Guided Bug Crush: Assist Manual GUI Testing of Android Apps via Hint Moves

Zhe Liu[*]
Institute of Software
Chinese Academy of Sciences
Beijing, China
liuzhe2020@iscas.ac.cn

Chunyang Chen
Monash University
Melbourne, Australia
Chunyang.chen@monash.edu

Junjie Wang[*] [†]
Institute of Software
Chinese Academy of Sciences
Beijing, China
junjie@iscas.ac.cn

Yuekai Huang[*]
Institute of Software
Chinese Academy of Sciences
Beijing, China
yuekai@iscas.ac.cn

Jun Hu[*]
Institute of Software
Chinese Academy of Sciences
Beijing, China
hujun@iscas.ac.cn

Qing Wang[*] [†]
Institute of Software Chinese
Academy of Sciences
Beijing, China
wq@iscas.ac.cn

ABSTRACT

Mobile apps are indispensable for people's daily life. Complementing with automated GUI testing, manual testing is the last line of defence for app quality. However, the repeated actions and easily missing of functionalities make manual testing time-consuming and inefficient. Inspired by the game candy crush with flashy candies as hint moves for players, we propose an approach named `NaviDroid` for navigating testers via highlighted next operations for more effective and efficient testing. Within `NaviDroid`, we construct an enriched state transition graph with the triggering actions as the edges for two involved states. Based on it, we utilize the dynamic programming algorithm to plan the exploration path, and augment the GUI with visualized hints for testers to quickly explore untested activities and avoid duplicate explorations. The automated experiments demonstrate the high coverage and efficient path planning of `NaviDroid` and a user study further confirms its usefulness. The `NaviDroid` can help us develop more robust software that works in more mission-critical settings, not only by performing more thorough testing with the same effort that has been put in before, but also by integrating these techniques into different parts of development pipeline.

KEYWORDS

GUI testing, Android App, Software Engineering, Quality Assurance

ACM Reference Format:
Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2022. Guided Bug Crush: Assist Manual GUI Testing of Android Apps via Hint Moves. In *CHI Conference on Human Factors in Computing Systems (CHI '22), April 29-May 5, 2022, New Orleans, LA, USA.* ACM, New York, NY, USA, 14 pages. https://doi.org/10.1145/3491102.3501903

[*]Also With University of Chinese Academy of Sciences, Beijing, China;
Laboratory for Internet Software Technologies, Beijing, China;
State Key Laboratory of Computer Sciences, Beijing, China;
Science & Technology on Integrated Information System Laboratory, Beijing, China
[†]Corresponding author

1 INTRODUCTION

Due to the portability and convenience of mobile phones, they are more and more popular in the current world. Given 3 million available Android applications (apps) [8] for different tasks such as reading, shopping, banking and chatting [1], mobile phones and apps now have become indispensable for our daily life in accessing the world [20, 23]. The importance of mobile apps makes it vital for the development team to carry out a thorough testing for ensuring the quality of the mobile apps. The quality of the mobile application also decides its success among many similar apps in the market.

However, it is challenging to guarantee the mobile application quality, especially considering that mobile applications are event-centric programs with rich graphical user interfaces (GUIs) [91], and interact with complex environments (e.g., users, devices, and other apps). To ensure the application quality, there are mainly two kinds of GUI testing i.e., automated GUI testing and manual GUI testing. There have been many automated GUI testing studies for mobile applications including model based [67, 99, 100], probability based [59, 61, 101] and deep learning based [54, 70] approaches. Most of them are dynamically exploring mobile apps by executing different actions such as scroll and click with random input, based on the analysis of the code structure of the current page. Albeit its convenience and scalability, it also has the following challenges. For example, automated testing may not have a high activity coverage, especially the deeper UI pages and functions are difficult to cover [59, 61, 82]. Complex operations are difficult to implement, especially for those functionalities which can only be reached by complicated inputs or a long sequence of actions [19, 79, 90]. Furthermore, the usability and accessibility bugs (e.g., color schema, font size, interaction) [22, 97, 98] are difficult to reveal by automated GUI testing [12, 27, 36, 72, 75, 77].

Therefore, in addition to automated GUI testing, companies also adopt manual testing as the last line of defence [35, 51, 55, 94]. Research also showed that due to the usability and learning curve of automated tools, manual testing is preferred by many software developers [41, 42, 55]. Within the manual testing, multiple testers

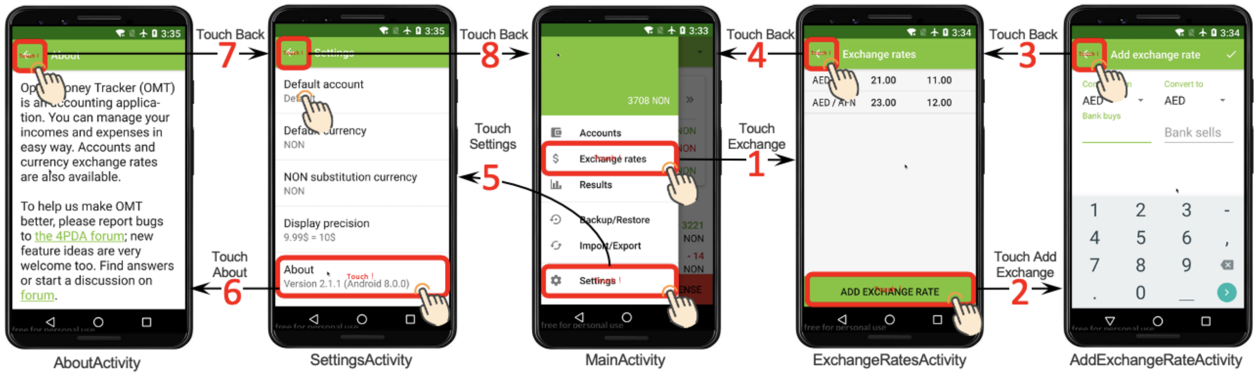


Figure 1: Example of the NaviDroid usage scenario.

mimic users' behaviors to explore different functionalities in the apps aiming to find more bugs [47, 58, 80, 94]. Some researchers optimize manual testing according to test cases and prioritization, but these techniques usually rely on specific data [43, 44, 49]. To harness crowdsourcing's diversity, crowdsourced testing [45, 76, 85] recently emerges in software testing which exploits the benefits, effectiveness, and efficiency of crowdsourcing and the cloud platform, to replace conventional manual testing which limits the fixed number of testers. Compared with automated GUI testing, human testers are able to discover more diverse and complicated bugs, especially those related to user experience. But there are also two problems with the manual GUI testing. First, it is time-consuming which requires a large number of testers to manually explore each screen of the app, and testers may execute repeated actions during the exploration [33, 34, 42]. Second, the performance of manual testing is unstable as it highly depends on the testers' capability and experience, and testers may miss some minor functionalities especially for those unfamiliar apps [42, 44, 68]. Therefore, the target users of this paper are the testers who are not familiar with the application they are testing.

There are always pros and cons of automated GUI testing and manual GUI testing, and separating them into two unrelated processes may further deepen their cons [47, 55]. To leverage the pros of both testing techniques, we propose a new hybrid method to assist manual GUI testing based on the insights from the automated GUI testing. Inspired by the automated GUI testing, we first distill the prior knowledge of one app including all states and state relationships. We then implement that prior knowledge into a tool NaviDroid. During manual GUI testing, our NaviDroid will trace testers' testing steps and help navigate or remind testers with unexplored pages by explicit visual annotations (e.g., red bounding box) in the run-time page as seen in Fig 1. Our NaviDroid can help human testers avoid missing some functionalities or making repeated exploration steps.

Within our approach, there are mainly three components including distilling prior knowledge via the program analysis, planning the exploration path, and providing visual-based path guidance for testing the applications. ***First***, STG_{action} is constructed, which is a state transition graph with each edge annotated with the trigger action between two states (e.g., clicking the "login" button in the state *login* to the next state *landingPage*). We combine the static program analysis and dynamic random exploration to ensure the states and trigger actions are accurately captured. We also design a context-aware state merging method to merge the near-duplicate states, by considering both the current state and the adjacent states. ***Second***, based on the extracted graph, we utilize dynamic programming algorithm to plan the exploration path to efficiently cover all the states with few repeated exploration steps. ***Third***, following the planned path, we augment the run-time GUI with visual hint moves to provide real-time guidance in mobile GUI testing. That process is similar to the flashing candies (hint/suggested moves) when a player hesitates to make a move in playing Candy Crush (It is a popular free-to-play match-three puzzle video game in Google Play) [7]. According to our observation, that suggested move is particularly useful when the trigger components are small or poorly designed/developed.

As the NaviDroid consists of STG extraction algorithm, dynamic programming algorithm and visual guidance. Therefore, we first evaluate the algorithm performance of the NaviDroid through an automated method. We evaluated the NaviDroid on 85 open-source apps from F-Droid (the largest repository of open-source Android Apps). Results show that NaviDroid can achieve 74% median activity coverage and 81% median state coverage with the extracted STG_{action}, outperforming five commonly-used and state-of-the-art baselines. It also saves 20% to 42% exploration steps compared with the three commonly-used baselines. We further carry out a user study to evaluate its usefulness in assisting manual GUI testing, with 20 apps from F-droid. Results show that, the participants with NaviDroid cover 62% more states and 61% more activities, detect 146% more bugs within 33% less time, compared with those without using NaviDroid. This confirms the usefulness of NaviDroid in avoiding missing functionalities and making repeated exploration steps, and helping detecting bugs during manual testing. The demo video link is https://youtu.be/7kR9-9-gPQ0. The contributions of this paper are as follows:

- The first Android manual testing assistant NaviDroid[1], to the best of our knowledge, by providing visual-based exploration path guidance based on the states and trigger actions, which can avoid missing functionalities or making repeated exploration steps.

[1]We release the source code, experiment detail, and demo videos of our NaviDroid in https://github.com/20200829/Navidroid. The demo video link is https://youtu.be/7kR9-9-gPQ0

- STG_{action} extraction method, which combines static analysis and dynamic exploration to acquire the states and related trigger actions, and incorporates context-aware state merging method for near-duplicate state merging.
- The DP-based (dynamic programming) path planning method, which guides the path exploration in covering all the states with few repeated exploration steps.
- Effectiveness evaluation on real-world mobile apps with promising results, and a user study demonstrating the usefulness of `NaviDroid` in assisting manual GUI testing and finding practical bugs.

2 RELATED WORK

With the rapid growth of the number of apps, their functions are also increasing. This leads to the researchers being more and more concerned about bugs from Android apps [91]. GUIs are the primary UI in the vast majority of today's commodity software [21, 37, 58, 64, 104]. However, creating GUI tests to cover a large number of Android UI pages is a challenging task.

