An Empirical Evaluation of Search Algorithms for App Testing

Leon Sell, Michael Auer, Christoph Frädrich, Michael Gruber, Philemon Werli, Gordon Fraser

An Empirical Evaluation of Search Algorithms for App Testing

Leon Sell, Michael Auer, Christoph Frädrich,
Michael Gruber, Philemon Werli, and Gordon Fraser

University of Passau, Germany

Abstract. Automated testing techniques can effectively explore mobile applications in order to find faults that manifest as program crashes. A number of different techniques for automatically testing apps have been proposed and empirically compared, but previous studies focused on comparing different *tools*, rather than *techniques*. Although these studies have shown search-based approaches to be effective, it remains unclear whether superior performance of one tool compared to another is due to fundamental advantages of the underlying search technique, or due to certain engineering choices made during the implementation of the tools. In order to provide a better understanding of app testing as a search problem, we empirically study different search algorithms within the same app testing framework. Experiments on a selection of 10 non-trivial apps reveal that the costs of fitness evaluations are inhibitive, and prevent the choice of algorithm from having a major effect.

Keywords: Software Testing, Android, Genetic Algorithm

1 Introduction

Mobile applications (apps) have become an important branch of software engineering. To support the development and analysis of mobile applications, automated testing techniques have been tailored towards the specifics of mobile applications. In particular, automated testing techniques which interact with apps through the user interface are frequently applied to automatically find faults leading to program crashes.

Different techniques for automatically testing apps have been proposed and empirically compared. In particular, techniques based on random exploration and on meta-heuristic search algorithms have emerged as the most effective [15, 19]. However, previous empirical studies focused on comparing different *tools*, rather than *techniques*. Consequently, it remains unclear whether superior performance of one tool compared to another is due to fundamental advantages of the underlying technique, or due to certain engineering choices made during the implementation of the tools.

In order to better understand the factors influencing the effectiveness of app test generators, we empirically study the best-working search algorithms and

technical choices, according to previous research (e.g., [4, 19]), re-implemented within the same app testing framework. This increases internal validity, as it reduces the influence of engineering aspects on the outcome of the comparison.

Since previous empirical comparisons suggest that multi-objective search achieves the best performance [19], we aim to scrutinise this insight in particular. We implemented the multi-objective search algorithm NSGA-II as used in the Sapienz [12] tool, a simple mono-objective genetic algorithm (GA), the many-objective optimisation algorithms MOSA [13] and MIO [3], and random testing as well as heuristic improvements based on the Stoat [17] tool into the MATE [8] framework for app testing. Given this setup, we aim to answer the following research questions:

RQ1: What is the influence of using multi-objective optimisation, rather than the generally common approach of optimising simply for code coverage?
RQ2: Do recent many-objective search techniques that have been specifically designed for test generation outperform multi-objective optimisation?
RQ3: Does meta-heuristic search perform better than basic random testing?

Experiments on 10 non-trivial apps suggest that there is little difference between the individual search algorithms. The main problem is that test executions, and thus fitness evaluations, take so much time on Android that the search algorithms hardly get a chance to perform meaningful evolution.

2 Background

In order to understand testing of Android apps, we first explain how an Android app is composed. Android apps can be divided into the following components [6]: Activities (user interface), Services (background processes), Broadcast Receivers (inter-process communication), and Content Providers (sharing data between apps). Components are configured in the Android manifest, an XML file that maintains permissions of each Android app, e.g., internet access. Activities, services and broadcast receivers can be invoked by internal messages called *Intents*. While progress on testing services has been made (e.g., intent fuzzing [16]), most effort is put into testing the UI. Activities implement handling of user interaction with the app (e.g., logic behind a button click) and can assign callback functionality to its life-cycle methods. Activities navigate to other activities and potentially trigger services, broadcast receivers and content providers. Exploring activities most often leads to code being executed. Therefore, testing the UI in a more sophisticated way often results in higher code coverage.

A UI test case consists of a sequence of UI events (e.g., clicks). Different methods to automatically generate such test cases have been proposed and empirically compared. Choudhary et al. [15] compared different tools in terms of the effectiveness of the search strategy (e.g., random or systematic), the fault detection ability (how many crashes were found), the ease of use (how difficult is it to install and run the testing tool) and the Android framework compatibility (is the testing tool compatible with several Android API levels). The study

Algorithm 1: Random exploration using only available events

```
   Input  : Termination Condition C, length of the test cases n
   Output: test suite TS with test cases of length n
1  TS ⟵ {}
2  while ¬C do
3      test ⟵ []
4      for 0 to n do
5          A ⟵ getListOfAvailableEvents()
6          a ⟵ selectNextEvent(A)
7          execute(a)
8          test.push(a)
9      TS ⟵ TS ∪ {test}
10 return TS
```

looked at 68 different open-source apps and reported that the tools Monkey [7] and Dynodroid [10] (both using random exporation) outperformed the remaining tools on average. More recently, Wang et al. [19] examined 68 industrial apps with state-of-the-art test generation tools, motivated by the assumption that industrial apps have higher complexity. However, Monkey still obtained the highest coverage on most apps, closely followed by the tool Sapienz [12], which in contrast builds on an evolutionary algorithm for the test generation. Based on this insight, Zeng et al. [20] added some heuristics to random exploration, resulting in overall highest coverage. In the following, we take a closer look at these test generation techniques.

