	TAGCN	Original, DICE, MMA, PGD, RAA, RAF, RAR, NEAR, NEAA
8	Cora	2708	5429	GraphSAGE	Original, DICE, MMA, PGD, RAA, RAF, RAR, NEAR, NEAA
9	LastFM	7624	27806	GCN	Original, DICE, PGD, RAA, RAF, RAR, NEAR, NEAA
10	LastFM	7624	27806	GAT	Original, DICE, PGD, RAA, RAF, RAR, NEAR, NEAA
11	LastFM	7624	27806	TAGCN	Original, DICE, PGD, RAA, RAF, RAR, NEAR, NEAA
12	LastFM	7624	27806	GraphSAGE	Original, DICE, PGD, RAA, RAF, RAR, NEAR, NEAA
13	PubMed	19717	44338	GCN	Original, DICE, RAA, RAF, RAR, NEAR, NEAA
14	PubMed	19717	44338	GAT	Original, DICE, RAA, RAF, RAR, NEAR, NEAA
15	PubMed	19717	44338	TAGCN	Original, DICE, RAA, RAF, RAR, NEAR, NEAA
16	PubMed	19717	44338	GraphSAGE	Original, DICE, RAA, RAF, RAR, NEAR, NEAA

4.2.1 GNN Models .

- **GCN** [45] GCN is a class of convolutional neural networks that can work directly on the graph. It solves the problem of classifying nodes (such as documents) in graphs (such as citation networks), of which only a small number of nodes are labeled. The core idea of GCN is to use the edge information of a graph to aggregate node information to generate new node representations. GCN has been used in several existing studies [31, 35, 89].
- **GAT** [79] GAT introduces a self-attention mechanism in the propagation process. Compared to GCN, which regards all neighbors of a node equally, the attention mechanism assigns different attention scores to each neighbor, thereby identifying more important neighbors.
- **GraphSAGE** [30] GraphSAGE is a generalized inductive framework that generates node embeddings by sampling and aggregating features of neighbor nodes.
- **TAGCN** [21] TAGCN introduces a systematic approach to design a set of fixed-size learnable filters to perform convolutions on graphs. These filters are topology-fit to the topology of the graph as they scan the graph for convolution.

4.2.2 Datasets .

- **Cora** [88] The Cora dataset is a citation graph composed of 2,708 scientific publications (nodes) and 5,429 links (edges) between them. Nodes represent ML papers, and edges represent citations between pairs of papers. Each paper is classified into one of seven classes, such as reinforcement learning and neural networks.
- **CiteSeer** [88] The CiteSeer dataset consists of 3,327 scientific publications (nodes) and 4,732 links (edges). Each paper belongs to one of six categories such as AI and ML.
- **PubMed** [88] The PubMed dataset contains 19,717 diabetes-related scientific publications (nodes) and 44,338 links (edges). Publications are classified into three classes such as Cancer and AIDS (i.e., Acquired Immune Deficiency Syndrome).
- **LastFM Asia Social Network** [70] The dataset LastFM Asia Social Network was collected from the social network of users on the Last.fm music platform in Asia. Nodes are LastFM users, and edges are mutual follower relationships between them. LastFM contains 7,624 nodes and 27,806 edges. The classification task of the LastFM dataset is to predict the home country of a user (e.g., Philippines, Malaysia, Singapore).

Notably, we evaluate GraphPrior on different types of test inputs (i.e., both natural test inputs and adversarial test inputs. We adopted eight graph adversarial attacks, presented in Section 4.4.

4.3 Compared Approaches

In our study, we considered 7 compared approaches in total, including one baseline (i.e., random selection), four DNN test prioritization approaches and two active learning approaches. We select these approaches due to the following reasons: 1) These approaches can be adapted for GNN test prioritization; 2) The selected approaches have been demonstrated as effective for DNNs in existing studies [26] [36] [82]; 3) The implementations of these approaches have been released by the authors.

- **DeepGini** DeepGini [26] prioritizes test inputs based on model confidence. DeepGini leverages the Gini coefficient to measure the likelihood of a test input being misclassified. DeepGini leverages Formula 1 to calculate the ranking scores.
$$\xi(x) = 1 - \sum_{i=1}^{N} (p_i(x))^2 \tag{1}$$
where $\xi(x)$ refers to the likelihood of the test input x being misclassified. $p_i(x)$ refers to the probability that the test input x is predicted to be label i. N refers to the number of labels.
- **Margin** Margin [80] regards a test input with less difference between the top two most confidence predictions as more likely to be misclassified. Margin score is calculated by Formula 2.
$$M(x) = p_k(x) - p_j(x) \tag{2}$$
where $M(x)$ refers to the margin score. $p_k(x)$ refers to the most confident prediction probability. $p_j(x)$ refers to the second most confident prediction probability.
- **Least Confidence** Least Confidence [80] regards test inputs for which the model has the least confidence as more likely to be misclassified. Least confidence is calculated by Formula 3.
$$L(x) = \max_{i=1:n} p_i(x) \tag{3}$$
where $L(x)$ refers to the confidence score. $p_i(x)$ refers to the probability that the test input x is predicted to be label i via a model M.
- **Vanilla Softmax** Vanilla Softmax [82] is computed by subtracting the highest activation probability in the output softmax layer from 1, resulting in a metric that is positively correlated with the misclassification probability. Formula 4 presents the calculation of the Vanilla Softmax metric.
$$\mathrm{V}(x) = 1 - \max_{c=1}^{C} l_c(x) \tag{4}$$
where $l_c(x)$ belongs to a valid softmax array in which all values are between 0 and 1, and their sum is 1.
- **Prediction-Confidence Score (PCS)** PCS [82] calculates the difference between the predicted class and the second most confident class in softmax likelihood.
- **Entropy** Entropy [82] calculates the entropy of the softmax likelihood.
- **Random selection** [22] In random selection, the execution order of the test inputs is determined randomly.

4.4 Graph Adversarial Attacks

In RQ3 and RQ4, we evaluate the effectiveness of GraphPrior on test inputs generated through diverse graph adversarial attacks, in which attackers aim to generate graph adversarial perturbations by manipulating the graph structure or node features to fool the GNN models. We introduce all the attacks we applied in our experiments as follows.

- **Disconnect Internally, Connect Externally (DICE)** [100] The DICE attack is a type of white-box attack whereby the adversary has access to all information about the targeted GNN model, including its parameters, training data, labels, and predictions. Specifically, the DICE attack randomly adds edges between nodes with different labels or removes edges between nodes sharing the same label. Through this, the attack can generate adversarial perturbations that can fool the targeted GNN model.

- **PGD attack** [86] The PGD attack leverages the Projected Gradient Descent (PGD) algorithm to search for optimal structural perturbations to attack GNNs.
- **Min-max attack (MMA)** [86] The min-max attack is a type of untargeted white-box GNN attack. The attack problem is formulated as a min-max problem, where the inner maximization is designed to update the model's parameters (θ) by maximizing the attack loss, and it can be solved using gradient ascent. On the other hand, the outer minimization can be achieved by using Projected Gradient Descent (PGD) [59].
- **Node Embedding Attack-Add (NEAA)** [3] In node embedding attack-add, the attackers are capable of modifying the original graph structure by adding new edges while adhering to a predefined budget constraint.
- **Node Embedding Attack-Remove (NEAR)** [3] In node embedding attack-remove, the attackers modify the original graph structure by removing edges.
- **Random Attack-Add (RAA)** [48] The Random Attack-Add approach randomly adds edges to the input graph to fool the targeted GNN model.
- **Random Attack-Flip (RAF)** [48] The Random Attack-Flip approach randomly flips edges to the input graph to fool the targeted GNN model.
- **Random Attack-Remove (RAR)** [48] The Random Attack-Add approach randomly removes edges to the input graph to fool the targeted GNN model.

4.5 Evaluation of mutation rules (RQ5)

In RQ5, we investigated the contribution of different mutation rules in generating top contributing mutated models. First, for each GNN model, we utilize the cover metric in XGBoost [15] to evaluate the importance of its mutation features and rank them according to the descending order of the importance scores. The cover metric can evaluate the importance of mutation features by quantifying the average coverage of each instance by the leaf nodes in a decision tree. Specifically, it calculates the number of times a particular feature is used to split the data across all trees in the ensemble and then sums up the coverage values for each feature over all trees. This coverage value is then normalized by the total number of instances to obtain the average coverage of each instance by the leaf nodes. The importance of a feature is then calculated based on its coverage value, and features with higher coverage values are considered more important.

Upon obtaining the importance of each mutation feature, which corresponds to a specific mutated model, we proceed to match and determine the importance of the respective mutated models. Subsequently, we select the top N critical mutated models and identify the specific mutated rules employed in their generation. This enables a comparative analysis of the contributions of various mutation rules.

4.6 Implementation and Configuration

We implemented GraphPrior in Python based on the PyTorch 1.11.0 framework [65]. We also integrate the available implementations of the compared approaches [26, 57, 80, 82] into our experimental pipeline to adapt to the GNN prioritization problem. Regarding our mutation rules, we set the number of mutated models as 80~240 across different subjects. Balancing the trade-off between execution time and the effectiveness of GraphPrior is a critical consideration in determining the number of mutants. Building on relevant literature [81], we identified a suitable range of mutants. Our preliminary investigations on multiple subjects demonstrate that these settings effectively maintain the effectiveness of GraphPrior while controlling the runtime within a reasonable range. In the case of subjects associated with longer mutant generation times, we choose to generate a comparatively smaller number of mutants compared to other subjects. Additionally, the range was achieved through the full execution of all pre-defined mutation rules. It is worth noting that the total number of mutation rules was predetermined and fixed. Thus, even with the addition of new mutants, the impact on the performancethe trade-off between excessive computational time and the preservation of method effectiveness of GraphPrior is minor, as the new mutants are created based on the existing mutation rules.

With regard to the specific mutation rules that change the integer/float training parameters, we define a parameter range close to the original parameter values, in order to achieve slight mutations. We conducted a preliminary study using multiple subjects, demonstrating the effectiveness of such settings. Moreover, to obtain parameter values from the specified range, we adopt uniform sampling [56] as the sampling methodology. This technique ensures an equitable probability of selecting each value within the parameter range and has been widely adopted across the ML testing field [56, 60, 96].

More specifically, we set the hidden channel parameter in the range of [15-20), epochs parameter as <= 50, heads parameter as <= 5, and negative slope parameter as <= 0.2. For the mutation rules that change the Boolean type parameters, if the parameter value of the original model is true, we set it to false. If the original value is false, we set it to true. The parameter ranges for our mutation rules are carefully selected to ensure the change to the original GNN model is slight.

With respect to the configuration of the ranking models utilized in GraphPrior, we made several parameter selections: for the random forest, XGBoost, and LightGBM ranking algorithms, we set the n_estimators parameter to 100. For the DNN ranking model, we set the learning_rate parameter to 0.01. Finally, for the logistic regression ranking algorithm, we set the max_iter parameter to 100.

We conducted the following experiments on a high-performance computer cluster, and each cluster node runs a 2.6 GHz Intel Xeon Gold 6132 CPU with an NVIDIA Tesla V100 16G SXM2 GPU. For the data process, we conducted corresponding experiments on a MacBook Pro laptop with Mac OS Big Sur 11.6, Intel Core i9 CPU, and 64 GB RAM.

4.7 Measurements

Following the existing study [26], we leverage Average Percentage of Fault-Detection (APFD) [92] to evaluate the prioritization effectiveness of GraphPrior and the compared approaches. APFD is a standard metric for prioritization evaluation. Typically, higher APFD values indicate faster misclassification detection rates. We calculate the APFD values by Formula 5

$$APFD = 1 - \frac{\sum_{i=1}^{k} o_i}{kn} + \frac{1}{2n} \tag{5}$$

where n is the number of test inputs in the test set T. k is the number of test inputs in T that will be misclassified by the GNN model G. o_i is the index of the i_{th} misclassified tests in the prioritized test set. More specifically, o_i is an integer that represents the position of the i_{th} misclassified tests in the test set that has been prioritized. When $\sum_{i=1}^{k} o_i$ is small (i.e., the total index sum of the misclassified tests within the prioritized list is small), indicating that that the misclassified tests are prioritized higher, the APFD will be large according to Formula 5. Therefore, large APFD indicates better prioritization effectiveness. Following the existing study [26], we normalize the APFD values to [0,1]. We consider a prioritization approach better when the APFD value is closer to 1. We present the comparison results in tables.

For more detailed analysis, we utilize PFD (Percentage of Fault Detected) [26] to evaluate the fault detection rate of each approach on different ratios of prioritized test inputs. High PFD values refer to high effectiveness in detecting misclassified test inputs.

$$PFD = \frac{F_c}{F_t} \tag{6}$$

where F_c is the number of faults (i.e., misclassified test inputs) correctly detected. F_t is the total number of faults. More specifically, we evaluate the fault detection rate of GraphPrior against different ratios of prioritized tests. We use **PFD-n** to represent the first n% prioritized test inputs.

5 RESULTS AND ANALYSIS

5.1 RQ1: Effectiveness of the killing-based GraphPrior approach (KMGP)

Objectives: We investigate the effectiveness of the killing-based GraphPrior approach, KMGP (cf. Section 3.3), comparing it with existing approaches that can be used to identify possibly-misclassified test inputs.

Experimental design: We used 16 pairs of datasets and GNN models as subjects to evaluate the effectiveness of GraphPrior. Table 1 exhibits their basic information. We carefully selected 7 compared approaches (i.e., DeepGini, least confidence, margin, Vanilla SM, PCS, entropy, and random selection), which can be adapted for GNN test prioritization. Random selection is considered the baseline. We adopt two metrics to measure the effectiveness of GraphPrior and the compared approaches: Average Percentage of Fault-Detection (APFD) and Percentage of Fault Detected (PFD), which are explained in Section 4.7.

Due to the randomness of the training process of a GNN model, we conduct a statistical analysis by repeating all the experiments 10 times. More specifically, for each subject (a dataset with a GNN model), 10 different GNN models are generated through separate training processes.

Results: The GraphPrior approach KMGP outperforms all the compared approaches (i.e., DeepGini, Least Confidence, Margin, Vanilla SM, PCS, Entropy, and Random) in GNN test prioritization. Table 2 presents the comparison results of the killing-based GraphPrior approach (KMGP) and a set of compared approaches using the APFD metric. We highlight the approach with the highest effectiveness for each case in grey. The results demonstrate that KMGP outperforms the other approaches in the majority of cases, specifically in 87.5% (14 out of 16) subjects. Vanilla SM, on the other hand, performs the best in only 12.5% of cases. Additionally, the average APFD value achieved by KMGP was 0.748, which is higher than that of the compared techniques, with improvements of 4.76%~49.6%. These results suggest that KMGP offers a promising solution for prioritizing GNN test inputs.

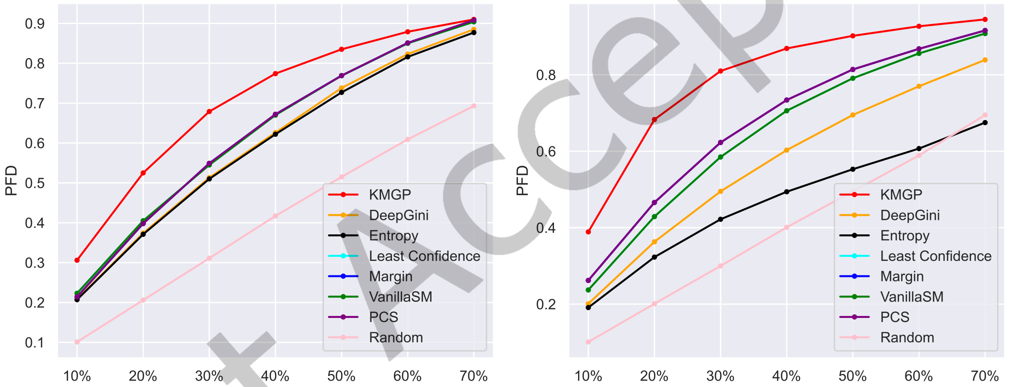
Table 3 exhibits the comparison results among the test prioritization techniques with respect to PFD. We highlight the approach with the highest effectiveness for each case in grey. The findings indicate that, for 68.75% (11 out of 16) of the subjects, KMGP performs best when prioritizing less than 50% of tests. Furthermore, for a majority of the subjects, specifically 87.5% (14 out of 16), KMGP exhibits the best performance when prioritizing less than 30% of tests. Furthermore, Table 4 exhibits the overall comparison results in terms of PFD. We can see that when prioritizing 10%~30% test inputs, the average effectiveness of KMGP outperforms that of the compared approaches in 100% cases. When prioritizing 10%~50% test inputs, the average effectiveness of KMGP outperforms that of the compared approaches in 90% cases. Figure 2 plots the ratio of detected misclassified tests against the prioritized tests. We see that GraphPrior achieves a higher APFD value in comparison to DeepGini, entropy, least confidence, margin, Vanilla SM, PCS, and random. These results confirm the effectiveness of KMGP in GNN test input prioritization.

To demonstrate the stability of our findings, a statistical analysis is performed. Specifically, all the experiments are repeated ten times for each subject, resulting in 10 distinct GNN model instances obtained through separate training processes for a given original GNN model. Based on the statistical analysis of the resulting data, the p-value was found to be lower than 10^{-05}, indicating that the KMGP approach can consistently outperform the compared approaches in terms of test prioritization.

> **Answer to RQ1:** *The GraphPrior approach KMGP outperforms all the compared approaches (i.e., DeepGini, Least Confidence, Margin, Vanilla SM, PCS, Entropy and Random) in GNN test prioritization.*

Table 2. Effectiveness comparison among KMGP and the compared approaches in terms of APFD

Data	Model	Approaches							
		KMGP	DeepGini	Least Confidence	Margin	Vanilla SM	PCS	Entropy	Random
CiteSeer	GAT	0.708	0.671	0.691	0.694	0.691	0.694	0.646	0.508
	GCN	0.701	0.641	0.677	0.682	0.677	0.682	0.638	0.502
	GraphSAGE	0.739	0.663	0.684	0.684	0.684	0.684	0.659	0.497
	TAGCN	0.712	0.658	0.691	0.694	0.691	0.694	0.620	0.499
Cora	GAT	0.841	0.742	0.770	0.763	0.770	0.763	0.733	0.487
	GCN	0.812	0.690	0.736	0.739	0.736	0.739	0.684	0.495
	GraphSAGE	0.792	0.727	0.781	0.784	0.781	0.784	0.704	0.515
	TAGCN	0.782	0.701	0.739	0.738	0.739	0.738	0.690	0.498
LastFM	GAT	0.801	0.633	0.695	0.713	0.695	0.713	0.534	0.498
	GCN	0.761	0.713	0.758	0.746	0.758	0.746	0.603	0.497
	GraphSAGE	0.702	0.734	0.761	0.754	0.761	0.754	0.626	0.502
	TAGCN	0.673	0.719	0.741	0.730	0.741	0.730	0.657	0.498
PubMed	GAT	0.735	0.642	0.670	0.661	0.670	0.661	0.645	0.502
	GCN	0.748	0.645	0.680	0.670	0.680	0.670	0.647	0.501
	GraphSAGE	0.747	0.631	0.685	0.675	0.685	0.675	0.634	0.498
	TAGCN	0.720	0.613	0.663	0.672	0.663	0.672	0.615	0.497
Average		0.748	0.677	0.714	0.712	0.714	0.712	0.646	0.500



(a) CiteSeer, GraphSAGE

(b) LastFM, GAT

Fig. 2. Test prioritization effectiveness among KMGP and the compared approaches for CiteSeer with GraphSAGE and LastFM with GAT. X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected miscalssified tests.

5.2 RQ2: Effectiveness of the feature-based GraphPrior approaches

Objectives: We investigate the effectiveness of feature-based approaches in GraphPrior, including XGGP, LRGP, RFGP, LGGP, and DNGP, compared with the killing-based approach KMGP.

Experimental design: We evaluated the effectiveness of feature-based GraphPrior approaches with the killing-based approach KMGP on 16 subjects (four graph datasets × four GNN models). Due to the randomness of the training process of a GNN model, we repeat all the experiments ten times and calculate the average results. For each subject (a dataset with a GNN model), 10 different GNN models are generated through separate training processes. For evaluation, we calculated the APFD (Average Percentage of Fault-Detection) values of all the approaches on each subject, which can reflect the misclassification detection rate. Moreover, we calculated the

Table 3. Effectiveness comparison among KMGP and the compared approaches in terms of PFD

Data	Model	Approaches	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70	Data	Model	Approaches	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
CiteSeer	GAT	**KMGP**	0.264	0.464	0.629	0.750	0.812	0.841	0.875	LastFM	GAT	**KMGP**	0.389	0.683	0.810	0.869	0.902	0.927	0.945
		DeepGini	0.211	0.382	0.521	0.646	0.748	0.828	0.895			DeepGini	0.201	0.363	0.495	0.603	0.695	0.770	0.839
		Entropy	0.203	0.373	0.506	0.621	0.716	0.788	0.844			Entropy	0.191	0.323	0.422	0.494	0.553	0.607	0.675
		Least Confidence	0.231	0.409	0.550	0.680	0.777	0.861	0.913			Least Confidence	0.237	0.429	0.585	0.706	0.791	0.856	0.908
		Margin	0.228	0.401	0.547	0.688	0.794	0.864	0.914			Margin	0.262	0.466	0.623	0.734	0.814	0.868	0.916
		Vanilla SM	0.231	0.409	0.550	0.680	0.777	0.861	0.913			Vanilla SM	0.237	0.429	0.585	0.706	0.791	0.856	0.908
		PCS	0.228	0.401	0.547	0.688	0.794	0.864	0.914			PCS	0.262	0.466	0.623	0.734	0.814	0.868	0.916
		Random	0.099	0.192	0.296	0.391	0.493	0.591	0.689			Random	0.101	0.201	0.300	0.401	0.495	0.589	0.695
	GCN	**KMGP**	0.278	0.492	0.652	0.723	0.771	0.811	0.865		GCN	**KMGP**	0.403	0.648	0.728	0.770	0.830	0.868	0.915
		DeepGini	0.200	0.355	0.490	0.600	0.697	0.783	0.858			DeepGini	0.267	0.467	0.600	0.715	0.799	0.875	0.928
		Entropy	0.201	0.354	0.487	0.595	0.692	0.779	0.856			Entropy	0.254	0.411	0.501	0.570	0.627	0.685	0.755
		Least Confidence	0.229	0.406	0.544	0.661	0.748	0.827	0.889			Least Confidence	0.298	0.530	0.691	0.799	0.880	0.927	0.956
		Margin	0.214	0.399	0.556	0.674	0.776	0.844	0.895			Margin	0.278	0.499	0.661	0.783	0.865	0.920	0.951
		Vanilla SM	0.229	0.406	0.544	0.661	0.748	0.827	0.889			Vanilla SM	0.298	0.530	0.691	0.799	0.880	0.927	0.956
		PCS	0.214	0.399	0.556	0.674	0.776	0.844	0.895			PCS	0.278	0.499	0.661	0.783	0.865	0.920	0.951
		Random	0.098	0.197	0.292	0.388	0.488	0.587	0.690			Random	0.098	0.199	0.302	0.397	0.503	0.600	0.704
	GraphSAGE	**KMGP**	0.306	0.525	0.679	0.774	0.835	0.879	0.910		GraphSAGE	**KMGP**	0.302	0.482	0.580	0.668	0.800	0.842	0.902
		DeepGini	0.208	0.374	0.513	0.626	0.738	0.823	0.885			DeepGini	0.285	0.501	0.655	0.765	0.836	0.893	0.929
		Entropy	0.207	0.371	0.510	0.622	0.727	0.816	0.877			Entropy	0.283	0.443	0.520	0.587	0.649	0.708	0.775
		Least Confidence	0.223	0.405	0.545	0.670	0.769	0.850	0.904			Least Confidence	0.294	0.527	0.709	0.825	0.892	0.922	0.946
		Margin	0.214	0.398	0.549	0.672	0.769	0.851	0.908			Margin	0.276	0.525	0.700	0.819	0.883	0.913	0.944
		Vanilla SM	0.223	0.405	0.545	0.670	0.769	0.850	0.904			Vanilla SM	0.294	0.527	0.709	0.825	0.892	0.922	0.946
		PCS	0.214	0.398	0.549	0.672	0.769	0.851	0.908			PCS	0.276	0.525	0.700	0.819	0.883	0.913	0.944
		Random	0.101	0.206	0.311	0.417	0.515	0.609	0.693			Random	0.095	0.194	0.298	0.398	0.498	0.596	0.697
	TAGCN	**KMGP**	0.295	0.490	0.617	0.723	0.795	0.845	0.888		TAGCN	**KMGP**	0.250	0.431	0.544	0.644	0.706	0.819	0.892
		DeepGini	0.216	0.375	0.512	0.622	0.719	0.808	0.877			DeepGini	0.260	0.461	0.615	0.731	0.821	0.885	0.934
		Entropy	0.214	0.366	0.492	0.592	0.693	0.749	0.801			Entropy	0.258	0.451	0.577	0.653	0.720	0.769	0.816
		Least Confidence	0.246	0.427	0.570	0.678	0.772	0.845	0.905			Least Confidence	0.260	0.475	0.642	0.769	0.865	0.928	0.966
		Margin	0.234	0.430	0.578	0.688	0.776	0.850	0.907			Margin	0.238	0.450	0.616	0.755	0.856	0.922	0.962
		Vanilla SM	0.246	0.427	0.570	0.678	0.772	0.845	0.905			Vanilla SM	0.260	0.475	0.642	0.769	0.865	0.928	0.966
		PCS	0.234	0.430	0.578	0.688	0.776	0.850	0.907			PCS	0.238	0.450	0.616	0.755	0.856	0.922	0.962
		Random	0.101	0.196	0.297	0.383	0.482	0.586	0.684			Random	0.100	0.203	0.299	0.401	0.497	0.596	0.697
Cora	GAT	**KMGP**	0.454	0.759	0.884	0.919	0.939	0.954	0.975	PubMed	GAT	**KMGP**	0.336	0.588	0.697	0.754	0.813	0.859	0.893
		DeepGini	0.295	0.509	0.669	0.781	0.852	0.892	0.928			DeepGini	0.205	0.359	0.495	0.607	0.702	0.782	0.856
		Entropy	0.293	0.503	0.658	0.766	0.842	0.886	0.918			Entropy	0.205	0.360	0.496	0.609	0.707	0.788	0.864
		Least Confidence	0.296	0.539	0.724	0.830	0.899	0.932	0.962			Least Confidence	0.213	0.384	0.532	0.657	0.758	0.841	0.895
		Margin	0.282	0.525	0.708	0.815	0.879	0.934	0.970			Margin	0.215	0.388	0.532	0.656	0.750	0.817	0.871
		Vanilla SM	0.296	0.539	0.724	0.830	0.899	0.932	0.962			Vanilla SM	0.213	0.384	0.532	0.657	0.758	0.841	0.895
		PCS	0.282	0.525	0.708	0.815	0.879	0.934	0.970			PCS	0.215	0.388	0.532	0.656	0.750	0.817	0.871
		Random	0.099	0.192	0.294	0.392	0.478	0.578	0.679			Random	0.101	0.201	0.298	0.396	0.497	0.595	0.696
	GCN	**KMGP**	0.384	0.704	0.854	0.884	0.909	0.933	0.952		GCN	**KMGP**	0.347	0.607	0.743	0.788	0.826	0.860	0.894
		DeepGini	0.249	0.418	0.569	0.682	0.776	0.853	0.908			DeepGini	0.215	0.395	0.534	0.624	0.698	0.771	0.838
		Entropy	0.245	0.411	0.559	0.676	0.763	0.840	0.897			Entropy	0.216	0.395	0.535	0.626	0.701	0.774	0.842
		Least Confidence	0.265	0.480	0.643	0.770	0.848	0.906	0.954			Least Confidence	0.223	0.407	0.560	0.686	0.782	0.844	0.890
		Margin	0.254	0.469	0.653	0.781	0.860	0.912	0.956			Margin	0.211	0.397	0.550	0.679	0.768	0.832	0.876
		Vanilla SM	0.265	0.480	0.643	0.770	0.848	0.906	0.954			Vanilla SM	0.223	0.407	0.560	0.686	0.782	0.844	0.890
		PCS	0.254	0.469	0.653	0.781	0.860	0.912	0.956			PCS	0.211	0.397	0.550	0.679	0.768	0.832	0.876
		Random	0.097	0.197	0.291	0.398	0.505	0.596	0.695			Random	0.098	0.202	0.302	0.403	0.503	0.602	0.704
	GraphSAGE	**KMGP**	0.489	0.705	0.777	0.820	0.848	0.886	0.919		GraphSAGE	**KMGP**	0.396	0.635	0.713	0.757	0.808	0.850	0.889
		DeepGini	0.323	0.498	0.623	0.736	0.829	0.878	0.922			DeepGini	0.214	0.364	0.488	0.589	0.676	0.756	0.829
		Entropy	0.318	0.482	0.604	0.710	0.792	0.846	0.885			Entropy	0.215	0.365	0.490	0.591	0.680	0.761	0.834
		Least Confidence	0.356	0.584	0.723	0.833	0.903	0.940	0.962			Least Confidence	0.229	0.407	0.561	0.682	0.774	0.846	0.901
		Margin	0.363	0.604	0.735	0.830	0.897	0.939	0.964			Margin	0.229	0.412	0.555	0.668	0.756	0.832	0.884
		Vanilla SM	0.356	0.584	0.723	0.833	0.903	0.940	0.962			Vanilla SM	0.229	0.407	0.561	0.682	0.774	0.846	0.901
		PCS	0.363	0.604	0.735	0.830	0.897	0.939	0.964			PCS	0.229	0.412	0.555	0.668	0.756	0.832	0.884
		Random	0.107	0.205	0.306	0.403	0.500	0.596	0.691			Random	0.096	0.200	0.303	0.400	0.505	0.606	0.704
	TAGCN	**KMGP**	0.372	0.668	0.788	0.841	0.863	0.888	0.914		TAGCN	**KMGP**	0.379	0.545	0.610	0.722	0.791	0.844	0.885
		DeepGini	0.249	0.450	0.586	0.696	0.783	0.857	0.914			DeepGini	0.210	0.352	0.468	0.553	0.644	0.732	0.811
		Entropy	0.246	0.442	0.578	0.689	0.771	0.838	0.895			Entropy	0.211	0.354	0.470	0.557	0.650	0.736	0.814
		Least Confidence	0.273	0.481	0.638	0.762	0.850	0.913	0.954			Least Confidence	0.223	0.397	0.541	0.658	0.744	0.815	0.867
		Margin	0.255	0.466	0.638	0.764	0.861	0.922	0.964			Margin	0.232	0.414	0.566	0.675	0.761	0.822	0.868
		Vanilla SM	0.273	0.481	0.638	0.762	0.850	0.913	0.954			Vanilla SM	0.223	0.397	0.541	0.658	0.744	0.815	0.867
		PCS	0.255	0.466	0.638	0.764	0.861	0.922	0.964			PCS	0.232	0.414	0.566	0.675	0.761	0.822	0.868
		Random	0.102	0.204	0.309	0.403	0.507	0.592	0.699			Random	0.099	0.196	0.297	0.398	0.499	0.601	0.700

PFD (Percentage of Fault Detected) values of all the approaches on different ratios of prioritized tests to further investigate the effectiveness of feature-based approaches.

Results: The experimental results of this research question are exhibited in Table 5, Table 6 and Table 7. Table 5 presents the comparison results in terms of APFD, while Table 6 and Table 7 present the comparison results in terms of PFD.

Among all the GraphPrior approaches, RFGP demonstrates the highest level of effectiveness in most cases. Table 5 exhibits the comparison results among KMGP (i.e., the killing-based GraphPrior approach) and the feature-based GraphPrior approaches in terms of APFD. The results demonstrate RFGP outperforms other GraphPrior approaches on average. Moreover, the average APFD values of RFGP exceed that of KMGP by around 0.02. Additionally, across different subjects, RFGP outperforms other GraphPrior approaches in the majority of cases. To provide a more detailed analysis, Table 6 and Table 7 exhibit the comparison results of all GraphPrior approaches in terms of PFD. The findings also confirm that RFGP is the most effective GraphPrior approach. Furthermore, Table 7 indicates that, on average, RFGP is consistently more effective than other GraphPrior approaches across different test prioritization ratios. Figure 3 presents some examples aimed at providing a more visually intuitive understanding of the performance of the various GraphPrior approaches. Collectively, these results suggest that RFGP is the most effective GraphPrior approach for the evaluated datasets.

