Evaluating Random Input Generation Strategies for Accessibility Testing

Diogo Oliveira Santos[1], Vinicius H. S. Durelli[2], Andre Takeshi Endo[3] and Marcelo Medeiros Eler[1]
[1]*University of São Paulo (EACH-USP), São Paulo, SP, Brazil*
[2]*Federal University of São João Del Rei, Minas Gerais, MG, Brazil*
[3]*Federal University of Technology - Paraná, Paraná, PR, Brazil*

Keywords: Accessibility, Automated, Testing, Tool, Evaluation, Random, Mobile.

Abstract: Mobile accessibility testing is the process of checking whether a mobile app can be perceived, understood, and operated by a wide range of users. Accessibility testing tools can support this activity by automatically generating user inputs to navigate through the app under evaluation and run accessibility checks in each new discovered screen. The algorithm that determines which user input will be generated to simulate the user interaction plays a pivotal role in such an approach. In the state of the art approaches, a Uniform Random algorithm is usually employed. In this paper, we compared the results of the default algorithm implemented by a state of the art tool with four different biased random strategies taking into account the number of activities executed, screen states traversed, and accessibility violations revealed. Our results show that the default algorithm had the worst performance while the algorithm biased towards different weights assigned to specific actions and widgets had the best performance.

1 INTRODUCTION

Accessibility evaluation aims at checking whether an information system can be perceived, understood, or operated by users with different characteristics, specially those with any disability (ISO, 2018). This is not a trivial task since the specific requirements for a wide range of users may be vast and diverse. Therefore, organizations have developed standards like the Web Content Accessibility Guidelines (WCAG), which defines guidelines and success criteria written as testable statements to guide the development of accessible digital products. Such standards can be used during both the development and evaluation phases.

Although organizations can rely on well-known standards, proper accessibility evaluations are costly and time consuming because ideally it should be conducted by end users and specialists. This poses the following challenges: *(i)* the number and diversity of users required to conduct a proper evaluation may be prohibitive for many organizations, and *(ii)* while specialists can emulate the behavior of many end users, they usually carry out this activity manually (Paz and Pow-Sang, 2016), which largely involves interacting with the system under evaluation to inspect each screen, component, or interaction, often relying on assistive features such as screen readers.

Recently, in hopes of mitigating costs associated with accessibility evaluations, researchers have come up with automated approaches (Siebra et al., 2018; Silva et al., 2018a; Agüero-Flores et al., 2019; Abascal et al., 2019; Frazão and Duarte, 2020). However, organizations should not solely rely on automated tests because there are many accessibility violations whose identification entails subjective judgment (Vigo et al., 2013). For example, it is easy for an automated tool to check whether the contrast ratio between a text element and its background color complies with agreed upon standards (i.e. 4.5/1), but it cannot decide whether the focus sequence of the elements in a screen is semantically correct for the user. Despite the limitations inherent to automated approaches, studies suggest that they can contribute with the evaluation process (Souza et al., 2019; Antonelli et al., 2019; Mateus et al., 2020).

Several tools tailored to mobile accessibility evaluation have been proposed (Eler et al., 2018; Park et al., 2019; Alshayban et al., 2020; Siebra et al., 2018; Silva et al., 2018a; Hao et al., 2014). Particularly, fully automated approaches generate test inputs to simulate user interaction and automatically explore the application under test while accessibility tests are run for each visited screen. Such approaches are based on uniform random input generation strategies with the goal of simulating user interaction (Alshayban et al., 2020;

Santos, D., Durelli, V., Endo, A. and Eler, M.
Evaluating Random Input Generation Strategies for Accessibility Testing.
DOI: 10.5220/0010456100660075
In *Proceedings of the 23rd International Conference on Enterprise Information Systems (ICEIS 2021) - Volume 2*, pages 66-75
ISBN: 978-989-758-509-8
Copyright © 2021 by SCITEPRESS – Science and Technology Publications, Lda. All rights reserved

Eler et al., 2018). In this strategy, a specific action is randomly selected from a list of on-screen actions (i.e., actions that the user can see and interact with on a given screen). While such an approach lends itself well to accessibility testing and thus has the potential to reveal lots of accessibility violations, it does not generate complex interactions or sensitive data to further explore the app under evaluation, which may let many parts of the evaluated applications uncovered (Eler et al., 2018; Alshayban et al., 2020).

Hence, an interesting question in this matter is to evaluate whether other random testing strategies would fare better in the detection of accessibility violations. Even though different strategies can be applied to guide test generation for mobile apps so the app under test can be further explored (e.g. search-based testing), Random Testing is cheaper and it does not require instrumentation (modification in the app); this makes it a robust effective strategy to explore a software under test (Choudhary et al., 2015; Tramontana et al., 2019; Pacheco and Ernst, 2007).

Therefore, this paper compares the effectiveness of five flavours of random approaches to generate accessibility tests for mobile applications. More specifically, we implemented four new random algorithms on top of an open source tool called MATE (Mobile Accessibility Testing) (Eler et al., 2018). The default strategy adopted by MATE is a uniform random algorithm in which each event of the mobile app under test has the same probability of being selected. The new algorithms are based on biased approaches that use frequency and weights to assign different probabilities for each event. Results show that the default strategy adopted by MATE had the worst performance and an algorithm that selects user input biased towards specific actions executed over specific widgets had the best performance.

This paper is organized as follows. Section 2 presents basic concepts regarding accessibility and accessibility testing approaches. Section 3 describes the tool in which we implemented the five evaluated strategies to generate test inputs for accessibility evaluation. Section 4 shows the setup and Section 5 presents the results and the discussion of our experiment. Finally, Section 6 brings some concluding remarks.

