Algorithm or Representation? An empirical study on how SAPIENZ achieves coverage

Iván Arcuschin Moreno
FCEyN-UBA/ICC-CONICET
Argentina
iarcuschin@dc.uba.ar

Juan Pablo Galeotti
FCEyN-UBA/ICC-CONICET
Argentina
jgaleotti@dc.uba.ar

Diego Garbervetsky
FCEyN-UBA/ICC-CONICET
Argentina
diegog@dc.uba.ar

ABSTRACT

Testing is a very important and expensive part of developing Android applications. Several tools for automatically testing Android applications have been proposed. In particular, Sapienz is a search-based tool that has been recently deployed in an industrial setting. Although it has been shown that Sapienz outperforms several state-of-the-art tools, it is still to be seen what features of Sapienz impact the most on its effectiveness.

We conducted an extensive empirical study where we compare the impact of the search algorithm and the usage of *motif* genes, a more compact representation of individuals. Our empirical study shows that the usage of *motif* genes improves statement coverage both for evolutionary algorithms and random approaches. In particular, our study shows that although the evolutionary algorithm used by Sapienz (i.e., NSGA-II) outperforms other search algorithms, it is not statistically distinguishable from Random Search. These facts cast doubts about the use of evolutionary algorithms in the context of Android test generation and suggest that *motif* genes have a great impact on the overall effectiveness.

KEYWORDS

Android, Sapienz, Empirical Study, Test Generation, Evolutionary Algorithms, Genetic Algorithms, Random Search

ACM Reference Format:
Iván Arcuschin Moreno, Juan Pablo Galeotti, and Diego Garbervetsky. 2020. Algorithm or Representation? An empirical study on how SAPIENZ achieves coverage. In *AST '20: International Conference on Automation of Software Test (AST '20), October 7–8, 2020, Seoul, Republic of Korea.* ACM, New York, NY, USA, 10 pages. https://doi.org/10.1145/3387903.3389307

1 INTRODUCTION

As software keeps becoming more important in our daily lives, the use of mobile devices such as smartphones and tablets increases as well. It is estimated that mobile technologies are now used by two-thirds of the global population. Furthermore, mobile users universally consume more digital minutes per person – more than double in the vast majority of countries and regions [2]. In this context, smartphones have become the dominant platform for mobile time consumption, in terms of total minutes across every market. About 80% of all mobile time [2] is spent in application consumption (commonly known as “apps”). As of January 2020, there are over 2.8 million applications available on Google’s Play App Store [4].

Despite their growing popularity, apps tend to contain defects which can ultimately manifest as failures (or *crashes*) to the end-user. Similarly to other software, testing mobile apps allows developers to ensure a minimum quality threshold for the applications they write. This process typically involves manually writing test cases. Testing intends to assure that new features behave as expected and that changes to the source code do not break previous existing functionality. However, testing is a very time consuming and error-prone task, and hence expensive [27]. To cope with this problem, different tools for automatically testing Android applications have been proposed [16].

Sapienz [33] is one of such tools, which has been proven to outperform several state-of-the-art tools like Dynodroid [31] and Monkey [5]. In recent years, a re-engineered version of the tool has been deployed in the software company Facebook [6]. Essentially, the Sapienz approach presented in Mao et al. [33] distinguishes from previous Android testing tools due to these two features:

i) A multi-objective evolutionary algorithm (NSGA-II [17]) that generates test sequences, simultaneously maximising statement coverage and fault detection while minimising test length.
ii) The representation of test cases as sequences of *atomic* and *motif* actions.

where an *atomic* action is an event that cannot be further decomposed (e.g., pressing down a key, taping the screen at a certain coordinate, etc.). On the other hand, *motif* actions are composed “events” (i.e., a sequence of *atomic* actions) that represents a usage pattern on the app.

Since these features (i.e., the NSGA-II algorithm combined with *motif* actions) were presented simultaneously, we would like to study the impact of each of them separately. What is more, Mao et al. [33] only performed cross-tool comparisons of their technique. This type of comparisons are undesirable since they might have conflating factors arising from implementation details.

