Functional Scenario Classification for Android Applications using GNNs

Guiyin Li
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, Jiangsu, China
nju_lgy@smail.nju.edu.cn

Fengyi Zhu
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, Jiangsu, China
MF20330128@smail.nju.edu.cn

Jun Pang*
FSTM & SnT, University of Luxembourg
Esch-sur-Alzette, Luxembourg
jun.pang@uni.lu

Tian Zhang*
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, Jiangsu, China
ztluck@nju.edu.cn

Minxue Pan*
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, Jiangsu, China
mxp@nju.edu.cn

Xuandong Li
State Key Lab for Novel Software Technology, Nanjing University
Nanjing, Jiangsu, China
lxd@nju.edu.cn

ABSTRACT

Functional scenario comprehension of screens in Android applications paves the way for Android app development and Android UI testing, especially in automated UI testing and test reuse. On the one hand, the screens of diverse Android applications contain widgets with many combinations. On the other hand, the screens of different scenarios may leverage similar widgets to fulfill the functionalities. Due to the above reasons, scenario comprehension is still hard to be solved by current approaches. In this paper, to fully understand the functionality of each screen, we propose a novel approach that employs Graph Neural Networks (GNN) to classify scenarios leveraging the transitions between screens and other available information of screens altogether. According to the result evaluated on 30 popular applications in the file management category, our approach improves the classification accuracy by at least 6% compared to previous work, demonstrating that GNN can fully utilize the potential relations and dependencies between the transitioned screens.

CCS CONCEPTS

• **Software and its engineering** → **Software testing and debugging**.

KEYWORDS

Graph neural networks, functional scenario classification, Android testing, UI testing

ACM Reference Format:
Guiyin Li, Fengyi Zhu, Jun Pang, Tian Zhang, Minxue Pan, and Xuandong Li. 2022. Functional Scenario Classification for Android Applications using GNNs. In *13th Asia-Pacific Symposium on Internetware (Internetware 2022), June 11–12, 2022, Hohhot, China.* ACM, New York, NY, USA, 9 pages. https://doi.org/10.1145/3545258.3545270

*Corresponding author.

Internetware 2022, June 11–12, 2022, Hohhot, China

ACM ISBN 978-1-4503-9780-3/22/06.
https://doi.org/10.1145/3545258.3545270

1 INTRODUCTION

Mobile applications often have multiple functionalities that provide services to users. As a result, for Android app developers, it is a critical mission to develop an Android app with rich functional scenarios that is easy to use and occupies the applications markets such as Google Play more quickly. Since most applications interact with users through the rich graphical user interface (GUI), GUI testing has become a necessary part of app testing before the release, and functional scenarios can help testers prepare GUI test scripts. In script-based testing, tests are recorded as scripts that can be automated executed using tools such as Appium [6], Robotium [17] and others [8]. These test scripts are often manually coded for specific applications and specific functional scenarios instead of testing the app as a whole, which tend to be a series of scenario tests and require testers to maintain frequently.

Furthermore, functional scenarios also play an essential role in Android GUI automated testing, which aims to achieve a high code coverage and generate test sequences in a relatively short period of time. The testing or exploring process often requires covering multiple functional scenarios because different scenarios are usually bound to various code and can help find more potential bugs. For example, the state-of-the-art tool Q-testing [14], which is based on reinforcement learning, guides the testing towards unfamiliar functionalities by recording previously visited states and judging whether the two states belong to the same functional scenario to determine the reward during the exploration process. As a result, the scenario division module can guide Q-testing to explore different functional scenarios preferentially and improve the efficiency of Q-testing. More accurate classification of functional scenarios can improve the exploration strategies of such tools.

There are ample opportunities to reuse tests in Android GUI testing since many applications belong to the same category and implement similar functionalities. For example, almost all file management applications provide services such as copying and pasting, browsing files, searching for files, setting the app, playing music, etc. Therefore, in the research of script generation, script migration, and script reuse [1, 10, 18, 23], they all try to reuse some modular

test scripts with the same functionalities in the same category of applications or different versions of the same app to reduce the manual work. When reusing scripts, it is necessary to classify or identify the screen of the new app into standard predefined functional screen categories in the test library to determine the purpose of the current screen and synthesize new test scripts.

Current scenario classification approaches are all based on available screen information extracted from three perspectives: the layout files and activity names of screens and the snapshots. For instance, Ariel et al. [18] and Luca et al. [1] both use the types and numbers of widget attributes in layout files, such as the number of clickable widgets. Q-testing encodes selected widget attributes in layout files into vectors using neural networks. Hu et al. [10] develop AppFlow that converts all the above three types of information into vectors. Through manual scenario classification of the screens, we find that screens are not isolated, and determining the category of a screen often requires checking one or more predecessor screens that reach the current one. This is reasonable since many functionalities are connected and interdependent. For example, in the file management applications, we need to select one or more files and then choose to cut or copy to reach the paste scenario. Therefore, we aim to utilize the transitions or connections between the screens for functional scenario classification. The transition corresponds to an event triggered by interactions with the GUI and can lead to a new screen state. We construct a directed transition graph by taking screens as nodes and transitions as edges and novelly leverage Graph Neural Networks (GNNs) [24] to learn this graph and mine potential functional dependencies and relations between screens. We apply our approach to scenario classifications on 30 file management applications. The results demonstrate the effectiveness of our approach by showing an improvement of 6% more classification accuracy than those produced by current approaches without transitions between screens.

The contributions of this paper are summarised as the following:

- We novelly employ the connections and transitions between the screens in classifying the functional scenarios of screens.
- We construct a directed transition graph whose nodes are screens and edges are transitions so that all screens are no longer isolated, and propose a GNN-based scenarios classification approach to learn the transition graph and capture the potential functional dependencies between screens.
- We implement our approach and evaluate it to classify scenarios on 30 file management applications. The evaluation results show its effectiveness compared with existing approaches.

Structure of the paper. Section 2 illustrates a motivating example. Section 3 provides the overview and the details of the approach. Section 4 presents the experimental evaluation. Section 5 reviews research studies that are closely related to this work. Section 6 concludes the paper.

2 MOTIVATION

In the scenario classification of screens, we notice that two relations between screens' connections and functional scenarios are not considered by previous work but have an impact on the classification accuracy. We use an app called OI File Manager (OIFM) as an illustrative example. OIFM is a very famous open-source file management app that has been installed more than 53,000 times on Google Play.

The first kind of relation is the dependency between consecutive screens. When we determine the scenarios for some screens, we may not be able to do so solely based on the information of the current screen, since we need to check the predecessor screens further. As the OIFM example in Figure 1 shows, we define a bookmarks scenario with coarse granularity, including bookmarks' creation and potential operation. Then screens S1 and S2 both belong to this scenario, and S1 can reach S2 by long-pressing the widget called *emulated* (the first item in ListView [7]). However, if we ignore the relation between S1 and S2, we may mistakenly determine that S2 is a screen of deleting files scenario, which means long-pressing and then deleting the file. In fact, the folder called *emulated* will not be deleted on storage after long-pressing and deleting it, and two scenarios are bound to different source codes. Let us check another scenario called searching, which is related to operations for searching for files or folders. Screens S3 and S4 both belong to this scenario, and we can reach S4 from S3 by clicking the enter key. If we ignore the connection between S3 and S4, we may misjudge S4 as a screen of browsing files scenario. Obviously, the two scenarios' source codes and subsequent operable element actions on their screens are entirely different. Therefore, relations between screens help us to classify functional scenarios more accurately.

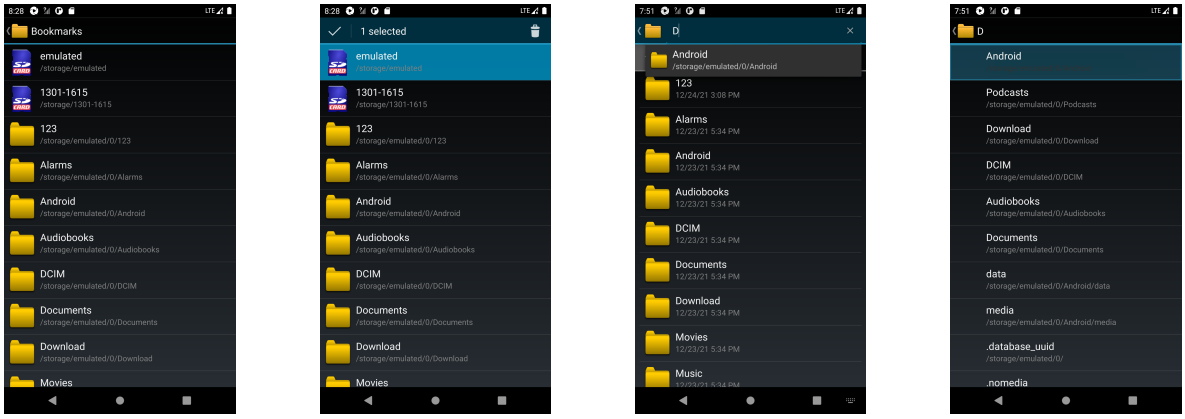
The other kind of relation is dependencies between functional scenarios since the designers considered the logic sequence relations between these functionalities during the design phase. If we define browsing files scenario, long-pressing files scenario, menu scenario, compressing files scenario, decompressing files scenario, viewing files' properties scenario and other scenarios in file management applications, we find that for common file management applications, these scenarios have the same logic sequence relations in Figure 2. For example, if we need to view the properties of a file, we must first click the menu button to reach the menu screen after long-pressing the file, and then click the button named properties or details on the menu screen. In other words, if we reach the menu scenario, the following scenario may be one of compressing files, decompressing files, or viewing files' properties. More generally, if we want to get to the pasting files scenario, we need to reach the cutting files scenario or copying files scenario first.

Through the above example, we can easily conclude that there exist relations between screens' connections and functional scenarios. By adding edges between screens, some of the relations mentioned above can be preserved, which in turn can help us mine the potential functional dependencies between screens and classify the functional scenarios of screens with higher accuracy. In the next section, we develop a new approach towards functional scenario classification for Android apps by utilizing these identified relations between screens.

3 APPROACH

3.1 Definitions

Definition 3.1 (Functional Scenario). Our definition of functional scenarios is the same as in Q-testing [14], that is, functional scenarios are used to describe the functionalities provided by an application from the user's perspective. Since the user interacts with



(a) Screen S1 (b) Screen S2 (c) Screen S3 (d) Screen S4

Figure 1: Snapshots of OIFM screens in two scenarios (bookmarks and searching)

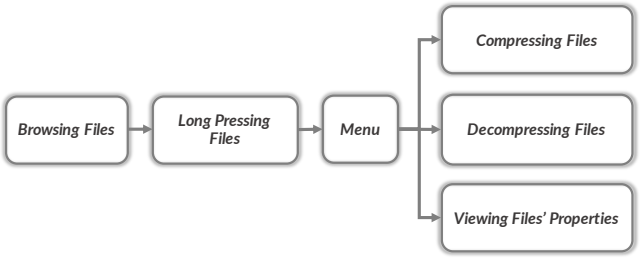


Figure 2: Part of the logic sequence relations in file management applications

the app through the rich GUI, we categorize the screen of a specific state when a user performs the task or uses the functionality as some functional scenario. Hence, the classification of functional scenarios is equivalent to the classification of screens. Since different users have different understandings of the same functional scenario, the granularity of the definition will be different. For example, the scenario of searching for files can be divided into two sub-scenarios: the searching process and the viewing of the search results. Therefore, we need to ensure the same understanding and classification granularity when defining and classifying functional scenarios.

Definition 3.2 (Directed Transition Graph). The directed transition graph of an application is similar to a state machine constructed by taking screens of the interface as nodes and transitions between screens as directed edges. Each node represents a GUI state during the running process, including a screenshot of the interface in the form of Portable Network Graphics (PNG) and a layout file in the form of XML. Each directed edge represents an event triggered by interactions with the GUI of the source state and can reach a new screen state. For example, if the user clicks button B on screen A and then arrives at screen C, there will be an edge e from node u representing screen A to node v representing screen C, where edge e represents the event of clicking button B on screen A.

Definition 3.3 (Problem Definition). Our problem is to perform functional scenario classification on the screens of the applications in the test set based on the training result on the training set. We use applications that belong to the same category as the experimental object. We predefine several functional scenarios for these applications, and each screen of these applications belongs to one of these scenarios. We select a part of these applications as the training set, that is, the labels of all the screens included are known, and select a part of them as the validation set and test set, respectively.