Table 4. Average comparison results among KMGP and the compared approaches in terms of PFD

Data	Approaches	#Best case in PFD							Average PFD						
		PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
CiteSeer	**KMGP**	4	4	4	4	3	1	1	0.285	0.492	0.644	0.742	0.803	0.844	0.884
	DeepGini	0	0	0	0	0	0	0	0.208	0.371	0.509	0.623	0.725	0.810	0.878
	Entropy	0	0	0	0	0	0	0	0.206	0.366	0.498	0.607	0.707	0.783	0.844
	Least Confidence	0	0	0	0	0	0	0	0.232	0.411	0.552	0.672	0.766	0.845	0.902
	Margin	0	0	0	0	0	0	0	0.222	0.407	0.557	0.680	0.778	0.852	0.906
	Vanilla SM	0	0	0	0	0	0	0	0.232	0.411	0.552	0.672	0.766	0.845	0.902
	PCS	0	0	0	0	1	3	3	0.222	0.407	0.557	0.680	0.778	0.852	0.906
	Random	0	0	0	0	0	0	0	0.099	0.197	0.299	0.394	0.494	0.593	0.689
Cora	**KMGP**	4	4	4	3	3	2	1	0.424	0.709	0.825	0.866	0.889	0.915	0.940
	DeepGini	0	0	0	0	0	0	0	0.279	0.468	0.611	0.723	0.810	0.870	0.918
	Entropy	0	0	0	0	0	0	0	0.275	0.459	0.599	0.710	0.792	0.852	0.898
	Least Confidence	0	0	0	0	0	0	0	0.297	0.521	0.681	0.798	0.875	0.922	0.958
	Margin	0	0	0	0	0	0	0	0.288	0.516	0.683	0.797	0.874	0.926	0.963
	Vanilla SM	0	0	0	1	1	1	0	0.297	0.521	0.681	0.798	0.875	0.922	0.958
	PCS	0	0	0	0		1	3	0.288	0.516	0.683	0.797	0.874	0.926	0.963
	Random	0	0	0	0	0	0	0	0.101	0.199	0.300	0.399	0.497	0.590	0.691
LastFM	**KMGP**	3	2	2	1	1	1	1	0.336	0.561	0.665	0.737	0.809	0.864	0.913
	DeepGini	0	0	0	0	0	0	0	0.253	0.448	0.591	0.703	0.787	0.855	0.907
	Entropy	0	0	0	0	0	0	0	0.246	0.407	0.505	0.576	0.637	0.692	0.755
	Least Confidence	0	0	0	0	0	0	0	0.272	0.490	0.656	0.774	0.857	0.908	0.944
	Margin	0	0	0	0	0	0	0	0.263	0.485	0.650	0.772	0.854	0.905	0.943
	Vanilla SM	1	2	2	3	3	3	3	0.272	0.490	0.656	0.774	0.857	0.908	0.944
	PCS	0	0	0	0	0	0	0	0.263	0.485	0.65	0.772	0.854	0.905	0.943
	Random	0	0	0	0	0	0	0	0.098	0.199	0.299	0.399	0.498	0.595	0.698
PubMed	**KMGP**	4	4	4	4	4	4	2	0.364	0.593	0.690	0.755	0.809	0.853	0.890
	DeepGini	0	0	0	0	0	0	0	0.211	0.367	0.496	0.593	0.679	0.760	0.833
	Entropy	0	0	0	0	0	0	0	0.211	0.368	0.497	0.595	0.684	0.764	0.838
	Least Confidence	0	0	0	0	0	0	0	0.222	0.398	0.548	0.670	0.764	0.836	0.888
	Margin	0	0	0	0	0	0	2	0.221	0.402	0.550	0.669	0.758	0.825	0.874
	Vanilla SM	0	0	0	0	0	0	0	0.222	0.398	0.548	0.670	0.764	0.836	0.888
	PCS	0	0	0	0	0	0	0	0.221	0.402	0.550	0.669	0.758	0.825	0.874
	Random	0	0	0	0	0	0	0	0.098	0.199	0.300	0.399	0.501	0.601	0.701

Additionally, although the killing-based GraphPrior approach, KMGP, shows good effectiveness in some specific datasets, its average effectiveness is lower than several feature-based GraphPrior approaches, such as RFGP, LGGP, and XGGP. This result suggests that KMGP is less stable compared to some feature-based approaches. For example, in Figure 3(b), we can see that KMGP (represented by the red line) is less effective than other GraphPrior approaches. In fact, the main difference between KMGP and feature-based GraphPrior approaches lies in their strategy for utilizing mutation results. Specifically, KMGP treats all mutated models as having equal importance, whereas feature-based GraphPrior approaches, such as RFGP, employ ranking models to assign higher weights to the more important mutated models, thereby better utilizing mutation results for test prioritization. The superior performance of RFGP indicates that the random forest algorithm it utilizes can effectively identify important mutated models and assign them high weights.

The efficiency of GraphPrior (all the six approaches) is acceptable. Table 8 illustrates the efficiency of GraphPrior in comparison with other approaches. The time cost of GraphPrior can be decomposed into three phases, namely mutant generation, training, and execution. Mutant generation involves the production of mutated models based on retraining the original GNN model. The training time represents the average duration needed for training a ranking model. Finally, execution time denotes the average duration expended on test prioritization. By decomposing the time cost into these distinct phases, we provide a more detailed understanding of the efficiency of GraphPrior in contrast to other approaches. As evident from Table 8, the average execution time of GraphPrior for test prioritization is 40 seconds, with the most time-consuming phase being mutant generation, which takes around 35 minutes. In contrast, the average execution time of the compared approaches is less than one second. Although GraphPrior is not as efficient as the compared approaches, it provides a viable alternative to costly and time-consuming manual labeling, and its total time cost remains acceptable in real-world scenarios.

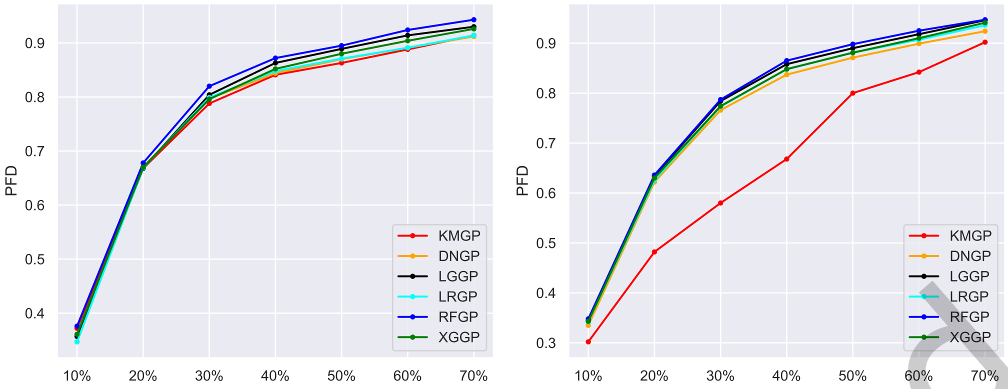
> **Answer to RQ2:** *Among all the GraphPrior approaches, RFGP demonstrates the highest level of effectiveness in most cases. The efficiency of GraphPrior (all the six approaches) is acceptable.*

Table 5. Effectiveness comparison among KMGP and the feature-based GraphPrior approaches in terms of APFD

Data	**Model**	**Approaches**					
		DGGP	LGGP	XGGP	LRGP	RFGP	**KMGP**
CiteSeer	GAT	0.633	0.678	0.669	0.651	0.675	0.708
	GCN	0.682	0.695	0.690	0.678	0.694	0.701
	GraphSAGE	0.656	0.694	0.699	0.682	0.710	0.739
	TAGCN	0.652	0.681	0.694	0.660	0.696	0.712
Cora	GAT	0.749	0.785	0.795	0.767	0.811	0.841
	GCN	0.778	0.791	0.791	0.784	0.806	0.812
	GraphSAGE	0.764	0.791	0.793	0.784	0.794	0.792
	TAGCN	0.777	0.785	0.785	0.778	0.800	0.782
LastFM	GAT	0.799	0.814	0.812	0.802	0.826	0.801
	GCN	0.796	0.811	0.809	0.802	0.816	0.761
	GraphSAGE	0.771	0.785	0.780	0.778	0.789	0.702
	TAGCN	0.763	0.781	0.776	0.770	0.779	0.673
PubMed	GAT	0.740	0.774	0.768	0.763	0.773	0.735
	GCN	0.743	0.749	0.745	0.746	0.750	0.748
	GraphSAGE	0.743	0.776	0.767	0.768	0.774	0.747
	TAGCN	0.701	0.780	0.773	0.765	0.768	0.720
Average		0.734	0.761	0.759	0.749	0.766	0.748

Table 6. Effectiveness comparison among KMGP and the feature-based GraphPrior approaches in terms of PFD

Data	**Model**	**Approaches**	**PFD-10**	**PFD-20**	**PFD-30**	**PFD-40**	**PFD-50**	**PFD-60**	**PFD-70**	**Data**	**Model**	**Approaches**	**PFD-10**	**PFD-20**	**PFD-30**	**PFD-40**	**PFD-50**	**PFD-60**	**PFD-70**
CiteSeer	GAT	**KMGP**	0.264	0.464	0.629	0.750	0.812	0.841	0.875	LastFM	GAT	**KMGP**	0.389	0.683	0.810	0.869	0.902	0.927	0.945
		DNGP	0.252	0.460	0.596	0.647	0.693	0.722	0.753			DNGP	0.382	0.728	0.848	0.863	0.883	0.905	0.926
		LGGP	0.251	0.465	0.621	0.715	0.759	0.791	0.833			LGGP	0.397	0.740	0.861	0.889	0.904	0.924	0.942
		LRGP	0.244	0.467	0.611	0.683	0.721	0.750	0.788			LRGP	0.389	0.729	0.848	0.876	0.892	0.910	0.929
		RFGP	0.257	0.470	0.619	0.697	0.743	0.781	0.827			RFGP	0.404	0.746	0.874	0.906	0.927	0.944	0.960
		XGGP	0.256	0.464	0.618	0.702	0.740	0.771	0.817			XGGP	0.393	0.737	0.856	0.886	0.901	0.921	0.942
	GCN	**KMGP**	0.278	0.492	0.652	0.723	0.771	0.811	0.865		GCN	**KMGP**	0.403	0.648	0.728	0.770	0.830	0.868	0.915
		DNGP	0.248	0.479	0.643	0.699	0.748	0.794	0.843			DNGP	0.412	0.717	0.814	0.849	0.877	0.906	0.93
		LGGP	0.273	0.483	0.651	0.717	0.764	0.803	0.856			LGGP	0.428	0.730	0.830	0.873	0.898	0.921	0.945
		LRGP	0.251	0.484	0.643	0.698	0.745	0.787	0.832			LRGP	0.424	0.717	0.817	0.859	0.886	0.912	0.937
		RFGP	0.272	0.486	0.653	0.716	0.762	0.807	0.852			RFGP	0.431	0.735	0.842	0.881	0.906	0.927	0.949
		XGGP	0.265	0.481	0.650	0.711	0.760	0.804	0.848			XGGP	0.424	0.724	0.826	0.869	0.895	0.918	0.942
	GraphSAGE	**KMGP**	0.306	0.525	0.679	0.774	0.835	0.879	0.910		GraphSAGE	**KMGP**	0.302	0.482	0.580	0.668	0.800	0.842	0.902
		DNGP	0.271	0.511	0.635	0.670	0.695	0.729	0.771			DNGP	0.335	0.622	0.766	0.837	0.871	0.899	0.924
		LGGP	0.287	0.515	0.680	0.733	0.767	0.797	0.831			LGGP	0.344	0.634	0.784	0.858	0.890	0.918	0.946
		LRGP	0.273	0.512	0.671	0.708	0.737	0.767	0.806			LRGP	0.342	0.626	0.773	0.848	0.881	0.907	0.936
		RFGP	0.287	0.515	0.684	0.730	0.775	0.816	0.865			RFGP	0.348	0.636	0.787	0.865	0.898	0.925	0.947
		XGGP	0.283	0.516	0.661	0.703	0.753	0.800	0.851			XGGP	0.343	0.630	0.774	0.848	0.881	0.910	0.941
	TAGCN	**KMGP**	0.295	0.490	0.617	0.723	0.795	0.845	0.888		TAGCN	**KMGP**	0.250	0.431	0.544	0.644	0.706	0.819	0.892
		DNGP	0.285	0.504	0.578	0.628	0.682	0.740	0.784			DNGP	0.294	0.552	0.742	0.840	0.884	0.915	0.936
		LGGP	0.298	0.513	0.651	0.700	0.737	0.773	0.811			LGGP	0.299	0.562	0.758	0.865	0.914	0.944	0.964
		LRGP	0.292	0.507	0.587	0.640	0.692	0.749	0.799			LRGP	0.295	0.555	0.747	0.846	0.896	0.927	0.950
		RFGP	0.294	0.511	0.662	0.694	0.747	0.793	0.845			RFGP	0.300	0.561	0.756	0.867	0.915	0.942	0.961
		XGGP	0.297	0.510	0.636	0.695	0.748	0.801	0.849			XGGP	0.297	0.558	0.751	0.860	0.911	0.936	0.960
Cora	GAT	**KMGP**	0.454	0.759	0.884	0.919	0.939	0.954	0.975	PubMed	GAT	**KMGP**	0.336	0.588	0.697	0.754	0.813	0.859	0.893
		DNGP	0.383	0.722	0.791	0.800	0.814	0.827	0.848			DNGP	0.334	0.631	0.730	0.767	0.803	0.843	0.883
		LGGP	0.427	0.724	0.823	0.836	0.848	0.867	0.894			LGGP	0.363	0.643	0.763	0.816	0.853	0.893	0.932
		LRGP	0.361	0.725	0.826	0.834	0.845	0.858	0.871			LRGP	0.354	0.632	0.746	0.803	0.841	0.881	0.919
		RFGP	0.428	0.730	0.869	0.882	0.894	0.909	0.928			RFGP	0.362	0.639	0.763	0.815	0.853	0.894	0.929
		XGGP	0.375	0.729	0.849	0.870	0.885	0.902	0.916			XGGP	0.360	0.640	0.756	0.806	0.844	0.886	0.921
	GCN	**KMGP**	0.384	0.704	0.854	0.884	0.909	0.933	0.952		GCN	**KMGP**	0.347	0.607	0.743	0.788	0.826	0.860	0.894
		DNGP	0.359	0.691	0.814	0.844	0.870	0.893	0.914			DNGP	0.347	0.629	0.739	0.779	0.816	0.851	0.885
		LGGP	0.357	0.678	0.831	0.862	0.889	0.913	0.932			LGGP	0.355	0.634	0.746	0.785	0.823	0.857	0.891
		LRGP	0.359	0.687	0.823	0.853	0.880	0.902	0.920			LRGP	0.353	0.629	0.741	0.782	0.818	0.854	0.888
		RFGP	0.379	0.691	0.848	0.876	0.900	0.928	0.947			RFGP	0.354	0.629	0.745	0.787	0.824	0.858	0.892
		XGGP	0.365	0.682	0.830	0.861	0.885	0.911	0.930			XGGP	0.348	0.629	0.740	0.780	0.818	0.853	0.886
	GraphSAGE	**KMGP**	0.489	0.705	0.777	0.820	0.848	0.886	0.919		GraphSAGE	**KMGP**	0.396	0.635	0.713	0.757	0.808	0.850	0.889
		DNGP	0.480	0.705	0.736	0.776	0.805	0.845	0.879			DNGP	0.396	0.670	0.717	0.753	0.791	0.833	0.872
		LGGP	0.475	0.721	0.772	0.818	0.857	0.895	0.924			LGGP	0.409	0.684	0.758	0.803	0.843	0.882	0.917
		LRGP	0.474	0.728	0.776	0.802	0.832	0.863	0.906			LRGP	0.406	0.677	0.744	0.791	0.833	0.874	0.909
		RFGP	0.487	0.736	0.771	0.803	0.848	0.896	0.923			RFGP	0.408	0.684	0.758	0.802	0.843	0.881	0.914
		XGGP	0.479	0.718	0.760	0.803	0.854	0.894	0.939			XGGP	0.404	0.678	0.748	0.793	0.832	0.871	0.907
	TAGCN	**KMGP**	0.372	0.668	0.788	0.841	0.863	0.888	0.914		TAGCN	**KMGP**	0.379	0.545	0.610	0.722	0.791	0.844	0.885
		DNGP	0.347	0.671	0.797	0.844	0.870	0.891	0.912			DNGP	0.402	0.593	0.644	0.692	0.734	0.777	0.828
		LGGP	0.357	0.668	0.804	0.863	0.889	0.914	0.930			LGGP	0.415	0.631	0.731	0.804	0.865	0.910	0.946
		LRGP	0.347	0.669	0.799	0.848	0.871	0.891	0.914			LRGP	0.409	0.618	0.707	0.784	0.840	0.889	0.927
		RFGP	0.376	0.678	0.820	0.872	0.895	0.924	0.943			RFGP	0.409	0.621	0.722	0.795	0.847	0.889	0.923
		XGGP	0.361	0.670	0.796	0.852	0.880	0.904	0.926			XGGP	0.410	0.627	0.722	0.796	0.852	0.899	0.934



(a) Cora, TAGCN

(b) LastFM, GraphSAGE

Fig. 3. Test prioritization effectiveness of the six GraphPrior approaches for Cora with TAGCN and LastFM with GraphSAGE. X-Axis: the percentage of prioritized tests; Y-Axis: the percentage of detected miscalssified tests

Table 7. Average effectiveness comparison among KMGP and the feature-based GraphPrior approaches in terms of PFD

Approaches	**Average PFD**						
	PFD-10	PFD-20	PFD-30	PFD-40	PFD-50	PFD-60	PFD-70
KMGP	0.353	0.589	0.707	0.775	0.828	0.869	0.907
DNGP	0.346	0.618	0.724	0.768	0.802	0.836	0.868
LGGP	0.358	0.627	0.754	0.809	0.844	0.875	0.906
LRGP	0.348	0.623	0.741	0.791	0.826	0.858	0.890
RFGP	0.362	0.629	0.761	0.812	0.849	0.882	0.913
XGGP	0.354	0.624	0.748	0.802	0.840	0.874	0.907

Table 8. Time comparison between GraphPrior and compared approaches

Time cost parts	**Approaches**							
	GraphPrior	DeepGini	Least Confidence	Margin	Vanilla SM	PCS	Entropy	Rndom
Mutant Generation	**35 min**	-	-	-	-	-	-	-
Training	**3 min**	-	-	-	-	-	-	-
Execution	**40 s**	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s	< 1 s

5.3 RQ3: Effectiveness of GraphPrior on adversarial test inputs

Objectives: We further investigate the effectiveness of GraphPrior on adversarial test data. Here, we adopt eight graph adversarial attacks (cf. Section 4.4) from the existing studies [3, 48, 86, 100]. The results can answer whether GraphPrior can perform well on adversarial test sets for GNNs, compared with existing approaches that can be used to identify possibly-misclassified test inputs.