2.1 Random Exploration

To create a test case for an Android application, the application needs to be explored. The simplest way to do this is to randomly click anywhere on the screen and record these events. Random exploration is one of the most frequently applied strategies as it is implemented in the state-of-current-practice tool Monkey, which comes with Android [11]. However, test cases that were created by random clicks on the screen can contain events that do not trigger any code in the source code [2], for example when clicking on an area of the screen that does not contain any widgets. To improve the exploration, the set of events to choose from can be minimized to the set of available events on the current activity. Algorithm 1 describes random selection based on available events (Line 6).

Since random selection of available events may likely choose events that do not lead to exploration of previously unvisited activities, a heuristic can be used to improve the selection process. In particular, a heuristic reported as effective in the literature is the Stoat approach [17]. Each available event is assigned an execution weight, which is used to rank the events based on the potential increase of test coverage. The execution weight consists of three parts:

Type of the Event T_e. A navigation event (scroll and back) is assigned a value of 0.5, whereas menu events are assigned the value 2. All other events have the value 1.

Number of Unvisited Child Widgets C_e. An event that has more previously unvisited child widgets will be executed more likely than ones with fewer unvisited child widgets.

Number of Executions E_e. If an event has been executed previously during exploration, it should have a lower priority of being selected again compared to other events that have not been executed before, in order to increase the probability of discovering new activities.

Each of these parts is multiplied with a factor α, β, γ, resulting in an overall execution weight computed as follows:

$$execution\_weight_e = \frac{\alpha \cdot T_e + \beta \cdot C_e}{\gamma \cdot E_e} \tag{1}$$

The event with the highest execution weight is then selected and added to the test case (cf. Algorithm 1, Line 6). If two or more events have the same execution weight, one of those events is randomly selected and added to the test case.

2.2 Searching for Test Suites with Multi-Objective Search

Search-based testing describes the use of meta-heuristic search algorithms, such as GAs, to evolve tests. A common approach is to encode test suites as chromosomes of the GA, and then guide evolution with a fitness function based on code coverage [9]. In the context of Android testing, this approach has been popularised by the Sapienz tool [12]. In contrast to previous work on whole test suite generation, Sapienz optimises not only for code coverage, but also the number of crashes and event sequence length: Maximising the code coverage assures that most parts of the app are tested. Finding as many as possible crashes is the main goal of automated Android testing in general. Minimising the total number of events in a test suite helps with reducing the number of steps that need to be taken to reproduce a crash and keeps the amount of time needed to execute one chromosome at reasonable level.

To address multiple objectives at once, Sapienz uses the Non-dominated Sorting Genetic Algorithm II (NSGA-II), a popular multi-objective algorithm, shown in Algorithm 2. In each iteration, NSGA-II successively applies ranks to non-dominated individuals to build Pareto-optimal sets, starting at optimum 1 (Line 7). An individual x dominates another individual y if x is at least as good in all objectives and better than y in at least one objective. Using non-domination, individuals with a good score for only one objective have a higher chance of surviving. To sort individuals with the same rank, crowding distance is applied to all individuals (Line 10). At the end of each iteration, individuals are sorted by crowding distance (Line 13) and the population is reduced to its size limit (Line 14). Once terminated, the current population is returned.

Algorithm 2: NSGA-II algorithm

Input : Termination Condition C, population size limit n
Output: P_t, population of individuals

```
1  t ⟵ 0 // generation count
2  P_t ⟵ GenerateRandomPopulation(n)
3  P_t ⟵ Non-Dominated-Sort(P_t)
4  while ¬C do
5      R_t ⟵ P_t ∪ GenerateOffspring(P_t)
6      r ⟵ 1
7      {F_1, F_2, ...} ⟵ Non-Dominated-Sort(R_t)
8      P_{t+1} ⟵ {}
9      while |P_{t+1}| ≤ n do
10         AssignCrowdingDistance(F_r)
11         P_{t+1} ⟵ P_{t+1} ∪ F_r
12         r ⟵ r + 1
13     CrowdingDistanceSort(P_t + 1)
14     P_{t+1} ⟵ P_{t+1}[1 : n]
15     t ⟵ t + 1
16 return P_t
```

2.3 Searching for Test Cases with Many-Objective Optimisation

Traditionally, search-based test generation used individual test cases as representation, and there are several approaches for Android testing that also use such a representation [1, 11, 14]. While the whole test suite generation approach has superseded the optimisation of individual goals in many scenarios, recently new specifically tailored many-objective optimisation algorithms have popularised test case representation again, in particular the Many-Objective Sorting Algorithm (MOSA) [13] and the Many Independent Objective Algorithm (MIO) [3]. Similar to the Sapienz approach, MOSA and MIO are population based GAs. Each coverage goal (e.g., individual branch) is viewed as an independent objective by both MOSA and MIO. An objective is represented by a corresponding fitness function, which allows us to evaluate whether an individual reaches an objective, i.e., whether a test case covers a certain branch or statement.