3.2 Background on Graph Neural Networks

A directed or undirected graph, denoted as $G = (V, E, \mathbf{X})$, is a triple where: (1) $V = \{v_1, \ldots, v_i, \ldots, v_n\}$ is a node set; (2) $E = \{e_{i,j} = (v_i, v_j)\} \subseteq (V \times V)$ is an edge set; (3) $\mathbf{X} = \{\mathbf{x}_1, \ldots, \mathbf{x}_i, \ldots, \mathbf{x}_n\}$ is a feature matrix, $\mathbf{x}_u \in \mathbb{R}^d$ is d-dimensional feature of the node v_u in V.

As a major field in deep learning [4], Graph Neural Networks (GNNs) [12, 20–22, 24] directly structure the learning process on the graph, which can effectively analyze graph structure data. GNNs learn a new representation or embedding vector of a node based on both graph structure information and node feature $\mathbf{X}$, which is called message passing scheme or neighborhood aggregation scheme. Propagation at layer (k) consists of three steps: (1) **Message passing**. For every linked nodes i, j, GNNs compute a message by utilizing their embeddings from the previous layer and edge features from node j to node i. (2) **Neighborhood aggregation**. For every node i, GNNs then aggregate the messages between node i and all its neighbors $N(i)$. (3) **Update**. For every node i, GNNs finally use a non-linear differentiable function to update embeddings $\mathbf{x}_i^k$ based on the aggregated messages and their embedding from the previous layer. Therefore, after (k) iterations of the propagation, every node's embedding captures the structural information from its k-hop network neighbors. Message passing graph neural networks can be described as:

$$\mathbf{x}_i^{(k)} = \gamma^{(k)}\left(\mathbf{x}_i^{(k-1)}, \square_{j \in \mathcal{N}(i)}\, \phi^{(k)}\left(\mathbf{x}_i^{(k-1)}, \mathbf{x}_j^{(k-1)}, \mathbf{e}_{j,i}\right)\right) \quad (1)$$

In the above equation, $\mathbf{x}_i^{(k-1)} \in \mathbb{R}^F$ represents the feature of node i at layer $(k-1)$ and $\mathbf{e}_{j,i} \in \mathbb{R}^D$ represents edge features from node j to node i. $\square$ corresponds to the aggregation way of step 2,

such as sum and mean, and ϕ and γ correspond to the message passing of step 1 and the updating of step 3.

3.3 Approach Overview

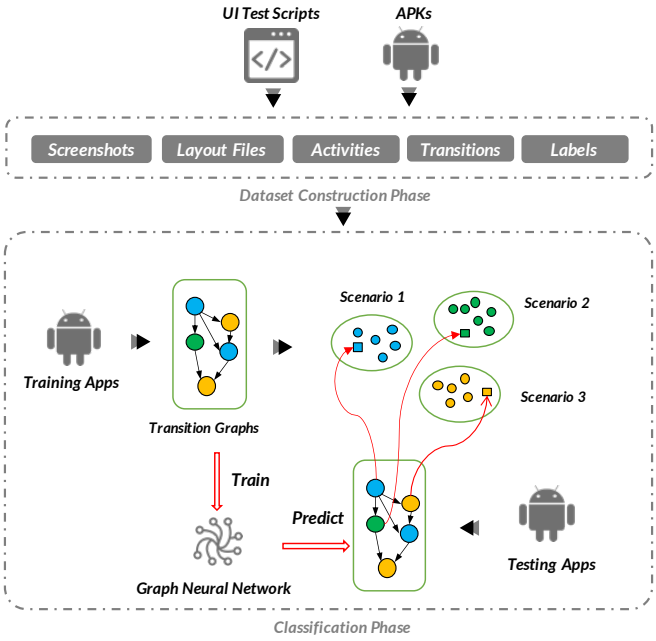


Figure 3: Approach Framework

Figure 3 briefly shows the overall framework of the approach in this paper. The classification of the functional scenarios in Android applications using GNNs can be mainly divided into two phases.

In the dataset construction phase, we select the APKs of applications in the same category and UI test scripts for those applications as input. Running these test scripts can automatically test the UI of these applications, and at the same time, save screenshots, layout files, and activity information of all screens and transitions between screens during the running process. After collecting all running data, we classify the functional scenarios of these screens manually. We obtain the labeled data in this phase.

The classification phase can be subdivided into three parts: the transition graph construction process, feature extraction, and the GNN classification process. In the transition graph construction process, we merge the same screens according to the saved layout files and screenshots and then build a directed transition graph based on the transitions between screens. In the feature extraction process, we extract useful information from the layout files, screenshots, and activity information as the feature vectors of the screens, which corresponds to the initial features of the nodes in the transition graphs. In the GNN classification process, we input the transition graphs, the initial features of the nodes, and the label information of functional scenarios of the applications as the training set, train a GNN model and save all the parameters. At last, we use the trained GNN model to classify all the nodes in the transition graphs of the test set.

3.4 Dataset Construction Process

Since the existing related work does not consider the transitions between screens, the existing datasets don't record the transitions. Therefore, we need to create a new labeled dataset with both screens and transitions to construct the transition graphs. The construction of the dataset is mainly divided into the following four steps.

3.4.1 Find APKs. Through research on the applications on Google Play and F-droid, we find that applications belonging to the same category often share multiple identical functional scenarios, so we selected applications of the same category to construct the dataset. For these applications of the selected category, we download and install manually based on the number of downloads on Google Play. After manually running and checking these applications, we record the common and identical functional scenarios and use these functional scenarios as the subsequent set of scenarios we want to classify, that is, the subsequent set of labels. Further, we can continue to look for applications of this category that contain these functional scenarios on Google Play to expand the app collection. The found applications will be randomly selected as a training set and a validation set in the subsequent steps, and the rest will be used as a test set.

3.4.2 Write UI Test Scripts. In the step of finding APKs, we have obtained the set of functional scenarios to be classified, so we mainly write test scripts for each app according to the above functional scenarios. Similar to the way it is written in AppFlow [10], we write a specific test flow for each functional scenario, that is, write a test script in the form of a function that covers a complete logical test of the functional scenario. In reality, tests of a functional scenario often contain different situations. For example, for the compression test, we can choose to compress into Zip, 7z, Rar, and other formats, and at the same time, we can choose "OK" or "Cancel the compression" in the last step of compression. We keep all possibilities in the form of function arguments for these options. Unlike AppFlow, the scripts we choose to write in the mainstream Python language are based on the Appium testing framework.