Experimental design: We evaluate GraphPrior on adversarial datasets generated by 8 graph attack techniques [100] [86] [3] [48]. In this research question, we set the attack level as 0.3, which means that 30% of the test inputs in the test set are adversarial tests. It is important to note that a high attack level, such as 90%, would result in a significant ratio of adversarial test inputs. Under such circumstances, a larger number of bug cases could be selected by any of the prioritization methods, making it difficult to demonstrate the effectiveness of GraphPrior. Thus, in order to ensure an effective evaluation of GraphPrior and the compared approaches, we selected a reasonable attack level (i.e., 0.3), which can limit the proportion of adversarial test inputs. Totally, in this research question, we evaluate GraphPrior on 108 subjects (4 GNN models, 4 datasets and 8 graph adversarial attacks). We then ran all six GraphPrior approaches and the compared approaches on the subjects, and calculated

Table 9. Effectiveness comparison among GraphPrior and the compared approaches in terms of APFD

Attack	**Approaches**												
	DNGP	**KMGP**	**LGGP**	**XGGP**	**LRGP**	**RFGP**	DeepGini	Least Confidence	Margin	Random	Vanilla SM	PCS	Entropy
DICE	0.672	0.710	0.707	0.706	0.695	0.713	0.667	0.698	0.693	0.500	0.698	0.693	0.642
MMA	0.691	0.725	0.721	0.724	0.705	0.731	0.684	0.717	0.718	0.499	0.717	0.718	0.672
NEAA	0.698	0.723	0.733	0.732	0.721	0.738	0.676	0.711	0.703	0.499	0.711	0.703	0.646
NEAR	0.737	0.735	0.767	0.764	0.757	0.774	0.678	0.719	0.717	0.499	0.719	0.717	0.644
PGD	0.718	0.730	0.743	0.743	0.729	0.753	0.693	0.728	0.727	0.498	0.728	0.727	0.656
RAA	0.659	0.701	0.697	0.696	0.684	0.703	0.671	0.702	0.695	0.499	0.702	0.695	0.648
RAF	0.657	0.702	0.696	0.696	0.683	0.703	0.670	0.701	0.694	0.500	0.701	0.694	0.646
RAR	0.703	0.724	0.735	0.734	0.723	0.742	0.673	0.708	0.707	0.498	0.708	0.707	0.645
Average	**0.692**	**0.718**	**0.725**	**0.724**	**0.712**	**0.732**	**0.677**	**0.711**	**0.707**	**0.499**	**0.711**	**0.707**	**0.650**

the APFD values of each approach with each graph adversarial attack. Moreover, we calculated the PFD values of each approach in terms of different ratios of prioritized values.

Results: GraphPrior approaches outperform the compared approaches (i.e., DeepGini, Least Confidence, Margin, Vanilla SM, PCS, Entropy and Random) in the context of graph adversarial attacks. Table 9 shows the test prioritization effectiveness (measured by APFD) of GraphPrior and the compared approaches across a variety of adversarial attacks. The experimental results indicate that the GraphPrior approaches exhibit superior performance, with the average APFD values ranging from 0.692 to 0.732, while the compared approaches range from 0.499 to 0.711. Notably, five GraphPrior approaches, namely RFGP, XGGP, LRGP, LGGP, and KMGP, outperform all the compared approaches on average across all the adversarial attacks. Table 10 presents the comparison results of GraphPrior and the compared approaches in terms of PFD, confirming the superior performance of GraphPrior from both the perspective of average effectiveness and the number of best cases. Furthermore, Table 11 presents the overall comparison results in terms of PFD, which further support the above conclusions by demonstrating that the largest average effectiveness of each case is achieved by the GraphPrior approaches, along with the largest number of best cases.

Among all the GraphPrior approaches proposed, the effectiveness of RFGP stands out as the most notable. From Table 9, in which the effectiveness is measured by the APFD values, we see that RFGP performs the best across different adversarial attacks, with the average improvement of 2.95%~46.69% compared with uncertainty-based test prioritization approaches. Table 10 presents the test prioritization effectiveness in terms of PFD. The column #Best case in PFD denotes the number of best cases a test prioritization approach achieved across all cases (i.e., all subjects of a graph adversarial attack). The results demonstrate that, against a majority of adversarial attacks, RFGP consistently outperforms all other GraphPrior approaches in terms of average effectiveness. Moreover, Table 11 presents the overall comparison results in terms of PFD, further indicating that RFGP outperforms all other approaches in terms of average effectiveness. Notably, when prioritizing 20% to 40% of the test inputs, RFGP consistently exhibits the highest number of best cases across a variety of subjects.

> **Answer to RQ3:** *GraphPrior approaches outperform the compared approaches (i.e., DeepGini, Least Confidence, Margin, Vanilla SM, PCS, Entropy and Random) in the context of graph adversarial attacks. Among all the GraphPrior approaches proposed, the effectiveness of RFGP stands out as the most notable.*

5.4 RQ4: Effectiveness of GraphPrior against adversarial attacks at varying attack levels

Objectives: We investigate the effectiveness of GraphPrior on adversarial test inputs with different attack levels.

Experimental design: To investigate the effectiveness of GraphPrior on test inputs generated via different levels of graph adversarial attacks, we set different attack levels (i.e., 0.1, 0.2, 0.3 and 0.4) on 8 graph adversarial techniques (i.e., DICE, Min-max attack, NEAA, NEAR, PGD attack, RAA, RAF, and RAR). As mentioned in RQ3, the attack level indicates the ratio of adversarial inputs in the dataset. For example, 0.4 means that 40% tests in the dataset are adversarial tests. We select these attack levels because a high attack level (e.g., 80%) would engender a

Table 10. Effectiveness comparison of GraphPrior and the compared approaches on adversarial test inputs in terms of PFD

Attack	Approaches	#Best cases in PFD				Average PFD				Attack	Approaches	#Best cases in PFD				Average PFD			
		PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40			PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
DICE	**DNGP**	0	2	0	0	0.289	0.553	0.705	0.769	PGD	**DNGP**	0	1	1	0	0.309	0.572	0.701	0.752
	KMGP	7	1	2	2	0.300	0.520	0.665	0.754		**KMGP**	7	4	3	3	0.336	0.573	0.714	0.789
	LGGP	1	0	0	0	0.301	0.557	0.719	0.801		**LGGP**	0	0	0	0	0.307	0.563	0.707	0.781
	LRGP	0	0	0	0	0.291	0.555	0.711	0.788		**LRGP**	0	2	0	0	0.305	0.570	0.697	0.762
	RFGP	4	9	10	10	0.304	0.561	0.729	0.818		**RFGP**	1	1	4	5	0.326	0.571	0.727	0.806
	XGGP	0	0	0	0	0.293	0.556	0.716	0.799		**XGGP**	0	0	0	0	0.300	0.558	0.703	0.774
	DeepGini	0	0	0	0	0.215	0.394	0.535	0.655		DeepGini	0	0	0	0	0.238	0.413	0.556	0.672
	Entropy	0	0	0	0	0.212	0.381	0.507	0.611		Entropy	0	0	0	0	0.236	0.409	0.547	0.660
	Least Confidence	0	0	0	0	0.233	0.428	0.590	0.713		Least Confidence	0	0	0	0	0.255	0.451	0.604	0.727
	Margin	0	0	0	0	0.225	0.423	0.584	0.711		Margin	0	0	0	0	0.242	0.449	0.606	0.730
	PCS	0	0	0	0	0.225	0.423	0.584	0.711		PCS	0	0	0	0	0.242	0.449	0.606	0.730
	Vanilla SM	0	0	0	0	0.233	0.428	0.590	0.713		Vanilla SM	0	0	0	0	0.255	0.451	0.604	0.727
	Random	0	0	0	0	0.100	0.200	0.299	0.398		Random	0	0	0	0	0.098	0.199	0.299	0.397
MMA	**DNGP**	0	2	0	0	0.320	0.598	0.729	0.785	RAA	**DNGP**	0	1	0	0	0.303	0.573	0.719	0.781
	KMGP	7	5	5	4	0.340	0.578	0.701	0.773		**KMGP**	5	3	2	4	0.308	0.544	0.675	0.755
	LGGP	1	1	0	0	0.327	0.597	0.739	0.809		**LGGP**	6	4	4	3	0.314	0.578	0.734	0.812
	LRGP	0	0	0	0	0.320	0.595	0.729	0.793		**LRGP**	0	1	0	0	0.307	0.574	0.727	0.800
	RFGP	4	4	7	8	0.341	0.598	0.754	0.829		**RFGP**	5	7	10	9	0.315	0.579	0.737	0.821
	XGGP	0	0	0	0	0.319	0.592	0.736	0.804		**XGGP**	0	0	0	0	0.307	0.574	0.730	0.808
	DeepGini	0	0	0	0	0.243	0.426	0.568	0.682		DeepGini	0	0	0	0	0.221	0.395	0.538	0.652
	Entropy	0	0	0	0	0.240	0.412	0.538	0.635		Entropy	0	0	0	0	0.219	0.387	0.518	0.623
	Least Confidence	0	0	0	0	0.263	0.469	0.622	0.741		Least Confidence	0	0	0	0	0.234	0.425	0.582	0.705
	Margin	0	0	0	0	0.253	0.463	0.622	0.743		Margin	0	0	0	0	0.220	0.411	0.570	0.698
	PCS	0	0	0	0	0.253	0.463	0.622	0.743		PCS	0	0	0	0	0.220	0.411	0.570	0.698
	Vanilla SM	0	0	0	0	0.263	0.469	0.622	0.741		Vanilla SM	0	0	0	0	0.234	0.425	0.582	0.705
	Random	0	0	0	0	0.102	0.202	0.303	0.402		Random	0	0	0	0	0.101	0.201	0.301	0.399
NEAA	**DNGP**	0	0	0	0	0.332	0.627	0.783	0.840	RAF	**DNGP**	0	1	0	0	0.295	0.565	0.715	0.780
	KMGP	3	2	2	1	0.335	0.589	0.733	0.805		**KMGP**	7	3	1	3	0.301	0.533	0.673	0.760
	LGGP	0	2	0	1	0.343	0.636	0.803	0.877		**LGGP**	4	5	5	4	0.307	0.568	0.731	0.812
	LRGP	1	0	0	0	0.334	0.630	0.795	0.860		**LRGP**	0	0	0	0	0.298	0.565	0.723	0.798
	RFGP	4	4	6	6	0.345	0.640	0.814	0.884		**RFGP**	5	7	10	9	0.308	0.570	0.736	0.821
	XGGP	0	0	0	0	0.336	0.631	0.800	0.869		**XGGP**	0	0	0	0	0.299	0.565	0.727	0.807
	DeepGini	0	0	0	0	0.245	0.433	0.579	0.694		DeepGini	0	0	0	0	0.218	0.394	0.536	0.650
	Entropy	0	0	0	0	0.240	0.414	0.538	0.632		Entropy	0	0	0	0	0.216	0.385	0.516	0.620
	Least Confidence	0	0	0	0	0.261	0.472	0.638	0.763		Least Confidence	0	0	0	0	0.230	0.422	0.580	0.706
	Margin	0	0	0	0	0.245	0.457	0.625	0.757		Margin	0	0	0	0	0.217	0.409	0.567	0.698
	PCS	0	0	0	0	0.245	0.457	0.625	0.757		PCS	0	0	0	0	0.217	0.409	0.567	0.698
	Vanilla SM	0	0	0	0	0.261	0.472	0.638	0.763		Vanilla SM	0	0	0	0	0.230	0.422	0.580	0.706
	Random	0	0	0	0	0.100	0.200	0.301	0.399		Random	0	0	0	0	0.100	0.202	0.301	0.402
NEAR	**DNGP**	0	0	0	0	0.322	0.618	0.787	0.848	RAR	**DNGP**	0	2	0	0	0.334	0.606	0.720	0.766
	KMGP	1	2	2	1	0.335	0.618	0.780	0.856		**KMGP**	6	1	1	4	0.341	0.568	0.697	0.772
	LGGP	1	0	0	0	0.336	0.620	0.793	0.871		**LGGP**	2	4	4	4	0.347	0.616	0.752	0.814
	LRGP	0	0	0	0	0.326	0.621	0.798	0.866		**LRGP**	1	0	0	0	0.338	0.611	0.740	0.799
	RFGP	2	2	2	3	0.339	0.627	0.810	0.893		**RFGP**	7	8	11	7	0.348	0.617	0.761	0.823
	XGGP	0	0	0	0	0.327	0.620	0.796	0.872		**XGGP**	0	1	0	1	0.339	0.613	0.749	0.810
	DeepGini	0	0	0	0	0.247	0.432	0.576	0.687		DeepGini	0	0	0	0	0.231	0.410	0.551	0.662
	Entropy	0	0	0	0	0.244	0.427	0.569	0.675		Entropy	0	0	0	0	0.229	0.400	0.528	0.627
	Least Confidence	0	0	0	0	0.256	0.458	0.621	0.747		Least Confidence	0	0	0	0	0.247	0.445	0.603	0.723
	Margin	0	0	0	0	0.233	0.431	0.600	0.737		Margin	0	0	0	0	0.243	0.444	0.605	0.727
	PCS	0	0	0	0	0.233	0.431	0.600	0.737		PCS	0	0	0	0	0.243	0.444	0.605	0.727
	Vanilla SM	0	0	0	0	0.256	0.458	0.621	0.747		Vanilla SM	0	0	0	0	0.247	0.445	0.603	0.723
	Random	0	0	0	0	0.101	0.198	0.294	0.391		Random	0	0	0	0	0.099	0.200	0.300	0.401

Table 11. Average effectiveness comparision among GraphPrior and the compared approaches on adversarial test inputs in terms of PFD

Approaches	#Best case in PFD				Average PFD			
	PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
DNGP	0	9	1	0	0.313	0.589	0.732	0.790
KMGP	43	21	18	22	0.325	0.565	0.705	0.783
LGGP	15	16	13	12	0.323	0.592	0.747	0.822
LRGP	2	3	0	0	0.315	0.590	0.740	0.808
RFGP	32	42	60	57	0.328	0.595	0.758	0.837
XGGP	0	1	0	1	0.315	0.589	0.745	0.818
DeepGini	0	0	0	0	0.232	0.412	0.555	0.669
Entropy	0	0	0	0	0.23	0.402	0.533	0.635
Least Confidence	0	0	0	0	0.247	0.446	0.605	0.728
Margin	0	0	0	0	0.235	0.436	0.597	0.725
PCS	0	0	0	0	0.235	0.436	0.597	0.725
Vanilla SM	0	0	0	0	0.247	0.446	0.605	0.728
Random	0	0	0	0	0.101	0.202	0.301	0.399

substantial proportion of adversarial test inputs. Consequently, such circumstances could yield a greater number of bug cases selected by any prioritization method, thereby affecting the evaluation of GraphPrior. Therefore, we carefully selected a range of attack levels that are not unduly high for the evaluation of GraphPrior. In this research question, we totally evaluate GraphPrior and the compared approaches on 432 subjects.

Results: GraphPrior outperforms all the compared approaches on the adversarial test inputs generated from different attack levels. More specifically, Table 12 presents the effectiveness of GraphPrior and the compared approaches under the attacks DICE, MMA, RAA and RAR, with the attack level ranging from 0.1 to 0.4. In this research question, we totally apply 8 adversarial attacks. The remaining experimental results (i.e., results of the other four adversarial attacks) are presented on our Github[2].

The experimental results presented in Table 12 demonstrate that GraphPrior, consisting of DNGP, KMGP, LGGP, LRGP, RFGP and XGGP, outperforms all the compared approaches across different levels of the adversarial attacks.

Table 13 demonstrates the overall comparison results among GraphPrior and the compared approaches across 8 adversarial attacks with differnet attack levels. Specifically, we evaluate the effectiveness of each test prioritization approach in terms of the number of cases where it performed the best, as well as its average PFD values across different attack levels. For example, the "All-0.1" refers to the overall results of each approach under all the adversarial attacks with an attack level of 0.1. Table 13 demonstrates that GraphPrior outperforms all compared approaches, achieving the best effectiveness in 99.94% of the tested cases. Only one best case is achieved by the compared approach margin. Furthermore, GraphPrior approaches such as RFGP and KMGP consistently exhibit the largest average PFD values across different attack levels.

Among all the GraphPrior approaches, RFGP and KMGP exhibit superior performance across different attack levels in comparison to other GraphPrior approaches. In Table 12, we see that, across the attack levels from 0.1 to 0.4, RFGP performs the best in the largest number of best cases, followed by KMGP. For example, when the attack level is 0.1, RFGP performs the best in 46.47% cases. KMGP performs the best in 35.33% cases. Notably, when prioritizing 10% test inputs, KMGP takes the largest number of best cases. When the attack level is 0.2~0.4, RFGP takes the largest number of best cases.

Additionally, our experimental results, as illustrated in Table 13, reveal that the RFGP technique exhibits the largest average PFD values when compared to the other evaluated approaches across varying attack levels. Specifically, when 40% of the test inputs are prioritized, RFGP achieves a PFD value ranging from 0.832 to 0.836, which indicates the ability to detect more than 80% of misclassified tests.

> **Answer to RQ4:** *GraphPrior outperforms all the compared approaches on the adversarial test inputs generated from different attack levels. Among all the GraphPrior approaches, RFGP and KMGP exhibit superior performance across different attack levels in comparison to other GraphPrior approaches.*

5.5 RQ5: Contribution analysis of different mutation rules

Objectives: For each evaluated GNN model, we investigate which mutated rules generate more top contributing mutated models for test prioritization.