MOSA: MOSA [13] is an extension of NSGA-II [5] and optimises test cases as individuals for not one, but multiple independent objectives (e.g., individual branches). It extends the ranking system of NSGA-II and adds a second population, called *archive*. Algorithm 3 illustrates how MOSA works. Extensions to NSGA-II are marked with a grey background.

The archive contains the best individual for each fulfilled objective. Individuals which fulfill an objective first are added to the archive; these objectives are ignored for future optimisation. If, however, an individual fulfills an already

Algorithm 3: MOSA algorithm

Input : Termination Condition C, population size limit n
Output: *archive*, a set of optimised test cases

```
1  t ⟵ 0 // generation count
2  P_t ⟵ GenerateRandomPopulation(n)
3  archive ⟵ UpdateArchive(P_t)
4  while ¬C do
5      R_t ⟵ P_t ∪ GenerateOffspring(P_t)
6      r ⟵ 0
7      {F_0, F_1, ...} ⟵ PreferenceSorting(R_t)
8      P_{t+1} ⟵ {}
9      while |P_{t+1}| + |F_r| ≤ n do
10         AssignCrowdingDistance(F_r)
11         P_{t+1} ⟵ P_{t+1} ∪ F_r
12         r ⟵ r + 1
13     CrowdingDistanceSort(F_r)
14     P_{t+1} ⟵ P_{t+1} ∪ F_r[1 : (n − |P_{t+1}|)]
15     archive ⟵ UpdateArchive(P_{t+1})
16     t ⟵ t + 1
17 return archive
```

satisfied objective, but with a shorter sequence length than the individual in the archive, it replaces the existing in the archive (Line 15).

NSGA-II selects individuals based on ranks, which start from an optimum of 1. MOSA extends the ranking system with a *preference sorting criterion* to avoid losing the best individual for each non-fulfilled objective, i.e., closest to fulfilling (Line 7). The best individuals for non-fulfilled objectives are given the new optimal rank 0, independent of their relation to other individuals. For all other individuals, NSGA-II ranking is applied.

At the end, MOSA returns the archive, which contains the individuals for all satisfied objectives (Line 17).

MIO: The Many Independent Objective Algorithm [3] is based on the (1+1) EA and uses archives with a dynamic population. For each testing target one archive exists. The best n tests are kept in the archive, where n is the maximum size of the archives. If a target is reached, then the maximum size of the corresponding archive is set to 1 and all but the covering test are removed from the archive.

To better control the trade-off between exploitation and exploration of the search landscape MIO uses a parameter P_r, according to which new tests are sampled or old ones mutated (Line 7). A second parameter F defines, when the focused search phase should start: Similar to Simulated Annealing the amount of exploration is reduced over time when the focused search starts (Line 20).

Algorithm 4: Many Independent Objective Algorithm

```
   Input  : Termination Condition C, Random sampling probability P_r, List of
            fitness functions L, Population size limit n, Start of focused search F
   Output: Archive of optimised individuals
 1 T ⟵ createInitialPopulation ()
 2 A ⟵ {}
 3 while ¬C do
 4     if rand() < P_r then
 5         p ⟵ createRandomIndividual ()
 6     else
 7         p ⟵ sampleIndividual (T)
 8         p ⟵ mutate (p)
 9     for k ⟵ L do
10         fitness = k.getFitness (p)
11         if target is covered then
12             addToArchive(A, p)
13             T_k ⟵ {p}
14             T ⟵ T \ {T_k}
15         else if target is partially covered then
16             T ⟵ T_k ∪ {p}
17             resetCounter(k)
18             if |T_k| > n then
19                 removeWorstTest(T_k, k)
20         updateParameters(F, P_r, n)
21 return A
```

An additional feature of MIO is Feedback-Directed Sampling. Simply put, it is an approach to prefer targets for which we continuously see improvements. To achieve this, each target has a counter c_k. It is increased every time we sample a test from the targets (Line 7) archive and is reset to 0 when the new test performs better than the ones already in the archive (Line 17). The algorithm always samples tests from the archive of the target with the lowest counter.

3 Experimental Setup

3.1 Study Subjects

For experiments we used ten different Android apps which we obtained from the F-Droid platform[1]. A main criterion for the apps was to allow a comparison of search algorithms without being inhibited by the technical limitations of the underlying Android testing framework. Therefore, the ten open-source apps (the code was necessary for the instrumentation) were primarily selected in a random fashion with the following restrictions:

[1] `https://f-droid.org/en/`

Table 1: Randomly selected apps.