Automated GUI Testing. To save human efforts in manual testing, many researchers explored the automatic generation approaches of large-scale test scripts for application testing [93]. There are a number of linting tools [4, 5, 99] based on the static program analysis to mark programming errors, bugs, style errors, and suspicious constructs to ensure that the application works properly. Due to the complexity of mobile apps, random GUI testing techniques based on dynamic exploration are proposed [19, 61], which aims to cover more components or activities. These dynamic GUI testing tools [18, 54, 79, 103] can simulate human operation (e.g., click, long press, slide, etc.) to test the application.

Although automated GUI testing can quickly spot functionality bugs of mobile apps, it has limitations in detecting the bugs related to complicated sequential operations or about app usability and accessibility [12, 27, 36, 72, 75, 77], which requires manual testing. And the automated GUI testing may not have a high activity coverage, such as the functions and interface that are easy to be ignored are difficult to cover [59, 61, 82]. Complex operations are difficult to implement, especially for those functionalities which can only be reached by complicated inputs or a long sequence of actions [19, 79, 90]. Empirical studies show that companies still rely on manual testing [35, 41, 42, 47, 51, 55, 62, 94]. The test execution is not a simple mechanical task, but a creative and experience-based process, and manual testing will never be replaced by automatic testing.

Manual GUI Testing. Manual testing can be defined as a process in which software testers manually verify the correctness of software functionalities according to the requirements provided by customers [29, 55, 56]. Many studies explored the factors influencing testers' performance such as training [65, 89], recruitment process [28, 92], and their experience [47]. However, all these studies are based on pure manual exploration which highly depends on the expertise and experience of testers. Some studies also optimize manual testing according to task priority [43, 44, 49], but they also rely on manually labeled data. None of them provides tools to guide testers to test applications more effectively, which is studied in this work. Crowdtesting is a newly developed manual testing schema [85–87] in which software development companies release test tasks through a crowd platform [2, 3]. Crowd workers conduct testing according to the description of testing tasks. However, researches showed that crowd workers always test the same function or repeated interfaces [84, 92], which leads to the waste of time and efforts of developers and testers. To avoid those repetitive operations, Zipt et al. [31] and Yan et al. [25] track each testers' behavior and aggregate them for reminding testers. In the context of GUI testing, a common method is to remove duplicate items from the list of test cases through post-event result analysis [84]. Other researchers have proposed incentive-based methods to reward testers who have found previously undetected cases [13]. Although these works can avoid duplicate efforts, they cannot help cover more activities or unrevealed bugs within the application. The testers may still miss some important application functionalities during testing. In addition to reminding users with testing repetition, our approach can guide the human testers during the manual exploration for achieving higher activity coverage and potentially uncovering more bugs.

Manual Test Assistant Technology. To give full play to the advantages of manual testing, previous studies utilized test assistant related technologies to improve the quality of manual testing [24, 40, 83]. SwiftHa[26] learned the models of Android applications and used them to discover unexplored states. Monkeylab [57] modeled user event interaction sequences on Android applications to generate new test cases. Polaris [62] simulated user interaction patterns learned from user behavior on Android applications, and then applied this simulation to different applications. Rico [30] proposed a hybrid method to record the application tracking of group workers for the first time, and then continued to explode programmatically to achieve a wider state space in the application. Patina [63] proposed an application-independent system for collecting and visualizing software application usage data. These methods combine the advantages of human intelligence and machine intelligence, so that the test cases are realistic and the testing tasks are scalable [39, 74]. However, the challenges of test duplication and incompleteness still remain. Chen et al. [25] proposed GUI level guidance on a web app to solve the problem of repeated testing by testers. There are three aspects distinguishing their work from ours. First, our approach targets at the Android apps, which has gained increasing popularity in people's daily life. Second, besides telling the testers where have been explored to avoid repetition, our approach can also guide the testers in exploring more UI pages to find more bugs. Third, we design an effective path exploration algorithm that helps to plan the optimized testing steps to void repetitions more effectively.

State Transition Graph Extraction. In the Android app, an state (e.g., activity) usually corresponds to an interface. The activity transition graph (ATG) reflects the relationship between activities. As the foundation of the downstream software testing task, ATG is usually used to analyze the relationship between activities in Android app. Zhang et al. [102] adopt the launch-mode-aware context-sensitive activity transition analysis method to extract ATG. Yang et al. [99] proposed a method of extracting static window transition graph (WTG) for Android based on window stack. Yet these methods can only obtain the coarse-grained ATG. Different from them, this work aims to obtain a fine-grained state transition graph

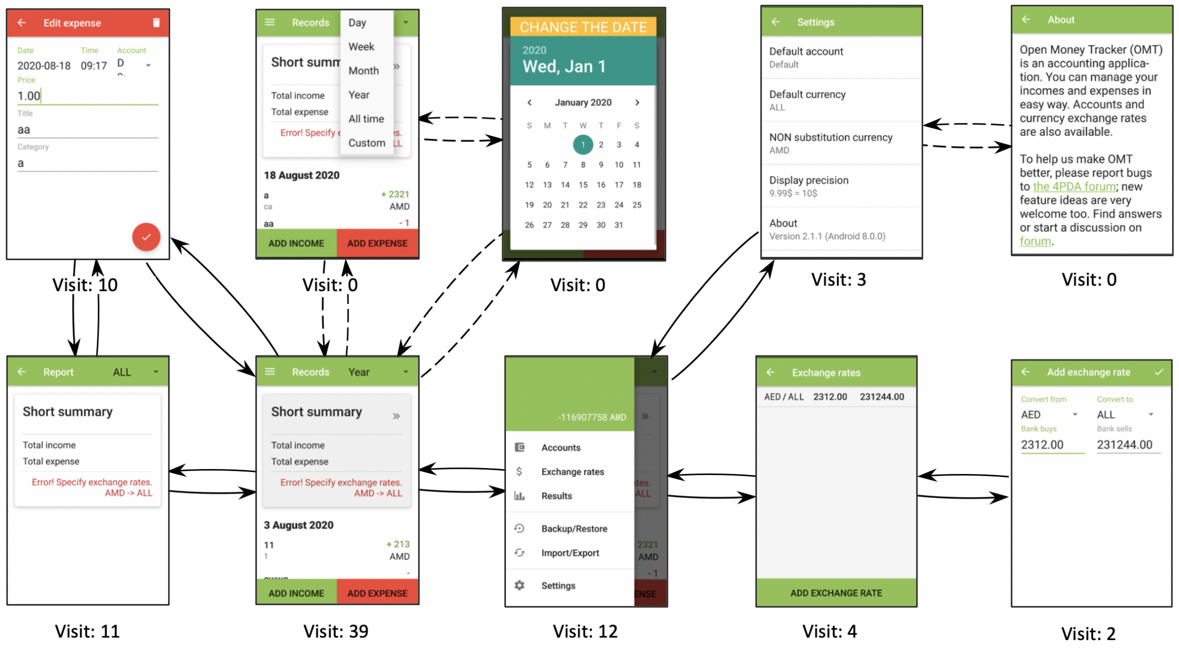


Figure 2: Example of one tester's exploration graph. The solid line represent the explored path while dotted line is the unexplored path. Number below each page is the visit time of the tester.

(STG) by combining both the static program analysis and dynamic exploration. Besides the nodes and edges in the graph, we enrich the STG by annotating the trigger actions along with the edge.

3 MOTIVATIONAL STUDY

To understand the bugs during manual exploratory testing [46, 48], we carry out an empirical study on observing testers' behavior. We randomly select 85 apps from F-Droid (the largest repository of open-source Android apps) according to the number of downloads, including 17 categories (e.g., connectivity, games, internet, money, reading, education, health) with 5 latest released apps in each category. Note that, to ensure the popularity of the experimental apps, these selected apps are also published in Google Play. Activity number in each app ranges from 10 to 30, and more detailed information of apps can be seen in our website[1]. We recruit 10 testers, all of whom major in computer science with more than 3 years of app testing experience. Six of them are from industry with practical working experience, while the other four are master students. Since the target users of this paper are testers who are not familiar with the application they are testing, none of the participants we selected have used the above apps.

Each tester independently completes the exploratory test of all apps, and the maximum test time of each application is set up 10 minutes. During the experiment, we ask testers to record their screen to ensure the validity of their testing and for further analysis. After the experiment, we carry out an informal interview with these 10 testers to further understand what they think about the testing. The following observations are found:

1) Low activity coverage vs. High confidence. The activity coverage rate of exploratory testing is only 57% on average, indicating that it is hard for human testers to cover all activities or functionalities of an application when conducting exploratory testing. As seen in Fig 2 (The MoneyTracker app), some functionalities are rarely explored such as "About" in the Setting page and "change the date" in the Main page by one user. Despite the low coverage, most of the participants are confident that they cover most functionalities of apps. Such blind confidence may significantly hurt the quality of testing.

2) Repeated visiting vs. Unawareness. 85% of the testers repeatedly visit the same page more than 10 times (e.g., "Account" and "Report" page in Fig 2), and 65% of the testers are trapped into the loop for more than 3 minutes according to our observation of the video recording. We also find that some testers hesitated for a long time on one page, without knowing the next step. Some of them mention that they visit some pages repeatedly, as they could not remember which page has been visited or which action can trigger a new page. However, there are still a large portion of participants that are unaware of their repeated testing within the app.

In summary, the low activity coverage of manual testing confirms the necessity of guidance during the testing process. The high confidence of testing and the unawareness of the repetition further indicates this practical need. In addition, the repeated visiting phenomenon motivates us in developing a state transition graph to record what has been explored and guide human testers to efficiently explore the uncover states in order to facilitate the exploratory app testing.

