NaviDroid: A Tool for Guiding Manual Android Testing via Hint Moves

Zhe Liu1,3, Chunyang Chen2, Junjie Wang1,2,3,∗, Yuhui Su1,3, Qing Wang1,2,3,∗
1Laboratory for Internet Software Technologies, 2State Key Laboratory of Computer Sciences,
Institute of Software Chinese Academy of Sciences, Beijing, China;
3University of Chinese Academy of Sciences, Beijing, China; ∗Corresponding author
4Monash University, Melbourne, Australia;
liuzhe181@mails.ucas.edu.cn, Chunyang.chen@monash.edu, junjie@iscas.ac.cn, wq@iscas.ac.cn

ABSTRACT
Manual testing, as a complement to automated GUI testing, is the last line of defense for app quality especially in spotting usability and accessibility issues. However, the repeated actions and easy missing of some functionalities make manual testing time-consuming, labor-intensive and inefficient. Inspired by the game candy crush with flashy candies as hint moves for players, we develop a tool named NaviDroid for navigating human testers via highlighted next operations for more effective and efficient testing. Within NaviDroid, it constructs an enriched state transition graph (STG) with the trigger actions as the edges for two involved states. Based on the STG, NaviDroid utilizes the dynamic programming algorithm to plan the exploration path, and augment the run-time GUI with visualized hint moves for testers to quickly explore untested states and avoid duplication. The automated experiments demonstrate the high coverage and efficient path planning of NaviDroid. A user study further confirms its usefulness in the participants covering more states and activities, detecting more bugs within less time compared with the control group.
NaviDroid demo video: https://youtu.be/lShFyg_nTA0.

KEYWORDS
GUI testing, Android App, State Transition Graph, Human testing

1 INTRODUCTION
With the development of mobile devices, mobile apps are indispensable for people’s daily life in accessing the world. The importance of mobile apps makes it vital for the development team to carry out a thorough testing for ensuring the quality of mobile apps [8]. However, mobile apps are event-centric programs with rich graphical user interfaces (GUIs) [24, 30, 31], and interact with complex environments. To ensure the app quality, there are mainly two kinds of GUI testing i.e., automated GUI testing and manual GUI testing. The automated GUI testing studies for mobile apps include model-based, probability-based and deep learning-based approaches [11, 17, 23, 28]. Albeit its convenience and scalability, such automated testing may not have a high activity coverage, especially for those functionalities which can only be reached by complicated inputs or actions [2, 13]. Furthermore, the usability and accessibility bugs (e.g., color schema, font size, interaction) [5, 15, 26, 27] are difficult to reveal by automated GUI testing.

Figure 1: Example NaviDroid usage scenario.

Therefore, in addition to automated testing, companies also adopt manual testing as the last line of defense [12]. Research also showed that due to the usability and learning curve of automated tools, manual testing is preferred by many software developers [12, 19–21]. Compared with automated testing, human testers can discover more diverse and complicated bugs, especially those related to user experience. However, manual GUI testing also has the following challenges. First, it is time-consuming which requires many testers to manually explore each UI page of the app, and they may execute repeated actions. 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. To leverage the pros of both testing techniques, we propose a new hybrid approach,
We also design a context-aware state merging method to merge when a player hesitates to make a move in playing Candy Crush (a popular free-to-play match-three puzzle videogame) [1].

This paper proposes NaviDroid to assist manual GUI testing based on the insights from the automated GUI testing.

NaviDroid is a user-friendly app. Testers can upload the Android APK and get real-time guidance. Inspired by the automated testing, NaviDroid distills the prior knowledge of one app including all states and state relationships. During manual GUI testing, 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. That process is similar to the flashing candies (hint/suggested moves) when a player hesitates to make a move in playing Candy Crush (a popular free-to-play match-three puzzle videogame) [1]. According to our observation, that suggested hint move is particularly useful when the trigger components are small or poorly designed/developed. It can help human testers avoid missing some functionalities or making repeated exploration steps.