App name (abbreviation)	Version	#Activities	#LoC
com.oriondev.moneywallet **(moneywallet)**	4.0.4.1	35	23081
com.woefe.shoppinglist **(shoppinglist)**	0.11.0	4	1242
de.arnowelzel.android.periodical **(periodical)**	1.22	7	1558
de.rampro.activitydiary **(activitydiary)**	1.4.0	11	3652
de.drhoffmannsoftware **(drhoffmansoftware)**	1.16-9	9	896
de.retujo.bierverkostung **(bierverkostung)**	1.2.1	13	7297
de.tap.easy_xkcd **(easy_xkcd)**	6.1.2	9	4972
net.gsantner.markor **(markor)**	1.6.0	7	5691
org.quantumbadger.redreader **(redreader)**	1.9.9	19	16019
protect.rentalcalc **(rentalcalc)**	0.5.1	12	1770

- ▷ The latest available version of each app has been used to avoid finding bugs that have already been fixed.
- ▷ Apps that require certain permissions, e.g., camera or GPS, for their core functionality were discarded as the MATE framework [8] cannot provide meaningful real world data for those sensors which could prevent a comprehensive exploration of the app.
- ▷ Apps that require authentication as a basic prerequisite were discarded as this too will prevent a comprehensive exploration of the app.
- ▷ Apps with less than four activities were discarded as those apps are often too small in scope to generate meaningful data during testing.
- ▷ Apps that do not primarily use native Android UI elements such as menus, text fields, buttons and sliders were discarded since we use native Android UI elements to deduce actions that can be performed on the current screen.

If any of those criteria do not apply for the selected app, a new random app was selected. This process was repeated until we found ten apps that fulfil all the criteria. The resulting apps are listed in Table 1.

3.2 Experimental Infrastructure

We implemented the different approaches described in Section 2 on top of the MATE framework [8]. MATE was previously used to identify accessibility flaws in apps by exploring the app using various heuristics. The framework was extended to support different search algorithms and to measure code coverage.

For the search algorithms random exploration is used as a basis for creating the initial population in order to be able to compare it to solely using the heuristic exploration introduced in Section 2.1. Unlike Android Monkey, random exploration chooses only from events that are available on the current activity. To ensure that there is no influence between individual test cases, for each test execution the app is re-installed and re-initialized to provide a consistent and clean starting state. This is also done by other test generation tools like Sapienz [12].

As proposed in the Sapienz paper [12], random selection was used for NSGA-II and also MOSA, to choose the chromosomes which are crossed over and mutated. MIO selects chromosomes according to Feedback-Directed Sampling. Simple GA uses a fitness proportionate selection. For the Sapienz reimplementation the crossover and mutation function were matched as closely as possible. For the other GAs (which all operate on test cases instead of test suites) a crossover function was implemented that tries to match the screen state after an event of the first parent with the screen state before an event of the second parent. The offspring is an event sequence that starts with the events leading up to the matching point of the first parent after which it is continued with the events succeeding the match in the second parent. The search is started from a random element of the event sequence of the first parent with a bias towards the middle and is continued in an alternating fashion until a matching screen state is found. While executing the resulting new test case if an event cannot be execute because a new screen state was reached, random events will be selected from that point onward until the original event sequence length has been reached. For the mutation function a random element of the event sequence is chosen after which new events will be generated that will be executed instead of the old events. Parameters of the GAs were chosen based on the values in the respective papers that introduced the search algorithms:

Population size	50	[12]
pMutate	0.3	[12]
pCrossover	0.7	[12]
Start of focused Search	0.5	[3]
pSampleRandom	0.5	[3]
Maximal number of events (per test case)	50	[12]
Number of test cases per test suite	5	[12]

All experiments were carried out on the Android Emulator version 28.0.22.0, using an x86 image of Android 7.1.1. For each run, the emulator had full access to 4 physical cores (8 hyper-threads) and 16 GB of RAM. Physically, we used Intel(R) Xeon(R) E5-2650 v2 @ 2.60 GHz CPUs. Each approach was executed ten times on each of the ten different study subjects for 8 hours per execution. We instrumented the apps using JaCoCo[2]. Measurement of line coverage and logging of occurred crashes were performed during the entire run. After a run has finished we calculate the unique number of crashes of the run by comparing the entire stacktrace of each crash. Two crashes are considered the same unique crash if the strings of both of the crashes' stacktraces are identical.

3.3 Experiment Procedure

For each research question we use the Vargha-Delaney $\hat{A}_{1,2}$ effect size and the Wilcoxon-Mann-Whitney U test, with a 95% confidence level, to statistically compare the performance of the algorithms. If the effect size is calculated for