4 APPROACH

This paper proposes `NaviDroid` to navigate human testers in exploring apps to avoid missing functionalities or making repeated exploration steps. Fig. 3 presents the overview of `NaviDroid`, which consists of three main components. ***First***, given the app Android

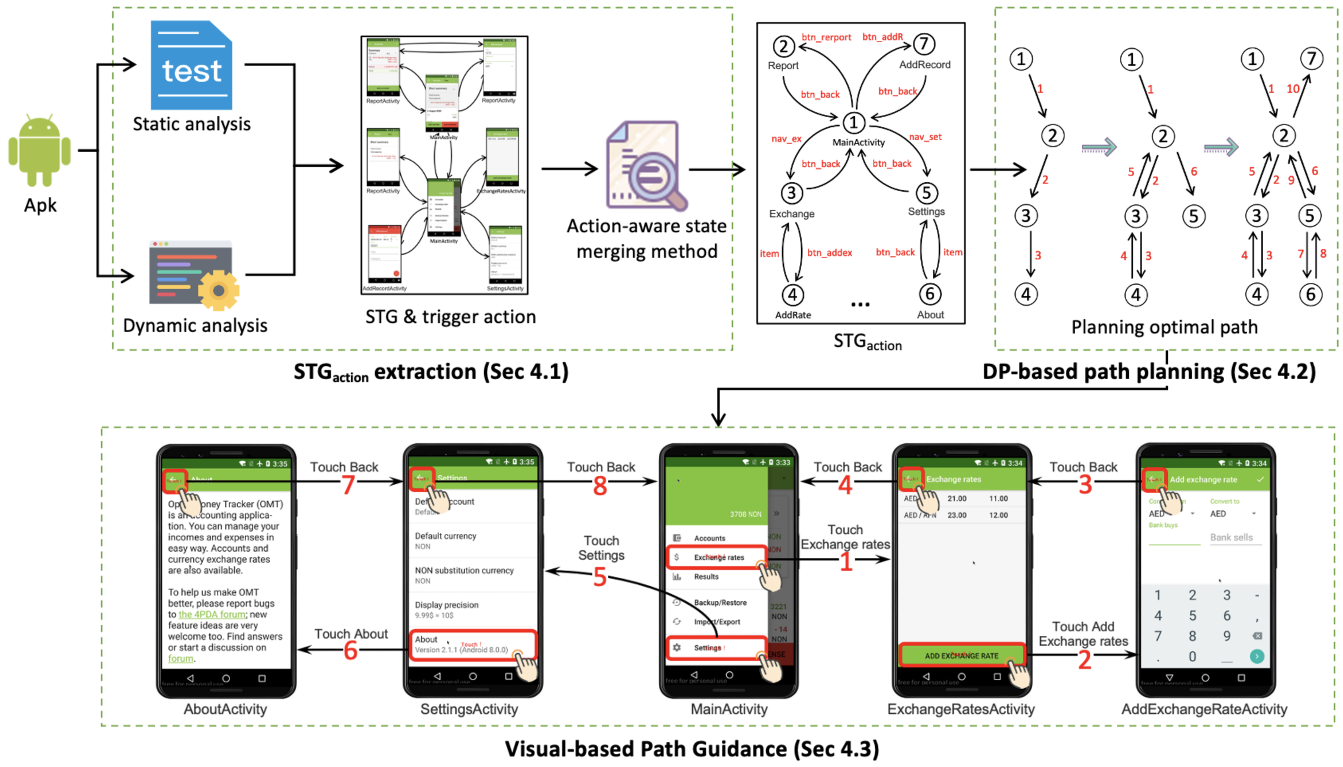


Figure 3: Overview of NaviDroid

package, the STG_{action} extraction component combines both static analysis and dynamic exploration to extract the state transition graph (STG) and its trigger actions between states (Section 4.1). We also design a context-aware state merging method to merge near-duplicate states by considering the current state and the adjacent states. ***Second***, based on the extracted STG_{action}, the DP-based path planning component plans the exploration path which aims at covering all the states of the app with few repeated exploration steps (Section 4.2). ***Third***, with the planned path, the visual-based path guidance component utilizes the visual augmentation technology to guide users' testing (Section 4.3).

4.1 STG_{action} Extraction

We first extract the state transition graph (STG), and then enrich it with trigger actions between the states to construct STG_{action}, which serves as the basis in guiding the users exploring the app. Both static analysis and dynamic explorations are used to ensure that accurate states and trigger actions are captured.

In our method, STG_{action} is defined as a graph $G < N, E >$ with node $N \in state$ and edge $E \in action$.

State: Previous automated testing works adopt different standards (e.g., activity, UI page) to abstract the *state* of an application [11, 14, 15, 70]. Research shows that more fine-grained *state* abstraction may lead to higher testing coverage [15]. Inspired by app GUI testing [32, 59, 70], we regard each unique UI page as one *state* and represent it by i.e., represented by UI components hierarchy tree, in which non-leaf nodes as layout components (e.g., LinearLayout, Framelayout) and leaf nodes as executable components (e.g., button); Each *activity* may have multiple *state*, that is, $N \in state \in activity$.

Action: Action is the trigger that results in *state* transition, which can be expressed as $E = ID,\ E \in action$. Where, ID is the identity of GUI component which receives the action.

4.1.1 *Static STG Extraction and Trigger Action Detection*.

In an Android application, activities can be started by invoking. For example, the $StartActivity(intent)$ is an inter-component communication (ICC) call [95], passing an intent that describes the activity to be launched [102]. By analyzing the ICC, the activity transition graph (ATG) [14, 67] can be extracted. That ATG can be regarded as an initial STG_{action}.

In detail, the target activity of ICC call is determined by querying the pointed-to values in the fields of an intent object. For example, $StartActivity(intent)$ determines the target activity to be started. By matching the parameter in $intent()$ method with the parameter in *AndroidManifest.xml* file, we obtain the transition between activities and build the initial ATG.

To guide the tester in manual testing, it is necessary to further extract detailed action (e.g., clicking a specific button) that triggers the activity transition. Given an application, the trigger action can be extracted by analyzing the call of $intent()$. We first transform the application source code to an abstract syntax tree (AST), and then traverse AST to locate $intent()$ method. There are two detailed types of it. First, if the $setOnClickListener()$ of a component (such as a button) directly calls the $intent()$ method, we will directly obtain the component name that calls the intent method. We traverse the AST again to obtain the corresponding trigger component which is represented by a certain ID (e.g., *R.id.btn_back*). Second, if the component calls the *intent*() method when calling other methods, we locate the component which calls *intent*() by traversing the method name *launchHome*(). After obtaining the component ID corresponding to the trigger action, the edge connecting two activities is determined, and we build the ATG_{action} (i.e., coarse-grained STG_{action}) accordingly.

4.1.2 *Dynamic STG Extraction and Trigger Action Detection.*

Some states and trigger actions, especially those in dynamic or mixed layout (such as dynamic rendering menu, list, navigation bar, appwidget), are difficult to be obtained by static analysis [88, 99]. Instead, it is easy to be captured with dynamic GUI rendering.

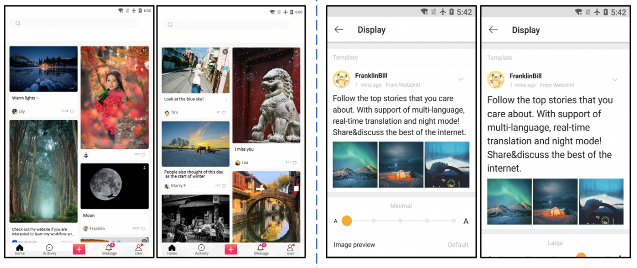
Therefore, to enrich the STG_{action} extracted in the above section, we further adopt dynamic exploration to detect more fine-grained states and actions. In detail, by leveraging the idea of dynamic app GUI testing [19, 32, 53, 79], we adopt an app explorer [53] to automatically explore the pages within an application through interacting with apps using random actions e.g., clicking, scrolling, and filling in text.

During the exploration, we record both *state* and trigger action *action* between states. Each *state* corresponds to the detailed view hierarchy file (XML file) of one Android application page. Each *action* corresponds to the component ID that triggers the state transition. For the case that the ID of some dynamic rendering components cannot be retrieved in the run-time GUI hierarchy, we analyze the 'text' corresponding to the component in the view hierarchy file and treat the text content as its ID in STG_{action}. If the component has no 'text', we treat the coordinates of the component as the component ID. We then combine the STG_{action} extracted from static analysis and dynamic exploration into one graph.

4.1.3 *Context-aware State Merging.*

Through static and dynamic analysis, we get STG_{action} which is composed of a large number of *states* and *actions*, in which some of them are duplicates [38, 60, 96]. To avoid state explosion, we develop a context-aware based approach to merge duplicate states. Existing works either consider run-time GUI hierarchy [70, 79] or visual features [96] to remove near-duplicate states. Nevertheless, some near-duplicate states can still be missed by these approaches, or some non-duplicate states might be wrongly detected. For example, streaming recommendation with different content in different time/users in Fig 4 (a) can hardly be correctly detected by visual features, while the same content in different font size settings in Fig 4 (b) could not be detected by run-time GUI hierarchy.

Therefore, to overcome those drawbacks, we consider not only the information within a certain state but also its context i.e., adjacent states, for state merging. Given STG_{action}, we first merge the states with the same GUI run-time hierarchy (XML file from *ADBdump* command) without considering detailed content (e.g., text or image) which may change dramatically. After that, we further merge states with similar GUI hierarchy by checking whether their $n-1$ *state* (i.e., the previous state which transits to the current state) and $n+1$ *state* (i.e., the next state to which current state transits) are similar. By referring to existing studies [53, 70, 96] and combining with our empirical observations, we set the threshold of similarity as 80%. Following these two operations, we finish the state merging.



Follow the top stories that you care about. With support of multi-language, real-time translation and night mode! Share&discuss the best of the internet.

Follow the top stories that you care about. With support of multi-language, real-time translation and night mode! Share&discuss the best of the internet.

(a) News & list scenario (b) Setting scenario

Figure 4: Example of failed cases with existing approaches.

Although we construct a STG_{action} for supporting the real-time path recommendation during testers' manual exploration, there is still potentially a gap between states stored in our STG_{action} and states in the real-world testing environment. As testers' input (e.g., text) or exploration path may be different from that in our static analysis and dynamic exploration, we need to map the live state to those in our STG_{action} which may be slightly different. We adopt the same approach described above for state mapping.

4.2 DP-based Path Planning

With the STG_{action} obtained in the previous section, we need to plan a path that can cover all the nodes (i.e., states) with few repeated steps, so as to serve as the basis for the follow-up testing guidance. To achieve this, we use a dynamic programming algorithm to derive the shortest path between each pair of nodes, and plan the path.

4.2.1 *Formalization of Planning Path.*

We formulate the path planning as a dynamic programming problem, and represent it by a 4-tuple: $\boldsymbol{< G, d, V, DP >}$.

***G*: Graph.** The STG_{action} ($G < \boldsymbol{N}, \boldsymbol{E} >$) obtained in the previous section, where $\boldsymbol{N}$ is the set of nodes *(i.e., states)*, and $\boldsymbol{E}$ is the set of edges *(i.e., triggered events)*.

***d*: Distance.** $\boldsymbol{d_{ij}}$ is the shortest distance between the state i and the state j.

***V*: Visit status.** $\boldsymbol{V}$ is the visit status of the current node, represented by binary numbers. 0 is not visited and 1 is visited.

***DP*: Dynamic programming.** $\boldsymbol{DP_{jV}}$ is the shortest distance from the current state i to state j in visit status $\boldsymbol{V}$. Since $\boldsymbol{V}$ is a binary number, $\boldsymbol{DP_{i(V \wedge (1 \ll (j-1))}}$ is the distance of reaching state i without accessing other states ($\ll$ is the bitwise operator).

Under the above formalization, to solve the path planning problem is to optimize the following two equations:

$$\begin{cases} d_{ij} = min(d_{ij}, d_{ik} + d_{kj}) \\ DP_{jV} = min(DP_{jV}, DP_{i(V \wedge (1 \ll (j-1))} + d_{ij}) \end{cases}$$

4.2.2 *Planning Strategies.*

The dynamic programming algorithm is shown in Algorithm 1. Its general idea is to use multi-stage optimal decision-making, where each decision depends on the current visit status, and then cause the visit status to transfer. In detail, the algorithm first traverses the graph STG_{action} to obtain the set of nodes N and edges E (line 1-2). It then employs Floyd algorithm [66] to calculate the shortest path between each pair of nodes $d[i][j]$, and record the node sequence for each shortest path in *NodeSequence* (line 3-10).

Algorithm 1: DP-based Path planning algorithm

```
   Input: $STG_{action}$: $STG_{action}$ graph;
   Output: $path$: Recommended path to users;
1  $N \leftarrow getNode(STG_{action})$;
2  $E \leftarrow getEdge(STG_{action})$;
3  for each $k \in n$ do
4      for each $i \in n$ do
5          for each $j \in n$ do
6              if $d[i][j] > (d[i][k] + d[k][j])$ then
7                  $NodeSequence[i, j] = k$;
8                  //Record nodes sequence of shortest path.
9              $d[i][j] = min(d[i][j], d[i][k] + d[k][j])$;
10             //Calculate the shortest distance between nodes.
11 $DP[n][n] \leftarrow inf$;
12 $bit \leftarrow (1 \ll n)$;
13 $DP[0][1] \leftarrow 0$;
14 for each $V \in bit$ do
15     for each $i \in n$ do
16         if $V$ & $(1 \ll i)$ then
17             for each $j \in n$ do
18                 if not $(V \& (1 \ll j)$ and $d[i][j]$ != $inf$ then
19                     if $DP[j][V] > DP[i][V \wedge (1 \ll (j-1))] + d[i][j]$ then
20                         $VisitStatus[i, (V \wedge (1 \ll j))] = (j, V)$;
21                         //Update node visit status.
22                     $DP[j][V] = min(DP[j][V], DP[i][V \wedge (1 \ll (j-1))] + d[i][j])$;
23 $minn \leftarrow inf$;
24 for each $node \in n$ do
25     if $minn > DP[node][bit-1]$ then
26         $path \leftarrow getpath(NodeSequence, VisitStatus, node, bit-1)$;
27         //Get the planned path according to the node status.
28     $minn = min(minn, dp[node][bit-1])$;
29 return $path$;
```

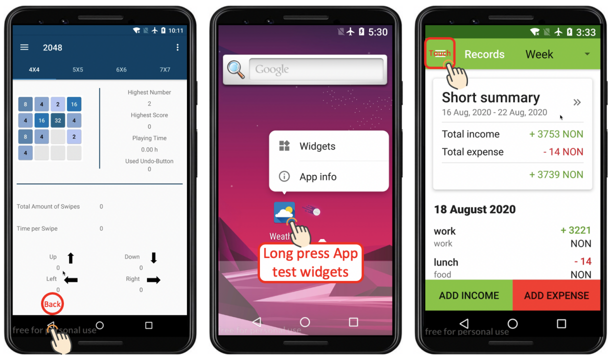
Based on the shortest path information, the algorithm then begins the path planning. Specifically, it maintains a buffer to store the visit status V, and gives priority to the nodes that have not been visited. Suppose the exploration is currently at node i, the algorithm will judge whether the visit status $DP[j][V]$ of node j is visited; if not, it finds a shortest path $d[i][j]$ between node i and node j, and update the node visit status *VisitStatus* (line 11-22). Finally, by calculating the shortest distance *minn* from the start node to the end node, the algorithm can get the end node *node* and its visit status $bit - 1$. According to the nodes sequence information of the shortest path in *NodeSequence* and the visit status information in *VisitStatus*, each intermediate node is backtracked from the tail node *node* to derive the visit order of nodes. After all nodes in the graph are visited, the algorithm can recommend the planned path for testers to explore (line 23-29).

Note that if the tester does not follow the path recommended by our approach in the process of exploration, we would record the state when he/she changes the path. Then according to the path that has been explored, NaviDroid will recalculate the path by running Algorithm 1 and take current state as the starting point.

4.3 Visual-based Path Guidance

We further implement the planned path into NaviDroid for guiding testers in testing mobile apps. It can suggest the next operation step by step in the user interface to help the testers covering the unexplored pages and reducing replication explorations. Specifically, we augment the run-time GUI with visual hint moves.

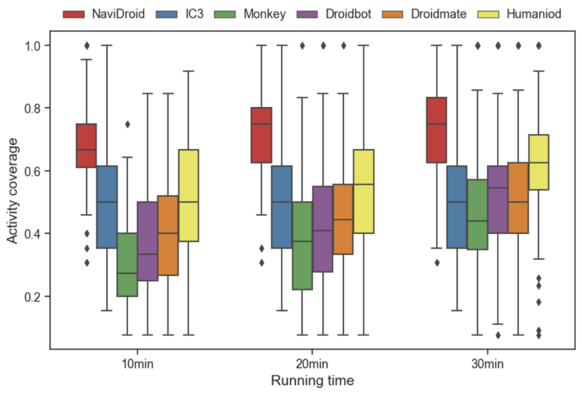
NaviDroid uses the Android debug bridge (adb) [6] command to start the app that testers need to test. To run an Android app, the source code is compiled and the mobile devices use rendering to realize the display of Android UI on the screen. During the tester's exploration, NaviDroid obtains the run-time information of the current interface including the state information and existing components within the current page on the backend. Specifically, NaviDroid can obtain the view hierarchy file corresponding to the current UI page (state) through "uiautomator dump" of the Android Debug Bridge (adb) command [9]. The view hierarchy file includes the component information (coordinate information, ID, component type, text description, etc.), and the layout information on the current state after rendering [10]. Given the state information of the current page, we search the obtained STG_{action} on the fly, find the next state on the planned path, and highlight the corresponding actions which can trigger that state in the page.



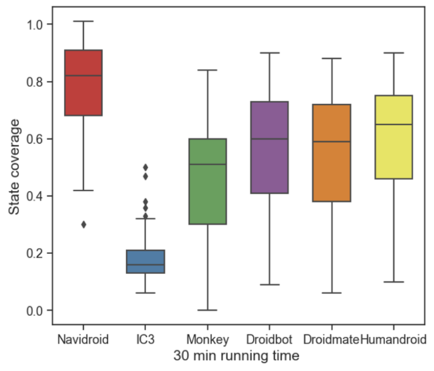
(a) Return back (b) Test widgets (c) Navigation bar

Figure 5: Highlighted suggested moves from NaviDroid.

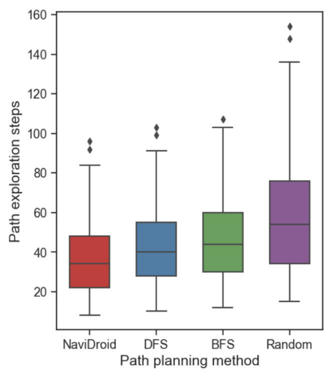
We adopt the Android floating window [25, 52] for visualizing the hint moves. It is a mobile window, which floats on the top of an app. As the Android interface drawing is realized through the services of *WindowManager*, which can add the floating window control to the screen through the *AddView*() method. After getting the trigger action from STG_{action}, our NaviDroid uses the adb command ("*adb shell uiautomator dump –compressed*") to get the



(a) Activity coverage of different methods



(b) State coverage in 30 minutes



(c) Path exploration steps

Figure 6: Result of effectiveness evaluation

coordinates and sizes of the component from the view hierarchy file [10] that testers need to operate. The system runs the floating window service in the backend, and sets the size and coordinates of the floating window (the rectangular box as shown in Figure 5) to make it in the same position and suitable size as the component and floating on the component, so as to guide the testers to explore the application interface.

There are three main types of trigger actions, i.e., returning back, long-pressing test widgets, and clicking a component(e.g., button, navigation bar) . The first action is easily visualized by a bounding box floating on the button as the hint. We present the last three cases in Fig 5 to facilitate understanding. If there is no back button in the current interface and the id is 'touch_back' as in Fig 5 (a), `NaviDroid` will suggest the tester with "back" action above the back key, otherwise, the tool will directly highlight the back button. As shown in Fig 5 (b), for the "appwidget activity", the tester will be suggested to operate "long press app test widgets" through the floating window when exploring the app. As shown in Fig 5 (c), for a button, navigation bar or fragment, the tester is suggested to click the UI component.

5 EFFECTIVENESS EVALUATION

`NaviDroid` consists of STG extraction algorithm, dynamic programming algorithm and visual guidance. Therefore, we first evaluate the algorithm performance of the `NaviDroid` through an automated method. We evaluate the effectiveness of `NaviDroid` from the points view of STG_{action} extraction component (Section 4.1) and DP-based path planning component (Section 4.2) respectively. For STG_{action} extraction, we compare the activity coverage and state coverage with 5 baseline methods to demonstrate its advantage (details are in Section 5.1.2). For path planning, we run `NaviDroid` and record the number of exploration steps, and then compare the path exploration efficiency of the `NaviDroid` with three baselines.

5.1 Experiment Setup

5.1.1 ***Dataset and Experiment Procedures.*** We use the 85 open-source mobile applications, as demonstrated in Section 3, for experiment, more detailed information of apps can be seen in our website[1].

For activity coverage, we collect all the activities defined in each app from AndroidManifest.xml following existing studies [19, 23, 53], and compare the percentage of the explored activities by running `NaviDroid` for 30 minutes. Note that there may be multiple states in one activity, so we also use state coverage [79] for evaluation. Since the app state cannot be obtained directly from Android files [79, 99], we invite two testers with more than five years of testing experience to manually label the states of 85 apps for the effectiveness evaluation. They conduct the manual labelling separately, and discuss the difference until a common consensus is reached. For path planning, we measure the number of exploration steps which is widely used [71, 78].

5.1.2 ***Baselines.*** For activity and state coverage, to further demonstrate the advantage of `NaviDroid`, we compare it with 5 common-used and state-of-the-art baselines. There are one static analysis based tool and four automated testing tools, i.e., IC3 [69], Monkey [32], Droidbot [53], DroidMate [18] and Humanoid [54]. The above automated test method is selected to evaluate the effect of `NaviDroid`'s algorithm on activity coverage. We run `NaviDroid` and these baselines on an Android virtual machine of Google Nexus 6 with Android 6.0 OS. We use the default configuration settings for each tool, record the activity and state coverage by running the test for 30 minutes. The experiment is repeated three times for robust consideration, and the average performance is used.