The contributions of this paper are as follows:

- We implement a static and dynamic based STG extraction method and a DP based path planning method to guide the path exploration in covering all the states with few repeated exploration steps.
- We develop a fully automated app NaviDroid. Users only need to upload an Apk file, and NaviDroid will automatically guide users to test the app. We release the implementation of NaviDroid on GitHub.
- An empirical study among professionals proves the usefulness of NaviDroid in assisting manual GUI testing and finding practical bugs.

2 APPROACH

This paper proposes NaviDroid to navigate testers in exploring apps to avoid missing functionalities or making repeated exploration steps. Figure 2 presents the overview of NaviDroid, which consists of three main components. First, given the app Android package (Apk), STG\(_{\text{action}}\) extraction component combines both static analysis and dynamic exploration to extract the state transition graph (STG) and its trigger actions between states (Section 2.1). 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\(_{\text{action}}\), the DP-based path planning component plans the exploration path which aims at covering all the states of the app with a few repeated exploration steps (Section 2.2). Third, with the planned path, the visual-based path guidance component utilizes the visual augmentation technology to guide users’ testing (Section 2.3). For more details please refer to our full paper [14].

2.1 STG\(_{\text{action}}\) Extraction

We first extract the state transition graph (STG), and then enrich it with trigger actions between the states to construct STG\(_{\text{action}}\), which serves as the basis in guiding the users exploring the app. In our method, STG\(_{\text{action}}\) is defined as a graph \( G < N, E > \) with node \( N \in \text{state} \) and edge \( E \in \text{action} \). State: Inspired by app GUI testing [17], we regard each unique UI page as one state and represent it by i.e., represented by UI components hierarchy tree. Each activity may have multiple state, that is, \( N \in \text{state} \in \text{activity} \). Action: Action is the trigger that results in state transition, which can be expressed as \( E = ID, E \in \text{action} \).

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, passing an intent that describes the activity to be launched [25, 29]. In detail, the target activity of ICC call is determined by querying the pointed-to values in the fields of an intent object. By matching the parameter in intent() method with the parameter in AndroidManifest.xml file, we obtain the transition between activities and build the initial STG\(_{\text{action}}\).

Dynamic STG Extraction and Trigger Action Detection. Some states and trigger actions, especially those in dynamic or mixed layout (such as dynamic rendering menu), are difficult to be obtained by static analysis. Instead, it is easy to be captured with dynamic GUI rendering. In detail, by leveraging the idea of dynamic app GUI testing [23], we adopt an app explorer [10] 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 between states. We then combine the STG\(_{\text{action}}\) extracted from static analysis and dynamic exploration into one graph.

Context-aware State Merging. Through static and dynamic analysis, we get STG\(_{\text{action}}\) which is composed of a large number of states and actions, in which some of them are duplicates [10, 23],

![Figure 2: Overview of NaviDroid.](image-url)
Given $STG_{\text{action}}$, we first merge the states with the same GUI run-time hierarchy 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 are similar.

2.2 DP-based Path Planning

With the $STG_{\text{action}}$ obtained in the previous section, we need to plan a path that can cover all the nodes (i.e., states) with a 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 and plan the shortest path.

**Formalization of Path Planning.** We formulate the path planning as a dynamic programming problem, and represent it by a 4-tuple: $<G, d, V, DP> : G$: Graph. The $STG_{\text{action}}$ $(G < N, E >)$ obtained in the previous section, where $N$ is the set of nodes (i.e., states), and $E$ is the set of edges (i.e., triggered events). $d$: Distance. $d_{ij}$ is the shortest distance between state $i$ and $j$. $V$: Visit status. $V$ is the visit status of the current node, represented by binary numbers. $0$ is not visited and $1$ is visited. $DP$: Dynamic programming. $DP_{ij}$ is the shortest distance from the current state $i$ to state $j$ in visit status $V$. Since $V$ is a binary number, $DP_{ij}(V^{\lambda(1<\lambda<\lambda-1)})$ is the distance of reaching state $i$ without accessing other states.

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

\[
\begin{align*}
    d_{ij} &= \min(d_{ij}, d_{ik} + d_{kj}) \\
    DP_{ij} &= \min(DP_{ikj}, DP_{i(V^{\lambda(1<\lambda<\lambda-1)})} + d_{ij})
\end{align*}
\]

**Planning Strategies.** The general idea of dynamic programming algorithm 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 $STG_{\text{action}}$ to obtain the set of nodes $N$ and edges $E$. It then employs Floyd algorithm to calculate the shortest path between each pair of nodes $d[i][j]$. 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 the shortest path $d[i][j]$ between node $i$ and node $j$, and update the node visit status $VisitStatus$. After all nodes in the graph are visited, the algorithm can recommend the planned path for testers to explore.

2.3 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 cover the unexplored pages and reduce the replication explorations. Specifically, we adopt the Android floating window [7] for visualizing the hint moves. 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. The system runs the floating window service in the backend, and sets the size and coordinates of the floating window to make it in the same position and suitable size and floating on the component, so as to guide the testers to explore the app’s interface.

3 TOOL IMPLEMENTATION AND USAGE

3.1 Implementation

NaviDroid can automatically extract the STG of the app and guide testers to test the app according to the dynamically planned path. Specifically, we augment the run-time GUI with visual hint moves. NaviDroid uses the Android debug bridge (adb) command to start the application that testers need to test. During the tester’s exploration, NaviDroid obtains the run-time information of the current state (interface) including the state information and existing components within the current page on the backend. Given the state information of the current page, NaviDroid searches the obtained STG on the fly, finds the next state on the planned path, and highlights the corresponding actions which can trigger that state in the Android GUI page.

Specifically, the NaviDroid mainly provides the following three modules: extracting the STG automatically and display, planning the exploration path based on the DP algorithm, guiding the testers in real time.

**Extracting the STG automatically and display:** Testers can upload the Apk of the application to NaviDroid. It will combine both static analysis and dynamic exploration to extract the STG of the app. The NaviDroid then automatically generates the STG.html to display the extracted STG.

**Planning the exploration path based on the DP algorithm:** According to the STG extracted from the previous module, the NaviDroid uses the dynamic programming algorithm to plan the test path in backend. 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, our NaviDroid will recalibrate the path by running the DP algorithm and take the current state as the starting point.

![Figure 3: Illustration of our NaviDroid.](image)

**Guiding the testers in real time:** Figure 3 gives the example of the hint moves in real-time. The NaviDroid runs the service in the backend, and sets the size and coordinates of the floating window (hint moves) to make it in the same position and floating on the component, so as to guide the testers to explore the app...
interface. We implement three types of trigger actions to provide a more friendly interactive experience, including:

- **Clicking a component**: In Figure 3 (a). For a button, a navigation bar or a fragment bar, the tester is suggested to click the UI component.
- **Returning back**: In Figure 3 (b). If there is no back button in the current interface and the id is ‘touch_back’, NaviDroid will suggest the tester with “back” action above the back key, otherwise, the tool will directly highlight the back button.
- **Long-pressing test widgets**: In Figure 3 (c). If “appwidget activity” exists in activity, for the “appwidget activity”, the tester will be suggested to operate “long press app test widgets” through the floating window when exploring the app.

3.2 Usage Scenarios

We present several examples to illustrate how testers would interact with NaviDroid. In some cases, when facing an unfamiliar app, testers usually know the function of the app by random clicking. However, testers may execute repeated actions during the exploration. At the beginning of the test, tester can directly input the Apk of the app to our NaviDroid. The NaviDroid will automatically extract the $STG_{\text{action}}$ and guide the user to explore the app according to the $STG_{\text{action}}$. It can help testers understand the application in the shortest time.

During the test process, in order to improve the activity coverage of manual testing, the NaviDroid dynamically plans the testing path according to the tested interface of the tester. Specifically, the NaviDroid uses adb to detect the user’s operation in the backend and records the access times of each state (interface). When the NaviDroid detects that the developer has not operated for more than 5 seconds, it will take the current state as the starting point and use the dynamic programming algorithm to re-plan the path. Because the new path considers the state that the user has tested, it can avoid repetition and guide the user to test more states. If the tester does not follow the path planned by NaviDroid, it will dynamically plan a new path according to the current state explored by the tester.

4 EVALUATION

4.1 Effectiveness Measurement

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 $STG_{\text{action}}$ extraction component and DP-based path planning component respectively.

Given the effectiveness of our NaviDroid for $STG_{\text{action}}$ extraction and path planning, we evaluate the NaviDroid on 85 open-source apps from F-Droid. This part is also published in our previous work [14] and we mainly use evaluation metrics of state coverage and exploration steps.

Figure 4 shows the performance comparison with the baselines. Results show that NaviDroid can achieve 81% median state coverage with the extracted $STG_{\text{action}}$, 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.

4.2 Usefulness Measurement

We further carry out a user study to evaluate its usefulness in assisting manual UI testing, with 20 apps from F-Droid. We recruit 32 testers to participate in the experiment from a crowdtesting platform TestIn. The experimental group (P1-16) who test the mobile apps guided by our NaviDroid, and the control group (P17-32) who conduct the testing without any assistant. This part is also published in our previous work [14]. 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 NaviDroid. This confirms the usefulness of NaviDroid in avoiding missing functionalities and making repeated exploration steps, and helping detecting bugs during manual testing.

Figure 5: Result of usefulness evaluation.

Regarding the user experience of NaviDroid, we create an online survey on 30 professional testers and researchers, whom major in computer science or mobile testing. We ask them to use and feedback the usefulness of NaviDroid, as well as its potential and scalability. In the end, they fill in the System Usability Scale (SUS) questionnaire (5-point Likert scale, e.g., 5 (strongly agree)).

Figure 6 summarizes the participants’ ratings of the 10 system design and usability questions in the SUS. The upper half of Figure 6 shows that they agree the features of the NaviDroid are well-devised. The lower half of figure 6 further confirms its simplicity and consistency. Furthermore, its average helpfulness for the tasks is 4.51, which indicates that they appreciate the help of NaviDroid. All of them appreciate that the hint moves can help guide them in exploring the inconspicuous UI pages, 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." Participants express they like our interaction design such as "Great, NaviDroid uses the float window for guidance is..."
a good idea.

"The guidance is much like the tutorial when the game software is first used. It can help us to understand a new app.

Participants also express that it can save their testing time such as "Nice! The NaviDroid saves our testing time!".

5 CONCLUSION

As the last line of defence, manual testing is crucial to improve app quality. Therefore, we develop the NaviDroid to guide human testers in exploring more states during app testing. We construct STGAction and generate the planned path based on a DP algorithm. On the app screen, we highlight the hint moves triggering the next unexplored state to users. 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 also improve the interaction between our NaviDroid and users, which can borrow the idea from the human-machine collaboration studies to better facilitate the testers.

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, and Youth Innovation Promotion Association Chinese Academy of Sciences.

REFERENCES

[1] 2022. Candy Crush Saga. https://www.king.com/game/candy crush
[2] Abdulaaziz Alshahan, Iftekhar Ahmed, and Sam Malek. 2020. Accessibility issues in Android apps: state of affairs, sentiments, and ways forward. In ICSE.
[3] Chunyang Chen, Ting Su, Guoju Meng, Zhengchao Xing, and Yang Liu. 2018. From user design image to gui skeleton: a neural machine translator to bootstrap mobile gui implementation. In Proceedings of the 40th International Conference on Software Engineering: 665–676.
[4] Qinyuan Chen, Chunyang Chen, Safwat Hassan, Zhengchao 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.
[5] 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).
[6] Sen Chen, Lingling Fan, Chunyang Chen, Ting Su, Wenli Li, Yang Liu, and Lifhua Xu. 2019. Storydroid: Automated generation of storyboard for android apps. In 2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE), IEEE, 596–607.
[7] Yan Chen, Masludhheer Pandey, Song, and Steve Oney. 2020. Improving Crowd-Supported GUI Testing with Structural Guidance. In CHI 2020.
[8] Sidong Feng, Suay Ma, Jinhong Yu, Chunyang Chen, TingTing Zhou, and Yankun Chen. 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.
[9] 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.
[10] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. Droidbot: a lightweight ui-guided test input generator for android. In ICSE 2017.
[11] 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.
[12] Mario Linares-Vazquez, Carlos Bernal-Cardenas, Kevin Moran, and Derek Poshyvanyk. 2017. How do developers test android applications? In ICSE 2017.
[13] Zhe Liu, Chunyang Chen, Junjie Wang, Yuehai Huang, Jun Hu, and Qing Wang. 2020. Owl Eyes: Spotting UI Display Issues via Visual Understanding. In 35th IEEE/ACM International Conference on Automated Software Engineering (ASE), 2020. Melbourne, Australia, September 21-25, 2020. IEEE, 398–409. https://doi.org/10.1109/ASE.2020.9314657
[14] Zhe Liu, Chunyang Chen, Junjie Wang, and Qing Wang. 2022. Guided Bug Crash. Assut Manual GUI Testing of Android Apps via Hint Moves. In CHI 2022. https://doi.org/10.1145/3491102.3501903
[15] Zhe Liu, Chunyang Chen, Junjie Wang, and Qing Wang. 2022. Nighthawk: Fully-Automated Localizing UI Display Issues via Visual Understanding. In IEEE Transactions on Software Engineering.
[16] 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.
[17] Minxue Pan, An Huang, Guoxin Wang, Tian Zhang, and Xuandong Li. 2020. Reinforcement learning based curiosity-driven testing of android apps. In ISSTA.
[18] Yuhu Su, Zhe Liu, Chunyang Chen, Junjie Wang, and Qing Wang. 2021. Owlyeyes: online a fully automated platform for detecting and localizing UI display issues. In IEEE/SE’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/10.1145/3468264.3471099
[19] Junjie Chen, Song Wang, Tongwei Wang, Tim Menzies, Qiang Cui, Miao Xie, and Qing Wang. 2021. Characterizing crowds to better optimize worker recommendations in crowdsourced testing. IEEE Trans. Software Eng. 47, 6 (2021), 1259–1276. https://doi.org/10.1109/TSE.2019.2915520
[20] Junjie Wang, Ye Yang, Rahul Krishna, Tim Menzies, and Qing Wang. 2019. iSENSE: Completion-Aware Crowdtesting Management. In ICSE’2019. 932–943.
[21] Junjie Wang, Ye Yang, Song Wang, Chunyang Chen, Dandan Wang, and Qing Wang. 2021. Context-aware Personalized Crowdsoftware Testing Task Recommendation. IEEE Transactions on Software Engineering (2021).
[22] Junjie Wang, Ye Yang, Song Wang, Yuanzhe Hu, Dandan Wang, and Qing Wang. 2020. Context-aware In-process Crowdworker Recommendation (ICSE 2020).
[23] Wenyu Wang, Wei Yang, Tianyuan Xu, and Tao Xie. 2021. Identify and Avoiding UI Exploration Tarps. In ESEC/FSE 2021.
[24] Muling Xie, Sidong Feng, Zhenchao Xing, Jishen Chen, and Chunyang Chen. 2020. UED: a hybrid tool for gui element detection. In ESEC/FSE ’20: 28th ACM Joint European Software Engineering Conference and Symposium on the Foundations of Software Engineering, Virtual Event, USA, November 8-13, 2020. P. Dembina, Myra B. Cohen, and Thomas Zimmermann (Eds.). ACM, 1655–1659. https://doi.org/10.1145/3360899.3417940
[25] Jiwri Yan, Linjie Pan, Jun Yan, and Bin Liang. 2020. Multiple-entry testing of android applications by constructing activity launching contexts. In ICSE 2020.
[26] Bo Yang, Zhenchao Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shaping 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.
[27] Bo Yang, Zhenchao Xing, Xin Xia, Chunyang Chen, Deheng Ye, and Shaping Li. 2021. US-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.
[28] Shengguan Yang, Haowei Wu, Haiulong Zhang, Yan Wang, Chandrasekar Vanyk, and Thomas Zimmermann. 2021. Context-aware In-process Crowdworker Recommendation in Crowdsourced Testing. In Proc. IEEE Trans. Software Eng.
[29] 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 Proceedings of the 25th International Symposium on Softw...