[2] `https://www.eclemma.org/jacoco/`

a comparsion of two algorithms X and Y, $\hat{A}_{X,Y} = 0.5$ indicates that both approaches achieve the same coverage or crashes, and $\hat{A}_{X,Y} = 0.0$ means that algorithm X has a 100% probability of achieving lower coverage or crashes.
RQ1: In order to answer RQ1, we compare the NSGA-II algorithm to a simple GA that solely optimises test suites for coverage. The simple GA uses the same representation, mutation, and crossover as NSGA-II, but only optimises for line coverage, whereas the NSGA-II algorithm additionally uses (1) a fitness function minimizing the number of events executed in the test suite and (2) a fitness function maximising the number of crashes produced by the test suite, both also used in the Sapienz approach. Based on the NSGA-II variant used in Sapienz, NSGA-II selects individuals for reproduction randomly; in contrast, the simple GA uses standard fitness proportionate selection. We compare the number of unique crashes found and the coverage achieved to determine if the multi-objective approach yields a benefit over a simple mono objective approach.
RQ2: In order to answer RQ2, we compare NSGA-II to the many-objective optimisation algorithms MOSA and MIO in terms of line coverage and unique crashes. For many-objective optimisation, each line of the app under test becomes an independent testing target to optimise. MOSA and MIO use the same fitness function for each line, which yields results between 0.0 and 1.0 (covered). An executed line in the same package as the target line adds 0.25 and same class as the target line adds 0.25. The remaining possible 0.5 are calculated from the distance inside the class to the target line. Since our line coverage fitness function maximises towards 1.0 and MOSA usually minimises its fitness functions, we adjust our implementation of MOSA to maximise towards 1.0 as well.
RQ3: In order to answer RQ3, we implemented a random exploration that only takes available events into account, and creates test cases of a fixed length. For our studies we set the length of a single test case to 50 events, which matches the length of individuals in the search algorithms. This strategy is then compared in terms of code coverage achieved and unique crashes found to the heuristic strategy based on Stoat (Section 2.1), and NSGA-II. The values for the weights we choose for the heuristic approach are the same ones that the researchers of Stoat [17] have used. These values are 1 for the value of α, the weight for the type of event, 1.4 for the value of β, the weight for the number of unvisited children widgets, and 1.5 for γ, the weight of the number of executions of the event. Since we do not know the number of unvisited children widgets for events that have not been executed before during exploration, we set this value twice as high as the highest number of unvisited children widgets from all known events.

3.4 Threats to validity

To reduce threats to *internal validity* caused by the randomised algorithms, we ran each experiment 10 times and applied statistical analysis. To avoid possible confounding factors when comparing algorithms, they were all implemented in the same tool. We used the same default values for all relevant parameters, and used recommended values for algorithm-specific ones. Threats to *external validity* result from our limited choice of algorithms and study subjects. However,

Table 2: Mean Line Coverage of the Different Algorithms

App	NSGA-II	Random	Heuristic	Simple GA	MOSA	MIO
moneywallet	0.20	0.21	0.35	0.22	0.22	0.19
shoppinglist	0.77	0.80	0.73	0.77	0.77	0.77
periodical	0.85	0.87	0.87	0.84	0.83	0.85
drhoffmannsoftware	0.44	0.48	0.43	0.44	0.46	0.48
activitydiary	0.74	0.75	0.78	0.73	0.74	0.75
bierverkostung	0.43	0.47	0.58	0.44	0.45	0.49
easy_xkcd	0.62	0.62	0.61	0.57	0.59	0.60
markor	0.62	0.63	0.58	0.61	0.60	0.60
redreader	0.55	0.55	0.45	0.52	0.54	0.52
rentalcalc	0.67	0.69	0.70	0.68	0.68	0.71
Mean	0.59	0.61	0.61	0.58	0.59	0.60

Table 3: Mean Unique Crashes of NSGA-II, Random, Heuristic, Simple GA, MOSA & MIO

App	NSGA-II	Random	Heuristic	Simple GA	MOSA	MIO
moneywallet	0.37	0.17	0.36	0.20	0.30	0.20
shoppinglist	0.00	0.00	0.00	0.00	0.00	0.00
periodical	0.00	0.00	0.00	0.00	0.00	0.00
drhoffmannsoftware	11.50	12.36	11.45	8.90	10.40	11.00
activitydiary	0.00	0.08	0.00	0.00	0.00	0.00
bierverkostung	3.30	4.09	8.00	4.00	3.60	3.80
easy_xkcd	3.30	3.38	0.64	2.80	3.40	2.80
markor	0.40	0.38	0.00	0.20	0.10	0.30
redreader	0.10	0.00	0.00	0.00	0.00	0.10
rentalcalc	1.40	2.17	4.18	1.90	1.80	2.90
Mean	2.04	2.26	2.46	1.80	1.96	2.11

we used the best algorithms based on previous empirical studies. The 10 open source apps were chosen as representatives for a wide range of functionality, but results may not generalise to other apps.

4 Results

In this section, we investigate each proposed research question with results from our experiments. Tables 2 and 3 summarise the results of our experiments in terms of the line coverage achieved and number of unique crashes found for each of the algorithms. Statistical comparisons are summarised in Tables 4 and 5.