Next, we briefly describe how we write a function for a specific functional scenario. We define a global counter to number all states in order of execution time. When writing a specific test flow for each functional scenario, we need to start counting from the current counter value and save the screenshots, layout files, and activity information of all UI states, and the transitions between states during the running process. For screenshots, layout files, and activity information of the UI state, we only need to call the three functions officially provided by Webdriver [19] every time we reach a new state, which can be understood as saving the current state after executing each test action. For each transition between states, we record it as a four-tuple $\langle S, D, A, E\rangle$, where S represents the source state before executing the test action, D represents the destination state after executing the test action, A represents the test action such as "click" and E represents the element or widget triggered by A. Common test actions include $click$, $longpress$, $\langle edit, text\rangle$, $clear$, $swipe$, $back$, $\langle press\_keycode, code\ number\rangle$ and so on. $\langle edit, text\rangle$ means entering "text" on the widget. $\langle press\_keycode, code\ number\rangle$ means sending a keycode "code number" to the device. For widget E, we use a 5-tuple consisting of class attribute, bounds attribute,

text attribute, resource-id attribute, and content-desc attribute to define and locate due to the fact that there are multiple widgets sharing the same class attribute or bounds attribute or resource-id attribute, and the content-desc attribute values of many widgets are empty or null. These 5 attributes can be directly obtained by calling the functions of Webdriver.

3.4.3 Run UI Tests and Collect Data. After running the UI test scripts written as required, we call automatically collect all necessary data. Like AppFlow and Q-testing, all screenshots are saved in the form of PNG, layout files are saved in the form of XML, and activity information is saved in the form of TXT. The only difference between our data and existing related works' data is that we save the transitions between interfaces. The format of the transitions we saved is shown in Figure 4. The numbers in the figure represent the global numbers of the source and destination states.

```
0 1 click ['android.widget.RelativeLayout', '[21,1456][1059,1624]', '', None, None]
1 2 back
2 3 longpress ['android.widget.ImageButton', '[0,189][148,311]', '', None, None]
3 4 click ['android.widget.TextView', '[586,604][1017,667]', 'Settings', 'android:id/title', '  ']
4 5 click ['android.widget.TextView', '[63,302][210,365]', 'Sort by', 'android:id/title', '  ']
5 6 click ['android.widget.CheckedTextView', '[70,824][1010,950]', 'Size', 'android:id/text1', None]
6 7 click ['android.widget.TextView', '[63,302][210,365]', 'Sort by', 'android:id/title', '  ']
7 8 click ['android.widget.CheckedTextView', '[70,696][1010,822]', 'Name', 'android:id/text1', None]
8 9 <edit, abc> ['android.widget.EditText', '[112,900][968,1002]', 'Folder name', filemanager:id/foldername', None]
9 10 clear  ['android.widget.EditText', '[112,900][968,1002]', 'Folder name', filemanager:id/foldername', None]
10 11 longpress ['android.widget.RelativeLayout', '[21,315][1059,483]', '', None, None]
11 12 click ['android.widget.TextView', '[638,63][785,189]', '', 'filemanager:id/menu_delete', 'Delete']
12 13 click ['android.widget.Button', '[541,953][1010,1079]', 'OK', 'android:id/button1', None]
13 14 longpress ['android.widget.RelativeLayout', '[21,315][1059,483]', '', None, None]
...
```

Figure 4: An example of transitions between screens

3.4.4 Label Screens. After collecting all the data, we need to label all the saved screenshots with scenario tags manually, and the label comes from the set of functional scenarios we defined in the first step. For example, if we define scenario 1, scenario 2, etc., we need to classify each screenshot into one of the above scenarios. In the end, we obtain a dataset with labeled screens that will fit our purposes.

3.5 Transition Graph Construction Process

Since there are operations such as returning to the previous state or clicking the "cancel this operation" button in the process of testing, the collected data has the same state, so we need to remove the duplicated states and renumber the remaining states.

Each GUI state contains a corresponding screenshot in PNG format and a layout file in XML. Each layout file corresponds to a screenshot, and each node it contains can correspond to a widget on the screenshot. Generally speaking, the contents of the layout file can be directly read into a long complete string, and comparing two states is equivalent to comparing whether two strings are equal. We find that the layout file contains some cached information belonging to the running process, which will lead to slight differences in the content of the layout files of two identical states. Still, the content of the screenshot is exactly the same when viewed visually. So the strategy is that if the layout files of the two states or the pixel values of their screenshots are the same, we consider them to be the same state.

After removing the duplicated states, we need to start numbering all the remaining states from 0, and at the same time, we need to update the number of screenshots, layout files, activity information, and labels related to these states. Finally, we need to update the numbers of the source state and destination state in the transitions. Therefore, we complete the state compression.

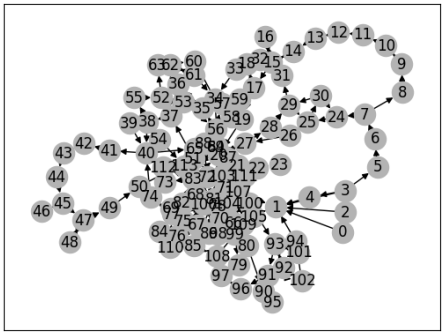


Figure 5: A transition graph from an app

For each app's state set, we build it into a directed transition graph. The graph has been defined in the above article. In short, each state corresponds to a node in the graph, and each transition is a directed edge. The transition graph of an application we have built is shown in Figure 5. In addition, our graphs are constructed following two principles:

- Discard all self-loops, that is, we do not add edges like from state A to state A to the graph. If there is an edge from state A to state A, it means that after performing an action on a widget of state A, it is still in state A. In fact, we can perform invalid operations on state A (such as clicking on a non-widget or long-pressing a widget whose long-clickable attribute is false, etc.), then any node has a self-loop.
- Any directed edge can only be added once, that is, there is only one directed edge from state A to state B. This is also in line with the expectations of most mainstream graph neural networks.

3.6 Feature Extraction Process

Feature extraction is critical to our classification results. We need to extract information relevant to our task from each screen and encode it into a fixed-length feature vector. Each screen involves three types of information, which have been obtained in the previous steps of the approach, namely screenshot, layout file, and activity information. The most relevant to the functional scenario classification is the text information and the icon information with a specific meaning. For example, in a searching scenario, we may see a search box or text information related to searching. As verified in AppFlow, these three types of information can provide text or icon information to help classify scenarios, so we choose to use their feature extraction approach. For the icon information, AppFlow converts it into text descriptions, so that we can finally get a text (keyword) list for each screen. The keywords of the three types of information in each screen are extracted in the following ways.