Experimental design: In our study, we employed one or more mutation rules to generate a mutated model. Each mutated model corresponds to one mutation feature. Thus, to evaluate the importance of different mutation rules, we initially evaluate the importance of various mutation features. We adopted the cover metric of the XGBoost algorithm to identify the importance of each mutation feature for ranking models. A detailed account of this approach is presented in Section 4.5. After computing the importance scores of all the mutated features, we

[2] https://github.com/yinghuali/GraphPrior/tree/main/mutation/adv_res

Table 12. Comparison results of GraphPrior and the compared approaches against different levels of the attacks DICE, MMA, RAA and RAR in terms of PFD

Attack	Approaches	#Best cases in PFD				Average PFD				Attack	Approaches	#Best cases in PFD				Average PFD			
		PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40			PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
DICE-0.1	**DNGP**	0	1	0	0	0.322	0.595	0.724	0.775	RAA-0.1	**DNGP**	0	0	0	0	0.333	0.604	0.725	0.774
	KMGP	7	3	2	5	0.335	0.568	0.700	0.776		**KMGP**	4	3	5	5	0.336	0.574	0.696	0.770
	LGGP	1	0	1	0	0.335	0.600	0.745	0.811		**LGGP**	3	6	4	4	0.343	0.611	0.751	0.814
	LRGP	0	0	0	0	0.323	0.597	0.736	0.797		**LRGP**	1	0	0	0	0.337	0.607	0.743	0.803
	RFGP	4	7	9	7	0.338	0.605	0.757	0.828		**RFGP**	8	7	7	7	0.345	0.613	0.757	0.822
	XGGP	0	1	0	0	0.325	0.599	0.743	0.810		**XGGP**	0	0	0	0	0.338	0.608	0.749	0.813
	DeepGini	0	0	0	0	0.237	0.419	0.559	0.674		DeepGini	0	0	0	0	0.232	0.410	0.549	0.660
	Entropy	0	0	0	0	0.233	0.405	0.528	0.627		Entropy	0	0	0	0	0.230	0.399	0.527	0.626
	Least Confidence	0	0	0	0	0.256	0.459	0.616	0.736		Least Confidence	0	0	0	0	0.248	0.445	0.602	0.722
	Margin	0	0	0	0	0.245	0.451	0.613	0.737		Margin	0	0	0	0	0.236	0.438	0.597	0.720
	PCS	0	0	0	0	0.245	0.451	0.613	0.737		PCS	0	0	0	0	0.236	0.438	0.597	0.720
	Vanilla SM	0	0	0	0	0.256	0.459	0.616	0.736		Vanilla SM	0	0	0	0	0.248	0.445	0.602	0.722
	Random	0	0	0	0	0.098	0.198	0.296	0.397		Random	0	0	0	0	0.100	0.200	0.301	0.401
DICE-0.2	**DNGP**	0	0	0	0	0.305	0.573	0.713	0.772	RAA-0.2	**DNGP**	0	0	0	0	0.311	0.584	0.717	0.773
	KMGP	6	3	2	3	0.314	0.545	0.678	0.762		**KMGP**	6	5	4	5	0.318	0.553	0.683	0.762
	LGGP	1	2	0	1	0.314	0.576	0.732	0.807		**LGGP**	4	5	3	3	0.323	0.587	0.736	0.807
	LRGP	0	0	0	0	0.304	0.575	0.724	0.795		**LRGP**	1	1	0	0	0.314	0.586	0.729	0.796
	RFGP	5	6	10	8	0.316	0.579	0.741	0.820		**RFGP**	5	5	9	8	0.324	0.590	0.744	0.818
	XGGP	0	1	0	0	0.305	0.574	0.729	0.804		**XGGP**	0	0	0	0	0.314	0.586	0.734	0.805
	DeepGini	0	0	0	0	0.228	0.409	0.552	0.667		DeepGini	0	0	0	0	0.223	0.397	0.540	0.653
	Entropy	0	0	0	0	0.225	0.395	0.522	0.624		Entropy	0	0	0	0	0.221	0.388	0.519	0.620
	Least Confidence	0	0	0	0	0.244	0.443	0.602	0.724		Least Confidence	0	0	0	0	0.237	0.431	0.588	0.713
	Margin	0	0	0	0	0.235	0.435	0.596	0.723		Margin	0	0	0	0	0.226	0.422	0.582	0.709
	PCS	0	0	0	0	0.235	0.435	0.596	0.723		PCS	0	0	0	0	0.226	0.422	0.582	0.709
	Vanilla SM	0	0	0	0	0.244	0.443	0.602	0.724		Vanilla SM	0	0	0	0	0.237	0.431	0.588	0.713
	Random	0	0	0	0	0.101	0.202	0.302	0.401		Random	0	0	0	0	0.099	0.199	0.298	0.398
DICE-0.3	**DNGP**	0	2	0	0	0.289	0.553	0.705	0.769	RAA-0.3	**DNGP**	0	1	0	0	0.303	0.573	0.719	0.781
	KMGP	7	1	2	2	0.300	0.520	0.665	0.754		**KMGP**	5	3	2	4	0.308	0.544	0.675	0.755
	LGGP	1	0	0	0	0.301	0.557	0.719	0.801		**LGGP**	6	4	4	3	0.314	0.578	0.734	0.812
	LRGP	0	0	0	0	0.291	0.555	0.711	0.788		**LRGP**	0	1	0	0	0.307	0.574	0.727	0.800
	RFGP	4	9	10	10	0.304	0.561	0.729	0.818		**RFGP**	5	7	10	9	0.315	0.579	0.737	0.821
	XGGP	0	0	0	0	0.293	0.556	0.716	0.799		**XGGP**	0	0	0	0	0.307	0.574	0.730	0.808
	DeepGini	0	0	0	0	0.215	0.394	0.535	0.655		DeepGini	0	0	0	0	0.221	0.395	0.538	0.652
	Entropy	0	0	0	0	0.212	0.381	0.507	0.611		Entropy	0	0	0	0	0.219	0.387	0.518	0.623
	Least Confidence	0	0	0	0	0.233	0.428	0.590	0.713		Least Confidence	0	0	0	0	0.234	0.425	0.582	0.705
	Margin	0	0	0	0	0.225	0.423	0.584	0.711		Margin	0	0	0	0	0.220	0.411	0.570	0.698
	PCS	0	0	0	0	0.225	0.423	0.584	0.711		PCS	0	0	0	0	0.220	0.411	0.570	0.698
	Vanilla SM	0	0	0	0	0.233	0.428	0.590	0.713		Vanilla SM	0	0	0	0	0.234	0.425	0.582	0.705
	Random	0	0	0	0	0.100	0.200	0.299	0.398		Random	0	0	0	0	0.101	0.201	0.301	0.399
DICE-0.4	**DNGP**	0	1	2	0	0.276	0.532	0.694	0.770	RAA-0.4	**DNGP**	0	1	0	0	0.290	0.554	0.713	0.783
	KMGP	7	2	1	1	0.288	0.510	0.647	0.740		**KMGP**	6	3	1	4	0.294	0.525	0.671	0.761
	LGGP	0	1	1	1	0.286	0.535	0.702	0.799		**LGGP**	4	5	3	3	0.300	0.559	0.726	0.812
	LRGP	0	0	1	0	0.277	0.533	0.699	0.785		**LRGP**	0	1	0	0	0.293	0.556	0.720	0.800
	RFGP	5	8	7	10	0.291	0.538	0.708	0.812		**RFGP**	6	6	12	9	0.302	0.560	0.731	0.823
	XGGP	0	0	0	0	0.280	0.532	0.700	0.795		**XGGP**	0	0	0	0	0.294	0.556	0.724	0.809
	DeepGini	0	0	0	0	0.211	0.388	0.533	0.654		DeepGini	0	0	0	0	0.215	0.392	0.535	0.650
	Entropy	0	0	0	0	0.209	0.376	0.508	0.613		Entropy	0	0	0	0	0.213	0.384	0.517	0.621
	Least Confidence	0	0	0	0	0.226	0.419	0.579	0.708		Least Confidence	0	0	0	0	0.226	0.418	0.576	0.702
	Margin	0	0	0	0	0.215	0.406	0.568	0.701		Margin	0	0	0	0	0.210	0.399	0.559	0.689
	PCS	0	0	0	0	0.215	0.406	0.568	0.701		PCS	0	0	0	0	0.210	0.399	0.559	0.689
	Vanilla SM	0	0	0	0	0.226	0.419	0.579	0.708		Vanilla SM	0	0	0	0	0.226	0.418	0.576	0.702
	Random	0	0	0	0	0.098	0.200	0.300	0.400		Random	0	0	0	0	0.098	0.200	0.300	0.400
MMA-0.1	**DNGP**	0	1	0	0	0.329	0.611	0.733	0.781	RAR-0.1	**DNGP**	0	0	0	0	0.342	0.613	0.721	0.767
	KMGP	7	5	4	4	0.346	0.583	0.708	0.776		**KMGP**	6	3	3	7	0.348	0.583	0.704	0.774
	LGGP	0	1	2	1	0.336	0.609	0.748	0.810		**LGGP**	4	4	4	1	0.352	0.621	0.753	0.809
	LRGP	0	0	0	0	0.330	0.608	0.738	0.800		**LRGP**	1	0	0	0	0.342	0.616	0.740	0.792
	RFGP	5	5	6	7	0.349	0.614	0.763	0.833		**RFGP**	5	9	8	8	0.357	0.624	0.760	0.817
	XGGP	0	0	0	0	0.329	0.607	0.746	0.808		**XGGP**	0	0	1	0	0.345	0.620	0.748	0.806
	DeepGini	0	0	0	0	0.246	0.428	0.568	0.682		DeepGini	0	0	0	0	0.240	0.416	0.552	0.662
	Entropy	0	0	0	0	0.241	0.413	0.536	0.634		Entropy	0	0	0	0	0.238	0.404	0.527	0.626
	Least Confidence	0	0	0	0	0.269	0.472	0.627	0.745		Least Confidence	0	0	0	0	0.257	0.455	0.609	0.726
	Margin	0	0	0	0	0.257	0.467	0.627	0.749		Margin	0	0	0	0	0.248	0.451	0.609	0.729
	PCS	0	0	0	0	0.257	0.467	0.627	0.749		PCS	0	0	0	0	0.249	0.451	0.609	0.729
	Vanilla SM	0	0	0	0	0.269	0.472	0.627	0.745		Vanilla SM	0	0	0	0	0.257	0.455	0.609	0.726
	Random	0	0	0	0	0.099	0.198	0.296	0.396		Random	0	0	0	0	0.100	0.200	0.301	0.401
MMA-0.2	**DNGP**	0	0	0	0	0.328	0.605	0.725	0.775	RAR-0.2	**DNGP**	0	1	0	0	0.341	0.614	0.723	0.771
	KMGP	6	5	5	4	0.344	0.581	0.705	0.774		**KMGP**	7	3	0	6	0.346	0.579	0.701	0.775
	LGGP	1	2	1	0	0.336	0.605	0.741	0.805		**LGGP**	4	3	3	3	0.353	0.619	0.751	0.812
	LRGP	0	0	0	0	0.331	0.603	0.734	0.794		**LRGP**	1	1	0	0	0.342	0.615	0.740	0.795
	RFGP	5	5	6	7	0.349	0.610	0.758	0.829		**RFGP**	4	7	13	7	0.356	0.623	0.762	0.822
	XGGP	0	0	0	0	0.330	0.601	0.738	0.802		**XGGP**	0	1	0	0	0.343	0.618	0.748	0.805
	DeepGini	0	0	0	0	0.248	0.431	0.573	0.686		DeepGini	0	0	0	0	0.237	0.411	0.550	0.659
	Entropy	0	0	0	0	0.245	0.417	0.541	0.639		Entropy	0	0	0	0	0.234	0.401	0.526	0.624
	Least Confidence	0	0	0	0	0.267	0.473	0.629	0.746		Least Confidence	0	0	0	0	0.250	0.449	0.604	0.725
	Margin	0	0	0	1	0.255	0.466	0.626	0.746		Margin	0	0	0	0	0.244	0.447	0.605	0.728
	PCS	0	0	0	0	0.255	0.466	0.626	0.746		PCS	0	0	0	0	0.244	0.447	0.605	0.728
	Vanilla SM	0	0	0	0	0.267	0.473	0.629	0.746		Vanilla SM	0	0	0	0	0.250	0.449	0.604	0.725
	Random	0	0	0	0	0.099	0.199	0.297	0.401		Random	0	0	0	0	0.098	0.199	0.297	0.397
MMA-0.3	**DNGP**	0	2	0	0	0.320	0.598	0.729	0.785	RAR-0.3	**DNGP**	0	2	0	0	0.334	0.606	0.720	0.766
	KMGP	7	5	5	4	0.340	0.578	0.701	0.773		**KMGP**	6	1	1	4	0.341	0.568	0.697	0.772
	LGGP	1	1	0	0	0.327	0.597	0.739	0.809		**LGGP**	2	4	4	4	0.347	0.616	0.752	0.814
	LRGP	0	0	0	0	0.320	0.595	0.729	0.793		**LRGP**	1	0	0	0	0.338	0.611	0.740	0.799
	RFGP	4	4	7	8	0.341	0.598	0.754	0.829		**RFGP**	7	8	11	7	0.348	0.617	0.761	0.823
	XGGP	0	0	0	0	0.319	0.592	0.736	0.804		**XGGP**	0	1	0	1	0.339	0.613	0.749	0.810
	DeepGini	0	0	0	0	0.243	0.426	0.568	0.682		DeepGini	0	0	0	0	0.231	0.410	0.551	0.662
	Entropy	0	0	0	0	0.240	0.412	0.538	0.635		Entropy	0	0	0	0	0.229	0.400	0.528	0.627
	Least Confidence	0	0	0	0	0.263	0.469	0.622	0.741		Least Confidence	0	0	0	0	0.247	0.445	0.603	0.723
	Margin	0	0	0	0	0.253	0.463	0.622	0.743		Margin	0	0	0	0	0.243	0.444	0.605	0.727
	PCS	0	0	0	0	0.253	0.463	0.622	0.743		PCS	0	0	0	0	0.243	0.444	0.605	0.727
	Vanilla SM	0	0	0	0	0.263	0.469	0.622	0.741		Vanilla SM	0	0	0	0	0.247	0.445	0.603	0.723
	Random	0	0	0	0	0.102	0.202	0.303	0.402		Random	0	0	0	0	0.099	0.200	0.300	0.401
MMA-0.4	**DNGP**	0	2	0	0	0.322	0.601	0.732	0.787	RAR-0.4	**DNGP**	0	1	0	0	0.333	0.607	0.723	0.770
	KMGP	7	6	5	5	0.345	0.582	0.711	0.778		**KMGP**	7	1	1	3	0.337	0.564	0.692	0.771
	LGGP	1	3	2	0	0.331	0.598	0.739	0.807		**LGGP**	3	5	4	3	0.341	0.611	0.749	0.815
	LRGP	0	0	0	0	0.324	0.597	0.728	0.790		**LRGP**	0	2	0	0	0.335	0.609	0.738	0.799
	RFGP	4	1	5	7	0.345	0.598	0.754	0.827		**RFGP**	6	7	10	10	0.345	0.613	0.758	0.824
	XGGP	0	0	0	0	0.323	0.592	0.731	0.797		**XGGP**	0	0	1	0	0.335	0.609	0.745	0.809
	DeepGini	0	0	0	0	0.245	0.427	0.568	0.681		DeepGini	0	0	0	0	0.233	0.406	0.547	0.658
	Entropy	0	0	0	0	0.242	0.413	0.539	0.636		Entropy	0	0	0	0	0.231	0.398	0.524	0.623
	Least Confidence	0	0	0	0	0.265	0.469	0.624	0.741		Least Confidence	0	0	0	0	0.248	0.439	0.596	0.719
	Margin	0	0	0	0	0.253	0.464	0.625	0.744		Margin	0	0	0	0	0.243	0.443	0.600	0.723
	PCS	0	0	0	0	0.253	0.464	0.625	0.744		PCS	0	0	0	0	0.243	0.443	0.600	0.723
	Vanilla SM	0	0	0	0	0.265	0.469	0.624	0.741		Vanilla SM	0	0	0	0	0.248	0.439	0.596	0.719
	Random	0	0	0	0	0.098	0.201	0.300	0.399		Random	0	0	0	0	0.097	0.199	0.299	0.398

selected the top-N important features for each subject and subsequently identified the top-N mutated models. We then identified the mutation rules utilized to generate each mutated model and compared the contributions of the mutation rules accordingly. Additionally, for different subjects in this research question, we generate 80~240 mutated models.

Results: The mutation rule HC made high contributions to the effectiveness of GraphPrior on all the four types of GNN models. Table 14 to Table 17 illustrate the contributions of different mutation rules to the effectiveness of GraphPrior on different GNN models (i.e., GCN, GAT, GraphSAGE and TAGCN). For each GNN model, we identify the top-N mutated models that made top contributions to the effectiveness of GraphPrior. The corresponding mutation rules applied to generate each mutated model are highlighted in grey. Table 14 presents the contributions of Top-N mutated models to the effectiveness of GraphPrior for the case of GCN model. Notably, the mutation rules **BIA** and **HC** made contributions to 100% of the top contributing mutated models, while **SL**, **NOR**, **CA**, and **IMP** contributed to a lower percentage of the top contributing mutated models. We

Table 13. Overall comparison results among GraphPrior and the compared approaches on adversarial tests with different attack levels

Attack Level	Approaches	#Best case in PFD				Average PFD			
		PFD-10	PFD-20	PFD-30	PFD-40	PFD-10	PFD-20	PFD-30	PFD-40
All-0.1	**DNGP**	0	3	0	0	0.334	0.615	0.738	0.784
	KMGP	42	27	28	33	0.349	0.594	0.723	0.791
	LGGP	13	18	14	11	0.346	0.619	0.760	0.820
	LRGP	3	0	0	0	0.336	0.617	0.752	0.808
	RFGP	34	43	48	46	0.352	0.624	0.772	0.836
	XGGP	0	1	2	1	0.336	0.617	0.758	0.819
	DeepGini	0	0	0	0	0.243	0.425	0.566	0.679
	Entropy	0	0	0	0	0.241	0.413	0.541	0.642
	Least Confidence	0	0	0	0	0.261	0.465	0.623	0.742
	Margin	0	0	0	1	0.249	0.457	0.619	0.742
	PCS	0	0	0	0	0.249	0.457	0.619	0.742
	Vanilla SM	0	0	0	0	0.261	0.465	0.623	0.742
	Random	0	0	0	0	0.099	0.200	0.301	0.402
All-0.2	**DNGP**	0	2	0	0	0.323	0.602	0.734	0.786
	KMGP	44	29	20	29	0.335	0.580	0.713	0.786
	LGGP	13	19	10	10	0.332	0.604	0.753	0.820
	LRGP	2	3	0	0	0.323	0.602	0.745	0.806
	RFGP	33	37	62	52	0.339	0.608	0.765	0.836
	XGGP	0	2	0	0	0.323	0.602	0.750	0.816
	DeepGini	0	0	0	0	0.238	0.419	0.561	0.675
	Entropy	0	0	0	0	0.235	0.408	0.538	0.640
	Least Confidence	0	0	0	0	0.254	0.456	0.614	0.736
	Margin	0	0	0	1	0.241	0.446	0.609	0.734
	PCS	0	0	0	0	0.241	0.446	0.609	0.734
	Vanilla SM	0	0	0	0	0.254	0.456	0.614	0.736
	Random	0	0	0	0	0.099	0.199	0.299	0.399
All-0.3	**DNGP**	0	9	1	0	0.313	0.589	0.732	0.790
	KMGP	43	21	18	22	0.324	0.565	0.704	0.783
	LGGP	15	16	13	12	0.322	0.591	0.747	0.822
	LRGP	2	3	0	0	0.314	0.590	0.740	0.808
	RFGP	32	42	60	57	0.328	0.595	0.758	0.836
	XGGP	0	1	0	1	0.315	0.588	0.744	0.817
	DeepGini	0	0	0	0	0.232	0.412	0.554	0.669
	Entropy	0	0	0	0	0.229	0.401	0.532	0.635
	Least Confidence	0	0	0	0	0.247	0.446	0.605	0.728
	Margin	0	0	0	0	0.234	0.435	0.597	0.725
	PCS	0	0	0	0	0.234	0.435	0.597	0.725
	Vanilla SM	0	0	0	0	0.247	0.446	0.605	0.728
	Random	0	0	0	0	0.100	0.200	0.299	0.398
All-0.4	**DNGP**	0	8	3	0	0.306	0.578	0.727	0.790
	KMGP	43	23	15	23	0.316	0.554	0.694	0.776
	LGGP	13	20	13	11	0.314	0.580	0.739	0.819
	LRGP	2	3	1	0	0.307	0.577	0.732	0.805
	RFGP	34	38	58	58	0.320	0.581	0.748	0.832
	XGGP	0	0	2	0	0.307	0.576	0.735	0.813
	DeepGini	0	0	0	0	0.228	0.408	0.552	0.669
	Entropy	0	0	0	0	0.226	0.399	0.532	0.636
	Least Confidence	0	0	0	0	0.242	0.439	0.599	0.724
	Margin	0	0	0	0	0.227	0.426	0.588	0.717
	PCS	0	0	0	0	0.227	0.426	0.588	0.717
	Vanilla SM	0	0	0	0	0.242	0.439	0.599	0.724
	Random	0	0	0	0	0.097	0.199	0.299	0.398

conclude that, for the GCN model, the mutation rules **SL** and **HC** were the most effective in generating the top important mutated models. Moving to GAT, GraphSAGE, and TAGCN, whose results are presented in Table 15, Table 16, and Table 17, the mutation rule HC also generates a large ratio (i.e., 100%, 90%, and 90% respectively) of top contributing mutated models. We can conclude that, across the four different types of GNN models, HC can continuously make top contributions to the effectiveness of GraphPrior.

Some mutated rules, such as NOR and BIA, made high contributions to the effectiveness of GraphPrior on some specific GNN models. Moreover, some mutation rules, such as BIA and NOR, also generate a considerable ratio (i.e., from 50% to 100%) of top-critical mutated models. For example, on GCN and GraphSAGE, BIA made contributions to 100% top-N mutated models. On TAGCN, NOR made contributions to 100% top-N mutated models.

> **Answer to RQ5:** *The mutation rule HC made high contributions to the effectiveness of GraphPrior on all the four types of GNN models. Some mutated rules, such as NOR and BIA made high contributions to the effectiveness of GraphPrior on some specific GNN models.*

Table 14. **The contributions of different mutation rules (GCN)**

Top-N	SL	BIA	CA	IMP	NOR	HC
0	✓	✓				✓
1	✓	✓	✓	✓		✓
2		✓	✓	✓		✓
3		✓				✓
4	✓	✓		✓	✓	✓
5	✓	✓	✓	✓	✓	✓
6	✓	✓			✓	✓
7	✓	✓	✓		✓	✓
8	✓	✓	✓			✓
9	✓	✓			✓	✓

Table 15. **The contributions of different mutation rules (GAT)**

Top-N	SL	BIA	CON	HDS	EP	NS	HC
0	✓	✓	✓	✓	✓	✓	✓
1	✓		✓	✓	✓		✓
2			✓		✓		✓
3	✓	✓	✓	✓		✓	✓
4	✓	✓	✓		✓	✓	✓
5		✓	✓		✓		✓
6			✓		✓	✓	✓
7			✓	✓		✓	✓
8		✓	✓		✓	✓	✓
9		✓	✓	✓	✓		✓

Table 16. **The contributions of different mutation rules (GraphSAGE)**

Top-N	BIA	NOR	HC	EP
0	✓	✓	✓	✓
1	✓	✓	✓	
2	✓	✓	✓	✓
3	✓	✓	✓	
4	✓	✓	✓	
5	✓	✓	✓	
6	✓		✓	✓
7	✓	✓	✓	
8	✓	✓	✓	✓
9	✓	✓	✓	

Table 17. **The contributions of different mutation rules to the (TAGCN)**

Top-N	NOR	HC	EP
0	✓		
1	✓	✓	✓
2	✓	✓	✓
3	✓	✓	
4	✓	✓	✓
5	✓	✓	
6	✓	✓	✓
7	✓	✓	✓
8	✓	✓	
9	✓	✓	✓

5.6 RQ6: Enhancing GNNs with GraphPrior

Objectives: We investigate whether GraphPrior and the uncertainty-based metrics can select informative retraining subsets to improve the performance of a GNN model.

Experimental design: Following the prior research by Ma *et al.* [57], our retraining experiments are structured as follows. Firstly, we randomly partitioned the dataset into three sets: an initial training set, a candidate set, and a test set, with a ratio of 4:4:2. The candidate set was reserved exclusively for retraining purposes, while the test set was kept untouched for the purpose of evaluation. In the first round, we trained a GNN model using only the initial training set and computed its accuracy on the test set. We employed the best model obtained over the training epochs for the subsequent retraining process. In the second round, we incorporate an additional 10% of new inputs from the candidate set into the existing training set without replacement. The inputs selected for inclusion are those that are prioritized in the first 10% by the test prioritization approaches, namely GraphPrior and the compared techniques. Following Ma *et al.* [57], we retrain the GNN models by utilizing the complete augmented training set. This approach ensures that the old and new training data are treated equally. We repeat the retraining process for multiple rounds until the candidate set is empty. We kept the test data untouched during the retraining process. Moreover, we account for the randomness involved in the model training process and repeat all the experiments ten times to report the average results (averaged over ten repetitions).

Results: Table 18 illustrates the average accuracy of GNN models after retraining with 10% to 100% prioritized test inputs. For each case, we highlight the approach with the highest effectiveness in grey to facilitate quick and easy interpretation of the results. **GraphPrior and the uncertainty-based test prioritization approaches outperform the random selection approach. However, the observed improvement is relatively small,**

Table 18. The GNNs' average accuracy value after retraining with 10%~100% prioritized tests.

Approaches	Accuracy of percentage of datasets										Average
	10%	20%	30%	40%	50%	60%	70%	80%	90%	100%	
KMGP	0.787	0.810	0.825	0.834	0.844	0.854	0.861	0.867	0.874	0.878	0.844
DNGP	0.787	0.811	0.827	0.836	0.844	0.853	0.859	0.867	0.873	0.877	0.843
LGGP	0.787	0.812	0.825	0.835	0.845	0.852	0.861	0.868	0.873	0.877	0.844
LRGP	0.787	0.811	0.825	0.835	0.845	0.854	0.864	0.869	0.873	0.877	0.844
XGGP	0.788	0.811	0.824	0.834	0.845	0.853	0.861	0.867	0.873	0.877	0.843
RFGP	0.787	0.813	0.825	0.835	0.845	0.853	0.860	0.869	0.874	0.877	0.844
DeepGini	0.788	0.801	0.814	0.826	0.836	0.844	0.851	0.858	0.866	0.870	0.835
Entropy	0.789	0.801	0.816	0.829	0.836	0.845	0.852	0.858	0.866	0.872	0.837
LeastConfidence	0.789	0.802	0.816	0.828	0.836	0.846	0.853	0.860	0.866	0.872	0.837
Margin	0.788	0.801	0.818	0.827	0.837	0.845	0.853	0.861	0.867	0.872	0.837
VanillaSM	0.788	0.804	0.819	0.829	0.837	0.846	0.853	0.861	0.867	0.873	0.838
PCS	0.787	0.802	0.817	0.827	0.837	0.845	0.854	0.860	0.866	0.872	0.837
Random	0.789	0.799	0.814	0.825	0.834	0.843	0.853	0.860	0.866	0.872	0.836

indicating that GNN test prioritization approaches can guide the retraining of GNN models but with limited effect. In Table 18, we observe that test prioritization methods, including GraphPrior and compared approaches, consistently demonstrate better performance across varying ratios of added data compared with the random selection. Furthermore, when incorporating prioritized tests exceeding 10% of the total, a significant majority of the test prioritization methods - specifically, 83.4% (10 out of 12) - outperform random selection in each case. However, the improvements achieved by these test prioritization methods compared to random selection are relatively small, with the highest increase being only 0.014. Additionally, Figure 4 visually depicts an example outcome of the retraining experiments conducted on the Cora dataset using the GCN model, showcasing a comparative evaluation of the performance of test prioritization approaches against random selection (indicated by the black line). As observed from the results, the test prioritization approaches demonstrate a better performance compared to random selection, but the improvement is visually slight.

One reason that leads to the effectiveness of GraphPrior and uncertainty-based test prioritization approaches being limited lies in their inadequate consideration of node importance (i.e., impact on other nodes in the dataset). In a GNN dataset, the complex interdependence among test inputs and their neighbors can lead to them having different importance. For example, nodes with greater connectivity can affect more of other nodes, making them relatively more critical. However, the current test prioritization approaches only focus on the ability of test inputs to reveal system bugs without regard to the importance of nodes. Although the selected test input by them can have a higher likelihood of misclassification, their importance within the dataset can be minor if they have a very small number of neighbors. Retraining such inputs would have less effect. Consequently, it is crucial to consider node importance in the selection of retraining data to achieve more effective outcomes.

GraphPrior achieved better effectiveness than the uncertainty-based test prioritization methods. In Table 18, we see that, when adding more than 20% (including 20%) test cases for retraining, the GraphPrior approaches perform the best in 100% cases. Figure 4 visually demonstrates that the GraphPrior approaches (solid line) perform better than the compared approaches (dotted line) in most cases.

Answer to RQ6: *GraphPrior and the uncertainty-based test prioritization approaches outperform the random selection approach. However, the observed improvement is relatively small, indicating that GNN test prioritization approaches can guide the retraining of GNN models but with limited effect. GraphPrior achieved better effectiveness than the uncertainty-based test prioritization methods.*

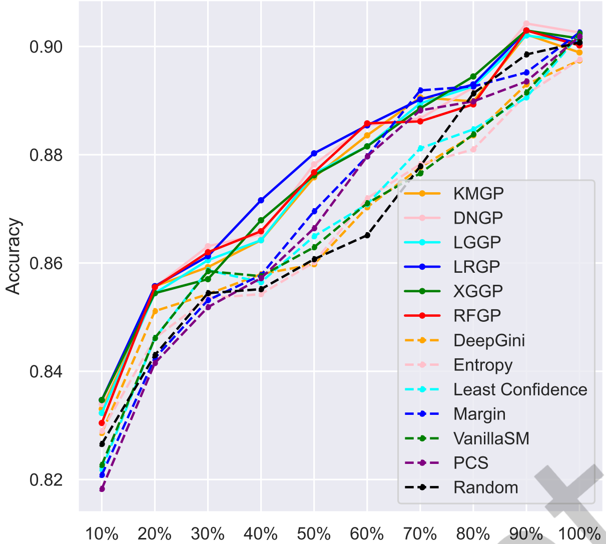


Fig. 4. Enhancing the accuracy of the GNN with prioritized tests (Cora with GCN)

6 DISCUSSION

6.1 Generality of GraphPrior

Although the confidence-based test prioritization approaches demonstrate excellent effectiveness in traditional DNNs, they do not consider the interdependencies between test inputs, which are particularly crucial in GNN test prioritization. Our proposed GraphPrior leverages the mutation analysis of GNN models to perform GNN test input prioritization, which has been demonstrated effective on graph classification tasks through 604 carefully designed subjects. In fact, the scheme of GraphPrior, (i.e., modifying training parameters to mutate the GNN model for test prioritization) can also be generalized to other dimensions of GNN tasks, including graph-level and edge-level tasks. In the future, we will further verify the extension of GraphPrior from this perspective.

[*The applicability of GraphPrior on regression tasks*] In this section, we will also discuss the potential applicability of GraphPrior to regression tasks. Currently, the mutation rules and ranking models of GraphPrior are specifically designed for classification tasks. To extend GraphPrior to regression tasks, modifications to the mutation rules and ranking models would be required. If appropriate mutation rules can be identified for regression tasks and suitable ranking models can be designed, GraphPrior could also be applied to regression tasks.

6.2 Limitations of GraphPrior

[*Diversity of the prioritized data*] One limitation of GraphPrior lies in guaranteeing the diversity of selected bug data. This limitation is also noted in prior work on the uncertainty-based test prioritization approaches [26], which did not consider the diversity of bugs when prioritizing test inputs. Similarly, GraphPrior also does not aim for diversity in the prioritized tests. However, GraphPrior has demonstrated the ability to identify a significant majority of misclassified test inputs using a small ratio of prioritized test cases. Specifically, RFGP (i.e., the most effective GraphPrior approach) has been shown to detect over 80% misclassified tests by prioritizing only 40% of the test inputs. This highlights GraphPrior's ability to efficiently identify a large proportion of bugs using a small set of prioritized tests, even without explicitly ensuring bug diversity. While prioritizing diverse bugs can improve the overall quality of testing, prioritizing a significant majority of bugs can still be a practical strategy in situations where time and resources are limited. Therefore, GraphPrior's ability to efficiently identify a large proportion of bugs using a small set of prioritized tests can be particularly useful in scenarios where time and resources are constrained.

[*GraphPrior in active learning scenarios.*] Active learning [68] operates under the assumption that samples within a dataset have varying contributions to the improvement of the current model and aims to select the most informative samples for inclusion in the training set. Our investigation in RQ6 has demonstrated that GraphPrior and uncertainty-based metrics can be utilized to select informative retraining tests. However, the effectiveness of these approaches is limited. Specifically, despite the demonstrated success of uncertainty-based metrics such as DeepGini and margin in previous studies [26] [36] on DNNs, their effectiveness in the context of GNNs is slight. We explore potential reasons for this phenomenon.

One crucial reason for their limited effectiveness lies in their inadequate consideration of node importance, i.e., the impact that a node has on other nodes in the graph dataset. In a GNN dataset, the complex interdependence among test inputs and their neighbors can result in differing levels of importance for different nodes. For instance, nodes with higher connectivity can be more influential and hence more critical. However, current test prioritization approaches only focus on the ability of test inputs to expose system bugs without taking into account the node importance. Although these approaches may identify inputs with a higher likelihood of misclassification, their importance within the dataset may be negligible if they have only a few neighbors. Retraining such inputs is, therefore, less effective.

Furthermore, we elaborate on the difference between GraphPrior and the existing active learning methods evaluated in our study. The active learning methods used for comparison in our paper are primarily uncertainty-based, aimed at datasets where each sample is independent of others. However, for graph datasets, these methods select retraining data without considering the interdependencies between nodes and also neglect the importance of nodes, merely selecting possibly-misclassifed nodes. In contrast, GraphPrior employs mutation analysis to identify test inputs that are more likely to be misclassified while considering the interdependencies between nodes during the mutation process. Despite this added consideration, GraphPrior's goal remains to select misclassified test inputs and does not explicitly consider node importance, leading to slight effectiveness as the uncertainty-based methods.

[*Generating mutants for large-scale GNN models*] In our experiments, which are based on our current model and datasets, the time cost of our retraining method (for generating mutants) is within an acceptable range. When dealing with large-scale GNN models, GraphPrior can require large computational resources, but it can remain feasible in situations where the cost of manual labeling outweighs the computational cost.

6.3 Threats to Validity

Threats to Internal Validity. The internal threats to validity mainly lie in the implementation of our proposed GraphPrior and the compared approaches. To reduce the threat, we implemented GraphPrior based on the widely used library PyTorch and adopted the implementations of the compared approaches published by their authors. Another internal threat lies in the randomness of the model training. To mitigate this threat and ensure the stability of our experimental results, we conducted a statistical analysis. Specifically, we repeated the training process ten times for both the original model and the mutated model and calculated the statistical significance of the experiments.

The selection of mutation rules in our study presents another internal threat to validity. Despite our best efforts to collect a comprehensive set of mutation rules, it is possible that other training parameters beyond our current knowledge could serve as mutation rules. To mitigate this threat, we selected mutation rules that can directly or indirectly affect node interdependence in the prediction process. The selection of parameter ranges for mutation rules is another internal threat that could affect the effectiveness of the rules. To mitigate this threat, we adopted a strategy in which we inverted the values of Boolean parameters, setting true to false and false to true. For integer and float parameters, we selected a range that introduces only slight changes to the original GNN model. Our experimental results demonstrated the effectiveness of GraphPrior, indicating that the mutation rules and selected parameter range are suitable for GNN test prioritization.

Threats to External Validity. The external threats to validity mainly lie in the GNN models under test and the testing datasets we used in our study. To mitigate this threat, we adopted a large number of subjects (pair of model and dataset) in our study and leveraged different types of test inputs. We applied 8 graph adversarial attacks from public studies to generate adversarial test inputs and varied the attack level for more detailed evaluation. In the future, we will apply GraphPrior to more GNN models and test datasets with diversity.

7 RELATED WORK

We present the related work in three aspects, which are test prioritization techniques, deep neural network testing, and mutation-based test prioritization for traditional software.

7.1 Test prioritization Techniques

In traditional software testing, test prioritization [11–13, 19, 20, 33, 69, 92] aims to find the ideal order of test cases to reveal system bugs earlier. Prioritizing test cases contributes to two critical constraints, time and budget for software testing, in order to detect more fault-revealing test cases in a limited time. Di Nardo *et al.* [19] conducted a case study of coverage-based prioritization strategies on real-world regression faults, evaluating the effectiveness of several test case prioritization techniques in bug detection. Rothermel *et al.* [69] presented and compared three types of test case prioritization techniques for regression testing that are based on test execution information. They demonstrated that each of the studied prioritization techniques increased the fault detection rate of the test suite. Henard *et al.* [33] conducted a comprehensive study to compare existing test prioritization approaches, finding that the difference between white-box [23, 24, 49, 92] and black-box strategies [32, 34, 46] are little. Chen *et al.* [13] proposed LET to prioritize test programs for compiler testing acceleration and demonstrated its effectiveness. LET works through two processes, the learning process to identify program features and predict the bug-revealing probability of a new test program and the scheduling process to prioritize test programs based on bug-revealing probabilities. Chen *et al.* [11] proposed to prioritize test programs based on the prediction information of the test coverage for compilers.

In terms of test prioritization for DNNs, Feng *et al.* [26] proposed the state-of-the-art approach, DeepGini, which identifies possibly-misclassified tests based on model uncertainty. DeepGini assumes a test is more likely to be mispredicted if the DNN outputs similar probabilities for each class. Byun *et al.* [6] evaluated several metrics that prioritize bug-revealing inputs based on the white-box measures of DNN's sentiment, including softmax confidence (i.e., predicted probability for output categories in DNNs that use softmax output layers), Bayesian uncertainty (i.e., the uncertainty of the prediction probability distributions for Bayesian Neural Networks), and input surprise (i.e., the distance of the neuron activation pattern between a test input and the training data). Wang *et al.* [81] proposed PRIMA to prioritize test inputs for DNNs via intelligent mutation analysis. PRIMA further improves DNN test prioritization in two main aspects. First, PRIMA can be applied not only to classification modes but also to regression models. Second, PRIMA can deal with the case in which test inputs are generated from adversarial input generation approaches [8] that can make the probability of the wrong class larger. Furthermore, some data selection approaches [80] are also proposed to detect possibly-misclassified tests for DNNs. Despite its effectiveness in DNN test prioritization, the PRIMA approach cannot be directly applied to GNNs. This is because PRIMA's mutation operators are not adapted to graph-structured data and GNN models.

More specifically, GNN models operate on graph-structured data, where nodes and edges represent entities and their relationships. Conversely, the input mutation rules of PRIMA were designed for independent test samples, rendering them unsuitable for GNNs. Moreover, GNNs incorporate unique graph operations and aggregation mechanisms, including graph convolution operations and message passing mechanisms. PRIMA's model mutation rules are not applicable to the graph-level mechanisms employed by GNNs. As such, GNNs require specialized test prioritization techniques, such as GraphPrior, which leverages the properties of GNN models in its mutation analysis for test prioritization. More specifically, to address the limitations of PRIMA, GraphPrior introduces mutation rules that are designed based on the graph operations and aggregation mechanisms of GNNs. These rules can directly or indirectly impact message passing. Consequently, GraphPrior enables prioritizing tests for graph-structured data.

7.2 Deep Neural Network Testing

Besides test input prioritization, some test selection approaches have also been proposed to improve the efficiency of DNN testing. Test selection aims to precisely estimate the accuracy of the whole set by only labeling the set of selected test inputs. In this way, the labeling cost for DNN testing is reduced. Li *et al.* [50] proposed CES (Cross Entropy-based Sampling) and CSS (Confidence-based Stratified Sampling) to select a small group of representative test inputs to estimate the accuracy of the whole testing set. CES minimizes the cross-entropy between the selected set and the entire test set to ensure that the distribution of the selected test set is similar to the original test set. CSS leverages the confidence features of test inputs to guarantee the similarity between the selected test set and the entire test set. Chen *et al.* [14] proposed PACE (Practical Accuracy Estimation), which selects test inputs practically based on clustering, prototype selection, and adaptive random testing. Pace first clusters all the test inputs into different groups based on their testing capabilities. Then, Pace utilizes the MMD-critic algorithm [43] to select prototypes from each group. For test inputs not in any group, Pace leverages adaptive random testing to select tests from them. Compared to the aforementioned research, our work focus on test prioritization, which ranks all the test inputs without discarding any test input. In this way, testers or developers can find the test inputs that reveal bugs earlier.

In addition to improving the efficiency of DNN testing, several existing studies [37, 44, 54–56, 66] have focused on measuring the adequacy of DNNs. Pei *et al.* [66] proposed a metric of neuron coverage to evaluate how adequate a test set covers the logic of a DNN model. Based on this metric, they proposed a white-box framework for testing DNNs. In the following study, Ma *et al.* [55] proposed DeepGauge, a set of DNN testing coverage criteria to measure the test adequacy of DNNs. DeepGauge also considers neuron coverage to be a good indicator of the effectiveness of a test input. Based on the basic neuron coverage metric, they proposed new metrics with different granularities to differentiate adversarial attacks from legit test data. Kim *et al.* [44] proposed the surprise adequacy for testing of DL models, which identify how effective a test input by measuring its surprise with respect to the training set. More specifically, the surprise of a test input refers to the difference in the activation value of neurons in the face of this new test.

7.3 Mutation Testing for DNNs

Several existing studies have explored the use of mutation testing for DNNs and developed different mutation operators and frameworks. Ma *et al.* [56] propose DeepMutation to measure the quality of test data for DL systems based on mutation testing. To this end, they design a set of source-level and model-level mutation operators to inject faults into the training data, training programs, and DL models. The quality of test data is evaluated by analyzing the extent to which the injected faults can be detected. The work by Ma *et al.* was later extended into a mutation testing tool for DL systems named DeepMutation++ [37], which proposed a set of new mutation operators for feed-forward neural networks (FNNs) and Recurrent Neural Networks (RNNs) and can dynamically mutate run-time states of an RNN. Humbatova *et al.* [39] proposed DeepCrime, which is the first mutation testing tool that implements a set of DL mutation operators based on real DL faults. Shen *et al.* [72] proposed MuNN, a mutation analysis method for neural networks. MuNN defined five mutation operators based on the characteristics of neural networks. The results reveal that mutation analysis has strong domain characteristics, indicating the need for domain mutation operators to enhance the analysis, and that new mutation mechanisms are required for deep neural networks.

The above studies in mutation testing have focused on traditional DNNs, which are typically evaluated on datasets with independent samples. However, the mutation rules employed in these studies do not account for the interdependence among test inputs, which is a crucial factor to consider in the context of GNN testing. In contrast, the mutation rules of GraphPrior are designed to impact the message passing mechanism in the GNN prediction process. In the mutated GNN model, the way nodes acquire information from their neighboring nodes differs slightly from that of the original GNN model. The mutation features generated based on these mutation rules are fed into ranking models to predict the likelihood of a test input being misclassified by the GNN model.

7.4 Mutation-based Test Prioritization for Traditional Software

In traditional software testing, mutation testing is a well-established technique to evaluate the quality of test sets. Mutation-based test prioritization focuses on prioritizing test cases based on their ability to detect mutants. The key idea is that test cases that can detect mutants are likely to be more effective at finding real faults in the code and, therefore, should be given higher priority. Several mutation-based approaches [52, 74] have been proposed. Lou *et al.* [52] proposed a test-case prioritization approach based on the fault detection capability of test cases. They introduced two models to calculate the fault detection capability: the statistics-based model and the probability-based model. Based on the experimental study, they found that the statistics-based model outperforms all the approaches. Shin *et al.* [74] proposed a test case prioritization technique guided by the diversity-aware mutation adequacy criterion and empirically evaluated the effectiveness of mutation-based prioritization techniques with large-scale developer-written test cases. Papadakis *et al.* [63] proposed mutating Combinatorial Interaction Testing models and using them to prioritize tests based on their ability to kill mutants and showed that the number of model-based mutants that are killed yields a strong correlation to code-level faults revealed by the test cases. The aforementioned DNN-oriented approaches consider each test input independent of each other, while in a graph dataset, there are usually complex connections between test inputs. Our proposed GraphPrior specifically targets GNNs and utilizes several mutation rules to generate GNN mutants for test prioritization. Moreover, to better leverage the mutation results, we adopt several ranking models [5, 42, 83] that can learn to predict the probability of a test input to be misclassified.

8 CONCLUSION

To improve the efficiency of GNN testing, we aim to prioritize possibly-misclassified test inputs to reveal GNN bugs earlier. However, a crucial limitation of existing test prioritization approaches is that, when applying to GNNs, they do not take into account the interdependence between test inputs (nodes). In this paper, we propose GraphPrior, a set of test prioritization approaches specifically for GNN testing. GraphPrior assumed that a test input is more likely to be misclassified if it can kill many mutated models. Based on it, GraphPrior leveraged carefully designed mutation rules to generate mutated models for GNNs. Subsequently, GraphPrior obtained the mutation results of test inputs based on the execution of the mutated models. GraphPrior utilized the mutation results in two ways, namely, killing-based and feature-based methods. In the process of scoring a test, killing-based methods considered each mutated model equally important, while feature-based methods learned different importance for each mutated model through ranking models. Finally, GraphPrior ranked all the test inputs based on their scores. We conducted an extensive study to evaluate the effectiveness of GraphPrior approaches on 604 subjects, comparing them with existing approaches that could detect possibly-misclassified test inputs. The experimental results demonstrate the effectiveness of GraphPrior. In terms of APFD, the killing-based GraphPrior approach, KMGP, exceeds the compared approaches (i.e., DeepGini, margin, Vanilla Softmax, PCS, Entropy, least confidence and random selection) by 0.034~0.248 on average. Furthermore, RFGP (i.e., the feature-based GraphPrior approach) exhibited better performance compared to other GraphPrior approaches. Specifically, RFGP outperforms the uncertainty-based test prioritization approaches against different adversarial attacks, with the average improvement of 2.95%~46.69%.

ACKNOWLEDGEMENTS

This work is supported by the Luxembourg National Research Funds (FNR) AFR Grant 17036341 and funding from the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation program (grant agreement No. 949014)

REFERENCES

[1] Bernhard K. Aichernig, Harald Brandl, Elisabeth Jöbstl, Willibald Krenn, Rupert Schlick, and Stefan Tiran. 2015. Killing strategies for model-based mutation testing. *Softw. Test. Verification Reliab.* 25, 8 (2015), 716–748. https://doi.org/10.1002/stvr.1522

[2] Paul Ammann and Jeff Offutt. 2008. *Introducation to Software Testing.* Cambridge University Press. 170–212 pages.

[3] Aleksandar Bojchevski and Stephan Günnemann. 2019. Adversarial attacks on node embeddings via graph poisoning. In *International Conference on Machine Learning*. PMLR, 695–704.

[4] Pietro Bongini, Monica Bianchini, and Franco Scarselli. 2021. Molecular generative graph neural networks for drug discovery. *Neurocomputing* 450 (2021), 242–252.

[5] Leo Breiman. 2001. Random forests. *Machine learning* 45, 1 (2001), 5–32.

[6] Taejoon Byun, Vaibhav Sharma, Abhishek Vijayakumar, Sanjai Rayadurgam, and Darren Cofer. 2019. Input prioritization for testing neural networks. In *2019 IEEE International Conference On Artificial Intelligence Testing (AITest)*. IEEE, 63–70.

[7] Hongyun Cai, Vincent W Zheng, and Kevin Chen-Chuan Chang. 2018. A comprehensive survey of graph embedding: Problems, techniques, and applications. *IEEE transactions on knowledge and data engineering* 30, 9 (2018), 1616–1637.

[8] Nicholas Carlini and David Wagner. 2017. Towards evaluating the robustness of neural networks. In *2017 ieee symposium on security and privacy (sp)*. Ieee, 39–57.

[9] Thierry Titcheu Chekam, Mike Papadakis, Yves Le Traon, and Mark Harman. 2017. An empirical study on mutation, statement and branch coverage fault revelation that avoids the unreliable clean program assumption. In *Proceedings of the 39th International Conference on Software Engineering, ICSE 2017, Buenos Aires, Argentina, May 20-28, 2017*, Sebastián Uchitel, Alessandro Orso, and Martin P. Robillard (Eds.). IEEE / ACM, 597–608. https://doi.org/10.1109/ICSE.2017.61

[10] Cen Chen, Kenli Li, Sin G Teo, Xiaofeng Zou, Kang Wang, Jie Wang, and Zeng Zeng. 2019. Gated residual recurrent graph neural networks for traffic prediction. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 485–492.

[11] Junjie Chen. 2018. Learning to accelerate compiler testing. In *Proceedings of the 40th International Conference on Software Engineering: Companion Proceeedings*. 472–475.

[12] Junjie Chen, Yanwei Bai, Dan Hao, Yingfei Xiong, Hongyu Zhang, and Bing Xie. 2017. Learning to prioritize test programs for compiler testing. In *2017 IEEE/ACM 39th International Conference on Software Engineering (ICSE)*. IEEE, 700–711.

[13] Junjie Chen, Guancheng Wang, Dan Hao, Yingfei Xiong, Hongyu Zhang, Lu Zhang, and Bing Xie. 2018. Coverage prediction for accelerating compiler testing. *IEEE Transactions on Software Engineering* 47, 2 (2018), 261–278.

[14] Junjie Chen, Zhuo Wu, Zan Wang, Hanmo You, Lingming Zhang, and Ming Yan. 2020. Practical accuracy estimation for efficient deep neural network testing. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 29, 4 (2020), 1–35.

[15] Tianqi Chen and Carlos Guestrin. 2016. Xgboost: A scalable tree boosting system. In *Proceedings of the 22nd acm sigkdd international conference on knowledge discovery and data mining*. 785–794.

[16] Hanjun Dai, Hui Li, Tian Tian, Xin Huang, Lin Wang, Jun Zhu, and Le Song. 2018. Adversarial attack on graph structured data. In *International conference on machine learning*. PMLR, 1115–1124.

[17] Richard A. DeMillo, Richard J. Lipton, and Frederick G. Sayward. 1978. Hints on Test Data Selection: Help for the Practicing Programmer. *IEEE Computer* 11, 4 (1978), 34–41. https://doi.org/10.1109/C-M.1978.218136

[18] Xavier Devroey, Gilles Perrouin, Mike Papadakis, Axel Legay, Pierre-Yves Schobbens, and Patrick Heymans. 2016. Featured model-based mutation analysis. In *Proceedings of the 38th International Conference on Software Engineering, ICSE 2016, Austin, TX, USA, May 14-22, 2016*, Laura K. Dillon, Willem Visser, and Laurie A. Williams (Eds.). ACM, 655–666. https://doi.org/10.1145/2884781.2884821

[19] Daniel Di Nardo, Nadia Alshahwan, Lionel Briand, and Yvan Labiche. 2013. Coverage-based test case prioritisation: An industrial case study. In *2013 IEEE Sixth International Conference on Software Testing, Verification and Validation*. IEEE, 302–311.

[20] Hyunsook Do and Gregg Rothermel. 2006. On the use of mutation faults in empirical assessments of test case prioritization techniques. *IEEE Transactions on Software Engineering* 32, 9 (2006), 733–752.

[21] Jian Du, Shanghang Zhang, Guanhang Wu, José MF Moura, and Soummya Kar. 2017. Topology adaptive graph convolutional networks. *arXiv preprint arXiv:1710.10370* (2017).