For path planning, based on the extracted STG_{action}, we choose 3 baselines of path exploration, i.e., Depth-First-Search [78, 81], Breadth-First-Search [16], Random Exploration [18, 32]. And then we record the number of exploration steps.

5.2 Results and Analysis

5.2.1 ***Activity and State Coverage.*** Fig 6(a) shows the activity coverage of `NaviDroid` and the baselines. `NaviDroid` achieves a median activity coverage of 0.74 with the range from 0.5 to 1.0 across 85 mobile apps. Fig 6(b) shows the state coverage of `NaviDroid` and the baselines in 30 minutes (Because the overall trend of state coverage and activity coverage is similar, we only give the state coverage in 30 minutes due to space constraints, and we upload the complete data to our website.). `NaviDroid` achieves a median state coverage of 0.81 with the range from 0.6 to 1.0 across 85 mobile apps. This indicates the effectiveness of NaviDroid in covering most of the activities and states so as to provide a full viewpoint for guiding testers in exploring the app.

NaviDroid is 12% (0.74 vs. 0.62) higher even compared with the best baseline (Humanoid) in activity coverage (with 30 minutes running time). This is because these baseline approaches rely on random user actions to explore the app, and the activity coverage could not be ensured.

Fig 6(a) also shows the coverage variation in terms of different time intervals. We can see that NaviDroid can quickly achieve a high coverage within 10 minutes. In contrast, automated testing tools need far more time to achieve their own optimal coverage. This is mainly because our NaviDroid can get a certain number of activities through static analysis, and combining them with the states obtained from dynamic analysis can achieve a superior activity coverage.

Although static analysis based baseline IC3 is not sensitive to the testing time, it misses two types of activities compared with our NaviDroid. In detail, it is unable to analyze the dynamic layout and mixed layout, and cannot handle the conversion of Android system event callback. For the automated GUI testing tools, the activity coverage increases along with the testing time. However, even given 30 minutes for testing one application, they can not reach as high activity coverage as that of our approach. That is because some apps require login or complicated input for triggering a certain activity which cannot be achieved by the random test case generation. It also indicates the necessity of manual testing and the importance of the development of our NaviDroid for assisting the manual testing.

We further examine the uncovered activities and states by our NaviDroid and summarize the following two reasons. First, some apps contain AppWidget activity, which is difficult to obtain the corresponding startup button. As running AppWidget activity requires a series of operations (i.e., exit the application to the home page of the device, hold down the application icon on the home page, then move the AppWidget to the device home page), obtaining the coordinates of the AppWidget is quite difficult. Second, there are some navigation bars or menus with numerical item ID (i.e., 1,2,3,...,N), rather than the meaningful descriptive one, and using the switch case method to call *intent*(). These numerical item IDs cannot be accurately obtained, and the trigger action cannot be extracted.

5.2.2 ***Path Exploration.*** Fig 6(c) shows the path exploration steps of our NaviDroid and the three baselines (DFS, BFS and random exploration). In 85 mobile applications, the average number of path exploration steps of NaviDroid is 36, outperforming the baselines. NaviDroid saves 20% (45 vs. 36), 23% (47 vs. 36) and 42% (60 vs. 35) steps compared with DFS, BFS and random exploration respectively (Each app is randomly explored once.). Since there are many leaf nodes and ring structures in STG_{action}, the three baseline approaches can occasionally fall into the ring structures, resulting in repeated operations. These baselines employ heuristic-driven strategies, and are far from achieving the globally optimal solution. By comparison, our proposed approach considers the whole graph during path planning, and can achieve more optimal exploration outcomes with fewer repeated steps.

6 USEFULNESS EVALUATION

To evaluate our NaviDroid, we also conduct a user study to demonstrate its usefulness in the real-world practice of manual testing. Our goal is to examine: (1) whether NaviDroid can effectively help explore the functionality of the application? (2) whether NaviDroid can help users find more bugs? (3) whether NaviDroid can save users' testing time?

6.1 Dataset of User Study

We begin with the 85 apps from F-Droid (also in Google Play) described in Section 3. To get realistic tasks for testing, we recruit an independent professional tester with four years of testing experience of Android app from Tencent. We ask him randomly choose apps, and find UI display bugs from each app. Note that, to ensure that testers can run the app smoothly, there are no functional bugs such as crashes in apps, and the UI display bugs will not cause any app crash. We end up with 20 apps with 30 UI display bugs (each app with at least one bug), and use them for the final evaluation, with details in Table 1.

6.2 Participants Recruitment

We recruit 32 testers to participate in the experiment (different from Section 3) from a crowdtesting platform TestIn. According to the background survey, all participants graduate with a computer science degree. They all have more than two years of app testing experience and practical work experience in the industry. Every participant receives a $50 shopping card as a reward after the experiment. At the beginning of the test, participants are asked to watch a short tutorial video and familiarize themselves with the apps. We also conduct a follow-up survey among the participants regarding their experiment experience.

The study involves two groups of 32 participants: the experimental group from P1 to P16 who test the mobile apps guided by our NaviDroid, and the control group from P17 to P32 who conduct the testing without any assistant. Each pair of participants ⟨ Px, P(x+16) ⟩ have comparable app testing experience to ensure that the experimental group has similar expertise and capability to the control group in total.

6.3 Experimental Design

To avoid potential inconsistency, we pre-install the 20 apps in the Google Nexus 6 on the emulator with Android 8.0 OS. The experiment begins with a brief introduction to the task. We show a demo to participants in the experimental group about how to use our NaviDroid with a new demo app (out of the 20 testing apps), and ask them to explore each app separately. NaviDroid will start to plan the test path for participants when they stay on the same page for 5 seconds without any operations.

The participants in the two groups need to test the 20 given mobile apps. They are required to fully explore all the interfaces of each app and find as many bugs as possible. Each participant has up to 10 minutes to test a mobile app which is far more than the typical app session (71.56 seconds) [17]. Each of them conducts the testing individually without any discussion with each other. During their testing, all their screen interactions are recorded, based on which we derive their testing performance.

Table 1: The comparison of the experiment and control group.

Basic information			State coverage		Activity coverage		Time (min)		# Bugs	
id	App	Categ	control	experiment	control	experiment	control	experiment	control	experiment
1	PHealth	Health	0.51	0.73	0.53	0.78	4.50	2.08	0.69	1.94
2	Weather	Internet	0.46	0.72	0.52	0.80	4.51	2.67	0.81	2.56
3	Contact	Phone	0.50	0.73	0.57	0.73	5.26	2.91	0.81	1.81
4	MoneyTK	Finance	0.39	0.68	0.45	0.82	7.79	4.31	0.94	2.00
5	FoodFacts	Health	0.53	0.78	0.55	0.83	8.28	6.71	1.44	2.63
6	GPSTest	Navig	0.44	0.71	0.53	0.90	5.92	4.18	0.50	1.00
7	PSStore	Security	0.44	0.68	0.49	0.76	8.18	6.68	0.38	0.94
8	NewPipe	Media	0.43	0.74	0.52	0.81	6.46	5.09	0.75	1.63
9	WallETH	Finance	0.48	0.80	0.49	0.84	8.52	7.76	0.38	0.94
10	Transistor	Media	0.38	0.76	0.40	0.82	5.33	3.65	0.31	0.94
11	Democracy	News	0.42	0.75	0.45	0.81	6.68	5.31	0.25	0.94
12	Metrodroid	Navig	0.31	0.60	0.39	0.72	7.37	5.98	0.31	0.88
13	INSTEAD	Game	0.41	0.74	0.54	0.80	7.62	4.79	0.44	0.94
14	ChaoChess	Game	0.47	0.73	0.51	0.79	7.90	6.49	0.31	0.94
15	RailwaySP	Navig	0.50	0.75	0.56	0.78	8.40	7.41	0.38	0.94
16	PocketMaps	Travel	0.49	0.75	0.45	0.70	8.09	5.95	0.56	1.88
17	Barinsta	Internet	0.50	0.74	0.52	0.77	5.50	3.65	0.44	1.00
18	FitNotif	Connect	0.49	0.77	0.45	0.81	7.38	6.86	0.50	1.00
19	SkyTube	Media	0.48	0.77	0.50	0.75	8.13	5.57	0.25	1.00
20	LibReader	Reading	0.45	0.79	0.44	0.77	7.78	6.71	0.44	0.81
Average			0.45	**0.73**	0.49	**0.79**	6.98	**5.24**	0.54	**1.33**

6.4 Evaluation Metrics

Following previous studies [23, 25], the activity and state coverage [19, 53, 79] are common evaluation metrics in the Android GUI testing. So we use the following metrics to evaluate the effectiveness of `NaviDroid`.

- State coverage: (number of discovered states) / (number of all possible states)
- Activity coverage: (number of discovered activities) / (number of all activities)
- Testing Time: average time spent per app (Each tester explore the entire app to the best of their ability)
- Bug number: number of discovered bugs

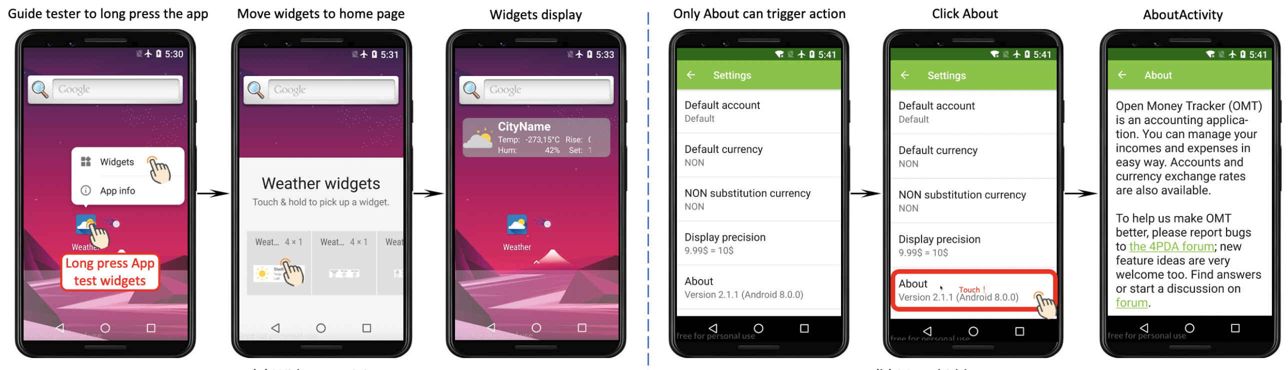
6.5 Results and Analysis

We present the `NaviDroid`'s average state and activity coverage, the average testing time, and the average detected bugs across the two groups, as shown in Table 1.

6.5.1 ***Higher State and Activity Coverage.*** As shown in Table 1, the state coverage of the experimental group is 0.73, which is about 62% ((0.73-0.45)/0.45) higher than that of the control group. And the activity coverage of the experimental group is 0.79, which is about 61% ((0.79-0.49)/0.49) higher than that of the control group. The results of Mann-Whitney U Test shows there is a significant difference (p-value <0.01, more detailed information of experiment can be seen in our website[1].) between these two groups in both metrics. It indicates that our `NaviDroid` can guide the testers in exploring more states and activities possibly through reducing the duplicate explorations as well as the hesitation time in choosing which to explore next. We also find that `NaviDroid` can help testers spot some activities which are hard to find without the help of the tool. We analyze those activities and summarize them into two categories.

First, there are certain activities that require specific actions to trigger. For example, only *long-pressing the app icon* can trigger the *AppWidgetActivity*, which contains a pop-up showing the setting of the weather bar in Fig 7 (a). Second, some clickable components are inconspicuous due to the poor app GUI design. Fig 7 (b) shows another example, in which "About" is a button that can be clicked to show detailed information about the app version. However, all other *TextView* in the list cannot be clicked. That is why all testers without using our tool missed it during manual testing. These examples indicate that without explicit guidance as our `NaviDroid`, human testers are very likely to miss those activities, resulting in the incompleteness of the app testing.

6.5.2 ***More Detected Bugs.*** With our `NaviDroid`, testers can find an average of 1.33 bugs for each app, resulting in 30 unique bugs for the 20 experimental apps. Due to the low activity coverage in the control group, some bugs are not discovered in this group, i.e., only an average of 0.54 bugs are reported for each app. For example, as showed in Fig 7 (a), testers did not notice the *AppWidgetActivity*, thus could not explore the pop-up window which displays the setting of the weather bar. Therefore no one in this group finds the bug in the pop-up window. The results of Mann-Whitney U Test shows there are significant difference (p-value <0.01) between these two groups for the detected bugs. Furthermore, these bugs are user interface bugs, which are difficult to be detected with current


Guide tester to long press the app
Move widgets to home page
Widgets display
Only About can trigger action
Click About
AboutActivity
Widgets
App info
Long press App test widgets
Weather widgets
Touch & hold to pick up a widget.
CityName
Weather
Settings
Default account
Default
Default currency
NON
NON substitution currency
NON
Display precision
9.99$ = 10$
About
Version 2.1.1 (Android 8.0.0)
Touch !
About

Open Money Tracker (OMT) is an accounting application. You can manage your incomes and expenses in easy way. Accounts and currency exchange rates are also available.

To help us make OMT better, please report bugs to the 4PDA forum; new feature ideas are very welcome too. Find answers or start a discussion on forum.

(a) Widgets activity (b) More hidden components

Figure 7: Two difficult situations for manual testing

automatic testing techniques, further indicating the practical value of our proposed NaviDroid.

In addition, for the control group, testers' attention and their reporting rate of bugs decreased after about 3 minutes of exploration during the process of the experiment, i.e., the duplication of testers' operation increased and the number of found bugs decreased.

6.5.3 ***Less Time Cost.*** It takes just 5.24 minutes for testers with our NaviDroid to finish exploring an app by covering all pages while 6.98 minutes in the control group. The results of Mann-Whitney U Test shows there are significant difference (p-value <0.01) between these two groups for the testing time. In fact, the average time of the control group is underestimated, because 14 participants fail to complete any apps even with 10 minutes, which means that they may need more time in exploring the apps.

We watch the video recording of the app testing in the control group for further discovering the reasons for the higher time cost. Without the NaviDroid, we find that testers repeated 0.69 (standard deviation $\sigma = 20\%$) of their own actions. In contrast, none of the testers using the NaviDroid generated duplicate actions. This shows that the guide can effectively prevent testers from interacting with previously explored pages and components, and then guide them to explore deeper pages. It seems that the testers are stuck in some activities or activity loops without knowing how to trigger new activities. After the initial exploration of the app, some testers forget where they have explored, resulting in repeated testing of some activities. This observation further confirms the importance of testing guidance tools as our NaviDroid.

7 DISCUSSION

In summary, we find that the combination of our NaviDroid can effectively guide testers to significantly increase their state and activity coverage. Consistent with Information Foraging Theory [50, 73], it suggests that providing visual navigation cues could help guide tester's attention and thus improve information access.

7.1 Testers' Experience With NaviDroid

According to the testers' feedback in Section 6, all of them confirm the usefulness of our NaviDroid in assisting their manual Android app testing. They all appreciate that the hint moves of our NaviDroid can help guide them in exploring the inconspicuous activities of the application, increasing the hit rate of potential bugs. For example, "NaviDroid is very helpful for us to discover new pages and functions. It effectively avoids repeated operations."(P1, P3, P8). Participants express they like our interaction design such as "Great, NaviDroid uses the float window for guidance is a good idea. I like it!"(P2, P7, P13, P15, P6); "The guidance is much like the tutorial when the game software is first used. It can help us to understand a new application."(P4). Participants express that our NaviDroid can save their testing time such as"Nice! The NaviDroid saves our testing time."(P11, P16, P10).

The participants also mention the drawback and potential improvement of our NaviDroid. First, besides the hint moves highlighted along with their testing, they also hope to see the overall STG_{action} before the application testing and also their real-time position in STG_{action} during the Android app testing (P5, P9). Second, more fine-grained transition states are needed as current ones can still not cover all functionality combinations or corner cases (P2,P3).

7.2 Generalization of NaviDroid and Findings

NaviDroid is designed to assist the manual testing of Android apps with the extracted STG on Android apps, and have achieved satisfactory performance. In addition to Android, there are also many other platforms such as iOS, web. To conquer the market, developers tend to develop either one cross-platform app or separated native apps for each platform considering the performance benefit of native apps. Although our NaviDroid is designed specifically for Android, it can also be extended to other platforms. Given an app from other platforms, we just need an exploration tool for getting the STG. With that STG, NaviDroid's path planning algorithm based on dynamic programming can be reused to facilitate the optimized exploration steps by avoiding repeated testing of UI page.

Therefore, we would expect that the idea of NaviDroid can be applied to apps in other platforms. Of course, the app usage pattern and the types of edits to predict are very likely to differ from one platform to another. For mobile platform like iOS, our empirical study and method may be easily adapted to it with some engineering effort. For platforms using different devices like desktop, the differences between these platforms with Android can be considerably big. In such cases, a detailed empirical study of the app usage pattern is required to determine the extension. In the future, we will try to extract STG of multi-platform apps to guide downstream testing tasks.

The NaviDroid will also obtain a large number of UI screenshots in the process of assisting the crowdworkers to test the app, including issues such as UI display issue and compatibility issues. We will further study these screenshots in the future and realize automatic issue detection and repair.

7.3 Potential Applications to End-users

In addition to assisting human testers in GUI testing during the software development, our NaviDroid can also be applied to help end-users in their daily app usage. Given the increasing complexity of mobile apps, fluent usage of mobile apps is a challenging task especially for the aged and disabled users. For example, there may be too many clickable components in one GUI page for senior users to locate the functionality which they want. They may stuck on one page with repetitive attempts but not working. Even for normal users, they may not notice some features especially news ones in the app.

Based on our approach, according to the users' interaction log, our NaviDroid can detect the situation of stuck mentioned above. With the future algorithm improvement by machine learning, our approach can smartly remind users the next moves to jump out of the dilemma and explore new features. In addition, our tool is running on users' device without any round-trip to a server, leading to the privacy preserving as privacy-sensitive data stays on the device.

7.4 Limitations

7.4.1 Incomplete STG_{action}. Although our hybrid approach can construct an STG_{action} with 74% activity coverage, it still misses some states. As analyzed in Section 5.2.1, different developers would have varied code writing styles for implementing states and transitions among them, and some of them are with poor coding conventions. All these could influence the complete extraction of STG_{action}. We will keep improving our approach for covering more corner cases in enriching STG_{action}.

7.4.2 Partial interaction and UI types. In term of actions triggering the state transition in NaviDroid, we only consider the clicking events, and could not handle other actions such as scrolling, text filling, etc. Besides, it is also hard to visualize a complicated combination of actions on the same page to the next state. For UI pages with more animations (highly visual and dynamic app UI interfaces), NaviDroid is difficult to operate on animations. The current method used by NaviDroid is to skip these animation components. Although those action combinations or animations only account for a small portion of state transition and state, we will also take them into consideration in the future.

8 CONCLUSION

As the last line of defence, manual testing is crucial to improve application quality. Despite its importance, manual testing is time-consuming, labor-extensive and highly dependent on testers' experience and capability. Therefore, we propose a method called NaviDroid to guide human testers in exploring more states during app testing. We first construct STG_{action} by both static analysis and dynamic exploration, and generate the planned path based on a dynamic programming algorithm. On the app screen, we highlight the hint moves triggering the next unexplored state to users by a visualized floating window of that page. The automated evaluation and user study demonstrate the accuracy and usefulness of NaviDroid in improving testing efficiency, reducing testing time and saving testers' efforts.

In the future, we will work in two directions. First, we will improve our approach in constructing enhanced STG_{action} by detecting more states and fine-grained transitions among them. According to the user feedback, we will also improve the interaction between our NaviDroid and users, which can borrow the idea from the human-machine collaboration studies to better facilitate the human testers. Second, we will not limit our NaviDroid to app testing, and plan to explore its potential usage into other areas like reminding users about the new functionalities of apps.

ACKNOWLEDGMENTS

This work is supported by the National Key Research and Development Program of China under grant No.2018YFB1403400, National Natural Science Foundation of China under Grant No. 62072442, No. 62002348, and Youth Innovation Promotion Association Chinese Academy of Sciences.