Layout File. Each screen has a corresponding layout file in XML format, which contains the attribute information of UI widgets on the screen and the relations between widgets. Each layout file

can be parsed into a tree, and each node in the tree corresponds to a UI widget. The resource-id attribute of each widget is the id information assigned to help developers cite key resources. The text attribute is used to describe the functionality or usage of the widget, usually the text displayed on the screens. The content-desc attribute helps users understand the purpose of the widgets. These three attributes are all represented in the form of strings in the layout file, and we can directly extract these texts and process them in a specific word segmentation approach. For some icon widgets that do not contain texts, the above attributes can often give their real functional descriptions. Although the class attribute cannot provide more specific functional information than the above three, some special widgets such as EditText and ImageView can still provide some functional information.

For some widgets that are not actually displayed in the interface or are not important, we determine their status as *hidden* by comparing the attribute information with the text on the screen recognized by OCR. So for a layout file, we traverse all nodes in pre-order. For each node, we first judge whether it belongs to the hidden state according to specific rules. If so, we skip the node. Otherwise, we first extract keywords from their resource-id, text, content-desc, class attribute. Because the position and size information of the widget can also provide some functional information, the bounds attribute is converted into keywords such as TOP, WIDE, BIG, XXL, etc. according to the AppFlow's approach, and then we calculate the Cartesian product of these keywords and keywords from resource-id attribute, these keywords and keywords from class attribute separately to supplement the extracted keyword list.

Screenshot. In the practice of manual classification, we will check the text prompts shown on the screenshot to judge the functionality of the current screen. So for each screenshot, we use Optical Character Recognition (OCR) to automatically extract all words as keywords for this part.

Activity Information. In Android applications, an activity is usually a separate screen, which can display some widgets or listen to and respond to user events, so each screen has its activity name. Although many developers make most screens share the same activity names for brevity, generally speaking, there are some activity names containing relevant functional scenario information. For example, the screens corresponding to ".bookmarks.BookmarkListActivity" belong to functional scenarios related to bookmarks. Therefore, we use a custom tokenization approach to segment the activity name to extract keywords.

Term Frequency–Inverse Document Frequency (TF-IDF) is used to evaluate the importance of a word in an article or in a corpus, so we use TF-IDF to convert the extracted keywords from each screen into numerical features. We take each keyword list from a screen as an article in TF-IDF, the keywords as its words, and all the screens of applications in the training set as a corpus, so that a feature vector with a fixed dimension can be calculated for each screen.

3.7 GNN Classification Process

In the practice of manual scenario classification of the screens, we found that the screens are not isolated and the connections of the screens can help us to classify the functional scenarios, so we connect the screens in the form of edges. The goal of a GNN is to recognize patterns in graph data, based both on the data within the nodes and the inter-connectivity. So we use a GNN to learn the transition graph and capture the potential functional dependencies between screens.

Common GNN architectures include the classic and basic Graph Convolutional Network (GCN) [12], Graph Attention Networks (GAT) [20]with multi-head attention, and Graph Sample And Aggregate (GraphSAGE) [9] with inductive learning. Each functional scenario often involves multiple consecutive screens, and the dependencies between functional scenarios also involve multiple scenarios. Therefore, as shown in Figure 5, the transition graph has a deep depth or long diameter. The GNN architectures mentioned above usually only support neighborhood aggregation with a small number of convolutional layers. When there are too many layers, these models may overfit due to too many parameters, or the gradient will be vanished or exploded during backpropagation. We employ the Gated Graph Neural Networks (GGNN) [13] combined with modern recurrent neural networks to solve this problem. Formula 2 below shows how it works. Node i first aggregates messages from its neighbors, then uses the Gated Recurrent Unit (GRU) [3] to combine messages from its previous layer with messages from neighbors to update its new hidden state or representation vector. When aggregating neighborhood messages, the aggregation function we use is not *Sum* but *Mean*, because we believe the information provided by multiple source screens is similar and *Mean* is conceptually better fitted to our setting.

$$\begin{aligned} h_i^0 &= [x_i \| \mathbf{0}] \\ a_i^t &= \sum_{j \in \mathcal{N}(i)} W_{e_{ij}} h_j^t \\ h_i^{t+1} &= \mathrm{GRU}(a_i^t, h_i^t) \end{aligned} \tag{2}$$

The transition graphs of the training set, the extracted features of the nodes in the graphs, and the scenario label information of nodes are input into GGNN for training. Since our case is a multi-class problem, the loss function uses the cross-entropy, and the optimizer uses the mainstream Adam [11]. The parameters of the initial GGNN model are random, and the parameters are updated through forward propagation, back propagation, and gradient descent. Finally, the GGNN model converges. When classifying screens of a new app, we just need to input the transition graph and initial node features into trained GGNN, and the model outputs the label results of these nodes, that is, the label information is the scenario classification result of the corresponding screens.

4 EVALUATION

To evaluate our approach, we conduct the following experiments on our constructed dataset to answer the following two questions:

RQ1: *How effectively does our approach classify functional scenarios? (Is ours more accurate than existing approaches?)*

RQ2: *Is GGNN the most effective approach in this situation among several classic GNN models?*

Table 1: Definition of 16 pre-defined scenarios

Scenario	Description
Browsing	Viewing files and folders in different folders.
Compressing	Compressing files in different forms, such as 7z, zip, encrypted, etc.
Searching	Searching for files, such as limited search, global search, etc.
Playing Music	The operations related to music files, such as playing, pausing, creating a new playlist, etc.
Sidebar	The sidebar of this app.
Deleting	Deleting files and emptying deleted files.
Decompressing	Decompressing the file to the specified folder, which involves inputting the unzip password, etc.
Editing	Editing the content of text files.
Creating	Creating a new file or folder.
Selecting	Selecting one or more files and folders after long pressing.
Pasting	Pasting the files that have been copied or cut.
Viewing properties	Viewing the properties of the file, such as size, modification information, md5, etc.
Renaming	Renaming the file or folder.
Menu	The menu of this app.
Setting	Setting various configurations of the applications, including background, color, font, etc.
Bookmarks	The operations related to bookmarks, such as adding, modifying, deleting, renaming, etc.