In particular, we are interested in comparing different choices of evolutionary algorithms for Android test generation. In [40], Sell et al. present a study comparing different algorithms for Android test generation, but these algorithms are evaluated on the testing tool MATE [19]. Among other differences, MATE differs from Sapienz in that it uses a widget-based representation of individuals. Widgets are interactive components on an Android UI (such as buttons, text fields, etc.). In contrast, Sapienz individuals are based on sequences of actions that do not depend on widgets (i.e., the *atomic* and *motif* actions mentioned above use only screen coordinates). This difference affects how evolutionary operators (such as crossover and mutation) are performed. Therefore, we would like to study how different evolutionary algorithms might behave by implementing them on the Sapienz tool. Also, it has been shown that (at least for unit test generation), due to flat fitness landscapes and often simple search problems, Random Search [28] can perform as well as evolutionary algorithms, and sometimes even outperform them [41]. Thus, we would also like to study the choice of Random Search for Android test generation.

Therefore, in this paper, we aim to gain more insight into the effects of the main features of Sapienz: the choice of the NSGA-II multi-objective algorithm and the representation of the individuals using *motif* actions. Specifically, the contributions of this paper are the following:

- **Experiment design:** We present an empirical study comparing the effectiveness in terms of statement coverage of 9 different algorithms for Android test generation (namely, Random Search, Random Search with *motif* actions, Standard GA, Monotonic GA, Steady-State GA, $(\mu + \lambda)$ EA, (μ, λ) EA, NSGA-II and NSGA-II with *motif* actions) using 8 experimental subjects (Section 4). This study involves both algorithms with and without *motif* actions, as well as a Random Search approach that will serve as a baseline for comparison. The total execution time was 180 days in a 16 core computer.
- **Experiment results:** We present the results of our empirical study (Section 4.2), leading to a total of 2160 data-points based on the 72 different configurations and 30 repetitions for each run.

Our empirical study yields the following findings:

- Both NSGA-II and Random Search improve their effectiveness when test cases include *motif* actions.
- Among all the evolutionary algorithms considered in our study, NSGA-II is the one achieving the highest statement coverage. Surprisingly, NSGA-II does not distinguish with statistical significance from Random Search.

In summary, our experiment provides evidence that the causes for Sapienz performance gains are more attributable to the representation of test cases including *motif* actions rather than to the usage of an evolutionary approach.

The remainder of this article is organized as follows: Section 2 and Section 3 present the necessary background. In particular, they briefly introduce the techniques considered for the empirical study. Section 5 discusses closely related work. Finally, Section 6 concludes the paper and outlines potential future works.

2 BACKGROUND

Evolutionary Algorithms (EAs) are a specific type of population-based meta-heuristic. These algorithms are used to solve optimisation problems and work by mimicking the process of natural selection. They typically start with a randomly generated population (i.e., a set of individuals). Each individual in the population represents a solution to the optimisation problem. Then, several iterations evolve the population towards a given goal. To produce a new generation, the fittest individuals are selected according to some selection mechanism (e.g., rank selection, tournament selection, etc.). After this, the new offspring is generated by applying genetic operators like crossover and mutation with certain parametric probabilities.

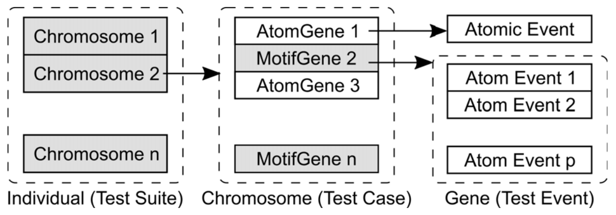

Figure 1: Representation of individuals in evolutionary algorithms as presented by Mao et al. [33]

2.1 Representation of individuals

The representation of individuals in Sapienz follows the Whole Test Suite generation (WTS) [22] principles. WTS evolves whole test suites for an entire coverage criterion at the same time (i.e., statements in the system under test). In WTS, each individual is a test suite (i.e., a set of test cases). Each test case can be seen as a "chromosome" of the individual. Then, each of these chromosomes will be represented as a sequence of genes (test events).

In our context, and to make comparisons between Sapienz and other algorithms fair, these genes will consist of a combination of *atomic* and *motif* genes. As defined by Mao et al. [33], an **atomic gene** is an event that cannot be further decomposed (e.g., press down a key, tap screen at a certain coordinate, etc.), while a **motif gene** is interpreted as a sequence of *atomic* events $\langle e_1, \ldots, e_p \rangle$. This representation is depicted in Figure 1.

Each *motif* gene defined represents a usage patern on the app. These patterns follow common user behaviour, such as filling-in all text fields in the current screen and then clicking a button. As such, *motif* actions are based on the User Interface (UI) information available on the current screen.

2.2 Optimisation goals

To guide the exploration towards a desired goal (i.e., covering all statements in the system), a fitness function must be defined. This function will evaluate each individual in the population. