JOURNAL OF INFORMATION SCIENCE AND ENGINEERING 39, 1129-1153 (2023)
DOI: 10.6688/JISE.202309_39(5).0008

A Feedback-Directed Approach to Crawl Android Apps for Increasing Code Coverage[*]

SHU-LING CHEN[1], CHIEN-HUNG LIU[2,+] AND WEN-QUAN XIAO[2]
[1]*Department of Industrial Management and Information*
Southern Taiwan University of Science and Technology
Tainan, 700 Taiwan
[2]*Department of Computer Science and Information Engineering*
National Taipei University of Technology
Taipei, 106 Taiwan
E-mail: slchen@stust.edu.tw; cliu@ntut.edu.tw[+]; t108598013@ntut.org.tw

To automate GUI testing for Android apps, a popular technique is to use a GUI crawler to systematically explore the GUIs of the apps while detecting possible app crashes. However, during GUI exploration, the crawler may get stuck and crawl some GUI states repeatedly, resulting in no increase in code coverage. This can significantly affect the efficiency of the GUI crawler and thus the effectiveness of app crash detection. To relieve this problem, this paper proposes a feedback-directed approach to guide the behavior of the crawler. Specifically, the approach can assess whether the GUI crawler gets trapped based on the feedback from the crawling results, and dynamically adjust the priority of GUI states to visit in order to guide the crawler to improve code coverage. Particularly, to update the priority of GUI states, two feedback-directed strategies, CoverageDirectedStrategy and StateDirectedStrategy, are presented to lead the crawler to exercise more code or explore more GUI states, respectively. To evaluate the proposed approach, we have extended our earlier Android crawler called ACE to support the approach and strategies. The experimental results show that both feedback-directed strategies can effectively detect whether ACE is trapped and guide ACE out of the traps, thereby improving code coverage.

Keywords: Android crawler, GUI testing, Android app testing, feedback-directed, code coverage

1. INTRODUCTION

As Android apps are widely-used by people all over the world for most of their daily activities, such as work, entertainment, and communication, the quality and reliability of Android apps become crucial. As such, testing Android apps has drawn much attention in recent years. One way to test Android apps is to use a GUI crawler to systematically and automatically explore the apps' GUIs to detect any possible crashes. Additionally, GUI crawler can be used to generate GUI state models for the apps under test [1, 2]. From the state models, test cases can be further derived to validate the behaviors of the apps.

Although GUI crawlers are useful for automatically testing Android apps, and various Android crawlers [3, 4] have been proposed, how to effectively explore and test Android apps is still challenging. One critical challenge of crawling Android apps is so-called "state explosion problem," where the size of the app's GUI state space grows exponentially as

Received October 31, 2022; revised December 13, 2022; accepted March 17, 2023.
Communicated by YungYu Zhuang.
[+] Corresponding author.
[*] This research was partially supported by the Ministry of Science and Technology, Taiwan, under contract No. MOST 111-2221-E-027-078.

the number of GUI components in the app increases. This can make timely and comprehensive GUI exploration of an Android app infeasible. For example, a typical Contacts app usually has GUI screens that allow users to add a new contact into the contact list. Through these GUI screens, the user can repeatedly add new contacts, thereby changing the result of the contact list as well as its GUI state. If the GUI crawler keeps exploring and adding new contacts, a huge number of GUI states can be generated by running the same blocks of code over and over. In this case, the crawler may repeatedly explore these GUI states without increasing its code coverage. To tackle this challenge, a common method is state abstraction, which merges similar GUI states together to reduce the size of state space during GUI exploration [5-7].

Although the state abstraction method can be used to reduce the size of GUI states and improve the code coverage of GUI crawler, designing a proper abstraction for different Android apps is still challenging [7]. For instance, in our earlier work [8], we presented an Android Crawler, called ACE, with several state-equivalent strategies that allow the crawler to determine whether two explored GUI states of an app are equivalent to reduce the GUI state space. In these strategies, different levels of state abstraction are applied to a hierarchical GUI component tree that captures the structure and content of an app screen. The experimental results of ACE indicate that the state-equivalent strategies can indeed improve the code coverage. However, we also found that the increase of code coverage can become saturated, even giving ACE a considerable amount of crawling time.

To illustrate this, Fig. 1 shows the code coverage of crawling an open-source app called Omni-Notes [9] using ACE for 60 minutes. It can be observed that, in this test run, the code coverage goes from 0% all the way up to 31% and then becomes saturated at about 35 minutes. The result also suggests that the GUI exploration did not increase any code coverage for 25 minutes over 60 minutes.

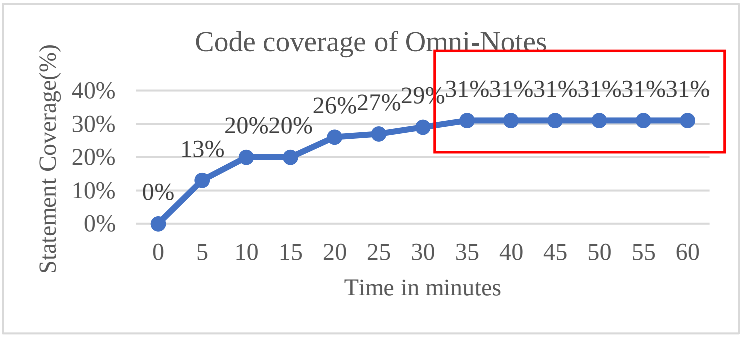


Fig. 1. The code coverage of crawling the Omni-Notes app using ACE for 60 minutes.

In order to avoid wasting crawling time and further improve the code coverage of ACE, we carefully analyzed the exploration results of different apps, and two cases caught our attention. The first is that ACE may continuously trigger some GUI events of an activity, which adds very little or even no code coverage. For example, as shown in Fig. 2, there are many GUI events in a setting activity of an app. Executing each event of the activity will pop up a dialog to the user for setting the tip percentage. However, ACE will add no code coverage if it executes the "Cancel" event of each dialog.

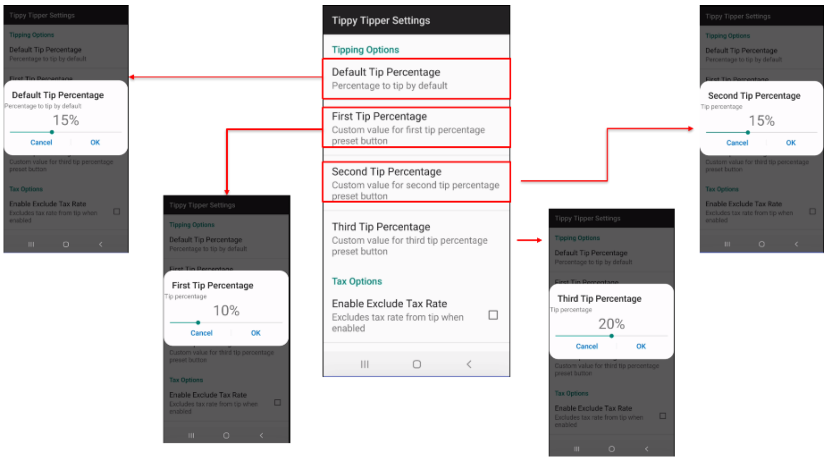

Tippy Tipper Settings
Tipping Options
Default Tip Percentage
Percentage to tip by default
First Tip Percentage
Custom value for first tip percentage preset button
Second Tip Percentage
Custom value for second tip percentage preset button
Third Tip Percentage
Custom value for third tip percentage preset button
Tax Options
Enable Exclude Tax Rate
Excludes tax rate from tip when enabled
15%
10%
20%
Cancel
OK

Fig. 2. An example to execute different events of an activity and add a little or no code coverage.

The second case is that ACE may repeatedly switch among several activities to execute some particular unexplored GUI events, such as the event for going back to parent activity. Similarly, switching between activities and executing the unexplored events may add very little or no code coverage. For instance, as shown in Fig. 3, there is a list of electronic templates (the parent activity in the center of figure), and the user can click the button of each electronic template to view the details of the electronic template (the child activity on the left or right of the figure). The user can click the backarrow in the upper left corner of the template details to navigate back to the activity's parent (*i.e.*, electronic template). Note that the GUI state of the template details could be different for each electronic in this app, such as Air Conditioner and Aquarium. Therefore, ACE should explore each GUI state of the template-details activity and execute the back-to-parent-activity event that returns to the GUI state of the electronic-template activity. However, switching between these two activities and executing the back-to-parent-activity event may add a little or no code coverage.

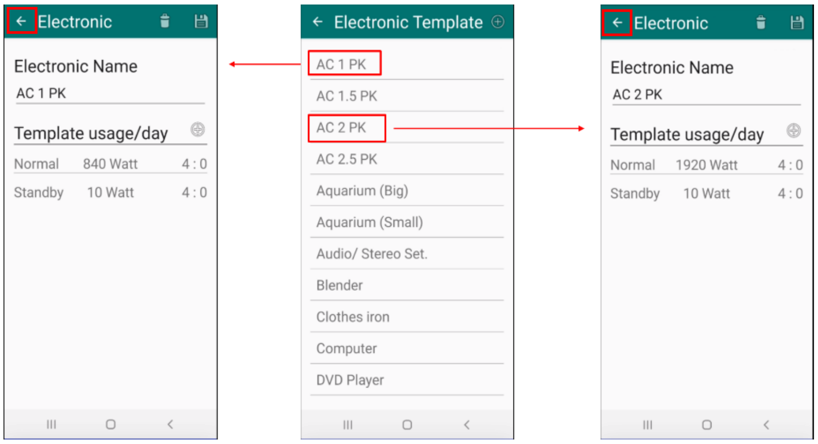


Fig. 3. An example to switch between several activities and add no code coverage.

The above two situations can cause the crawler to get stuck while exploring app's GUI states within or between activities without increasing code coverage, known as *Intra-Activity Loop* and *Inter-Activity Loop* respectively. To automatically detect and get rid of such situations, this paper proposes a feedback-directed approach, called Priority First Search (PFS), to guide the crawler to explore Android apps based on the feedback from the crawling results and dynamically adjust the priority of GUI states to visit in order to increase code coverage. Specifically, two feedback-directed strategies, CoverageDirected-Strategy and StateDirectedStrategy, are presented to guide the crawler to exercise more code or explore more GUI states, respectively. To evaluate the approach, ACE has been extended to support PFS and strategies, and several experiments have been conducted. The experimental results show that both feedback-directed strategies can effectively detect whether ACE gets trapped in a loop and guide ACE to get out of traps to increase code coverage.

The rest of the paper is organized as follows. Section 2 briefly reviews related work. Section 3 presents the proposed feedback-directed approach to detect the intra- and inter-activity loops and guide the crawler to get out of traps and improve code coverage. Section 4 describes and discusses the experimental results. The conclusion remarks and future work are given in Section 5.

2. RELATED WORK

To automate GUI testing of Android apps and improve the code coverage of the app under test, many Android crawlers have been proposed. This section briefly reviews existing studies related to this work. Based on the techniques used by the crawlers, these studies are classified into three categories: random or systematic, model-based, and learning-based crawling approaches.

• Random or Systematic Crawling

Machiry *et al.* [3] proposed a tool called Dynodroid to explore and test Android apps. Specifically, the tool can select and execute an event from the current GUI state according to some random selection strategies. Additionally, Dynodroid supports both GUI and system events, and allows users to manually add events. Thus, it can interleave input events from machine and human. However, the tool uses the exploration results only for computing the relevant UI events to be selected. It does not address whether the crawler can get trapped in the exploration or not.

Song *et al*. [10] proposed a method and tool called EHBDroid for testing Android apps. Particularly, instead of generating events from GUI states, the proposed method directly invokes the callbacks of event handlers to simulate a large number of events for improving test efficiency. In order to invoke callbacks, the source code of the apps is analyzed, and corresponding callback functions of event handlers are overridden. The experimental results show that, as compared to state-of-the-art tools, including Monkey [11] and Dynodroid [3], EHBDroid on average has higher statement coverage and better error detection rate.

Mao *et al.* [12] proposed a multi-objective search-based approach and a tool called Sapienz to explore and test Android apps. Particularly, the tool can minimize the length of event sequences to explore the apps while maximizing code coverage. It combines different techniques, including random fuzzing, search-based testing, string seeding, and multi-level instrumentation, to generate test inputs. The experimental results indicate that the proposed tool has better performance than other state-of-the-art tools. Similarly, the tool does not address the issue whether the crawler can get trapped during the exploration.

• Model-based Crawling

Amalfitano *et al.* [13], proposed a tool called AndroidRipper that can automatically explore the GUI of Android apps and detect runtime crashes. This work was extended to another tool called MobiGUITAR [2], which can dynamically generate a GUI state graph during the exploration instead of creating a GUI tree. This tool can provide satisfactory code coverage. However, it does not use the exploration results to guide the behavior of the crawler.

Cao *et al.* [14] presented a model-based GUI testing approach for Android apps. A tool called CrawlDroid is developed to support the approach. Particularly, in the tool a feedback-based exploration strategy is proposed. This strategy groups equivalent widgets in a state and assigns a priority value to each supported action of a group. Based on the crawling result, the priority will be adjusted so that the crawler can have more chances to select the actions to explore new GUI states. The experimental results show that the proposed approach is effective. However, the evaluation is based on the activity and method coverage metrics instead of statement and branch coverage.

Yan *et al.* [15] proposed a Multiple-Entry Testing (MET) approach and a tool called Fax to explore and test Android apps. Unlike traditional exploration strategies that start the exploration from a single default entry (*i.e.,* MainActivity), this can lead to uneven coverage on activity components and may cause some marginal activities not to be explored. To solve the problem, the MET approach aims to lunch activities directly under various contexts without executing long event sequences. Specifically, in the approach, the activity launching models are constructed through static analysis and the complete launching contexts are generated via dynamic exploration. An adaptive exploration framework is also provided to reassign events to multiple entries in order to achieve an in-depth exploration. The experimental results show that Fax can have higher activity and method coverage than that of Monkey [11] and can also reveal hidden bugs.

Su *et al.* [16] proposed an approach and a tool called Stoat (STOchastic model App Tester) for testing Android apps. Particularly, the tool first explores the app under test to construct a GUI model in the form of stochastic finite state machine. It then mutates the GUI model by iteratively perturbing the probability values associated with the transitions of the model. The mutated stochastic model is used to generate the test suites which are further injected with system events to uncover possible errors of the app. The proposed tool can effectively detect app crashes caused system events. However, the approach does not employ the crawling results to further improve the code coverage of the exploration.

Gu *et al.* [17] proposed an automatic approach and a tool called AimDroid for testing Android apps. The approach aims to reduce search complexity and app restart time by avoiding unnecessarily long transitions and minimizing the number of app restart. Specifically, the approach first systematically explores and discovers every unexplored activity using a BFS algorithm. It then insulates the discovered activity and intensively exploits the activity with a fuzzing algorithm guided by reinforcement learning. The experimental results show that, as compared to other works, AimDroid on average has better performance in activity, method and instruction coverage.

Dong *et al.* [18] proposed an approach and a tool called TimeMachine for testing Android apps. The main idea of the approach is to explore the GUI states of the app under test while identifying the "interesting states" based on the change of GUI or code coverage. The identified "interesting" state will be snapshotted and restored later when needed. During the exploration, if the crawler gets stuck (*i.e.*, lack of progress), select and restore the most progressive state from previously recorded state's snapshots and then resume the exploration. The progressive state is determined according to the number of times the state has been visited and the number of times the state has been identified as an "interesting" state. The experimental results show that, as compared to state-of-the-art tools, the proposed approach on average has better performance.

• Learning-based Crawling

Adamo *et al.* [19] presented a reinforcement learning approach for Android GUI Testing. Specifically, the approach systematically selects events from the GUI to execute and maximizes the cumulative reward based on Q-learning algorithm without requiring a preexisting GUI model. The reward for executing an event is simply calculated inversely proportional to how many times the event has been executed before. The experimental result indicates that the presented approach can have better block coverage on average than random test generation.

Vuong and Takada [20] presented an approach for Android app testing using Q-learning algorithm. In the approach, a Q-learning agent is used to select the next event to execute based on a greedy policy and the current behavioral model of the app. The reward is calculated based on two aspects: GUI change and event execution frequency. The larger the GUI changes between two states in terms of the number of events, the higher the reward the agent receives. However, the reward decreases as the event execution frequency increases. The experimental results suggest that the presented approach can improve code coverage on average and is able to discover faults as compared to existing random and model-based test tools.

Koroglu *et al.* [21] proposed a QLearning-Based Exploration (QBE) approach for Android GUI testing. Particularly, the approach first obtains the GUI models of the apps from a training set by exploring the apps automatically with a random exploration strategy. These models are then used to obtain a transition prioritization matrix (*i.e.* Q-Matrix) based on a reinforcement learning technique called QLearning for choosing event actions to explore the GUI of app under test (AUT) while maximizing the reward of activity coverage or crash detection. The evaluation results show that QBE performs better than other tools in terms of the activity coverage as well as the number of distinct detected crashes.

Li *et al.* [22] proposed a deep learning-based method for Android GUI test input generation and a tool called Humanoid. Specifically, Humanoid learns the interactions between human users and Android apps from a dataset, and builds a neural network model to guide the tool to choose components and actions from the GUI of apps for achieving higher test coverage. This allows Humanoid to generate test inputs just like a human user interacts with the GUI components of an Android app. The evaluation results show that Humanoid is able to achieve higher test coverage than other existing tools.

Pan *et al.* [23] proposed a Q-learning based approach for testing Android apps and a tool called Q-testing. In particular, the approach uses Q-table as a lightweight model to find out the executable actions while exploring unfamiliar states (*i.e.*, curious states) with a curiosity-driven strategy to improve code coverage and fault revelation. Moreover, to effectively determinate the reward and guide the exploration, a scenario division module is proposed to extract and distinguish whether two GUI states in the same functional scenario are similar using a neural network. The experimental results show that Q-testing performs better than other tools in terms of code coverage and fault detection.

YazdaniBanafsheDaragh and Malek [24] proposed a black box testing tool for Android apps called Monkey++. The proposed tool aims to extend Google Monkey [11]. Although Monkey is significantly robust and highly efficient, it does not have any specific knowledge about which event can be executed in the current GUI screen of an app. Thus, many events sent by Monkey can be inexecutable (*i.e.*, invalid). Monkey++ uses a deep learning method to train a neural network that can determine the probability of an event being executable (*i.e.*, valid) in an app's GUI screen, so as to guide Google Monkey to send valid events and improve the performance of Monkey.

As compared to the above related work, the proposed method is a model-based approach that primarily focuses on detecting and guiding the crawler to get out of loops during exploration to increase code coverage. Specifically, the approach selects the events to execute based on the GUI model and the feedback-directed strategies, which enables the crawler to exercise more code or explore more GUI states of the app under test, thereby improving code coverage.

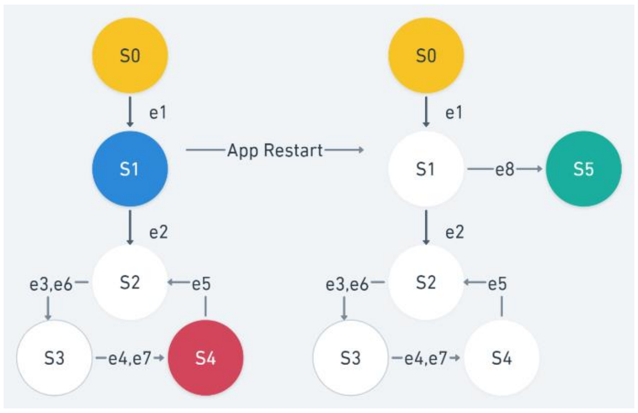
3. THE PROPOSED APPROACH

This section describes the proposed approach, including an overview of the approach, the PFS algorithm, and the feedback-directed strategies, used to detect whether the clawer get trapped in the intra- and inter-activity loops and guide the crawler to get out of the loops.

3.1 The Feedback-Directed Approach

In order to detect and get out of the intra- and inter-activity loops, a feedback-directed crawling approach is proposed. The main idea of the approach is to detect whether the crawler gets stuck in an intra- or inter-activity loop without increasing code coverage after executing each GUI event. If the crawler gets trapped in a loop, then guide the crawler to select a target GUI state, and continue the app exploration from the target state, so that the crawler can get out of the loop to improve its crawling efficiency while increasing code coverage. To determine the target GUI state from which has high probability to increase code coverage of the subsequent exploration, the proposed approach adopts the concept of "GUI state weight". A GUI state with a higher weight indicates that there may be a high probability that more new GUI states can be discovered from that state, thereby improving code coverages.

To illustrate the idea of the feedback-directed approach, Fig. 4 shows an example of GUI state graph generated by the crawler during the app exploration, given that the crawler starts from the initial state *S*0 and executes events *e*1, *e*2, *e*3, *e*4, *e*5, *e*6, *e*6, *e*7, *etc.* Assume the crawler detects that it gets trapped in a loop at *S*4 after executing *e*7 (Fig. 4 (a)). The crawler then can restart the app, go to the GUI state that has high probability to increase code coverage from *S*0, and continue the exploration. Assume that in Fig. 4 (a) *S*1 is the GUI state with the highest weight when the crawler restarts. The crawler will go to *S*1 and then continue the exploration, which discovers a new GUI state *S*5 as shown in Fig. 4 (b). The process will continue until exploration time out or no more unexplored GUI states and events.



(a) The crawler gets trapped in a loop. (b) The crawler gets out of the loop after restart.
Fig. 4. An example of GUI state graph to illustrate the idea of feedback-directed crawling.

To implement the feedback-directed approach, an algorithm called PFS (Priority First Search) is presented, as shown in Fig. 5. Basically, in the algorithm Lines 2 to 7 are the initialization of the crawler and the configuration of the feedback-directed strategy to be used, and Lines 8 to 23 are the main body of the crawling algorithm. Specifically, in Lines 9-11, the algorithm will take a candidate unexplored event from the current GUI state to explore. If such an event exists, the crawler does not get trapped, and the current state is not equivalent to a previously visited GUI state, then the algorithm will call the procedure executeEventAndUpdateStateRepository() in Lines 26-33. This procedure first executes the event, changes current state, and updates the state model (*i.e.*, the GUI state repository) accordingly. It then updates the weights of the corresponding GUI states in the GUI state repository and checks if the crawler gets trapped in the changed state based on the feedback-directed strategy. Finally, it examines if the changed state is equivalent to any previously explored GUI state.

Otherwise, the algorithm will restart the app under test and change the current state to the root state (Lines 13-14). It then finds a target GUI state that has the highest priority from the set of explored states (*i.e.*, GUI state repository). If such a target state exists, go to the target state from the root state (Lines 15-17), take a candidate unexplored event from the target state (Line 18), and call the procedure executeEventAndUpdateStateRepository() (Line 19) again to execute the event and update the weights of the GUI states accordingly. The algorithm will stop crawling when timeout is reached or current state remains the root state (*i.e.*, no more GUI states need to explore).

Algorithm: Priority_First_Search

Input: Android *app*, GUIStateRepository *stateRepository*
Output: log, CodeCoverageReport

```
 1 begin
 2   Initialize the crawling environment
 3   Start app
 4   currentState = stateRepository.getCurrentState() = rootState
 5   feedbackDirectedStrategy ← CoverageDirectedStrategy or StateDirectedStrategy //configured by user
 6   isTrapped ← false
 7   isEquivalent ← false
 8   while(!timeout)
 9       event = find a candidate GUI event of the currentState
10       if event ≠ null && !isTrapped && !isEquivalent then
11           executeEventAndUpdateStateRepository(event, stateRepository)
12       else
13           app.restart()
14           currentState = rootState
15           targetState = find a target GUI state with the highest priority from stateRepository
16           if targetState ≠ null then
17               goto the targetState from rootState
18               event = find a candidate GUI event of the targetState
19               executeEventAndUpdateStateRepository(event, stateRepository)
20           end if
21           if currentState == rootState then break // all states have been explored and end crawling
22       end if
23   end while
24 end
25
26 procedure executeEventAndUpdateStateRepository(e, stateRepo)
27 begin
28   execute e
29   stateRepo.updateCurrentState()    // update current GUI state
30   feedbackDirectedStrategy.updatePriority(stateRepo) // update the weights of corresponding GUI states
31   isTrapped = feedbackDirectedStrategy.isTrapped()
32   isEquivalent = stateRepo.isEquivalent()
33 end
```

Fig. 5. The algorithm of PFS (Priority First Search).

Note that the PFS algorithm will detect whether the crawler gets trapped in an intra- / inter-activity loop or goes to an equivalent GUI state. For both cases, PFS uses the weights of GUI states to guide the crawler to get out of the loop or to stop further exploring the equivalent state and restart the exploration from a target GUI state for the purpose of increasing code coverage. To achieve this objective, two feedback-directed strategies, CoverageDirectedStrategy and StateDirectedStrategy, are presented to (1) detect whether the crawler gets trapped in the intra- and inter-activity loops, and (2) define and update the weights of GUI states during the crawling. Specifically, the CoverageDirectedStrategy uses code coverage information of AUT to guide the exploration of the crawler. The StateDirectedStrategy uses GUI state information of AUT instead. Despite the performance of both strategies can be affected by the implementation of AUT, the CoverageDirectedStrategy can only be used when the source code of AUT is available. The StateDirectedStrategy, however, has no such restriction and can be used when the source code or apk of AUT is available. The details of these two strategies are described in Sections 3.2 and 3.3, respectively.

3.2 The Coverage-Directed Strategy

The CoverageDirectedStrategy is proposed to guide the behavior of crawler in the PFS algorithm to improve the code coverage by directly using the information of how much code of the app was executed through the exploration. Particularly, in this strategy, the code coverage of the app measured after executing a GUI event is used to detect if the crawler gets into trapped in the intra- or inter-activity loop as well as to update the weights of GUI states.

• Detecting Intra- and Inter-Activity Loops using Code Coverage Information

The basic idea to detect if the crawler gets trapped in the intra- or inter-activity is to count the number of times the crawler explores the same activity or switches between different activities and examines the average increase in code coverage. If the increase of code coverage is less than the expected improvement, the crawler is considered trapped in an intra- or inter-activity loop. Specifically, Eqs. (1) and (2) are the conditions used to determine if the crawler gets trapped in the intra- and inter-activity, respectively. In Eq. (1), $C(s_i^x, t)$ denotes the code coverage of the app when the crawler visits any GUI state s_i of an activity x for the t-th times, $Count_x$ is the counter that the crawler repeatedly revisits the GUI states of the activity x, TH_{intra} is the threshold of times that the crawler can repeatedly revisit the GUI states within an activity, and $CovAvg_{intra}$ represents the expected average improvement of code coverage when the crawler revisits a GUI state within an activity. Note that $Count_x$ will be increased by 1 when the next GUI state visited by the crawler remains in the same activity. Otherwise, it will be reset to 0. In Eq. (1), when the number of times the crawler repeatedly revisits the GUI states of the same activity exceeds a specified threshold (*i.e.*, $Count_x \geq TH_{intra}$), the average increase of code coverage is computed for each revisit from the first time to the last (*i.e.*, $(C(S_j^x, Count_x) - C(S_i^x, 1)) / Count_x$). If the average increase of code coverage is less than the expected improvement, $CovAvg_{intra}$, then the crawler is considered to be trapped in the intra-activity loop.

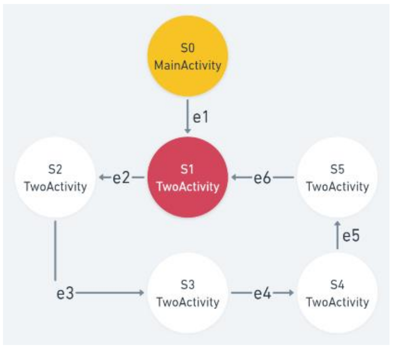
Similarly, in Eq. (2) $C(s_j^y, t)$ denotes the code coverage of the app when the crawler visits any GUI state s_j of the activity y for the t-th times, $Count_a$ is the counter that the craw-ler continuously visits the GUI states between different activities, TH_{inter} is the threshold of times that the crawler can continuously visit the GUI states of different activities, and $CovAvg_{inter}$ represents the expected average improvement of code coverage when the crawler continuously visits the GUI states of different activities. Note that unlike $Count_x$, $Count_a$ will be increased by 1 when the next GUI state visited by the crawler is in a different activity. Otherwise, it will be reset to 0. In Eq. (2), when the number of times the crawler continuously visits the GUI states of different activities exceeds the specified threshold (*i.e.*, $Count_a \geq TH_{inter}$), the average increase of code coverage is computed for each visit from the first time to the last (*i.e.*, $(C(S_j^y, Count_a) - C(S_i^x, 1)) / Count_a$). If the average increase of code coverage is less than the expected improvement, $CovAvg_{inter}$, then the crawler is considered to be trapped in the inter-activity loop.

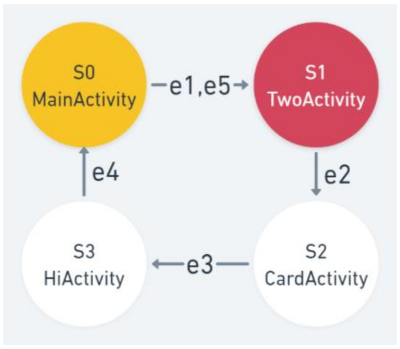
$$Trap_{intra\text{-}loop} = true \text{ if } Count_x \geq TH_{intra} \;\&\; (C(s_j^x, Count_x) - C(s_i^x, 1)) / Count_x < CovAvg_{intra} \quad (1)$$

$$Trap_{inter\text{-}loop} = true \text{ if } Count_a \geq TH_{inter} \;\&\; (C(s_j^y, Count_x) - C(s_i^x, 1)) / Count_a < CovAvg_{inter} \quad (2)$$

To illustrate, Figs. 6 (a) and (b) show examples of detecting an intra- and inter-activity loop with CoverageDirectedStrategy, respectively. Particularly, in Fig. 6 (a) the crawler starts the exploration form the root state $S0$, where $S0$ is a GUI state of the MainActivity. After executing event $e1$, the state of the app under test changes to $S1$, where $S1$ is a GUI state of the TwoActivity. Assume that in the subsequent exploration, the crawler repeatedly explores the states of the TwoActivity. Suppose that after executing event $e2$, the state changes to $S2$ and the code coverage of the app is $C(S2, 2)$. At this moment, the value of $Count_x$ is set to 1, where x is the TwoActivity. Let TH_{intra} be 5. Thus, after executing event $e6$, the state changes back to $S1$ and the value of $Count_x$ now becomes to 5. Assume the code coverage of the app is now $C(S1, 6)$. At this moment, the condition $Count_x \geq TH_{intra}$ becomes true and the crawler will be considered to be trapped in an intra-activity if $(C(S1, 6) - C(S2, 2))/5$ is less than $CovAvg_{intra}$.

Similarly, in Fig. 6 (b) the crawler starts the exploration form state $S0$, where $S0$ is a GUI state of the MainActivity. After executing event $e1$, the state of the app changes to $S1$ of the TwoActivity and the code coverage of the app is $C(S1, 1)$. Since states $S0$ and $S1$ belong to different activities, set the value of $Count_a$ to 1. Assume that in the subsequent exploration the crawler executes events $e2$, $e3$, $e4$, $e5$, and continuously visits the GUI states of different activities in the app. Let TH_{inter} be 5. Therefore, after executing event $e5$, the state changes to $S1$ again, the value of $Count_a$ now becomes to 5, and the code coverage of the app is $C(S1, 5)$. At this moment, the condition $Count_a \geq TH_{inter}$ becomes true and the crawler will be considered to be trapped in an inter-activity if $(C(S1, 5) - C(S1, 1))/5$ is less than $CovAvg_{inter}$.





(a) An example of intra-activity loop. (b) An example of inter-activity loop.

Fig. 6. Examples of detecting an intra- and inter-activity loop with Coverage-Directed Strategy.

• The Weight Design of GUI State for Coverage-Directed Strategy

To enable the crawler to have higher probability to explore the GUI states that may improve the code coverage of the app under test, a heuristic method is used to design and update the weights of GUI states based on the code coverage information of the app. Specifically, Eq. (3) gives the weight of a GUI state. If the GUI state *s* is a new state, the initial weight of *s* is *Init*(*s*) which is further defined in Eq. (4). Particularly, *Init*(*s*) = 2 if the code coverage of the app increases when the new state *s* is discovered; otherwise, *Init*(*s*) = 1. Such a design aims to encourage the crawler to explore a GUI state that leads to increase code coverage when the state was visited last time. The assumption behind this design is that visiting such GUI state again may be more likely to lead to additional unexplored GUI states which could increase the code coverage of the app. Note that even though a new state does not increase code coverage when discovered, new state may still contain events that lead to other unexplored states, thereby increasing the code coverage. As a result, there is deliberately a small difference between two possible initial weights.

Moreover, in Eq. (3) if the GUI state *s* is an equivalent state or an old state visited before, then the weight of *s* is updated to $(1+\alpha)W'(s)$, where the parameter α is defined in Eq. (5) and $W'(s)$ is the weight of *s* before update. Specifically, in Eq. (5) the value of α is determined according to the type of *s* and the code coverage result when visiting the state *s* again. Note that the weight of *s* increases when $\alpha > 0$, and decreases otherwise. Thus, when *s* is an equivalent state or has no executable events, or the code coverage remains unchanged when visiting *s*, the weight of *s* is decreased, thereby reducing the probability that the crawler will visit *s* again.

$$W(s) = \begin{cases} Init(s) & \text{if } s \text{ is a new state} \\ (1+\alpha)W'(s) & \text{if } s \text{ is an equivalent or existing state} \end{cases} \tag{3}$$

$$Init(s) = \begin{cases} 2 & \text{if code coverage increases when } s \text{ is discovered} \\ 1 & \text{otherwise} \end{cases} \tag{4}$$

$$\alpha = \begin{cases} -0.2 & \text{if } s \text{ is an equivalent state} \\ -0.1 & \text{if } s \text{ has no executable GUI events} \\ 0.5 & \text{if code coverage increases when visisting } s \\ -0.1 & \text{otherwise} \end{cases} \tag{5}$$

3.3 The State-Directed Strategy

While using code coverage information to detect if the crawler gets trapped in a loop and update the weights of the GUI states is kind of easy and straightforward, it will require considerable overheard if the collection and computation of code coverage need to be done for every execution of GUI event. This overhead can consume a lot of crawling time and therefore can significantly reduce the actual time a crawler is allowed to explore the app. As a result, the overall improvement of code coverage may be affected when using the CoverageDirectedStrategy. To avoid such overhead, the StateDirectedStrategy is proposed. Instead of using code coverage information after executing each GUI event, this strategy employs GUI state information to detect if the crawler gets into trapped in the intra- or inter-activity loop as well as to update the GUI state's weight.

The assumption behind the StateDirectedStrategy is that the number of GUI states explored by the crawler in an app can be proportional related to the percentage of app's code executed during the exploration. That is, the more GUI states the crawler discovers, the greater the chance of executing the app's code. Notice that although StateDirectedStrategy avoids the overhead of collecting code coverage, its result may be influenced by the state abstraction method used in the crawler. This is because the state abstraction method can determine whether a GUI state is a new state or is considered an equivalent state to reduce the state space of the exploration.

• Detecting Intra- and Inter-Activity Loops using GUI State Information

The basic idea of StateDirectedStrategy to detect if the crawler gets trapped in an intra- or inter-activity loop is to count the number of new GUI states discovered by the crawler within a sequence of event executions while exploring the same activity or navigating between different activities. The average increase of the number of GUI states is then examined. If the average increase is less than the expected improvement, the crawler is considered to be trapped in an intra- or inter-activity loop. In StateDirectedStrategy, Eqs. (6) and (7) define how to decide whether the crawler is trapped in the intra- and inter-activity, respectively. In particular, $N(s_i^x, t)$ denotes the number of new GUI states discovered when the crawler visits any GUI state s_i of an activity x for the t-th times, and $NumAvg_{intra}$ represents the expected average increase of the number of new GUI states discovered when the crawler revisits a GUI state of an activity. The definitions of $Count_x$ and TH_{intra} are similar to those specified in Eq. (1). Therefore, similar to Eq. (1), the crawler is considered to be trapped in an intra-activity loop if the number of new GUI states (*i.e.*, $(N(s_j^x, Count_x) - N(s_i^x, 1)) / Count_x$) increases on average is less than the expected improvement (*i.e.*, $NumAvg_{intra}$).

Likewise, in Eq. (7) $N(s_j^y, t)$, $Count_a$, and TH_{inter} are similar to those defined in Eq. (2), and $NumAvg_{inter}$ represents the expected average improvement in the number of new GUI states discovered when the crawler continuously visits a GUI state of different activity. Hence, like Eq. (2), the crawler is considered to be trapped in an inter-activity loop if the average increase in the number of new GUI states (*i.e.*, $(N(s_j^y, Count_a) - N(s_i^x, 1)) / Count_a$) is less than the expected improvement (*i.e.*, $NumAvg_{inter}$).

$$Trap_{intra\text{-}loop} = true \text{ if } Count_x \geq TH_{intra} \;\&\; (N(s_j^x, Count_x) - N(s_i^x, 1)) / Count_x < NumAvg_{intra} \tag{6}$$

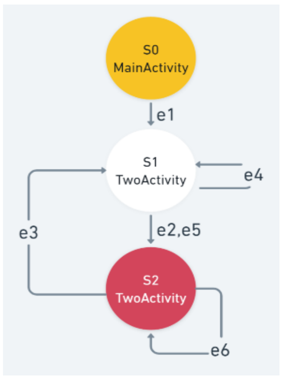
$$Trap_{inter\text{-}loop} = true \text{ if } Count_a \geq TH_{inter} \;\&\; (N(s_j^y, Count_a) - N(s_i^x, 1)) / Count_a < NumAvg_{inter} \tag{7}$$

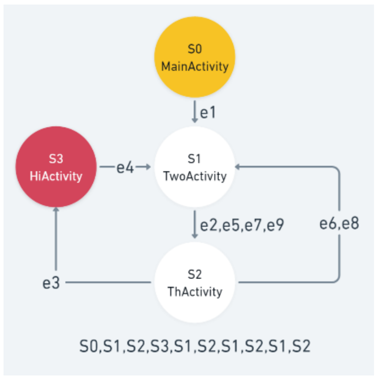
Alternatively, to ease the implementation of detecting if the crawler gets trapped in a loop using the GUI state information, the number of revisits for the GUI states within a sequence can be used. If the average number of revisits for each GUI state is greater than a threshold (*i.e.*, the acceptable minimum number of revisits for a state), the crawler can be considered to be trapped. The idea is that, in a sequence of GUI states, the more states the crawler revisits, the fewer new states the crawler can discover. Eq. (8) defines the condition whether the crawler gets trapped in an inter-activity loop, where $Len(s_i, s_j)$ denotes the number of GUI states in the sequence from s_i to s_j, L is a pre-defined length of state sequence, $\sum_k NumOfRevisits((s_i, s_j))$ represents the sum of the number of revisits for each state k within the state sequence (s_i, s_j), and $RevisitTH$ denotes the threshold of revisit.

$$Trap_{inter\text{-}loop} = true \text{ if } Len(s_i, s_j) \geq L \;\&\; \textstyle\sum_k NumOfRevisits((s_i, s_j))/Len(s_i, s_j) \geq RevisistTH \tag{8}$$

For illustration, Fig. 7 (a) shows an example of detecting an intra-activity loop using Eq. (6). Particularly, in Fig. 7 (a) the crawler starts the exploration form the state $S0$ of the MainActivity. After executing events $e1$ and $e2$, the state changes to $S1$ and then to $S2$. Both states $S1$ and $S2$ are belong to TwoActivity. At this moment, the value of $Count_x$ is set to 1, where x is TwoActivity, and the number of new GUI states is $N(S2,2) = 3$ (*i.e.*, $S0$, $S1$, and $S2$). Assume that in the subsequent exploration, the crawler repeatedly explores the states of TwoActivity by executing events $e3$, $e4$, $e5$, and $e6$. Therefore, after executing event $e6$, the state remains at $S2$ and the value of $Count_x$ now becomes to 5. Assume that TH_{intra} is 5. Now the number of new GUI states discovered is $N(S2,6) = 3$. At this time, $Count_x \geq TH_{intra}$ becomes true and the crawler is considered to be trapped in an intra-activity if $(N(S2,6) - N(S2,2))/5$ is less than $NumAvg_{intra}$.

Similarly, Fig. 7 (b) shows an example of detecting an inter-activity loop using Eq. (8). In Fig. 7 (b), the crawler starts the exploration form the state $S0$ of MainActivity. Assume that in the subsequent exploration the crawler executes events $e1$, $e2$, $e3$, …, and continuously visits the GUI states of different activities in the app. Suppose $L = 10$, so the state sequence starting from $S0$ is <$S0$, $S1$, $S2$, $S3$, $S1$, $S2$, $S1$, $S2$, $S1$, $S2$> after executing event $e9$. At this moment, the numbers of revisits for states $S0$, $S1$, $S2$, and $S3$ in the sequence are 0, 3, 3, and 0, respectively. Therefore, the revisits for each state in the sequence is <0, 3, 3, 0, 3, 3, 3, 3, 3, 3> and $\sum_k NumOfRevisits((S0, S2))$ is 24. Since $Len(S0, S2) = L = 10$, the condition $Len(S0, S2) \geq L$ becomes true and the crawler is considered to be trapped in an inter-activity if $\sum_k NumOfRevisits((S0, S2))/ Len(S0, S2)$ is greater than *RevisitTH*.



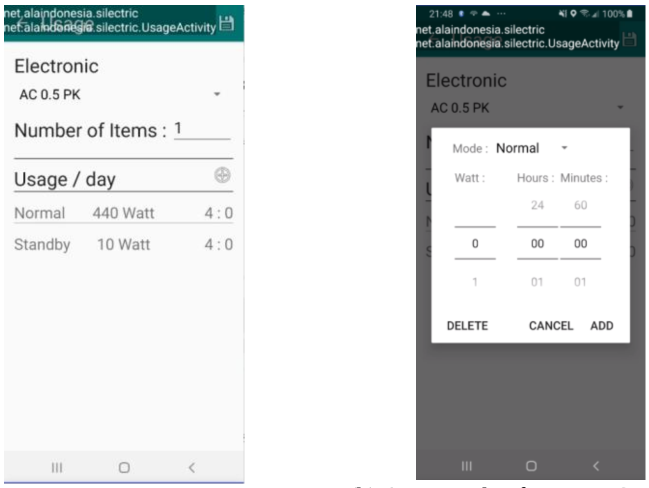


(a) An example of intra-activity loop. (b) An example of inter-activity loop.

Fig. 7. Examples of detecting an intra- and inter-activity loop with State-Directed Strategy.

• The Weight Design of GUI State for State-Directed Strategy

Similar to CoverageDirectedStrategy, a heuristic method is used to design and update the weights of GUI states. However, instead of using code coverage information, StateDirectedStrategy uses the GUI state information as feedback to initialize and update the weights of the GUI states dynamically. Eq. (9) defines the weight of a GUI state in the strategy. Specifically, if the GUI state s is a new state, the initial weight of s is $Init(s)$, which is further specified in Eq. (10) depending on the type of the GUI state. In the heuristic method, a GUI state is classified into two types, *normal state* and *popup state*.



(a) An example of a normal GUI state. (b) An example of a popup GUI state.
Fig. 8. Examples of different kinds of GUI state with different initial weight.

As shown in Fig. 8, a normal state represents a regular activity's view and a popup state represents a popup view in an Android app. The popup view can be a popup window or dialog and is usually created using Android UI components library, such as DatePicker, TimePic-ker, and Alert Dialog. Therefore, visiting a popup view usually may not lead to significant increase of code coverage. Thus, to encourage the crawler to explore an activity's view and not to keep crawling a popup view, such as TimePicker in Fig. 8 (b), the normal state is designed to have a much higher initial weight than the popup state. Additionally, in Eq. (10) a normal state of a new activity also has a higher weight than that of a previously visited activity. Such design is to encourage the crawler to explore newly visited activity rather than the one visited several times before since the newly visited activity may have more chances to lead to unexplored states, thereby increasing code coverage.

$$W(s) = \begin{cases} Init(s) & \text{if } s \text{ is a new state} \\ W'(s) + \Delta w(s_t) & \text{if } s \text{ is an equivalent or existing state, and } s_t = \text{toState}(s, e) \end{cases} \tag{9}$$

$$Init(s) = \begin{cases} 100 & \text{if } s \text{ is a normal state of a new activity} \\ 50 & \text{if } s \text{ is a normal state of a visited activity} \\ 25 & \text{if } s \text{ is a popup state of a visited activity} \end{cases} \tag{10}$$

Moreover, in Eq. (9) if the GUI state s is an equivalent state or an existing state visited before, then the weight of s is updated to $W'(s) + \Delta w(s_t)$, where $W'(s)$ is the weight of s before update, $\Delta w(s_t)$ is the change of the weight and is defined in Eq. (11), and s_t is the next state of s after executing an event e. This means that when the crawler executes an event e and changes the GUI state from s to s_t, the weight of s is updated accordingly using $\Delta w(s_t)$ with respect to the weight of s_t. The idea of such design is that if the state s_t has a high weight such as a new state of a new activity, then its previous state s should also have a high weight (relatively high compared to others) to lead the crawler to visit s_t. On the other hand, if the state s_t has a low weight such as a popup state, then its previous state s

should have a low weight too, so as to reduce the chance of visiting the state s_t from s.

In addition, in Eq. (11) the value of $\Delta w(s_t)$ is determined according to the type of s_t. Particularly, when the crawler visits s_t from s, if s_t is an equivalent state or has no executable events, the value of $\Delta w(s_t)$ is negative (*i.e.*, $-1 * W(s_t)$ or $-0.1 * W(s_t)$). Consequently, the weight of s will be decreased, and such update will discourage the crawler to visit s_t from s again. Otherwise, the value of $\Delta w(s_t)$ is increased by $2^{-RC(s_t)} * W(s_t)$, where $RC(s_t)$ represents the revisit count of s_t. Initially, when the state s_t is discovered the first time, $RC(s_t) = 0$, thereby $2^{-RC(s_t)} = 1$. Therefore, $\Delta w(s_t) = W(s_t)$ and at this time $\Delta w(s_t)$ has the largest value. The value of $\Delta w(s_t)$ will gradually become smaller and smaller as the number of revisit count of s_t increases. As the result, the increase of the weight of s will become very small if s_t is visited frequently through s. Once all the events of s_t are executed (*i.e.*, s_t has no executable events), then the weight of s will start to decrease. The main concept of this design is to encourage the crawler to explore a newly discovered state, which may have a better chance to lead to the improvement of code coverage, while discouraging the crawler to explore the state being visited many times which may not further increase code coverage.

$$\Delta w(s_t) = \begin{cases} -1 * W(s_t) & \text{if } s_t \text{ is an equivalent state} \\ -0.1 * W(s_t) & \text{if } s_t \text{ has no executable events} \\ 2^{-RC(s_t)} * W(s_t) & \text{otherwise} \end{cases} \tag{11}$$

4. EVALUATION

To evaluate the effectiveness of the proposed approach, we has extended ACE to support the proposed approach and feedback-directed strategies. Several experiments were conducted to evaluate the performance of the approach and the strategies. Specifically, the following three research questions are addressed.

- RQ1 What is the performance of ACE when CoverageDirectedStrategy is used to guide the crawler?
- RQ2 What is the performance of ACE when StateDirectedStrategy is used to guide the crawler?
- RQ3 Can the proposed approach of the extended ACE improve the code coverage as compared to the search-based algorithm of original ACE?

The subject apps used in the evaluation are shown in Table 1, including name, line of code, number of classes, number of methods, number of activities, and category of the apps. These subjects are open source and available in F-Droid Market [25]. Table 2 shows the hardware/software equipment used in the experiments. Moreover, the parameter settings of the experiments are given in Table 3. The parameter values, such as threshold and expected average improvement, are chosen to enable ACE to quickly detect potential intra- or inter-activity loops in the subject apps, while the timeout is set to 1 hour for the purpose of comparing the experimental results with those obtained using original ACE. In addition, for detecting if ACE gets trapped in the intra- and inter-activity loops in StateDirectedStrategy, Eqs. (6) and (8) are used. Two experiments were conducted to address the above research questions. Each experiment is described in the following subsections.

Table 1. The subject apps of the experiments.

App	Line of code (LOC)	Number of classes	Number of methods	Number of activities	Category
A2DP Volume	6984	23	847	9	Navigation
AnyMemo	19891	193	4158	28	Education
Bierverkostung	15649	190	3086	14	Note
BudgetWacth	4527	46	471	12	Financial
Caloriescope	2985	23	474	10	Healthy
CarReport	16181	177	2593	9	Financial
Chubbyclick	2361	18	510	2	Music
EpMobile	9564	97	1341	80	Medical Tools
FruitRadar	11010	109	1958	17	Tourism
Omninotes	14841	176	2678	13	Note
ParnenDD	2285	20	337	6	Navigation
Reminder	9058	43	4602	3	Clock
Rentalc	3489	27	291	13	Note
Silectric	3008	17	366	5	Financial
Tippytipper	2275	13	297	6	Financial

Table 2. The experimental equipment.

Hardware/Software	Specifications
Desktop CPU	Intel(R) Core(TM) i7-9700 CPU @ 3.00GHz
Desktop Memory	16GB LPDDR3 2133MHz
Desktop OS	Windows 10
Smartphone model	Samsung Galaxy A8
Smartphone CPU	Samsung Exynos 7885 (8 cores, 2×2.2GHz + 6×1.6GHz)
Smartphone Memory	4 GB
Android version	9.0

Table 3. The parameter settings of the experiments.

Parameter	Value	Parameter	Value
TH_{intra}	5	$NumAvg_{intra}$	1
TH_{inter}	5	L	10
$CovAvg_{intra}$	1%	$RevisitTH$	1
$CovAvg_{inter}$	1%	Timeout	1 hour

4.1 Experiment 1

The first experiment addresses RQ1 and RQ2. We use the extended ACE to explore the behavior of subject apps. To reduce the GUI state space, the state-equivalent strategies of ACE [8] is used. The results are shown in Table 4, including the number of the crawler trapped into intra- and inter-activity loops, the total time the crawler trapped in the loops, and the improvement of statement code coverage. The results indicate that the intra- or inter-activity loops can exist in many apps and both CoverageDirectedStrategy and State-DirectedStrategy can successfully detect if ACE gets trapped in the loops. In addition, the code coverage can be improved in average when ACE is guided to get out of the loops. Moreover, the results also suggest that it takes more time for CoverageDirectedStrategy to guide ACE to get out the loops than that of StateDirectedStrategy.

Table 4. The crawling results of ACE.

App	CoverageDirectedStrategy			StateDirectedStrategy		
	The Number of Loops	Total Time Trapped in Loops	Stmt. Coverage Improvement	The Number of Loops	Total Time Trapped in Loops	Stmt. Coverage Improvement
A2DP Volume	12	08:52	11%	6	02:13	1%
AnyMemo	10	14:03	7%	9	05:26	9%
Bierverkostung	3	04:32	0%	2	01:06	0%
BudgetWacth	6	04:47	13%	1	00:49	1%
Caloriescope	12	09:30	43%	1	00:19	13%
CarReport	5	05:17	1%	2	01:43	1%
Chubbyclick	23	15:01	36%	26	14:01	34%
EpMobile	11	09:01	6%	3	02:02	10%
FruitRadar	1	01:48	0%	4	04:04	1%
Omninotes	9	10:39	6%	3	01:54	2%
ParnenDD	18	12:31	14%	1	00:42	0%
Reminder	12	11:58	9%	2	00:57	13%
Rentalc	2	01:26	0%	4	03:35	3%
Silectric	4	02:08	23%	3	01:54	0%
Tippytipper	9	06:45	30%	5	01:21	17%
Average	9.1	07:52	13.2%	4.8	02:35	6.9%

Note that even though the intra- and inter-activity loops can be detected, a few apps, including Bierverkostung, FruitRadar, CarReport, and Rentalc, still do not seem to improve code coverage during the experimental runs using CoverageDirectedStrategy or StateDirectedStrategy (see Fig. 9). One possible reason for such results can be that the numbers of intra- and inter-activity loops detected in these apps are all small. As a result, there is no much improvement on code coverage introduced by getting out of loops. Another reason is that the crawling behavior is partially affected by the state-equivalent strategy used by ACE. Thus, after ACE gets out of loops, the overall code coverage does not increase apparently in the experimental runs.

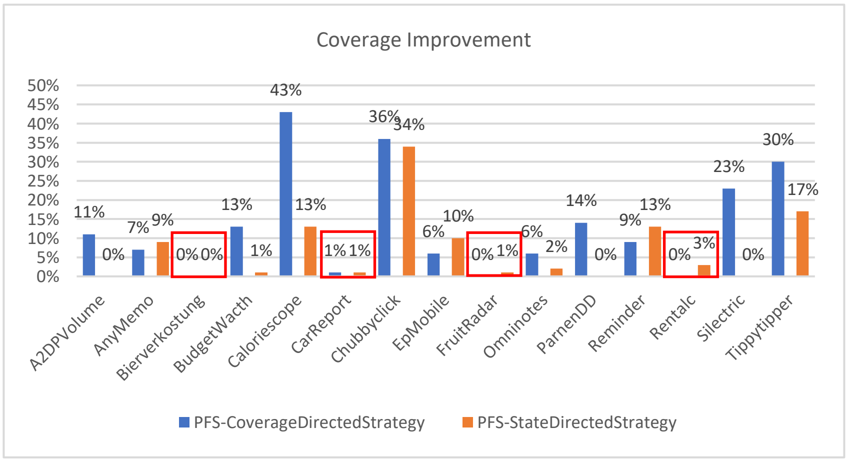


Fig. 9. The code coverage improvement of subject apps.

Overall, the answer to RQ1 is "CoverageDirectedStrategy can successfully detect if the crawler gets trapped in the intra- and inter-activity loops and guide the crawler to get out of the loops to increase code coverage." Specifically, the average number of intra- and inter-activity loops detected by the strategy is 9.1, the average time to guide the crawler to get out of loops is 7:52 (mm:ss), and the average improvement of statement code coverage is 13.2%.

Additionally, the answer to RQ2 is "StateDirectedStrategy can successfully detect whether the crawler gets trapped in the intra- and inter-activity loops and guide the crawler to get out of the loops to increase code coverage." Specifically, the average number of intra- and inter-activity loops detected by the strategy is 4.8, the average time to guide the crawler to get out of loops is 2:35 (mm:ss), and the average improvement of statement code coverage is 6.9%.

4.2 Experiment 2

The second experiment addresses RQ3. For the comparison of the proposed approach, the NFS (Nearest unvisited event First Search) algorithm [8] of original ACE is used. The NFS is a search-based algorithm where the nearest unvisited event of the current GUI state will be selected and executed to minimize the number of app restarts to improve the crawling efficiency. The experimental results indicated that on average NFS has better code coverage than that of Depth-first search (DFS) algorithm [8]. Table 5 shows the experimental results of NFS and PFS with CoverageDirectedStrategy, including the statement coverage (SC), the branch coverage (BC), the number of app restarts, the improvement of statement coverage (ΔSC), and the improvement of branch coverage (ΔBC). The results indicate that, as compared to NFS, the CoverageDirectedStrategy of PFS has better or similar code coverage in 9 apps (over 15 apps). The percentage of coverage improvement is ranged from 1% to 11% depending on the implementation of individual apps under test.

However, for some particular apps, such as AnyMemo, CoverageDirectedStrategy seems to perform poorly as compared to NFS. By examining the crawling details of this app, we found that it took much of the crawling time (35 over 60 minutes) to create code coverage report (*i.e.*, overhead) for this large app (with 19,891 LOC) during the exploration (see Fig. 10). Thus, the actual time spent to execute GUI events for exploring the app is only 25 minutes. This may be the reason that CoverageDirectedStrategy has less code coverage than NFS within 60 minutes time limit.

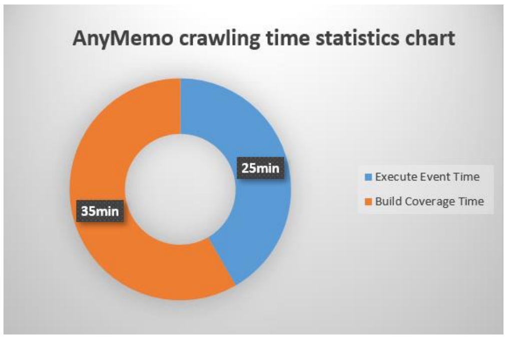


Fig. 10. The crawling time statistics of AnyMemo app using CoverageDirectedStrategy.

Table 5. The crawling results of NFS and PFS (CoverageDirectedStrategy).

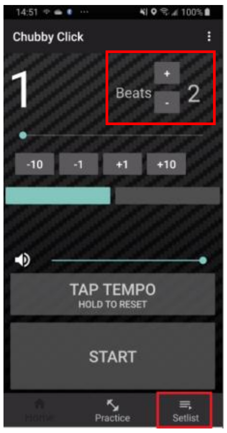
App	Algorithm	Statement Coverage (SC)	Branch Coverage (BC)	# of Restart	ΔSC	ΔBC
A2DPVolume	NFS	44%	25%	47	−6%	−5%
	PFS-CoverageDirectedStrategy	38%	20%	45		
AnyMemo	NFS	49%	31%	25	−14%	−7%
	PFS-CoverageDirectedStrategy	35%	24%	21		
Bierverkostung	NFS	47%	24%	41	−4%	−3%
	PFS-CoverageDirectedStrategy	43%	21%	29		
BudgetWacth	NFS	43%	21%	51	**11%**	**11%**
	PFS-CoverageDirectedStrategy	**54%**	**32%**	44		
Caloriescope	NFS	66%	46%	49	−6%	−8%
	PFS-CoverageDirectedStrategy	60%	38%	41		
CarReport	NFS	33%	24%	8	**1%**	0%
	PFS-CoverageDirectedStrategy	**34%**	**24%**	42		
Chubbyclick	NFS	77%	53%	31	0%	0%
	PFS-CoverageDirectedStrategy	77%	53%	59		
EpMobile	NFS	37%	16%	38	−5%	3%
	PFS-CoverageDirectedStrategy	32%	19%	59		
FruitRadar	NFS	**44%**	26%	19	**8%**	**5%**
	PFS-CoverageDirectedStrategy	52%	**31%**	27		
Omninotes	NFS	27%	21%	34	**4%**	**2%**
	PFS-CoverageDirectedStrategy	**31%**	**23%**	25		
ParnenDD	NFS	65%	49%	29	**2%**	**2%**
	PFS-CoverageDirectedStrategy	**67%**	**51%**	52		
Reminder	NFS	48%	35%	40	**4%**	**1%**
	PFS-CoverageDirectedStrategy	**52%**	**36%**	50		
Rentalc	NFS	64%	34%	55	**10%**	**12%**
	PFS-CoverageDirectedStrategy	**74%**	**46%**	29		
Silectric	NFS	91%	69%	75	**2%**	**2%**
	PFS-CoverageDirectedStrategy	93%	**71%**	84		
Tippytipper	NFS	80%	44%	105	−5%	−3%
	PFS-CoverageDirectedStrategy	75%	41%	91		
				Average	**0.13%**	**0.8%**

Table 6 shows the experimental results of NFS and PFS with StateDirectedStrategy, including the statement coverage, branch coverage, the number of app restarts, the improvement of statement coverage (ΔSC), and the improvement of branch coverage (ΔBC). The results indicate that, as compared to NFS, StateDirectedStrategy of PFS has better code coverage in 9 apps (over 15 apps). The percentage of coverage improvement is ranged from 1% to 18% depending on the implementation of individual apps under test. Moreover, it can be observed that for some apps CoverageDirectedStrategy performs better than StateDirectedStrategy, while others do not.

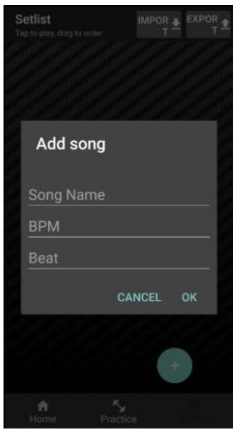
Similar to CoverageDirectedStrategy, StateDirectedStrategy seems to perform poorly for some apps as compared to NFS. For example, in Chubbyclick app, StateDirectedStrategy seems to perform worse than NFS. By examining the crawling details of the app, we found that this app can generate a lot of GUI states during the exploration, and a particular event that can lead to increase code coverage in NFS was not executed within 60 minutes time limit by the crawler of StateDirectedStrategy. As a result, StateDirectedStrategy has less code coverage than NFS. Specifically, as shown in Fig. 11 (a), when the crawler triggers the "+" or "–" button of the Chubbyclick app to turn up or down the volume on the beats, the value of beat volume on the upper-left corner will be changed accordingly.

Table 6. The crawling results of NFS and PFS (StateDirectedStrategy).

App	Algorithm	Statement Coverage (SC)	Branch Coverage (BC)	# of Restart	ΔSC	ΔBC
A2DPVol-ume	NFS	44%	25%	47	−1%	−3%
	PFS-StateDirectedStrategy	43%	22%	61		
AnyMemo	NFS	49%	31%	25	−3%	−1%
	PFS-StateDirectedStrategy	46%	30%	51		
Bierverkost ung	NFS	47%	24%	41	−5%	−4%
	PFS-StateDirectedStrategy	42%	20%	60		
Budg-etWacth	NFS	43%	21%	51	**10%**	**10%**
	PFS-StateDirectedStrategy	**53%**	**31%**	52		
Calo-riescope	NFS	66%	46%	49	−7%	−7%
	PFS-StateDirectedStrategy	59%	39%	59		
CarReport	NFS	33%	24%	8	**2%**	−2%
	PFS-StateDirectedStrategy	**35%**	22%	53		
Chubby-click	NFS	77%	53%	31	−17%	−16%
	PFS-StateDirectedStrategy	60%	37%	58		
EpMobile	NFS	37%	16%	38	**18%**	**12%**
	PFS-StateDirectedStrategy	**55%**	**28%**	64		
FruitRadar	NFS	44%	26%	19	**2%**	**2%**
	PFS-StateDirectedStrategy	**46%**	**28%**	42		
Omninotes	NFS	27%	21%	34	**6%**	**4%**
	PFS-StateDirectedStrategy	**33%**	**25%**	37		
ParnenDD	NFS	65%	49%	29	**5%**	**8%**
	PFS-StateDirectedStrategy	**67%**	**57%**	48		
Reminder	NFS	48%	35%	40	**5%**	**4%**
	PFS-StateDirectedStrategy	**53%**	**39%**	46		
Rentalc	NFS	64%	34%	55	**8%**	**9%**
	PFS-StateDirectedStrategy	**72%**	**43%**	37		
Silectric	NFS	91%	69%	75	−5%	−5%
	PFS-StateDirectedStrategy	86%	64%	111		
Tippytipper	NFS	80%	44%	105	**1%**	**1%**
	PFS-StateDirectedStrategy	**81%**	**45%**	96		
				Average	**1.1%**	**0.8%**



(a) The state that has many similar states.



(b) The unexploded state in StateDirectedStrategy.

Fig. 11. The GUI states of the Chubbyclick app being analyzed.

Therefore, a lot of GUI states with different values in beat volume can be generated. Unfortunately, StateDirectedStrategy seemed to be influenced in this special case, and before the timeout expiration it did not have chances to trigger the "Setlist" button in the bottom-right corner of Fig. 11 (a) that can lead to increase code coverage by executing a new activity to allow the user to add a new song as shown in Fig. 11 (b).

Overall, the proposed approach on average has slightly better code coverage than that of NFS in both feedback-directed strategies. Therefore, the answer to RQ3 is "yes, both the CoverageDirectedStrategy and StateDirectedStrategy of PFS can have better code coverage for most of the apps under test as compared to NFS." The improvement of statement coverage and branch coverage on average are 0.13% and 0.8% for CoverageDirectedStrategy and are 1.1% and 0.8% for StateDirectedStrategy, respectively.

5. CONCLUSIONS AND FUTURE WORK

This paper proposed a feedback-directed approach to guide the GUI crawler to explore Android apps automatically to increase code coverage. In particular, the paper presents two situations where the crawler may continuously execute GUI events of an activity or switch between different activities while adding very little or even no code coverage. These two situations are called intra- and inter-activity loops, respectively. To detect if the crawler gets trapped in such loops and guide the crawler to get out of the loops for improving code coverage, a PFS algorithm and two feedback-directed strategies, CoverageDirectedStrategy and StateDirectedStrategy, are proposed to guide the behavior of the crawler to exercise more code or explore more GUI states, respectively. The experimental results show that the proposed approach and both feedback-directed strategies indeed can detect intra- and inter-activity loops and guide the crawler to get out of loops to increase code coverage.

The proposed feedback-directed strategies can have different improvements on code coverage for different apps. Moreover, the experimental results can also be sensitive to parameter settings. Therefore, in the future, we plan to conduct more experiments on different kinds of apps with different parameter settings to study how to select the feedback-directed strategy with parameter settings for the types of apps under test to achieve better code coverage. We also plan to study if the code coverage can be further improved through combining two strategies. Further, the outcomes of the proposed approach can be influenced by the state-equivalent strategy used by the crawler. Therefore, we plan to study how the state abstraction method can affect the feedback-directed strategies and how to leverage different state-equivalent strategies for the proposed approach to guide the crawler to further improve code coverage.

REFERENCES

1. W. Yang, M. R. Prasad, and T. Xie, "A Grey-box approach for automated GUI-model generation of mobile applications," in *Proceedings of the 16th International Confer ence on Fundamental Approaches to Software Engineering*, 2013, pp. 250-265.
2. D. Amalfitano, A. R. Fasolino, P. Tramontana, B. D. Ta, and A. M. Memon, "MobiGUITAR – Automated model-based testing of mobile apps," *IEEE Software*, Vol. 32, 2015, pp. 53-59.

3. A. Machiry, R. Tahiliani, and M. Naik, "Dynodroid: an input generation system for Android apps," in *Proceedings of Joint Meeting on Foundations of Software Engineering*, 2013, pp. 224-234.
4. W. Choi, G. Necula, and K. Sen, "Guided GUI testing of Android apps with minimal restart and approximate learning," in *Proceedings of ACM SIGPLAN International Conference on Object Oriented Programming Systems Languages and Applications*, 2013, pp. 623-640.
5. Y.-M. Baek and D.-H. Bae, "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*, 2016, pp. 238-249.
6. B. Jiang, Y. Zhang, W. K. Chan, and Z. Zhang, "Which factor impacts GUI traversal-based test case generation technique most? A controlled experiment on Android applications," in *Proceedings of IEEE International Conference on Software Quality, Reliability and Security*, 2017, pp. 21-31.
7. T. Gu, C. Sun, X. Ma, C. Cao, C. Xu, Y.-Y. Yang, Q. Zhang, J. Lu, and Z. Su, "Practical GUI testing of Android applications via model abstraction and refinement," in *Proceedings of IEEE/ACM 41st International Conference on Software Engineering*, 2019, pp. 269-280.
8. C.-H. Liu, W.-K. Chen, and S.-H. Ho, "NFS: An algorithm for avoiding restarts to improve the efficiency of crawling Android applications," in *Proceedings of the 42nd IEEE International Conference on Computers, Software, and Applications*, 2018, pp. 69-74.
9. Omni-Notes, https://omninotes.app/, 2022.
10. W. Song, X. Qian, and J. Huang, "EHBDroid: Beyond GUI testing for Android applications," in *Proceedings of the 32nd IEEE/ACM International Conference on Automated Software Engineering*, 2017, pp. 27-37.
11. Android UI/Application Exerciser Monkey, http://developer.android.com/tools/help/monkey.html, 2022.
12. K. Mao, M. Harman, and Y. Jia, "Sapienz: multi-objective automated testing for Android applications," in *Proceedings of the 25th International Symposium on Software Testing and Analysis*, 2016, pp. 94-105.
13. D. Amalfitano, A. R. Fasolino, P. Tramontana, S. de Carmine, and A. M. Memon, "Using GUI ripping for automated testing of Android applications," in *Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering*, 2012, pp. 258-261.
14. Y. Cao, G. Wu, W. Chen, and J. Wei, "CrawlDroid: Effective model-based GUI testing of Android apps," in *Proceedings of the 10th Asia-Pacific Symposium on Internetware*, 2018, Article 19, pp. 1-6.
15. J. Yan, H. Liu, L. Pan, J. Yan, J. Zhang, and B. Liang, "Multiple-entry testing of Android applications by constructing activity launching contexts," in *Proceedings of IEEE /ACM 42nd International Conference on Software Engineering*, 2020, pp. 457-468.
16. T. Su, G. Meng, Y. Chen, K. Wu, W. Yang, Y. Yao, G. Pu, Y. Liu, and Z. Su, "Guided, stochastic model-based GUI testing of Android apps," in *Proceedings of the 11th Joint Meeting on Foundations of Software Engineering*, 2017, pp. 245-256.
17. T. Gu *et al.*, "AimDroid: Activity-insulated multi-level automated testing for Android

applications," in *Proceedings of IEEE International Conference on Software Maintenance and Evolution*, 2017, pp. 103-114.
18. Z. Dong, M. Böhme, L. Cojocaru, and A. Roychoudhury, "Time-travel testing of Android apps," in *Proceedings of IEEE/ACM 42nd International Conference on Software Engineering*, 2020, pp. 481-492.
19. D. Adamo, M. K. Khan, S. Koppula, and R. Bryce, "Reinforcement learning for Android GUI testing," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 2-8.
20. T. A. T. Vuong and S. Takada, "A reinforcement learning based approach to automated testing of Android applications," in *Proceedings of the 9th ACM SIGSOFT International Workshop on Automating TEST Case Design, Selection, and Evaluation*, 2018, pp. 31-37.
21. Y. Koroglu *et al.*, "QBE: QLearning-based exploration of Android applications," in *Proceedings of IEEE 11th International Conference on Software Testing, Verification and Validation*, 2018, pp. 105-115.
22. Y. Li, Z. Yang, Y. Guo, and X. Chen, "Humanoid: A deep learning-based approach to automated black-box Android app testing," in *Proceedings of the 34th IEEE/ACM International Conference on Automated Software Engineering*, 2019, pp. 1070-1073.
23. M. Pan, A. Huang, G. Wang, T. Zhang, and X. Li, "Reinforcement learning based curiosity-driven testing of Android applications," in *Proceedings of the 29th ACM SIGSOFT International Symposium on Software Testing and Analysis*, 2020, pp. 153-164.
24. F. YazdaniBanafsheDaragh and S. Malek, "Deep GUI: Black-box GUI input generation with deep learning," in *Proceedings of the 36th IEEE/ACM International Conference on Automated Software Engineering*, 2021, pp. 905-916.
25. F-Droid, https://www.f-droid.org/, 2022.

Shu-Ling Chen (陳淑玲) received her Ph.D. degree in Industrial and Manufacturing Systems Engineering from the University of Texas at Arlington in 2002. She is currently an Assistant Professor of the Industrial Management and Information Department at Southern Taiwan University of Science and Technology. Her research interests include information management systems, software testing, data mining, and big data analytics.

Chien-Hung Liu (劉建宏) received his Ph.D. degree in Computer Science and Engineering from the University of Texas at Arlington in 2002. He is currently a Professor of Computer Science and Information Engineering Department at National Taipei University of Technology, Taiwan. His research interests include software testing, software engineering, deep learning applications, and vocal detection.

Wen-Quan Xiao (蕭文全) received his BS and MS degrees in Computer Science and Information Engineering Department from National Formosa University and National Taipei University of Technology at Taiwan in 2019 and 2021, respectively. He is currently a Software Engineer of Titansoft Pte Ltd. His research interests include software engineering and software testing.