4.1 Experiment Setup

Application Collection. We choose file management as the category of experimental applications because we find that such applications tend to contain many of the same functional scenarios. We download the top 40 file management applications by popularity from Google Play and F-droid. In the manual installation and operation stage, we remove ten applications that do not meet our needs, such as having no common functional scenarios, failing to run with the Appium testing framework, etc. At the same time, we discover and record 16 common functional scenarios, including searching for files, creating new files, and compressing files and other scenarios. The specific scenario definitions are shown in Table 1.

Table 2: Data of 30 applications in experiments

App	Package	Version	Screens	Transitions
OI File Manager	org.openintents.filemanager	2.2.3	310	550
File Manager Pro	com.michaldabski.filemanager	0.5	146	308
AmazeFileManager	com.amaze.filemanager	3.5.3	364	555
MaterialFiles	me.zhanghai.android.files	1.2.0	564	882
AnExplorer	dev.dworks.applications.anexplorer	4.1.1	298	461
Simple File Manager	com.simplemobiletools.filemanager.pro	6.8.7	276	496
Mi File Manager	com.mi.android.globalFileexplorer	1-210304	172	333
RS File Manager	com.rs.explorer.filemanager	1.7.1	316	521
Cx File Explorer	com.cxinventor.file.explorer	1.5.1	147	249
File Manager Plus	com.alphainventor.filemanager	2.6.3	407	645
Files by Google	com.google.android.applications.nbu.files	1.0.357865958	218	381
FileMaster	com.root.clean.boost.explorer.filemanager	1.2.4	298	446
DV	dv.fileexplorer.filebrowser.filemanager	1.9.18	331	515
Solid Explorer File Manager	pl.solidexplorer2	2.8.9	288	483
FX File Explorer	nextapp.fx	8.0.1.0	410	789
Explorer	com.speedsoftware.explorer	3.9.1	171	350
FE File Explorer	com.skyjos.applications.fileexplorerfree	4.3.3	440	738
File Manager by Lufick	com.cvinfo.filemanager	5.0.3	406	755
Simple FileManager	com.simple.filemanager	1.1.06	240	390
FS File Explorer	com.ioapplications.fsexplorer	4.0.6	589	805
ESx File Manager & Explorer	filemanager.fileexplorer.manager	1.4.2	199	362
ES File Explorer	com.estrongs.android.pop	4.2.0.3.4	496	794
File Manager	files.fileexplorer.filemanager	1.0.3.3	153	256
Simple Explorer	com.dnielfe.manager	2.3.1	347	499
File Manager	com.cv.filemanager	3.3.4	285	531
File Manager	com.itel.filemanager	3.0.10	256	441
MK Explorer	pl.mkexplorer.kormateusz	2.5.4	394	681
File browser	filebrowser.filemanager.file.folder.app	1.0.5	232	423
File Manager	fm.clean	1.13.0	185	333
File Manager	com.photovideotools.file.manager	1.5.0	139	245

Screenshot Collection. To get the screenshots of each app, we write test UI test scripts for these 30 applications, which mainly revolve around the above 16 functional scenarios. Running these test scripts, we can automatically collect all the required data, and manually label these screenshots for the following stages (i.e., training and testing), corresponding to the 16 functional scenarios. According to the approach of Section 3.5, we remove the duplicated UI states and build a transition graph for each app respectively. The specific data of the 30 applications are shown in Table 2, where the number of screens and transitions is the number after deduplication. For instance, column 4 and 5 in row 2 separately denotes the transition graph of OI File Manager containing 310 nodes (screens) and 550 edges (transitions).

Data Split. We randomly select 24 applications from the 30 applications as the training set, 3 as the validation set, and 3 as the test set. That is, the split for training/validation/test set is 80%: 10%: 10%. We randomly split 5 times in this way (denoted in later tables as *Data1*, *Data2*, etc.) to avoid training bias. Our task is to perform a 16-class classification for each dataset respectively.

4.2 RQ1: Effectiveness compared to other approaches

To test the validity of the feature extraction and learning model for scenario classification from the perspective of ablation, we introduce the similarities and differences between our approaches and others. We use the notation of *feature extraction + model* to represent an approach.

Our Approach. The widget information in layout files, widgets in screenshots, and activity names contain abundant semantic information, which can be fully utilized to understand the functionality of different scenarios. We call our procedure for extracting semantic features *Text Extraction* (abbreviated as *Text* in Table 3 and 4).

Different from AppFlow [10], we preserve the transitions between screens and build a transition graph. Due to the long diameter of the graph, we choose Gated Graph Neural Networks (GGNN) as our classifier. Because the keyword features are sparse, we use a fully connected layer (FC) to reduce the feature dimension. Based on actual verification and experience, we use the 4-layer neighborhood aggregation, which is more suitable for our task. Therefore our GGNN model contains one fully connected layer, four convolutional layers, one fully connected layer for classification, trained with Adam optimizer with a learning rate of 0.01. We use Pytorch [16] and Pytorch Geometric [5] for our implementation. We run our approach for 200 epochs for each dataset. We denote our approach as *Text + GGNN*.

Other Approaches. In AppFlow, they utilize Multilayer Perceptron (MLP) as a classifier, which contains a hidden layer with 68 neurons, optimized by a stochastic gradient-based optimizer. We denote their approach as $Text + MLP$. According to our experiment settings, we find that the results are slightly better with 60 neurons, so we modify their model parameter configuration.

Since the approach proposed by Luca et al. [1] has a slight improvement of the approach proposed by Ariel et al. [18], and they demonstrate better classification results, we do not compare with the latter [18]. In a recent work [1], the approach divides the screen into three areas by the ratio of 20%: 60%: 20%, and count the number of various widgets of these three areas as features, such as the number of Clickable widgets on the top 20% of the screen, the number of EditText widgets in the bottom 20% of the screen, etc.

Table 3: Test accuracy compared to existing approaches

	Data1	Data2	Data3	Data4	Data5	Average
Text+MLP	78.48%	72.05%	88.16%	85.86%	89.56%	82.82%
Count+LR	43.87%	57.50%	62.01%	53.36%	52.89%	53.93%
LayoutMD5+GGNN	41.00%	40.33%	51.91%	35.66%	50.55%	43.89%
Text+GGNN	**89.52%**	**81.65%**	**93.69%**	**89.48%**	**92.50%**	**89.37%**

Table 4: Test weighted-F1 compared to existing approaches

	Data1	Data2	Data3	Data4	Data5	Average
Text+MLP	81.36%	72.09%	88.00%	85.36%	89.18%	83.20%
Count+LR	38.00%	50.00%	58.00%	48.00%	49.00%	48.60%
LayoutMD5+GGNN	26.27%	39.36%	47.18%	33.18%	44.45%	38.09%
Text+GGNN	**89.36%**	**81.64%**	**93.18%**	**89.36%**	**92.27%**	**89.16%**

In their experiments, it is verified that logistic regression performs the best. So we reproduce and apply their feature extraction and classification algorithm to our dataset. We denote their approach as *Count + LR*.