Table 4: Comparison of Mean Coverage of NSGA-II vs. Random, Heuristic, Simple GA, MOSA & MIO

NSGA-II	vs. Random		vs. Heuristic		vs. Simple GA		vs. MOSA		vs. MIO	
App	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$
moneywallet	0.13	0.26	<0.01	0.00	0.23	0.31	0.27	0.32	0.71	0.56
shoppinglist	0.16	0.30	0.02	0.81	0.79	0.46	0.79	0.46	0.38	0.38
periodical	<0.01	0.10	0.14	0.30	0.41	0.61	0.16	0.69	0.91	0.48
drhoffmannsoftware	0.03	0.20	0.15	0.69	1.00	0.49	0.34	0.36	0.01	0.15
activitydiary	0.36	0.37	<0.01	0.09	0.88	0.52	0.31	0.64	0.60	0.42
bierverkostung	0.01	0.15	<0.01	0.00	0.57	0.58	0.43	0.39	<0.01	0.12
easy_xkcd	0.96	0.51	0.41	0.61	<0.01	0.94	0.16	0.69	0.12	0.71
markor	0.92	0.52	<0.01	1.00	0.08	0.74	0.01	0.84	<0.01	0.89
redreader	0.31	0.64	<0.01	1.00	<0.01	0.90	0.52	0.59	<0.01	0.90
rentalcalc	0.43	0.39	0.21	0.33	1.00	0.50	0.82	0.46	0.05	0.23
Mean	0.33	0.34	0.09	0.48	0.50	0.61	0.38	0.54	0.28	0.48

4.1 RQ1: Mono-objective vs. multi-objective optimisation

In order to determine if multi-objective optimisation is an improvement over mono-objective GAs optimising for code coverage, we compare the coverage achieved by both approaches. As shown in Tables 2 and 4, NSGA-II and the simple GA perform similar in terms of overall coverage with NSGA-II having a slight edge over the simple GA. NSGA-II achieves better coverage for the apps `easy_xkcd` and `redreader`, where the results are well within the margin of significance, with a mean difference in overall coverage of 5% for `easy_xkcd` and 3% for `redreader`. For all other apps neither algorithm significantly outperforms the other. Overall, NSGA-II achieves about 1% more line coverage on average.

In terms of mean unique crashes per app, which are shown in Tables 3 and 5, NSGA-II is also doing slightly better. There is only one statistically significant difference in mean crashes, i.e., for the `markor` app where NSGA-II discovers about 0.2 crashes more than the simple GA on average. Overall NSGA-II discovers about 0.24 crashes more than the simple GA.

Our conjecture is that NSGA-II triggers a few more unique crashes than the simple GA because it keeps test suites containing crashes in its population. Intuitively, because of defect clustering crashes may often occur in proximity to other crashes. This could explain why NSGA-II discovers more unique crashes by mutating and crossing test suites that already contain crashes.

This could also explain the slight edge in coverage NSGA-II has over the simple GA: More unique crashes found might mean covering more catch-blocks, whereas the "easy" parts of the code will be covered similarly by both approaches.

4.2 RQ2: Multi-objective vs. many-objective optimisation

To test how multi-objective optimisation compares to many-objective optimisation we compare NSGA-II with MOSA and also with MIO.

Table 5: Comparison of Mean Unique Crashes of NSGA-II vs. Random, Heuristic, Simple GA, MOSA & MIO

NSGA-II	vs. Random		vs. Heuristic		vs. Simple GA		vs. MOSA		vs. MIO	
App	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$	**P-Value**	$\hat{\mathbf{A}}_{\mathbf{1,2}}$
moneywallet	0.64	0.55	0.94	0.50	0.64	0.55	0.94	0.50	0.94	0.51
shoppinglist	-	-	-	-	-	-	-	-	-	-
periodical	-	-	-	-	-	-	-	-	-	-
drhoffmannsoftware	0.25	0.35	0.57	0.57	<0.01	0.91	0.04	0.77	0.17	0.68
activitydiary	0.37	0.45	-	-	-	-	-	-	-	-
bierverkostung	0.29	0.36	<0.01	0.00	0.41	0.39	0.94	0.48	0.53	0.41
easy_xkcd	0.29	0.35	<0.01	1.00	0.45	0.60	0.88	0.47	0.45	0.60
markor	0.67	0.56	0.03	0.70	0.37	0.60	0.14	0.65	0.68	0.55
redreader	0.37	0.55	0.37	0.55	0.37	0.55	0.37	0.55	0.94	0.50
rentalcalc	0.50	0.41	<0.01	0.04	0.31	0.36	0.67	0.44	0.10	0.28
Mean	0.42	0.44	0.27	0.48	0.36	0.57	0.57	0.55	0.54	0.50

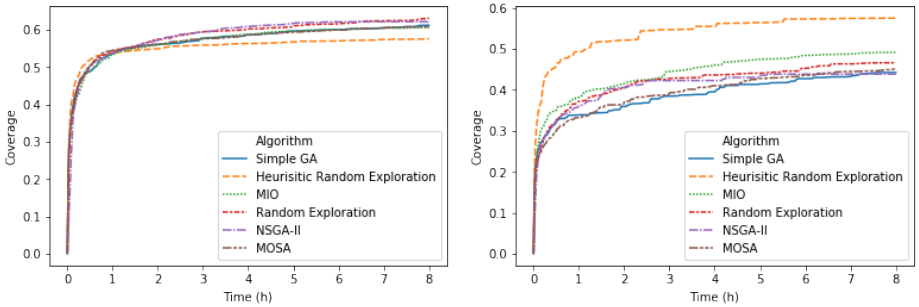
In terms of coverage, Table 2 indicates an insignificant advantage of NSGA-II compared to MOSA. The comparison in Table 4 shows one significant difference for the `markor` app. In this case, NSGA-II outperforms MOSA with an effect size of 0.84 as NSGA-II achieves 62% coverage vs. the 60% achieved by MOSA. In terms of unique crashes, we observe a similar behaviour. As shown in Table 3, NSGA-II has a minor advantage over MOSA. For the only significant result, the `drhoffmannsoftware` app, NSGA-II covers 11.5 unique crashes while MOSA only covers 10.4 as depicted in Table 5.