2 BACKGROUND

Android apps are composed by *Activities*, components that offer user interfaces and can be invoked individually. As such, they work as a container of components called *widgets*, e.g., buttons, text fields, toggle switches, etc., which allow the user to interact with the app and navigate through other activities. An activity is usually presented to the user as a full-screen window, but they can also be used as floating or embedded windows. One single activity may present different widget configurations (i.e., different arrangement, states and colors) as the user interacts with the app.

From a software testing perspective, it is important to identify the different screen configurations (screen states) reached during a test session because it might represent that different functionalities of the app under test have been covered (Baek and Bae, 2016). From the accessibility testing standpoint, different screen states may present different properties and user interaction options that might be of interest.

Figure 1 shows four examples of screen states of the activity *New List* of an app that provides users with shopping lists facilities. Screen state A is the entry point of this activity and presents the user with input fields, radio buttons, check boxes and buttons. Screen state B is different from screen state A because the input field for the shopping list name is not empty anymore. Screen state C is different from screen state B because the radio button on the top right corner is activated. Finally, screen state D is different from screen state C because the checkbox *deadline* is checked and it has additional components (widgets).

Several tools have been proposed to support the automated testing for mobile accessibility (Silva et al., 2018a; Siebra et al., 2018). Even though many accessibility barriers can only be detected based on human judgement (Vigo et al., 2013; Mateus et al., 2020), supporting tools can increase the productivity of the evaluation task. Automated tools for accessibility testing can perform static or dynamic analysis. Static analysis tools can only detect violations on the source code, but the lion's share of accessibility violations are caused by dynamic changes that might happen during execution.

Dynamic analysis tools can be partially or fully automated. Partially automated tools rely on manual interaction or test scripts created by developers or testers to find accessibility violations. The limitation of such strategy is that manual interaction is error prone and labor intensive, because it requires to navigate through the application while the underlying tool performs accessibility checks. Additionally, tests cannot be re-executed when the interface changes. With respect to script-based approaches, their effectiveness is determined by the thoroughness of the test set. Fully automated approaches generate user inputs to simulate user interaction while accessibility checks are executed in each new screen explored.

As far as we know, there are only three fully automated tools that support accessibility testing and

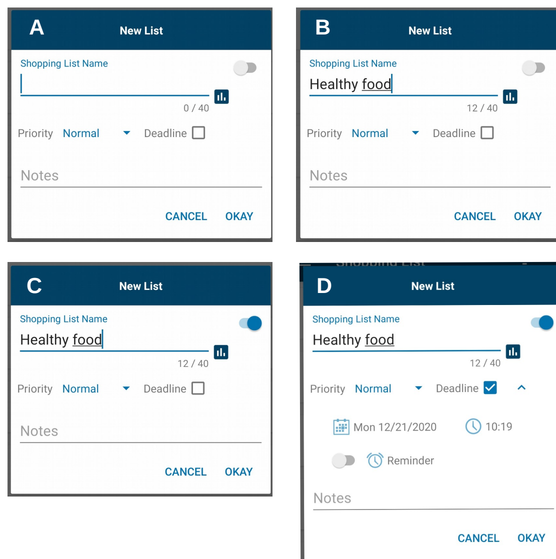

Figure 1: Examples of different screen states of a single activity.

automatically generate test inputs to navigate through the application. PUMA is a generic framework that can be customized for various types of dynamic analyses, such as checks for button sizes (Hao et al., 2014). Alshayban et. al (2020) developed a tool to assess the accessibility features of Android applications, which takes advantage of the accessibility checks provided by Google's Accessibility Testing Framework. The accessibility assessment tool has two main parts. One part simulates user interactions and the other part monitors the device for accessibility events.

MATE (Eler et al., 2018) is a tool designed with the goal of automatically exploring applications while applying different checks for accessibility violations according to the WCAG 2.1 (Web Content Accessibility Guidelines). MATE is compatible with the Android environment (emulators or physical devices) and uses the UIAutomator structure to interact with the application without requiring access to the source code of the application under test. One of the limitation of this approach is that the algorithm that generates user inputs may not generate complex interactions or sensitive data to further explore the app under evaluation.

3 APPROACH DESCRIPTION

The results of comparing different algorithms that generates test inputs using different tools may be influenced by the underlying implementation and the engineering choices of the testing tool. Therefore, we decided to implement the new flavours of random test generation algorithms we devised for this study on top of the open source tool called MATE (Eler et al., 2018). In this section, we provide an overview of how this tool operates, the types of accessibility violations the tool can find, and the new algorithms we implemented.

3.1 MATE: Mobile Accessibility Testing

MATE automatically generates inputs (e.g. tap, swipe) to simulate user interaction and run accessibility tests in each visited screen. The tool runs as a mobile app in the Android environment and execute the following steps repeatedly until a time budget is met.

Step 1: the tool extracts detailed information from the visual components presented on the device or emulator screen using the *Accessibility Service*[1] API. The information available for each visual component can vary, but they usually include: coordinates in the screen, component type, content description, label, hint, text, drawing order, and so forth. This API also allows to retrieve the events each component or the screen itself can handle, such as if the component can be taped, or checked, or scrolled.

Step 2: the tool uses the information retrieved in *Step 1* to check whether that screen follows specific accessibility criteria aiming at finding accessibility violations. MATE checks for more than accessibility criteria defined by the WCAG 2.1, such as for *Non-text Content* (guideline 1.1.1), which checks whether non-text components (e.g. images) have a text alternative to describe their purpose, of for *Target Size* (guideline 2.5.5), which checks whether actionable components have at least a size of 44 by 44 pixels.

Step 3: the tool uses the information on the possible events the components or the screen can handle (collected at *Step 1*) to create a list of actions that will simulate user interaction, such as "tap on button add product" or "insert text in the input field product name". MATE selects an action to be executed using an algorithm called *Uniform Random (UR)*, that selects a possible action from a list in which each action has the same probability to be chosen.

The selected action is executed by means of the *UiAutomator*[2] API. The action selected to simulate user interaction is important because it is used to navigate through the app while accessibility checks are run. Depending on the actions executed, the tool can explore more or less functionalities and different screens; this impacts on the application coverage and the number of accessibility violations revealed.

[1] https://developer.android.com/reference/android/accessibilityservice/AccessibilityService

[2] https://developer.android.com/training/testing/ui-automator

Step 4: the tool registers whether the executed action (in *Step 3*) moved the app to a screen that had not been visited before.

3.2 New Algorithms

In this work, we implemented four different flavours of random algorithms that receive a list of possible actions that a screen can handle and select one action based on different strategies. Following we present the different random approaches each algorithm use, which were largely inspired in testing tools of general purposes (Machiry et al., 2013; Su et al., 2017).

Biased Random - New Action (BRNA): all actions of the list have the same probability of being selected, as the default algorithm implemented by MATE, but once an action is selected for a given screen, it is removed from the list. When the list of actions of a given screen is empty, the original list is restored.

Biased Random - Fixed Weight (BRFW): some type of actions have more probability of being selected than others by assigning different weights to different types of actions (e.g. tap, type, back). Such weights were defined based on an investigation in which we tested 30 mobile applications during 30 minutes each using the *Uniform Random* algorithm to collect information of which type of action lead to the exploration of new screens. Our results are presented in Table 1.

Table 1: Percentage of new states and weights assigned to each type of event.

Action type	New Screen exp.	Weight
TAP	65.20%	8
TYPE-TEXT	10.80%	5
SWIPE-DOWN	9.80%	5
MENU	3.1%	3
BACK	3.0%	3
SWIPE-UP	3.0%	3
ENTER	1.7%	2
LONG-PRESS	1.2%	2
SWIPE-RIGHT	1.2%	2
SWIPE-LEFT	1.1%	2
OTHER-TYPES	<1%	1

Instead of using the absolute values presented at the column *New screen exploration*, we decided to use the Fibonacci scale to reduce the gap between the weights assigned to each type of action, similarly to what is generally done in the software estimation in agile projects (Dybå et al., 2014). In this case, we decided that the weights would be assigned based on the percentagem of new screens explored by that action as follows: 1 (below 1%), 2 (1% to 2%), 3 (3% to 5%), 5 (6% to 10%), 8 (above 11%).

Biased Random - Fixed Weighted Widget-Action (BRFWWA): some types of actions combined with some types of components (e.g. tap on a button, tap on a spinner, or type text in an input box) have more probability of being selected than others. Some actions are associated with the screen itself, such as *BACK* or even SWIPE actions. Weights were assigned based on the same investigation mentioned in the description of the *Fixed Weighted Action Random* algorithm. Here are the list of assigned weights: *TAP* on LinearLayout(8), ImageButton (8), Button" (8), CheckBox (8), TextView (5), RadioButton (3), ImageView (3), FrameLayout (2), RelativeLayout (2), LinearLayoutCompat (2); *TYPE-TEXT* in any type of input box (8); *SWIPE-DOWN* (8), *SWIPE-UP* (2), *MENU* (3), *BACK* (3), *ENTER* (2), all other actions and components (1).

Biased Random - Uniform Variable Weight (BRUVW): all actions have the same probability of being selected at first because they are all assigned the same weight: 5. However, the weight assigned for each type of action (e.g. tap, type, back) is variable during the execution. Each time an action is selected, that type of action has its weight reduced in one point. When the weight reaches 0, it is restored to 5.

4 EXPERIMENTAL DESIGN

Considering that different approaches to generating inputs for uncovering accessibility violations are likely to perform differently across various apps, we set out to examine how five algorithms perform when compared with each other. We believe that the results of our experiment can shed some light into the merits and limitations of each algorithm. Specifically, our analysis is centered on the following research question:

RQ$_1$: How do the algorithms compare to each other with respect to their effectiveness?

4.1 Scope of the Experiment

We defined the scope of our experiment by setting its goals, which were summarized according to the Goal/Question/Metric (GQM) (Wohlin et al., 2012) template as follows:

Analyze: a uniform random algorithm and four adaptive random input generation algorithms
for the purpose of: evaluation
with respect to their: effectiveness
from the point of view of: the researcher
in the context of: mobile apps.

SCITEPRESS SCIENCE AND TECHNOLOGY PUBLICATIONS

4.2 Variables Selection

Given that it is challenging to compare algorithms in terms of their effectiveness, prior to carrying out the experiment, we needed to come up with an *operationalization* of effectiveness. We decided to evaluate the effectiveness of the algorithms in terms of three dependent variables: *(i)* activity coverage, *(ii)* traversed state coverage, and *(iii)* the amount of uncovered accessibility violations.

The independent variables are the algorithms we set out to examine, *Uniform Random* (UR), *No Repetition Random*, *Fixed Weight Random*, *Variable Weight Random*, and *Widget-Action Random*. An in-depth description of these algorithms is presented in Section 3.

The control variables are the *(i)* time budget allocated for each algorithm to generate user inputs automatically, *(ii)* the number of times each algorithm is executed, and *(iii)* the execution environment. The time budget we chose for all algorithms is exactly 30 minutes, which has been used in several studies regarding test generation (Eler et al., 2018; Choudhary et al., 2015). Due to the inherently random behavior of the algorithms, they were executed three times and the outcome used with respect to the independent variables is the average between the three executions.

The execution environment for each app and each input generation algorithm was the same: an Intel Core i7 processor, 16GB RAM. Prior to each execution, the emulator running the application had all its data erased and it was rebooted.

In the next subsection we describe the criteria we adopted to select our study sample.

4.3 Sample Selection

As described in Section 3, the accessibility testing tool we chose can test applications that run on Android emulators or actual devices. Although the testing tool does not require the source code of the application, having access to deployable files of the applications under evaluation makes running experiments that demand several executions more feasible. Therefore, in the context of our experiment, we selected applications from a F-Droid,[3] which is a well-known repository that indexes more than 3000 Android apps of different categories, sizes and complexities. This repository has been used in several studies on testing mobile applications (Samir et al., 2019; Li et al., 2019; Jha et al., 2019; Mao et al., 2016; Li et al., 2014; Lamothe and Shang, 2018; Eler et al., 2018; Silva et al., 2018b).

We applied the following criteria during sample construction: *(i)* the application must also be available in the official app store of the Android platform, the Google Play Store, to make sure it is not a toy project; *(ii)* the target version of the Android API must be the 28th or a newer version since the testing tool was built based on the API 28. Only 323 out of 3,168 met the two selection criteria we set[4]. Next, we randomly selected 63 out of these 323 apps to create the sample for our study. Table 2 shows the apps that comprise our sample; it includes apps with one to up 90 activities.

5 RESULTS AND DISCUSSION

Activity Coverage. To measure activity coverage, we collected how many different activities were explored for each algorithm. Figure 2 shows that, when we evaluate activity coverage across all apps of our sample, the Uniform Random (UR) performed slightly worse than the other algorithms. However, there are no significant differences between all algorithms.

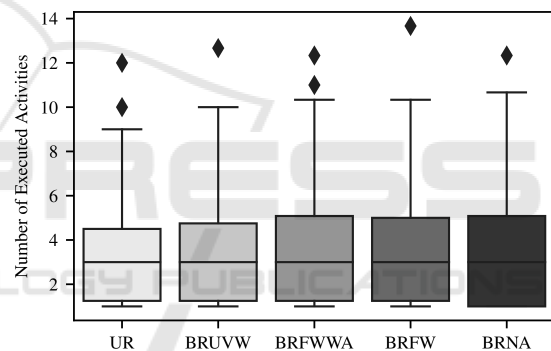

Figure 2: Number of executed activities for each algorithm across all apps in our sample.

Results seem to be different when we compare the performance of each algorithm on an app-by-app basis. Figure 3 shows for how many apps each algorithm performed best in terms of activity coverage. Since the number of activities in the apps is low, most algorithms covered about the same number of activities; hence, many algorithms were the best with respect to activity coverage for each app. Notice that the *BiasedRandomNewAction* performed best for 45 apps, followed by *BiasedRandomFixedWeight* and *BiasedRandomFixedWeightWA* which yield the best results (i.e., covered more activities) for 42 apps; and then the *UniformRandom* and *BiasedRandomUniformVariableWeight* performed best for 38 apps.

Traversed States. By looking at the current activity, the UI elements displayed, and their contents, we can

[3] https://f-droid.org/

[4] Data extracted from FDroid on August 27th of 2020.

Table 2: Mobile apps of our sample.

app	#act.s	app	#act.s
cityfreqs.com.pilfershushjammer	1	net.tjado.passwdsafe	7
com.dosse.speedtest	1	com.adguard.android.contentblocker	8
de.tutao.tutanota	1	com.james.status	8
fr.jnda.android.ipcalc	1	com.menny.android.anysoftkeyboard	8
net.guildem.publicip	1	com.orpheusdroid.screenrecorder	8
net.mullvad.mullvadvpn	1	nitezh.ministock	8
se.tube42.drum.android	1	net.eneiluj.moneybuster	9
androdns.android.leetdreams.ch.androdns	2	net.gsantner.markor	9
btools.routingapp	2	ai.susi	10
cf.fridays.fff_info	2	org.secuso.privacyfriendlytodolist	10
com.github.axet.hourlyreminder	2	de.rampro.activitydiary	11
io.github.easyintent.quickref	2	org.connectbot	11
me.tsukanov.counter	2	org.nutritionfacts.dailydozen	11
net.asceai.meritous	2	org.zephyrsoft.trackworktime	11
com.aa.mynotes	3	com.eventyay.organizer	12
com.atelieryl.wonderdroid	3	org.secuso.privacyfriendlyweather	12
com.github.axet.filemanager	3	com.yacgroup.yacguide.dev	13
com.physphil.android.unitconverterultimate	3	org.kiwix.kiwixmobile	13
eu.roggstar.luigithehunter.batterycalibrate	3	de.tadris.fitness	15
mn.tck.semitone	3	org.openintents.shopping	15
org.billthefarmer.diary	3	org.xbmc.kore	15
org.billthefarmer.notes	3	com.gpl.rpg.AndorsTrail	17
org.billthefarmer.tuner	3	de.danoeh.antennapod	17
org.jitsi.meet	3	au.com.wallaceit.reddinator	18
ru.meefik.busybox	3	de.syss.MifareClassicTool	19
com.darshancomputing.BatteryIndicator	4	org.quantumbadger.redreader	19
info.papdt.blackblub	4	io.github.hidroh.materialistic	23
it.rignanese.leo.slimfacebook	5	com.fsck.k9	29
org.billthefarmer.currency	5	io.pslab	32
uk.co.busydoingnothing.prevo	5	org.epstudios.epmobile	81
com.google.android.stardroid	7	me.ccrama.redditslide	90
name.boyle.chris.sgtpuzzles	7		

SCITEPRESS
SCIENCE AND TECHNOLOGY PUBLICATIONS

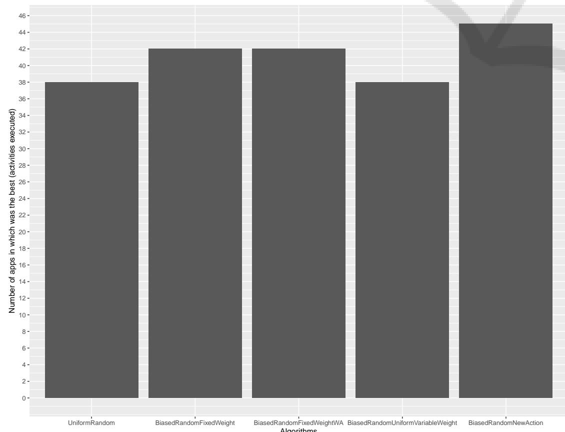

Figure 3: Number of apps for which an algorithm is the best w.r.t. activity coverage.

define a state of the app. In this part, we counted the number of unique app states traversed by each algorithm. Figure 4 shows that the number of different states reached by the *Biased Random - Fixed Weight Widget-Action* algorithm slightly covers more states, followed by *Biased Random - Fixed Weight* and *Biased Random - New Action*. Algorithms *Uniform Random* and *Biased Random - Uniform Variable Weight* are the least effective ones in terms of covering screen states. However, the differences between algorithms are not significant when we consider all apps in our sample.

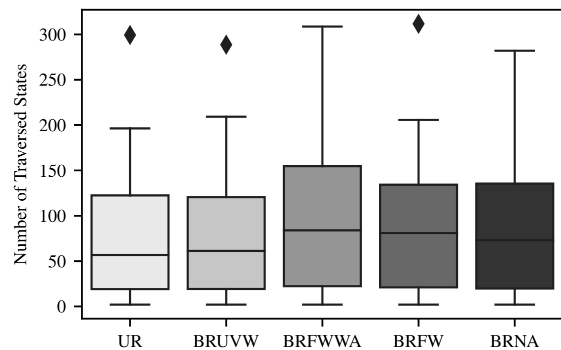

Figure 4: Number of traversed states for each algorithm across all apps in our sample.

When it comes to each app individually, however, the

number of apps for which each algorithm performed best with respect to screen state coverage is significantly different. Figure 5 shows that the algorithm *Biased Random - Fixed Weight Widget-Action* is clearly the one that traverses more states for 31 apps, followed by *Biased Random - Fixed Weight* with 19 apps, *Biased Random - Uniform Variable Weight* with 15 apps, *Biased Random - New Action* with 14 apps, and finally *Uniform Random* with 9 apps. Such results would seem to indicate that, all random algorithms we employed managed to traverse more states than the default implementation of MATE: the *Uniform Random* algorithm. In addition, algorithms that resort to weights to decide which user input should be generated next tend to traverse more states. It is noticeable that more than two algorithms were the best for the same app when it comes to the screen state coverage; in that case, all algorithms are counted as the best for that app.

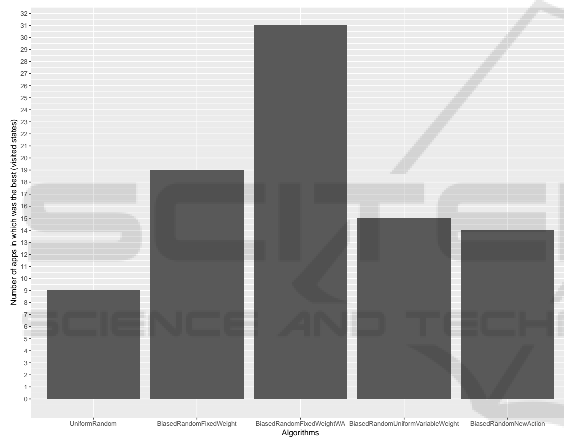

Figure 5: Number of apps for which an algorithm is the best w.r.t. screen state coverage.

The five algorithms were not able to explore many screen states for many apps due to two reasons mainly: many apps have only one activity and possibly few states to be explored; and for many apps the algorithms could not go through the first screen because they were not able to generate more complex user inputs or produce sensitive data (e.g. login and password). In such cases, the results achieved by the algorithms are not much different. Therefore, we analyzed the results of two subsets of the whole sample to check whether results would be different. First, we created a subset of our sample containing only apps for which the traversed states are equal or greater than the average number of states explored in the experiment (80). Then, we created a second subset of our sample containing only apps for which the traversed states are equal or greater than the average number of states explored in the experiment divided by two (40).