Q-testing utilizes the siamese network [2] to determine whether two UI states are in the same functional scenario, which is different from all the work mentioned above and our work. However, we can still use their feature extraction approaches for comparison. They extract features from the layout file of each screen and encode the attributes of selected nodes in the layout file. For example, they use the MD5 message-digest algorithm to hash resource-id attributes, text attributes, etc., into numbers and normalize the four integers of the bounds attribute. We apply it in conjunction with GGNN to our data and denote this approach as *LayoutMD5 + GGNN*.

For five sets of data split, the parameters of each of the above models are fixed. We set 10 random seeds, respectively, and each approach is run ten times to take the mean value as the result. The accuracy of all results is shown in Table 3. Because our task is a multi-class problem and there is a class imbalance in our data, we also leverage weighted-F1 scores of all results as another evaluation metric, as shown in Table 4.

According to Table 3 and Table 4, feature extraction *Text* contributes the best to scenario classification, whether it is combined with simple model *MLP* (row 2) or novel model *GGNN* (row 5). As expected, *Text* is the most reasonable feature extraction method for classifying scenarios among all the other ways, which fully utilizes the corresponding semantic information via layout files, screenshots, and activity information. Comparing row 2 and row 3, it can be indicated that utilizing *Count* as features extraction ignores a lot of important text information, which proves that relying only on the position and statistical information of widgets has less benefit on classifying functional scenarios. According to rows 4 and 5, the features of *LayoutMD5* are as high as 21400 dimensions, but the results are the worst, which may be attributed to the fact that GGNN cannot precisely capture the semantic information from high-dimensional features like Siamese LSTM.

For each dataset, our approach achieves the best results for both accuracy and weighted-F1. The average accuracy and weighted-F1 on the 5 datasets can both reach about 89%, the average accuracy is improved by 6.55% compared to AppFlow, and even the accuracy is improved by 11.04% on data 1, which also verifies that in the case of the same screen features, the transitions between screens can help us classify functional scenarios.

Table 5: Test accuracy compared to other GNNs

	Data1	Data2	Data3	Data4	Data5	Average
GCN (K=4)	52.21%	35.40%	54.43%	56.92%	59.39%	51.67%
GraphSAGE (K=4)	81.00%	67.73%	87.57%	88.19%	86.25%	82.15%
GAT (K=4)	52.98%	37.62%	59.40%	62.88%	63.05%	55.19%
GIN (K=4)	45.61%	35.78%	50.32%	51.18%	52.99%	47.18%
GGNN (K=4)	**89.52%**	**81.65%**	**93.69%**	**89.48%**	**92.50%**	**89.37%**
GCN (K=1)	69.78%	67.43%	78.08%	78.72%	83.45%	75.49%
GraphSAGE (K=2)	83.31%	80.43%	93.30%	88.17%	91.58%	87.36%
GAT (K=1)	68.57%	64.90%	75.51%	77.58%	78.48%	73.01%
GIN (K=1)	65.74%	62.88%	75.84%	76.28%	79.01%	71.95%

Table 6: Test weighted-F1 compared to other GNNs

	Data1	Data2	Data3	Data4	Data5	Average
GCN (K=4)	51.73%	34.18%	53.73%	56.00%	57.46%	50.62%
GraphSAGE (K=4)	81.82%	69.09%	87.82%	88.27%	85.73%	82.55%
GAT (K=4)	51.64%	38.36%	57.64%	62.36%	61.55%	54.31%
GIN (K=4)	44.09%	34.82%	48.28%	50.36%	52.64%	46.04%
GGNN (K=4)	**89.36%**	**81.64%**	**93.18%**	**89.36%**	**92.27%**	**89.16%**
GCN (K=1)	69.64%	63.36%	76.64%	77.91%	82.91%	74.09%
GraphSAGE (K=2)	83.64%	79.73%	93.09%	88.18%	91.09%	87.15%
GAT (K=1)	67.45%	61.09%	74.36%	76.82%	77.82%	71.51%
GIN (K=1)	64.64%	63.27%	75.82%	75.64%	78.00%	71.47%

4.3 RQ2: Effectiveness compared to other GNNs

To evaluate the effectiveness of GGNN and find the better GNN structures, we replace the GGNN with other common GNN structures mentioned above, including GCN, GraphSAGE, and GAT. We also experiment with the most popular expressive GNN framework: Graph Isomorphism Network (GIN) [22]. All models use the same *Text* features as initial node features and use the same hyperparameters: 1 fully connected layer for reducing sparse node features, 200 dimensional hidden units, four GNN layers (K=4), one fully connected layer for classification. We choose four layers considering that the diameter of our graph is relatively long, which corresponds to a functional scenario with multiple continuous screens. The final results are shown in the first six rows in Tables 5 and 6. We can see that when dealing with deeper networks, the GGNN combined with GRU has the best accuracy and weighted-F1 results on each data, the results of GraphSAGE are sub-optimal, and the results of other models can only be around 50%. This also verifies our conjecture in Section 3.7: other models are only suitable for dealing with networks containing 1-3 layers.

To ensure the completeness of the experiment and verify the influence of layer numbers on model performance, we control the variable of layer number with better results for other models based on our data. GCN, GAT, and GIN have better results with one layer (K=1), respectively, and GraphSAGE has two layers (K=2). The final results are shown in the bottom four rows of Tables 5 and 6. Even if we adjust the number of layers for each model, they are still not suitable for our task, and GGNN (K=4) is still optimal. All results confirm the effectiveness of the GGNN.

5 RELATED WORK

There is a large number of related works on the classification and matching of screens in Android applications [1, 10, 14, 15, 18, 23], most of which are related to UI automated testing and test reuse. Their work uses one or more layout files, screenshots, and activity names of screens to help classification. For example, Ariel et al. [18] and Luca et al. [1] both extract the number of widgets they consider important from the layout files and adopt a basic classification algorithm similar to KStar or logistic regression. They only utilize a small amount of screen information: the position and number of widgets, so their feature dimensions are only two digits. Their approach can only be applied to relatively simple and rough scenario classification. Pan et al. [14] utilize scenario comparison to help optimize the exploration strategy of automated testing tools. Hu et al. [23] use accurate scenario classification to help synthesize reusable UI tests and combine the three types of information of the screen in a better and more reasonable way with a neural network. Compared with these tools, we take advantage of the transitions between screens with the GNN model for the first time and improve the classification accuracy.