Tables 2 and 3 show that NSGA-II and MIO have the same mean code coverage and number of unique crashes. When comparing code coverage, NSGA-II and MIO perform significantly better than the other for two apps as shown in Table 4. In comparison, NSGA-II achieves significant results for `markor` and `redreader`, while MIO does so for `drhoffmannsoftware` and `bierverkostung`. For unique crashes, no results are close to statistical significance (Table 5).

Our results indicate that neither multi-objective nor many-objective optimisation have a clear advantage over the other. We conjecture that the search space contains many states that are equally easy to reach for both types of optimisations, while very specific sequences of actions or inputs are needed to reach the remaining states. Figure 1a illustrates how coverage evolves over time: Even after eight hours the search has not converged. Consequently, the search algorithms are likely to require much more time to cover more difficult states.

4.3 RQ3: Random vs. search-based testing

To answer RQ3, we compare the random and the heuristic approach to NSGA-II. Tables 2 and 3 suggest that, on average, NSGA-II achieves slightly less coverage and crashes. Table 4 shows that NSGA-II achieved significantly lower coverage than random exploration in three cases, and for heuristic exploration it is



(a) markor (b) bierverkostung

Fig. 1: Cumulative coverage over time for the tested apps. Values averaged every full minute using linear interpolation of the nearest respective previous and subsequent data point.

significantly worse in three cases but significantly better in three other cases. Table 5 shows that in terms of crashes, there are no significant differences between NSGA-II and random, and compared to heuristic exploration NSGA-II is significantly better in two and significantly worse in two other cases. Overall, differences are small and mostly insignificant, although averages are slightly in favour of random and heuristic exploration.

This is also reflected by Figure 1a and Figure 1b which illustrate that, depending on whether it is better in terms of coverage to interact with unexplored widgets or to interact with the same widgets multiple times, the heuristic exploration performs a little better or a little worse. Random exploration, on the other hand, very closely matches the search-based approaches.

Overall, our results suggest that mono-objective and multi-objective optimisations do not achieve better results than random or heuristic exploration, and on average even appear to be slightly worse.

4.4 Discussion

Our experiments suggest that the actual search algorithm used only has a minor impact on the results. A closer look at the number of generations executed suggests that the search-based approaches simply do not receive sufficient time for meaningful evolution to take place. For example, NSGA-II on average over all apps achieved only 12.3 generations per run. This means that a substantial part of the search budget is used simply for evaluating the initial population, which means that the search behaves identical to random exploration for that time. For successive generations, the search mainly makes small changes through mutations and crossover, and the low number of generations leads to only small coverage increases over the initial population. Random exploration, in contrast, in the same time executes independently generated random tests, which likely explore more diverse aspects of the app under test, thus leading to the slight

improvement in terms of coverage and crashes. To better understand what causes the high test execution costs, we took a closer look at the main cost factors:

- MATE uses a delay of 0.5s in between actions to conservatively allow the UI to react to user events. However, the UIAutomator, which is used by Android testing tools to send user events, has an additional overhead, such that on average, each action takes 1.5s.
- Code coverage information is stored in a local file on the emulated device, and this file needs to be retrieved after each test execution. This action takes another 0.7 - 2.5s.
- In between test executions, the app is reset to a clean state. This is necessary to avoid dependencies between tests, and thus flaky tests. On average, it takes 8s to reset an app.

When optimising test suites using NSGA-II, we used a population size of 50 test suites, each with 5 test cases and a maximum of 50 events per test case (i.e., a maximum of 12.500 events per generation of the search). Thus, only considering the values listed above, evaluating a single generation can take up to $50 \times 5 \times 1.5s + 50 \times 5 \times 8s + 50 \times 50 \times 5 \times 1.5s = 21125s = 5.87h$. However, test execution results are cached, and thus tests are only executed if they are mutated, which only happens with a certain probability. In practice, we observed that evaluating the initial generation takes around 3.6h, and successive generations take about 38 minutes. The search algorithms using test cases rather than test suites potentially require fewer test executions, since each individual in the search is a test case rather than a whole test suite. However, the different representation means that chances of the tests being mutated are higher (when optimising test suites, on average only one test in the test suite is mutated if an individual is mutated). In our experiments, evaluating the initial population of test case based algorithms took only 46 minutes, but then evaluating successive generations took around 76 minutes, leading to an overall comparable number of generations to NSGA-II. Note that these execution costs are not specific to MATE but apply to other tools as well; for example, Vogel et al. [18] report execution times of up to 5 hours to run Sapienz for 10 generations.