Table 3 shows the number of apps for which a particular algorithm is the best with respect to screen state coverage considering different samples. The *Whole Sample* is the original sample and it has 63 apps; the *#St>=average* has 26 apps; and *#St>=average/2* has 35 apps. Considering the Whole Sample, it is noticeable that more than two algorithms were the winners with respect to the number of screen state traversed during the test execution. In that case, more than one algorithm is defined as the best for that app; that is why the sum of the first second column is more than the number of apps in the sample. As presented in Figure 5, *Biased Random - Fixed Weight Widget-Action* is clearly the most effective algorithm as it is the best for half of the apps (49.21%), followed by *Biased Random - Fixed Weight* (30.16%), *Biased Random - Uniform Variable Weight* (23.81%), *Biased Random - New Action* (22.22%), and *Uniform Random* (14.29%).

Considering the subset of the sample for which the algorithms reached at least the average number of states explored by the whole sample (80); notice that there is only one best algorithm for each app, which may imply that differences between algorithms increase as the number of states explored increases. In this subset, the algorithm *Biased Random - Fixed Weight Widget-Action* is still the most effective as it is the best algorithm for 46.15% of the apps of this sample subset, followed by *Biased Random - New Action* (23.08%), *Biased Random - Fixed Weight* (15.38%), *Biased Random - Uniform Variable Weight* (11.54%), and *Uniform Random* (3.85%). When apps with more states traversed are considered, the percentage of the apps for which the algorithm *Biased Random - Fixed Weight Widget-Action* is the best remained about the same while it was reduced for the others, especially for the default algorithm *Uniform Random* which was the best for only one app this time.

Considering the subset of the sample for which the algorithms achieved at least the average number of states explored by the whole sample divided by two (40), notice that there is also only one best algorithm for each app. Similarly to the other sample subset, the algorithm *Biased Random - Fixed Weight Widget-Action* is also the most effective as it is the best algorithm for almost half of the apps (48.57%), followed by *Biased Random - New Action* (17.14%) and *Biased Random - Fixed Weight* (17.14%), *Biased Random - Uniform Variable Weight* (11.43%), and *Uniform Random* (5.71%).

Accessibility Violations. To probe into the violation detection effectiveness of each algorithms, we collected the absolute number of unique accessibility

Table 3: Number of apps for which an algorithm is the best w.r.t. screen coverage in different sample subsets.

	Whole Sample	Proportion	#St>=average	Proportion	#St>=average/2	Proportion
BRFW	19	30,16%	4	15,38%	6	17,14%
BRFWWA	31	49,21%	12	46,15%	17	48,57%
BRNA	14	22,22%	6	23,08%	6	17,14%
BRUVW	15	23,81%	3	11,54%	4	11,43%
UR	9	14,29%	1	3,85%	2	5,71%

violations revealed during the exploration of each algorithm. Figure 6 shows that there is no significant difference in the number of accessibility violations revealed when different flavors of random algorithms are applied to generate user inputs.

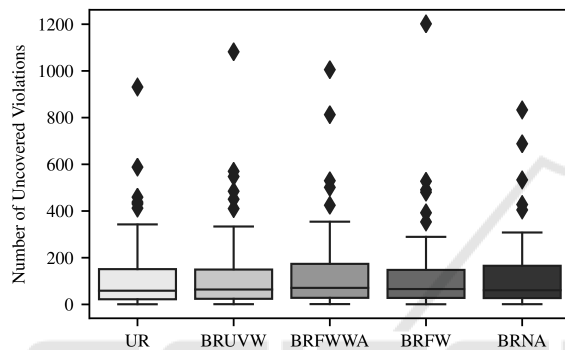

Figure 6: Number of violations uncovered by each algorithm when applied to all apps in our sample.

Nevertheless, the number of apps for which each algorithm performed the best with respect to accessibility violations found is different. Figure 7 shows that the algorithm *Biased Random - Fixed Weight Widget-Action* clearly generates user inputs that uncovered more accessibility violations for 27 apps, followed by *Biased Random - Fixed Weight* for 16 apps, *Uniform Random* for 12 apps, *Biased Random - New Action* for 9 apps, and *Biased Random - Uniform Variable Weight* for 7 apps. Notice that, even though the *Uniform Random* algorithm was the worst algorithm with respect to state coverage, it was the third in terms of uncovering accessibility violations. It means that the number of states explored may not correlate with uncovering more violations. We surmise that this is the case because uncovering violations depends on the diversity of the states explored.

Table 4 shows the number of apps for which a particular algorithm is the best with respect to the number of accessibility violations found considering different subsets of our sample: apps for which the number of states traversed is equal or greater than the average for the whole sample (80) and half of the average (40).

Considering the Whole Sample, notice that more

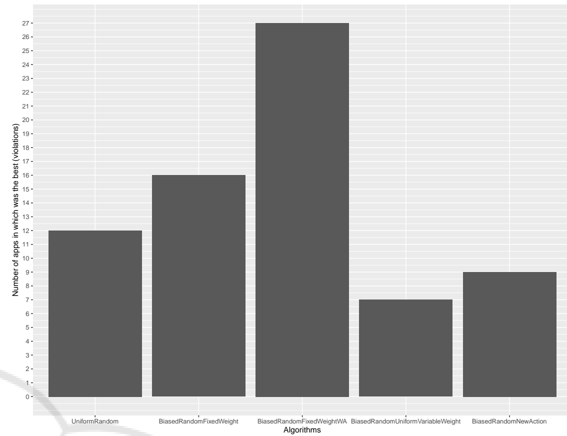

Figure 7: Number of apps for which an algorithm is the best w.r.t. accessibility violations found.

than two algorithms were the best for the same app with respect to the number of accessibility violations found. As presented in Figure 7, *Biased Random - Fixed Weight Widget-Action* is clearly the most effective algorithm as it is the most effective for 42.86% of the apps, followed by *Biased Random - Fixed Weight* (25.40%), *Uniform Random* (19.05%), *Biased Random - New Action* (14.19%) and *Biased Random - Uniform Variable Weight* (11.11%).