6 CONCLUSION AND FUTURE WORK

In this paper, we focused on the problem of classifying functional scenarios in Android applications and proposed transitions and dependencies between screens to improve classification accuracy. We constructed a directed transition graph to preserve transition information, whose nodes are screens and edges are transitions. We proposed a novel GNN-based scenario classification approach using GNNs to learn directed transition graphs and classify screens. We evaluated our approach on 30 popular file management apps. The results showed that both the classification accuracy and weighted-F1 of our approach are improved by about 6% compared to the state-of-the-art approach. Currently, we use transitions as edges in a direct way. In the future, we will try to extract useful widget information from transitions, optimize feature extraction, and experiment with other categories of apps to further improve our work.

ACKNOWLEDGMENTS

This research is supported by the National Natural Science Foundation of China under Grant No.61972193.

REFERENCES

[1] Luca Ardito, Riccardo Coppola, Simone Leonardi, Maurizio Morisio, and Ugo Buy. 2020. Automated test selection for Android apps based on APK and activity classification. *IEEE Access* 8 (2020), 187648–187670.
[2] Jane Bromley, Isabelle Guyon, Yann LeCun, Eduard Säckinger, and Roopak Shah. 1993. Signature verification using a" siamese" time delay neural network. *Advances in neural information processing systems* 6 (1993), 737–744.
[3] Kyunghyun Cho, Bart van Merriënboer, Dzmitry Bahdanau, and Yoshua Bengio. 2014. On the Properties of Neural Machine Translation: Encoder–Decoder Approaches. In *Proceedings of SSST-8, Eighth Workshop on Syntax, Semantics and Structure in Statistical Translation*. 103–111.
[4] Li Deng and Dong Yu. 2014. Deep learning: methods and applications. *Foundations and trends in signal processing* 7, 3-4 (2014), 197–387.
[5] Matthias Fey and Jan Eric Lenssen. 2019. Fast graph representation learning with PyTorch Geometric. In *Proceedings of the 7th International Conference on Learning Representations, RLGM Workshop (ICLR'19)*.
[6] JS Foundation. 2022. Appium: Mobile App Automation Made Awesome. http://appium.io/. Online; accessed 15-Feb-2022.
[7] Google. 2022. ListView. https://developer.android.google.cn/reference/android/. Online; accessed 15-Feb-2022.
[8] Google. 2022. Monkeyrunner. https://developer.android.com/studio/test/monkeyrunner. Online; accessed 15-Feb-2022.
[9] William L Hamilton, Zhitao Ying, and Jure Leskovec. 2017. Inductive representation learning on large graphs. In *Proceedings of the 31st International Conference on Neural Information Processing Systems (NIPS'17)*. MIT Press, 1025–1035.
[10] Gang Hu, Linjie Zhu, and Junfeng Yang. 2018. AppFlow: using machine learning to synthesize robust, reusable UI tests. In *Proceedings of the 2018 26th ACM Joint Meeting on European Software Engineering Conference and Symposium on the Foundations of Software Engineering (ESEC/FSE'18)*. ACM, 269–282.
[11] Diederik P Kingma and Jimmy Ba. 2015. Adam: A method for stochastic optimization. In *Proceedings of the 3th International Conference on Learning Representations (ICLR'15)*.
[12] Thomas N Kipf and Max Welling. 2017. Semi-supervised classification with graph convolutional networks. In *Proceedings of the 5th International Conference on Learning Representations (ICLR'17)*.
[13] Yujia Li, Richard Zemel, Marc Brockschmidt, and Daniel Tarlow. 2016. Gated graph sequence neural networks. In *Proceedings of the 4th International Conference on Learning Representations (ICLR'16)*.
[14] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of Android applications. In *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'20)*. ACM, 153–164.
[15] Minxue Pan, Tongtong Xu, Yu Pei, Zhong Li, Tian Zhang, and Xuandong Li. 2020. GUI-Guided Test Script Repair for Mobile Apps. *IEEE Transactions on Software Engineering* (2020), 1–1.
[16] Adam Paszke, Sam Gross, Soumith Chintala, Gregory Chanan, Edward Yang, Zachary DeVito, Zeming Lin, Alban Desmaison, Luca Antiga, and Adam Lerer. 2017. Automatic differentiation in pytorch. In *Proceedings of the 31st International Conference on Neural Information Processing Systems,Workshop (NIPS'17)*. MIT Press.
[17] Robotium.org. 2022. Android UI Testing. http://www.robotium.org. Online; accessed 15-Feb-2022.
[18] Ariel Rosenfeld, Odaya Kardashov, and Orel Zang. 2017. *Automation of Android Applications Testing Using Machine Learning Activities Classification.* http://arxiv.org/abs/1709.00928
[19] Selenium. 2022. WebDriver. https://www.selenium.dev/documentation/webdriver/. Online; accessed 15-Feb-2022.
[20] Petar Veličković, Guillem Cucurull, Arantxa Casanova, Adriana Romero, Pietro Lio, and Yoshua Bengio. 2018. Graph attention networks. In *Proceedings of the 6th International Conference on Learning Representations (ICLR'18)*.
[21] Zonghan Wu, Shirui Pan, Fengwen Chen, Guodong Long, Chengqi Zhang, and Philip S. Yu. 2021. A Comprehensive Survey on Graph Neural Networks. *IEEE Transactions on Neural Networks and Learning Systems* 32, 1 (2021), 4–24.
[22] Keyulu Xu, Weihua Hu, Jure Leskovec, and Stefanie Jegelka. 2019. How powerful are graph neural networks?. In *Proceedings of the 7th International Conference on Learning Representations (ICLR'19)*.
[23] Tongtong Xu, Minxue Pan, Yu Pei, Guiyin Li, Xia Zeng, Tian Zhang, Yuetang Deng, and Xuandong Li. 2021. GUIDER: GUI structure and vision co-guided test script repair for Android apps. In *Proceedings of the 30th ACM SIGSOFT International Symposium on Software Testing and Analysis (ISSTA'21)*. ACM, 191–203.
[24] 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.