REFERENCES

[1] 2016. Mobile Internet use passes desktop for the first time. https://techcrunch.com/2016/11/01/mobile-internet-use-passes-desktop-for-the-first-time-study-finds/.
[2] 2018. Amazon Mechanical Turk. https://www.mturk.com/.
[3] 2019. UTest. https://www.utest.com/.
[4] 2020. http://tools.android.com/tips/lint.
[5] 2020. https://github.com/stylelint/stylelint.
[6] 2020. Android Debug Bridge (adb). https://developer.android.com/studio/command-line/adb.html#forwardports.
[7] 2020. Candy Crush Saga. https://play.google.com/store/apps/details?id=com.king.candycrushsaga.
[8] 2020. Number of available applications in the Google Play Store from December 2009 to June 2020. https://www.statista.com/statistics/266210/number-of-available-applications-in-the-google-play-store.
[9] 2021. Android development. http://developer.android.com/reference/android.
[10] 2021. View hierachy. https://developer.android.google.cn/topic/performance/rendering/optimizing-view-hierarchies/.
[11] David Adamo, Md Khorrom Khan, Sreedevi Koppula, and Renée Bryce. 2018. Reinforcement learning for android gui testing. In *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation.* 2–8.
[12] Abdulaziz Alshayban, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility issues in Android apps: state of affairs, sentiments, and ways forward. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE).* IEEE, 1323–1334.
[13] Josh M Attenberg, Pagagiotis G Ipeirotis, and Foster Provost. 2011. Beat the machine: Challenging workers to find the unknown unknowns. In *Workshops at the Twenty-Fifth AAAI Conference on Artificial Intelligence.*
[14] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *Proceedings of the 2013 ACM SIGPLAN international conference on Object oriented programming systems languages & applications.* 641–660.
[15] Young-Min Baek and Doo-Hwan Bae. 2016. Automated model-based android gui testing using multi-level gui comparison criteria. In *Proceedings of the 31st IEEE/ACM International Conference on Automated Software Engineering.* ACM, 238–249.
[16] Scott Beamer, Krste Asanovic, and David Patterson. 2012. Direction-optimizing breadth-first search. In *SC'12: Proceedings of the International Conference on High Performance Computing, Networking, Storage and Analysis.* IEEE, 1–10.
[17] Matthias Böhmer, Brent Hecht, Johannes Schöning, Antonio Krüger, and Gernot Bauer. 2011. Falling asleep with Angry Birds, Facebook and Kindle: a large scale study on mobile application usage. In *Proceedings of the 13th international*

conference on Human computer interaction with mobile devices and services. 47–56.

[18] Nataniel P Borges, Jenny Hotzkow, and Andreas Zeller. 2018. DroidMate-2: a platform for Android test generation. In *2018 33rd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 916–919.

[19] Tianqin Cai, Zhao Zhang, and Ping Yang. 2020. Fastbot: A Multi-Agent Model-Based Test Generation System Beijing Bytedance Network Technology Co., Ltd.. In *Proceedings of the IEEE/ACM 1st International Conference on Automation of Software Test*. 93–96.

[20] Chunyang Chen, Ting Su, Guozhu Meng, Zhenchang Xing, and Yang Liu. 2018. From ui design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In *Proceedings of the 40th International Conference on Software Engineering*. ACM, 665–676.

[21] Qiuyuan Chen, Chunyang Chen, Safwat Hassan, Zhengchang Xing, Xin Xia, and Ahmed E Hassan. 2021. How Should I Improve the UI of My App? A Study of User Reviews of Popular Apps in the Google Play. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 30, 3 (2021), 1–38.

[22] Sen Chen, Chunyang Chen, Lingling Fan, Mingming Fan, Xian Zhan, and Yang Liu. 2021. Accessible or Not An Empirical Investigation of Android App Accessibility. *IEEE Transactions on Software Engineering* (2021).

[23] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenhe Li, Yang Liu, and Lihua Xu. 2019. Storydroid: Automated generation of storyboard for Android apps. In *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 596–607.

[24] Yan Chen, Sang Won Lee, and Steve Oney. 2021. CoCapture: Effectively Communicating UI Behaviors on Existing Websites by Demonstrating and Remixing. In *Proceedings of the 2021 CHI Conference on Human Factors in Computing Systems*. 1–14.

[25] Yan Chen, Maulishree Pandey, Jean Y Song, Walter S Lasecki, and Steve Oney. 2020. Improving Crowd-Supported GUI Testing with Structural Guidance. In *Proceedings of the 2020 CHI Conference on Human Factors in Computing Systems*. 1–13.

[26] Wontae Choi, George Necula, and Koushik Sen. 2013. Guided gui testing of android apps with minimal restart and approximate learning. *Acm Sigplan Notices* 48, 10 (2013), 623–640.

[27] Ilinca Ciupa, Bertrand Meyer, Manuel Oriol, and Alexander Pretschner. 2008. Finding faults: Manual testing vs. random+ testing vs. user reports. In *2008 19th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 157–166.

[28] Qiang Cui, Song Wang, Junjie Wang, Yuanzhe Hu, Qing Wang, and Mingshu Li. 2017. Multi-Objective Crowd Worker Selection in Crowdsourced Testing.. In *SEKE*, Vol. 17. 1–6.

[29] Nikita Danilov, Tatiana Shulga, Natalya Frolova, Nina Melnikova, Nataliia Vagarina, and Elena Pchelintseva. 2016. Software usability evaluation based on the user pinpoint activity heat map. In *Computer Science On-line Conference*. Springer, 217–225.

[30] Biplab Deka, Zifeng Huang, Chad Franzen, Joshua Hibschman, Daniel Afergan, Yang Li, Jeffrey Nichols, and Ranjitha Kumar. 2017. Rico: A mobile app dataset for building data-driven design applications. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 845–854.

[31] Biplab Deka, Zifeng Huang, Chad Franzen, Jeffrey Nichols, Yang Li, and Ranjitha Kumar. 2017. Zipt: Zero-integration performance testing of mobile app designs. In *Proceedings of the 30th Annual ACM Symposium on User Interface Software and Technology*. 727–736.

[32] Android Developers. 2012. Ui/application exerciser monkey.

[33] Sebastian Eder, Benedikt Hauptmann, Maximilian Junker, Rudolf Vaas, and Karl-Heinz Prommer. 2014. Selecting manual regression test cases automatically using trace link recovery and change coverage. In *9th International Workshop on Automation of Software Test, AST 2014, Hyderabad, India, May 31 - June 1, 2014*. ACM, 29–35. https://doi.org/10.1145/2593501.2593506

[34] Bernard Elodie, Julien Botella, Fabrice Ambert, Bruno Legeard, and Mark Utting. 2020. Tool Support for Refactoring Manual Tests. In *13th IEEE International Conference on Software Testing, Validation and Verification, ICST 2020, Porto, Portugal, October 24-28, 2020*. IEEE, 332–342. https://doi.org/10.1109/ICST46399.2020.00041

[35] 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.

[36] Mattia Fazzini and Alessandro Orso. 2017. Automated cross-platform inconsistency detection for mobile apps. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 308–318.

[37] Sidong Feng, Suyu Ma, Jinzhong Yu, Chunyang Chen, TingTing Zhou, and Yankun Zhen. 2021. Auto-icon: An automated code generation tool for icon designs assisting in ui development. In *26th International Conference on Intelligent User Interfaces*. 59–69.

[38] Dennis Fetterly, Mark Manasse, and Marc Najork. 2003. On the evolution of clusters of near-duplicate web pages. In *Proceedings of the IEEE/LEOS 3rd International Conference on Numerical Simulation of Semiconductor Optoelectronic Devices (IEEE Cat. No. 03EX726)*. IEEE, 37–45.

[39] Adam Fourney. 2015. Web search, web tutorials & software applications: characterizing and supporting the coordinated use of online resources for performing work in feature-rich software. (2015).

[40] Christian Frisson, Sylvain Malacria, Gilles Bailly, and Thierry Dutoit. 2016. Inspectorwidget: A system to analyze users behaviors in their applications. In *Proceedings of the 2016 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. 1548–1554.

[41] Sarah E. Garcia. 2020. Usability Testing: Creative Techniques for Answering Your Research Questions. In *Extended Abstracts of the 2020 CHI Conference on Human Factors in Computing Systems, CHI 2020, Honolulu, HI, USA, April 25-30, 2020*. ACM, 1–2. https://doi.org/10.1145/3334480.3375064

[42] Roman Haas, Daniel Elsner, Elmar Jürgens, Alexander Pretschner, and Sven Apel. 2021. How can manual testing processes be optimized? developer survey, optimization guidelines, and case studies. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 1281–1291. https://doi.org/10.1145/3468264.3473922

[43] Hadi Hemmati, Zhihan Fang, and Mika V. Mäntylä. 2015. Prioritizing Manual Test Cases in Traditional and Rapid Release Environments. In *8th IEEE International Conference on Software Testing, Verification and Validation, ICST 2015, Graz, Austria, April 13-17, 2015*. IEEE Computer Society, 1–10. https://doi.org/10.1109/ICST.2015.7102602

[44] Hadi Hemmati and Fatemeh Sharifi. 2018. Investigating NLP-Based Approaches for Predicting Manual Test Case Failure. In *11th IEEE International Conference on Software Testing, Verification and Validation, ICST 2018, Västerås, Sweden, April 9-13, 2018*. IEEE Computer Society, 309–319. https://doi.org/10.1109/ICST.2018.00038

[45] Juha Itkonen and Mika V Mäntylä. 2014. Are test cases needed? Replicated comparison between exploratory and test-case-based software testing. *Empirical Software Engineering* 19, 2 (2014), 303–342.

[46] Juha Itkonen, Mika V Mantyla, and Casper Lassenius. 2007. Defect detection efficiency: Test case based vs. exploratory testing. In *First International Symposium on Empirical Software Engineering and Measurement (ESEM 2007)*. IEEE, 61–70.

[47] Juha Itkonen, Mika V Mantyla, and Casper Lassenius. 2009. How do testers do it? An exploratory study on manual testing practices. In *2009 3rd International Symposium on Empirical Software Engineering and Measurement*. IEEE, 494–497.

[48] Juha Itkonen and Kristian Rautiainen. 2005. Exploratory testing: a multiple case study. In *2005 International Symposium on Empirical Software Engineering, 2005*. IEEE, 10–pp.

[49] Remo Lachmann, Sandro Schulze, Manuel Nieke, Christoph Seidl, and Ina Schaefer. 2016. System-Level Test Case Prioritization Using Machine Learning. In *15th IEEE International Conference on Machine Learning and Applications, ICMLA 2016, Anaheim, CA, USA, December 18-20, 2016*. IEEE Computer Society, 361–368. https://doi.org/10.1109/ICMLA.2016.0065

[50] Joseph Lawrance, Christopher Bogart, Margaret Burnett, Rachel Bellamy, Kyle Rector, and Scott D Fleming. 2010. How programmers debug, revisited: An information foraging theory perspective. *IEEE Transactions on Software Engineering* 39, 2 (2010), 197–215.

[51] Andreas Leitner, Ilinca Ciupa, Bertrand Meyer, and Mark Howard. 2007. Reconciling manual and automated testing: The autotest experience. In *2007 40th Annual Hawaii International Conference on System Sciences (HICSS'07)*. IEEE, 261a–261a.

[52] Baichao Li and Chenglong Ao. 2016. Visualizing content referenced in an electronic document. US Patent 9,495,334.

[53] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In *2017 IEEE/ACM 39th International Conference on Software Engineering Companion (ICSE-C)*. IEEE, 23–26.

[54] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2019. Humanoid: a deep learning-based approach to automated black-box Android app testing. In *2019 34th IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 1070–1073.

[55] Mario Linares-Vásquez, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2017. How do developers test android applications?. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 613–622.

[56] Mario Linares-Vásquez, Kevin Moran, and Denys Poshyvanyk. 2017. Continuous, evolutionary and large-scale: A new perspective for automated mobile app testing. In *2017 IEEE International Conference on Software Maintenance and Evolution (ICSME)*. IEEE, 399–410.