Considering the subset *#St>=average*, the algorithm *Biased Random - Fixed Weight Widget-Action* is still the most effective for 42.31% of the apps, followed by *Biased Random - New Action* (26.92%), *Biased Random - Fixed Weight* (15.38%), *Biased Random - Uniform Variable Weight* (11.54%), and *Uniform Random* (11.54%). When apps with more states traversed are considered, the percentage of the apps for which the algorithms *Biased Random - Fixed Weight Widget-Action* and *Biased Random - Uniform Variable Weight* are the best remained about the same while it increased for *Biased Random - New Action* and decreased for *Uniform Random* and *Biased Random - Fixed Weight*.

Considering the subset of the sample for which the algorithms achieved at least the average number of states explored by the whole sample divided by two (40), the algorithm *Biased Random - Fixed Weight Widget-Action* remains the most effective for 45.71% of the apps, followed by *Biased Random -*

Table 4: Number of apps for which an algorithm is the best w.r.t. accessibility violations found in different sample subsets.

	Whole Sample	Proportion	#St>=average	Proportion	#St>=average/2	Proportion
BRFW	16	25,40%	4	15,38%	8	22,86%
BRFWWA	27	42,86%	11	42,31%	16	45,71%
BRNA	9	14,29%	7	26,92%	7	20,00%
BRUVW	7	11,11%	3	11,54%	5	14,29%
UR	12	19,05%	3	11,54%	4	11,43%

Fixed Weight (22.86%), *Biased Random - New Action* (20%), *Biased Random - Uniform Variable Weight* (14.29%), and *Uniform Random* (11.43%).

The algorithm *Biased Random - Fixed Weight Widget-Action* seems to be the most effective considering both the number of states traversed and the number of accessibility violations found. Nevertheless, for most winning algorithms, the difference between the number of states explored and the accessibility violations found is not significantly different from the second best algorithm. Another noticeable fact regarding our results is that the *Uniform Random*, the default algorithm used to generate user inputs by the underlying tool we used in this experiment, has the worst performance in most cases.

6 CONCLUDING REMARKS

We probed into different strategies for random testing in the context of automated accessibility testing. To this end, we proposed four biased algorithms and implemented them on top of state of the art open source tool MATE (Eler et al., 2018). These algorithms were compared with the default strategy *Uniform Random*, taking into account the number of activities executed, screen states traversed, and accessibility violations revealed.

Comparing results for the our sample altogether, the differences between the results achieved by the five algorithms are negligible in most cases. However, when we look at the results on an app-by-app basis, the algorithm *Biased Random - Fixed Weight Widget-Action* is clearly the most effective at exploring more screen states and revealing more accessibility violations. In addition, the default strategy *Uniform Random* had the worst performance in most scenarios.

Such results evince that different flavors of random algorithms can be explored to achieve better results with respect to the automated accessibility testing of mobile apps. As future work, we intend to evaluate how adaptive random strategies, which systematically generate more diverse test inputs, perform against the best performing algorithm (i.e., *Biased Random - Fixed Weight Widget-Action*) according to our results.

ACKNOWLEDGMENTS

The authors would like to thank some Brazilian research agencies for their financial support. Andre Takeshi Endo is partially financially supported by the grant grant nr. of CNPq, and Marcelo Medeiros Eler if partially financially supported by the grant nr. 18/12287-6 of the São Paulo Research Foundation (FAPESP).

REFERENCES

Abascal, J., Arrue, M., and Valencia, X. (2019). *Tools for Web Accessibility Evaluation*, pages 479–503. Springer London, London.

Agüero-Flores, P., Quesada-López, C., Martínez, A., and Jenkins, M. (2019). Tools for the evaluation of web accessibility: A systematic literature mapping. In *2019 IV Jornadas Costarricenses de Investigación en Computación e Informática (JoCICI)*, pages 1–7.

Alshayban, A., Ahmed, I., and Malek, S. (2020). Accessibility issues in android apps: State of affairs, sentiments, and ways forward. In *Proceedings of the ACM/IEEE 42nd International Conference on Software Engineering*, ICSE '20, page 1323–1334, New York, NY, USA. ACM.

Antonelli, H. L., Sensiate, L., Watanabe, W. M., and de Mattos Fortes, R. P. (2019). Challenges of automatically evaluating rich internet applications accessibility. In *Proceedings of the 37th ACM International Conference on the Design of Communication*, SIGDOC '19, New York, NY, USA. ACM.

Baek, Y.-M. and Bae, D.-H. (2016). Automated model-based android GUI testing using multi-level GUI comparison criteria. pages 238–249. ACM.

Choudhary, S. R., Gorla, A., and Orso, A. (2015). Automated test input generation for android: Are we there yet? (e). In *2015 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, pages 429–440.

Dybå, T., Dingsøyr, T., and Moe, N. B. (2014). *Agile Project Management*, pages 277–300. Springer Berlin Heidelberg, Berlin, Heidelberg.

Eler, M. M., Rojas, J. M., Ge, Y., and Fraser, G. (2018). Automated accessibility testing of mobile apps. In *2018 IEEE 11th International Conference on Software Testing, Verification and Validation*, pages 116–126.

Frazão, T. and Duarte, C. (2020). Comparing accessibility evaluation plug-ins. In *Proceedings of the 17th International Web for All Conference*, W4A '20, New York, NY, USA. ACM.