5 Conclusions

Search-based testing is one of the most popular approaches for testing mobile apps, and has been popularised by the Sapienz tool. In this paper we aimed to better understand how search algorithms influence the effectiveness of search-based testing for Android apps. In contrast to prior studies, we implemented different search algorithms within the same framework, such that our comparisons are not skewed by the engineering of the underlying tool. Our experiments suggest that the high costs of executing tests on Android devices or emulators makes fitness evaluations so expensive that the search algorithms can run only for few iterations, and hardly get a chance to perform meaningful evolution. As a result, random exploration performs slightly better than the search algorithms.

This finding is also in line with previous, tool-based comparisons, in which random exploration tools like Android Monkey performed well. As future work, we therefore plan to investigate ways of reducing the execution costs, to allow search algorithms to perform better, and we plan to investigate other technical choices such as the influence of choosing widgets rather than random positions.

Acknowledgements

This work is supported by EPSRC project EP/N023978/2, Erasmus+ project IMPRESS 2017-1-NL01-KA203-035259 and DFG grant FR 2955/2-1.

References

1. Amalfitano, D., Amatucci, N., Fasolino, A.R., Tramontana, P.: Agrippin: a novel search based testing technique for android applications. In: Proceedings of the 3rd International Workshop on Software Development Lifecycle for Mobile. pp. 5–12. ACM (2015)
2. Amalfitano, D., Fasolino, A.R., Tramontana, P., De Carmine, S., Memon, A.M.: Using GUI ripping for automated testing of Android applications. In: Proceedings of the 27th IEEE/ACM International Conference on Automated Software Engineering. pp. 258–261 (2012)
3. Arcuri, A.: Test suite generation with the many independent objective (mio) algorithm. Information and Software Technology **104**, 195–206 (2018)
4. Campos, J., Ge, Y., Fraser, G., Eler, M., Arcuri, A.: An empirical evaluation of evolutionary algorithms for test suite generation. In: Search Based Software Engineering. pp. 33–48 (2017)
5. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: Nsga-ii. In: Parallel Problem Solving from Nature PPSN VI. pp. 849–858 (2000)
6. Developers, A.: Application fundamentals (October 2018), `https://developer.android.com/guide/components/fundamentals`
7. Developers, A.: Ui/application exerciser monkey (September 2018), `https://developer.android.com/studio/test/monkey`
8. Eler, M.M., Rojas, J.M., Ge, Y., Fraser, G.: Automated accessibility testing of mobile apps. In: 2018 IEEE 11th International Conference on Software Testing, Verification and Validation (ICST). pp. 116–126 (2018)
9. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering **39**(2), 276–291 (2013)
10. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: An input generation system for android apps. In: Proceedings of the 9th Joint Meeting on Foundations of Software Engineering. pp. 224–234. ESEC/FSE, ACM (2013)
11. Mahmood, R., Mirzaei, N., Malek, S.: Evodroid: Segmented evolutionary testing of android apps. In: Proceedings of the 22nd ACM SIGSOFT International Symposium on Foundations of Software Engineering. pp. 599–609 (2014)
12. Mao, K., Harman, M., Jia, Y.: Sapienz: Multi-objective automated testing for android applications. In: Proceedings of the 25th International Symposium on Software Testing and Analysis. pp. 94–105. ISSTA, ACM (2016)

13. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on. pp. 1–10 (2015)
14. Rohella, A., Takada, S.: Testing android applications using multi-objective evolutionary algorithms with a stopping criteria. In: 30th International Conference on Software Engineering and Knowledge Engineering, SEKE 2018. pp. 308–313. Knowledge Systems Institute Graduate School (2018)
15. Roy Choudhary, S., Gorla, A., Orso, A.: Automated test input generation for android: Are we there yet? (e). In: Proceedings of the IEEE/ACM International Conference on Automated Software Engineering (ASE). pp. 429–440 (2015)
16. Sasnauskas, R., Regehr, J.: Intent fuzzer: Crafting intents of death. In: Proceedings of Joint International Workshop on Dynamic Analysis (WODA) and Software and System Performance Testing, Debugging, and Analytics (PERTEA). pp. 1–5 (2014)
17. Su, T., Meng, G., Chen, Y., Wu, K., Yang, W., Yao, Y., Pu, G., Liu, Y., Su, Z.: Guided, stochastic model-based gui testing of android apps. In: Proceedings of the 11th Joint Meeting on Foundations of Software Engineering. pp. 245–256 (2017)
18. Vogel, T., Tran, C., Grunske, L.: Does diversity improve the test suite generation for mobile applications? In: Proceedings of the 11th Symposium on Search-Based Software Engineering (SSBSE 2019). Springer, to appear
19. Wang, W., Li, D., Yang, W., Cao, Y., Zhang, Z., Deng, Y., Xie, T.: An empirical study of Android test generation tools in industrial cases. In: Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering. pp. 738–748 (2018)
20. Zeng, X., Li, D., Zheng, W., Xia, F., Deng, Y., Lam, W., Yang, W., Xie, T.: Automated test input generation for android: are we really there yet in an industrial case? In: SIGSOFT FSE (2016)