[22] Sebastian Elbaum, Alexey G Malishevsky, and Gregg Rothermel. 2002. Test case prioritization: A family of empirical studies. *IEEE transactions on software engineering* 28, 2 (2002), 159–182.

[23] Sebastian Elbaum, Gregg Rothermel, and John Penix. 2014. Techniques for improving regression testing in continuous integration development environments. In *Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 235–245.

[24] Emelie Engström, Per Runeson, and Mats Skoglund. 2010. A systematic review on regression test selection techniques. *Information and Software Technology* 52, 1 (2010), 14–30.

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

[26] Yang Feng, Qingkai Shi, Xinyu Gao, Jun Wan, Chunrong Fang, and Zhenyu Chen. 2020. Deepgini: prioritizing massive tests to enhance the robustness of deep neural networks. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 177–188.

[27] Thomas Gaudelet, Ben Day, Arian R Jamasb, Jyothish Soman, Cristian Regep, Gertrude Liu, Jeremy BR Hayter, Richard Vickers, Charles Roberts, Jian Tang, et al. 2021. Utilizing graph machine learning within drug discovery and development. *Briefings in bioinformatics* 22, 6 (2021), bbab159.

[28] Simon Geisler, Tobias Schmidt, Hakan Şirin, Daniel Zügner, Aleksandar Bojchevski, and Stephan Günnemann. 2021. Robustness of graph neural networks at scale. *Advances in Neural Information Processing Systems* 34 (2021), 7637–7649.

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

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

[31] Xiangnan He, Kuan Deng, Xiang Wang, Yan Li, Yongdong Zhang, and Meng Wang. 2020. Lightgcn: Simplifying and powering graph convolution network for recommendation. In *Proceedings of the 43rd International ACM SIGIR conference on research and development in Information Retrieval*. 639–648.

[32] Hadi Hemmati, Andrea Arcuri, and Lionel Briand. 2013. Achieving scalable model-based testing through test case diversity. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 22, 1 (2013), 1–42.

[33] Christopher Henard, Mike Papadakis, Mark Harman, Yue Jia, and Yves Le Traon. 2016. Comparing white-box and black-box test prioritization. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 523–534.

[34] Christopher Henard, Mike Papadakis, Gilles Perrouin, Jacques Klein, Patrick Heymans, and Yves Le Traon. 2014. Bypassing the combinatorial explosion: Using similarity to generate and prioritize t-wise test configurations for software product lines. *IEEE Transactions on Software Engineering* 40, 7 (2014), 650–670.

[35] Danfeng Hong, Lianru Gao, Jing Yao, Bing Zhang, Antonio Plaza, and Jocelyn Chanussot. 2020. Graph convolutional networks for hyperspectral image classification. *IEEE Transactions on Geoscience and Remote Sensing* 59, 7 (2020), 5966–5978.

[36] Qiang Hu, Yuejun Guo, Maxime Cordy, Xiaofei Xie, Wei Ma, Mike Papadakis, and Yves Le Traon. 2021. Towards Exploring the Limitations of Active Learning: An Empirical Study. In *2021 36th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 917–929.

[37] Qiang Hu, Lei Ma, Xiaofei Xie, Bing Yu, Yang Liu, and Jianjun Zhao. 2019. Deepmutation++: A mutation testing framework for deep learning systems. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1158–1161.

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

[39] Nargiz Humbatova, Gunel Jahangirova, and Paolo Tonella. 2021. DeepCrime: mutation testing of deep learning systems based on real faults. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis*. 67–78.

[40] Kanchan Jha, Sriparna Saha, and Hiteshi Singh. 2022. Prediction of protein–protein interaction using graph neural networks. *Scientific Reports* 12, 1 (2022), 1–12.

[41] Weiwei Jiang and Jiayun Luo. 2022. Graph neural network for traffic forecasting: A survey. *Expert Systems with Applications* (2022), 117921.

[42] Guolin Ke, Qi Meng, Thomas Finley, Taifeng Wang, Wei Chen, Weidong Ma, Qiwei Ye, and Tie-Yan Liu. 2017. Lightgbm: A highly efficient gradient boosting decision tree. *Advances in neural information processing systems* 30 (2017).

[43] Been Kim, Rajiv Khanna, and Oluwasanmi O Koyejo. 2016. Examples are not enough, learn to criticize! criticism for interpretability. *Advances in neural information processing systems* 29 (2016).

[44] Jinhan Kim, Robert Feldt, and Shin Yoo. 2019. Guiding deep learning system testing using surprise adequacy. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 1039–1049.

[45] Thomas N Kipf and Max Welling. 2016. Semi-supervised classification with graph convolutional networks. *arXiv preprint arXiv:1609.02907* (2016).

[46] Yves Ledru, Alexandre Petrenko, Sergiy Boroday, and Nadine Mandran. 2012. Prioritizing test cases with string distances. *Automated Software Engineering* 19, 1 (2012), 65–95.

[47] Cheng Li, Jiaqi Ma, Xiaoxiao Guo, and Qiaozhu Mei. 2017. Deepcas: An end-to-end predictor of information cascades. In *Proceedings of the 26th international conference on World Wide Web*. 577–586.

[48] Yaxin Li, Wei Jin, Han Xu, and Jiliang Tang. 2020. Deeprobust: A pytorch library for adversarial attacks and defenses. *arXiv preprint arXiv:2005.06149* (2020).

[49] Zheng Li, Mark Harman, and Robert M Hierons. 2007. Search algorithms for regression test case prioritization. *IEEE Transactions on software engineering* 33, 4 (2007), 225–237.

[50] Zenan Li, Xiaoxing Ma, Chang Xu, Chun Cao, Jingwei Xu, and Jian Lü. 2019. Boosting operational dnn testing efficiency through conditioning. In *Proceedings of the 2019 27th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering*. 499–509.

[51] Yiling Lou, Junjie Chen, Lingming Zhang, and Dan Hao. 2019. A survey on regression test-case prioritization. In *Advances in Computers*. Vol. 113. Elsevier, 1–46.

[52] Yiling Lou, Dan Hao, and Lu Zhang. 2015. Mutation-based test-case prioritization in software evolution. In *2015 IEEE 26th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 46–57.

[53] Jiaqi Ma, Shuangrui Ding, and Qiaozhu Mei. 2020. Towards more practical adversarial attacks on graph neural networks. *Advances in neural information processing systems* 33 (2020), 4756–4766.

[54] Lei Ma, Felix Juefei-Xu, Minhui Xue, Bo Li, Li Li, Yang Liu, and Jianjun Zhao. 2019. Deepct: Tomographic combinatorial testing for deep learning systems. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*. IEEE, 614–618.

[55] Lei Ma, Felix Juefei-Xu, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Chunyang Chen, Ting Su, Li Li, Yang Liu, et al. 2018. Deepgauge: Multi-granularity testing criteria for deep learning systems. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*. 120–131.

[56] Lei Ma, Fuyuan Zhang, Jiyuan Sun, Minhui Xue, Bo Li, Felix Juefei-Xu, Chao Xie, Li Li, Yang Liu, Jianjun Zhao, et al. 2018. Deepmutation: Mutation testing of deep learning systems. In *2018 IEEE 29th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 100–111.

[57] Wei Ma, Mike Papadakis, Anestis Tsakmalis, Maxime Cordy, and Yves Le Traon. 2021. Test selection for deep learning systems. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 2 (2021), 1–22.

[58] Yao Ma, Suhang Wang, Tyler Derr, Lingfei Wu, and Jiliang Tang. 2019. Attacking graph convolutional networks via rewiring. *arXiv preprint arXiv:1906.03750* (2019).

[59] Aleksander Madry, Aleksandar Makelov, Ludwig Schmidt, Dimitris Tsipras, and Adrian Vladu. 2017. Towards deep learning models resistant to adversarial attacks. *arXiv preprint arXiv:1706.06083* (2017).

[60] Volodymyr Mnih, Koray Kavukcuoglu, David Silver, Andrei A Rusu, Joel Veness, Marc G Bellemare, Alex Graves, Martin Riedmiller, Andreas K Fidjeland, Georg Ostrovski, et al. 2015. Human-level control through deep reinforcement learning. *nature* 518, 7540 (2015), 529–533.

[61] Quang Hung Nguyen, Hai-Bang Ly, Lanh Si Ho, Nadhir Al-Ansari, Hiep Van Le, Van Quan Tran, Indra Prakash, and Binh Thai Pham. 2021. Influence of data splitting on performance of machine learning models in prediction of shear strength of soil. *Mathematical Problems in Engineering* 2021 (2021), 1–15.

[62] Niccolò Pancino, Alberto Rossi, Giorgio Ciano, Giorgia Giacomini, Simone Bonechi, Paolo Andreini, Franco Scarselli, Monica Bianchini, and Pietro Bongini. 2020. Graph Neural Networks for the Prediction of Protein-Protein Interfaces.. In *ESANN*. 127–132.

[63] Mike Papadakis, Christopher Henard, and Yves Le Traon. 2014. Sampling Program Inputs with Mutation Analysis: Going Beyond Combinatorial Interaction Testing. In *Seventh IEEE International Conference on Software Testing, Verification and Validation, ICST 2014, March 31 2014-April 4, 2014, Cleveland, Ohio, USA*. IEEE Computer Society, 1–10. https://doi.org/10.1109/ICST.2014.11

[64] Mike Papadakis, Marinos Kintis, Jie Zhang, Yue Jia, Yves Le Traon, and Mark Harman. 2019. Chapter Six - Mutation Testing Advances: An Analysis and Survey. *Adv. Comput.* 112 (2019), 275–378. https://doi.org/10.1016/bs.adcom.2018.03.015

[65] Adam Paszke, Sam Gross, Francisco Massa, Adam Lerer, James Bradbury, Gregory Chanan, Trevor Killeen, Zeming Lin, Natalia Gimelshein, Luca Antiga, et al. 2019. Pytorch: An imperative style, high-performance deep learning library. *Advances in neural information processing systems* 32 (2019).

[66] Kexin Pei, Yinzhi Cao, Junfeng Yang, and Suman Jana. 2017. Deepxplore: Automated whitebox testing of deep learning systems. In *proceedings of the 26th Symposium on Operating Systems Principles*. 1–18.

[67] Michael Prince. 2004. Does active learning work? A review of the research. *Journal of engineering education* 93, 3 (2004), 223–231.

[68] Pengzhen Ren, Yun Xiao, Xiaojun Chang, Po-Yao Huang, Zhihui Li, Brij B Gupta, Xiaojiang Chen, and Xin Wang. 2021. A survey of deep active learning. *ACM computing surveys (CSUR)* 54, 9 (2021), 1–40.

[69] Gregg Rothermel, Roland H. Untch, Chengyun Chu, and Mary Jean Harrold. 2001. Prioritizing test cases for regression testing. *IEEE Transactions on software engineering* 27, 10 (2001), 929–948.

[70] Benedek Rozemberczki and Rik Sarkar. 2020. Characteristic functions on graphs: Birds of a feather, from statistical descriptors to parametric models. In *Proceedings of the 29th ACM international conference on information & knowledge management*. 1325–1334.

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

[72] Weijun Shen, Jun Wan, and Zhenyu Chen. 2018. Munn: Mutation analysis of neural networks. In *2018 IEEE International Conference on Software Quality, Reliability and Security Companion (QRS-C)*. IEEE, 108–115.

[73] Chence Shi, Minkai Xu, Zhaocheng Zhu, Weinan Zhang, Ming Zhang, and Jian Tang. 2020. Graphaf: a flow-based autoregressive model for molecular graph generation. *arXiv preprint arXiv:2001.09382* (2020).

[74] Donghwan Shin, Shin Yoo, Mike Papadakis, and Doo-Hwan Bae. 2019. Empirical evaluation of mutation-based test case prioritization techniques. *Software Testing, Verification and Reliability* 29, 1-2 (2019), e1695.

[75] Karen Simonyan and Andrew Zisserman. 2014. Very deep convolutional networks for large-scale image recognition. *arXiv preprint arXiv:1409.1556* (2014).

[76] Chen Sun, Abhinav Shrivastava, Carl Vondrick, Rahul Sukthankar, Kevin Murphy, and Cordelia Schmid. 2019. Relational action forecasting. In *Proceedings of the IEEE/CVF Conference on Computer Vision and Pattern Recognition*. 273–283.

[77] Lichao Sun, Yingtong Dou, Carl Yang, Ji Wang, Philip S Yu, Lifang He, and Bo Li. 2018. Adversarial attack and defense on graph data: A survey. *arXiv preprint arXiv:1812.10528* (2018).

[78] Vivienne Sze, Yu-Hsin Chen, Tien-Ju Yang, and Joel S Emer. 2017. Efficient processing of deep neural networks: A tutorial and survey. *Proc. IEEE* 105, 12 (2017), 2295–2329.

[79] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2017. Graph attention networks. *arXiv preprint arXiv:1710.10903* (2017).

[80] Dan Wang and Yi Shang. 2014. A new active labeling method for deep learning. In *2014 International joint conference on neural networks (IJCNN)*. IEEE, 112–119.

[81] Zan Wang, Hanmo You, Junjie Chen, Yingyi Zhang, Xuyuan Dong, and Wenbin Zhang. 2021. Prioritizing test inputs for deep neural networks via mutation analysis. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 397–409.

[82] Michael Weiss and Paolo Tonella. 2022. Simple techniques work surprisingly well for neural network test prioritization and active learning (replicability study). In *Proceedings of the 31st ACM SIGSOFT International Symposium on Software Testing and Analysis*. 139–150.

[83] Raymond E Wright. 1995. Logistic regression. (1995).

[84] Le Wu, Peijie Sun, Richang Hong, Yanjie Fu, Xiting Wang, and Meng Wang. 2018. Socialgcn: An efficient graph convolutional network based model for social recommendation. *arXiv preprint arXiv:1811.02815* (2018).

[85] Shiwen Wu, Fei Sun, Wentao Zhang, Xu Xie, and Bin Cui. 2022. Graph neural networks in recommender systems: a survey. *Comput. Surveys* 55, 5 (2022), 1–37.

[86] Kaidi Xu, Hongge Chen, Sijia Liu, Pin-Yu Chen, Tsui-Wei Weng, Mingyi Hong, and Xue Lin. 2019. Topology attack and defense for graph neural networks: An optimization perspective. *arXiv preprint arXiv:1906.04214* (2019).

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

[88] Zhilin Yang, William Cohen, and Ruslan Salakhudinov. 2016. Revisiting semi-supervised learning with graph embeddings. In *International conference on machine learning*. PMLR, 40–48.

[89] Liang Yao, Chengsheng Mao, and Yuan Luo. 2019. Graph convolutional networks for text classification. In *Proceedings of the AAAI conference on artificial intelligence*, Vol. 33. 7370–7377.

[90] Ruiping Yin, Kan Li, Guangquan Zhang, and Jie Lu. 2019. A deeper graph neural network for recommender systems. *Knowledge-Based Systems* 185 (2019), 105020.

[91] Rex Ying, Ruining He, Kaifeng Chen, Pong Eksombatchai, William L Hamilton, and Jure Leskovec. 2018. Graph convolutional neural networks for web-scale recommender systems. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 974–983.

[92] Shin Yoo and Mark Harman. 2012. Regression testing minimization, selection and prioritization: a survey. *Software testing, verification and reliability* 22, 2 (2012), 67–120.

[93] Junliang Yu, Hongzhi Yin, Jundong Li, Min Gao, Zi Huang, and Lizhen Cui. 2020. Enhance social recommendation with adversarial graph convolutional networks. *IEEE Transactions on Knowledge and Data Engineering* (2020).

[94] Long Zhang, Xuechao Sun, Yong Li, and Zhenyu Zhang. 2019. A noise-sensitivity-analysis-based test prioritization technique for deep neural networks. *arXiv preprint arXiv:1901.00054* (2019).

[95] Qin Zhang, Keping Yu, Zhiwei Guo, Sahil Garg, Joel JPC Rodrigues, Mohammad Mehedi Hassan, and Mohsen Guizani. 2021. Graph neural network-driven traffic forecasting for the connected internet of vehicles. *IEEE Transactions on Network Science and Engineering*

9, 5 (2021), 3015–3027.

[96] Dongbin Zhao, Haitao Wang, Kun Shao, and Yuanheng Zhu. 2016. Deep reinforcement learning with experience replay based on SARSA. In *2016 IEEE symposium series on computational intelligence (SSCI)*. IEEE, 1–6.

[97] Hang Zhou, Weikun Wang, Jiayun Jin, Zengwei Zheng, and Binbin Zhou. 2022. Graph Neural Network for Protein–Protein Interaction Prediction: A Comparative Study. *Molecules* 27, 18 (2022), 6135.

[98] Jie Zhou, Ganqu Cui, Shengding Hu, Zhengyan Zhang, Cheng Yang, Zhiyuan Liu, Lifeng Wang, Changcheng Li, and Maosong Sun. 2020. Graph neural networks: A review of methods and applications. *AI Open* 1 (2020), 57–81.

[99] Daniel Zügner, Amir Akbarnejad, and Stephan Günnemann. 2018. Adversarial attacks on neural networks for graph data. In *Proceedings of the 24th ACM SIGKDD international conference on knowledge discovery & data mining*. 2847–2856.

[100] Daniel Zügner and Stephan Günnemann. 2019. Adversarial attacks on graph neural networks via meta learning. *arXiv preprint arXiv:1902.08412* (2019).