[57] Mario Linares-Vásquez, Martin White, Carlos Bernal-Cárdenas, Kevin Moran, and Denys Poshyvanyk. 2015. Mining android app usages for generating actionable gui-based execution scenarios. In *2015 IEEE/ACM 12th Working Conference on Mining Software Repositories*. IEEE, 111–122.

[58] Zhe Liu, Chunyang Chen, Junjie Wang, Yuekai Huang, Jun Hu, and Qing Wang. 2020. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In *2020*

35rd IEEE/ACM International Conference on Automated Software Engineering (ASE). IEEE.

[59] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*. 224–234.

[60] Gurmeet Singh Manku, Arvind Jain, and Anish Das Sarma. 2007. Detecting near-duplicates for web crawling. In *Proceedings of the 16th international conference on World Wide Web*. 141–150.

[61] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: Multi-objective automated testing for Android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*. 94–105.

[62] Ke Mao, Mark Harman, and Yue Jia. 2017. Crowd intelligence enhances automated mobile testing. In *2017 32nd IEEE/ACM International Conference on Automated Software Engineering (ASE)*. IEEE, 16–26.

[63] Justin Matejka, Tovi Grossman, and George Fitzmaurice. 2013. Patina: Dynamic heatmaps for visualizing application usage. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*. 3227–3236.

[64] Domenico Mazza. 2017. Reducing cognitive load and supporting memory in visual design for HCI. In *Proceedings of the 2017 CHI Conference Extended Abstracts on Human Factors in Computing Systems*. 142–147.

[65] Mark Micallef, Chris Porter, and Andrea Borg. 2016. Do exploratory testers need formal training? an investigation using hci techniques. In *2016 IEEE Ninth International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*. IEEE, 305–314.

[66] Antonio E Mirino et al. 2017. Best routes selection using Dijkstra and Floyd-Warshall algorithm. In *2017 11th International Conference on Information & Communication Technology and System (ICTS)*. IEEE, 155–158.

[67] Nariman Mirzaei, Joshua Garcia, Hamid Bagheri, Alireza Sadeghi, and Sam Malek. 2016. Reducing combinatorics in GUI testing of android applications. In *2016 IEEE/ACM 38th International Conference on Software Engineering (ICSE)*. IEEE, 559–570.

[68] Takao Nakagawa, Kazuki Munakata, and Koji Yamamoto. 2019. Applying Modified Code Entity-Based Regression Test Selection for Manual End-to-End Testing of Commercial Web Applications. In *2019 IEEE International Symposium on Software Reliability Engineering Workshops (ISSREW)*. 1–6. https://doi.org/10.1109/ISSREW.2019.00033

[69] Damien Octeau, Daniel Luchaup, Matthew Dering, Somesh Jha, and Patrick McDaniel. 2015. Composite constant propagation: Application to android inter-component communication analysis. In *2015 IEEE/ACM 37th IEEE International Conference on Software Engineering*, Vol. 1. IEEE, 77–88.

[70] 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*. 153–164.

[71] Noé Pérez-Higueras, Fernando Caballero, and Luis Merino. 2018. Learning human-aware path planning with fully convolutional networks. In *2018 IEEE International Conference on Robotics and Automation (ICRA)*. IEEE, 5897–5902.

[72] Helen Petrie and Omar Kheir. 2007. The relationship between accessibility and usability of websites. In *Proceedings of the 2007 Conference on Human Factors in Computing Systems, CHI 2007, San Jose, California, USA, April 28 - May 3, 2007*, Mary Beth Rosson and David J. Gilmore (Eds.). ACM, 397–406. https://doi.org/10.1145/1240624.1240688

[73] Peter Pirolli and Stuart Card. 1999. Information foraging. *Psychological review* 106, 4 (1999), 643.

[74] Antoine Ponsard and Joanna McGrenere. 2016. Anchored customization: anchoring settings to the application interface to afford customization. In *Proceedings of the 2016 CHI Conference on Human Factors in Computing Systems*. 4154–4165.

[75] Christopher Power, André Pimenta Freire, Helen Petrie, and David Swallow. 2012. Guidelines are only half of the story: accessibility problems encountered by blind users on the web. In *CHI Conference on Human Factors in Computing Systems, CHI '12, Austin, TX, USA - May 05 - 10, 2012*. ACM, 433–442. https://doi.org/10.1145/2207676.2207736

[76] Dudekula Mohammad Rafi, Katam Reddy Kiran Moses, Kai Petersen, and Mika V Mäntylä. 2012. Benefits and limitations of automated software testing: Systematic literature review and practitioner survey. In *2012 7th International Workshop on Automation of Software Test (AST)*. IEEE, 36–42.

[77] Navid Salehnamadi, Abdulaziz Alshayban, Jun-Wei Lin, Iftekhar Ahmed, Stacy M. Branham, and Sam Malek. [n.d.].

[78] Safaa H Shwail, Alia Karim, and Scott Turner. 2013. Probabilistic multi robot path planning in dynamic environments: A comparison between A* and DFS. *International Journal of Computer Applications* 975 (2013), 8887.

[79] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*. 245–256.

[80] Yuhui Su, Zhe Liu, Chunyang Chen, Junjie Wang, and Qing Wang. 2021. OwlEyes-online: a fully automated platform for detecting and localizing UI display issues. In *ESEC/FSE '21: 29th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Athens, Greece, August 23-28, 2021*. ACM, 1500–1504. https://doi.org/10.1145/3468264.3473109

[81] Robert Tarjan. 1972. Depth-first search and linear graph algorithms. *SIAM journal on computing* 1, 2 (1972), 146–160.

[82] Sander Van Der Burg and Eelco Dolstra. 2010. Automating system tests using declarative virtual machines. In *2010 IEEE 21st International Symposium on Software Reliability Engineering*. IEEE, 181–190.

[83] Gang Wang, Xinyi Zhang, Shiliang Tang, Christo Wilson, Haitao Zheng, and Ben Y Zhao. 2017. Clickstream user behavior models. *ACM Transactions on the Web (TWEB)* 11, 4 (2017), 1–37.

[84] Junjie Wang, Mingyang Li, Song Wang, Tim Menzies, and Qing Wang. 2019. Images don't lie: Duplicate crowdtesting reports detection with screenshot information. *Information and Software Technology* 110 (2019), 139–155.

[85] Junjie Wang, Ye Yang, Rahul Krishna, Tim Menzies, and Qing Wang. 2019. iSENSE: Completion-Aware Crowdtesting Management. In *ICSE'2019*. 932–943.

[86] Junjie Wang, Ye Yang, Song Wang, Chunyang Chen, Dandan Wang, and Qing Wang. 2021. Context-aware Personalized Crowdtesting Task Recommendation. *IEEE Transactions on Software Engineering* (2021).

[87] Junjie Wang, Ye Yang, Song Wang, Yuanzhe Hu, Dandan Wang, and Qing Wang. 2020. Context-aware In-process Crowdworker Recommendation *(ICSE 2020)*.

[88] Yan Wang, Hailong Zhang, and Atanas Rountev. 2016. On the unsoundness of static analysis for Android GUIs. In *Proceedings of the 5th ACM SIGPLAN International Workshop on State Of the Art in Program Analysis*. 18–23.

[89] James A Whittaker. 2009. *Exploratory software testing: tips, tricks, tours, and techniques to guide test design*. Pearson Education.

[90] Kristian Wiklund, Sigrid Eldh, Daniel Sundmark, and Kristina Lundqvist. 2017. Impediments for software test automation: A systematic literature review. *Software Testing, Verification and Reliability* 27, 8 (2017), e1639.

[91] Tianyong Wu, Xi Deng, Jun Yan, and Jian Zhang. 2019. Analyses for specific defects in android applications: A survey. *Frontiers of Computer Science* (2019), 1–18.

[92] Miao Xie, Qing Wang, Guowei Yang, and Mingshu Li. 2017. Cocoon: Crowdsourced testing quality maximization under context coverage constraint. In *2017 IEEE 28th International Symposium on Software Reliability Engineering (ISSRE)*. IEEE, 316–327.

[93] Qing Xie and Atif M Memon. 2007. Designing and comparing automated test oracles for GUI-based software applications. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 16, 1 (2007), 4–es.

[94] Baron Yan and Yitmen Koray. 2018. ISTQB® Worldwide Software Testing Practices 2017-2018.

[95] Jiwei Yan, Hao Liu, Linjie Pan, Jun Yan, Jian Zhang, and Bin Liang. 2020. Multiple-entry testing of android applications by constructing activity launching contexts. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 457–468.

[96] Rahulkrishna Yandrapally, Andrea Stocco, and Ali Mesbah. 2020. Near-duplicate detection in web app model inference. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*. 186–197.

[97] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. Don't Do That! Hunting Down Visual Design Smells in Complex UIs against Design Guidelines. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 761–772.

[98] Bo Yang, Zhenchang Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shanping Li. 2021. UIS-Hunter: Detecting UI Design Smells in Android Apps. In *2021 IEEE/ACM 43rd International Conference on Software Engineering: Companion Proceedings (ICSE-Companion)*. IEEE, 89–92.

[99] Shengqian Yang, Haowei Wu, Hailong Zhang, Yan Wang, Chandrasekar Swaminathan, Dacong Yan, and Atanas Rountev. 2018. Static window transition graphs for Android. *Automated Software Engineering* 25, 4 (2018), 833–873.

[100] Wei Yang, Mukul R Prasad, and Tao Xie. 2013. A grey-box approach for automated GUI-model generation of mobile applications. In *International Conference on Fundamental Approaches to Software Engineering*. Springer, 250–265.

[101] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for android: Are we really there yet in an industrial case?. In *Proceedings of the 2016 24th ACM SIGSOFT International Symposium on Foundations of Software Engineering*. 987–992.

[102] Yifei Zhang, Yulei Sui, and Jingling Xue. 2018. Launch-mode-aware context-sensitive activity transition analysis. In *Proceedings of the 40th International Conference on Software Engineering*. 598–608.

[103] Dehai Zhao, Zhenchang Xing, Chunyang Chen, Xiwei Xu, Liming Zhu, Guoqiang Li, and Jinshui Wang. 2020. Seenomaly: vision-based linting of GUI animation effects against design-don't guidelines. In *2020 IEEE/ACM 42nd International Conference on Software Engineering (ICSE)*. IEEE, 1286–1297.

[104] Tianming Zhao, Chunyang Chen, Yuanning Liu, and Xiaodong Zhu. 2021. GUIGAN: Learning to Generate GUI Designs Using Generative Adversarial Networks. In *2021 IEEE/ACM 43rd International Conference on Software Engineering (ICSE)*. IEEE, 748–760.