Hao, S., Liu, B., Nath, S., Halfond, W. G., and Govindan, R. (2014). Puma: Programmable ui-automation for large-scale dynamic analysis of mobile apps. In *Proceedings of the 12th Annual International Conference on Mobile Systems, Applications, and Services*, MobiSys '14, page 204–217, New York, NY, USA. Association for Computing Machinery.

ISO (2018). Iso 9241: Ergonomics of human-system interaction – part 11: Usability: Definitions and concepts. Standard ISO 9241-11:2018, International Organization for Standardization, Geneva, CH.

Jha, A. K., Lee, S., and Lee, W. J. (2019). Characterizing android-specific crash bugs. In *2019 IEEE/ACM 6th International Conference on Mobile Software Engineering and Systems (MOBILESoft)*, pages 111–122.

Lamothe, M. and Shang, W. (2018). Exploring the use of automated api migrating techniques in practice: An experience report on android. In *Proceedings of the 15th International Conference on Mining Software Repositories*, MSR '18, page 503–514, New York, NY, USA. Association for Computing Machinery.

Li, W., Jiang, Y., Xu, C., Liu, Y., Ma, X., and Lü, J. (2019). Characterizing and detecting inefficient image displaying issues in android apps. In *2019 IEEE 26th International Conference on Software Analysis, Evolution and Reengineering (SANER)*, pages 355–365.

Li, X., Jiang, Y., Liu, Y., Xu, C., Ma, X., and Lu, J. (2014). User guided automation for testing mobile apps. In *2014 21st Asia-Pacific Software Engineering Conference*, volume 1, pages 27–34.

Machiry, A., Tahiliani, R., and Naik, M. (2013). Dynodroid: An input generation system for android apps. In *Proceedings of the 2013 9th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2013, pages 224–234, New York, NY, USA. ACM.

Mao, K., Harman, M., and Jia, Y. (2016). Sapienz: Multi-objective automated testing for android applications. In *Proceedings of the 25th International Symposium on Software Testing and Analysis*, ISSTA 2016, page 94–105, New York, NY, USA. ACM.

Mateus, D. A., Silva, C. A., Eler, M. M., and Freire, A. P. (2020). Accessibility of mobile applications: Evaluation by users with visual impairment and by automated tools. In *Proceedings of the 18th Brazilian Symposium on Human Factors in Computing Systems.*, IHC '20. ACM.

Pacheco, C. and Ernst, M. D. (2007). Randoop: Feedback-directed random testing for java. In *Companion to the 22nd ACM SIGPLAN Conference on Object-Oriented Programming Systems and Applications Companion*, OOPSLA '07, page 815–816, New York, NY, USA. Association for Computing Machinery.

Park, E., Han, S., Bae, H., Kim, R., Lee, S., Lim, D., and Lim, H. (2019). Development of automatic evaluation tool for mobile accessibility for android application. In *2019 International Conference on Systems of Collaboration Big Data, Internet of Things Security (SysCoBIoTS)*, pages 1–6.

Paz, F. and Pow-Sang, J. A. (2016). A systematic mapping review of usability evaluation methods for software development process. *International Journal of Software Engineering and Its Applications*, 10:165–178.

Samir, A., Maghawry, H. A., and Badr, N. (2019). Enhanced approach for maximizing coverage in automated mobile application testing. In *2019 Ninth International Conference on Intelligent Computing and Information Systems (ICICIS)*, pages 402–407.

Siebra, C. A., Correia, W., Penha, M., Macêdo, J., Quintino, J., Anjos, M., Florentin, F., Silva, F. Q. B., and Santos, A. L. M. (2018). An analysis on tools for accessibility evaluation in mobile applications. In *Proceedings of the XXXII Brazilian Symposium on Software Engineering*, SBES '18, page 172–177, New York, NY, USA. Association for Computing Machinery.

Silva, C., Eler, M. M., and Fraser, G. (2018a). A survey on the tool support for the automatic evaluation of mobile accessibility. In *Proceedings of the 8th International Conference on Software Development and Technologies for Enhancing Accessibility and Fighting Info-exclusion*, DSAI 2018, pages 286–293, New York, NY, USA. ACM.

Silva, D. B., Eler, M. M., Durelli, V. H., and Endo, A. T. (2018b). Characterizing mobile apps from a source and test code viewpoint. *Information and Software Technology*, 101:32 – 50.

Souza, N., Cardoso, E., and Perry, G. T. (2019). Limitations of automated accessibility evaluation in a mooc platform: Case study of a brazilian platform. *Revista Brasileira de Educacao Especial*, 25:603 – 616.

Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., and Su, Z. (2017). Guided, stochastic model-based gui testing of android apps. In *Proceedings of the 2017 11th Joint Meeting on Foundations of Software Engineering*, ESEC/FSE 2017, pages 245–256, New York, NY, USA. ACM.

Tramontana, P., Amalfitano, D., Amatucci, N., and Fasolino, A. R. (2019). Automated functional testing of mobile applications: a systematic mapping study. *Software Quality Journal*, 27(1):149–201.

Vigo, M., Brown, J., and Conway, V. (2013). Benchmarking web accessibility evaluation tools: Measuring the harm of booktitle = Proceedings of the 10th International Cross-Disciplinary Confere Sole Reliance on Automated Tests,nce on web accessibility. W4A '13, pages 1:1–1:10, New York, NY, USA. ACM.

Wohlin, C., Runeson, P., Höst, M., Ohlsson, M. C., Regnell, B., and Wesslén, A. (2012). *Experimentation in Software Engineering*. Springer.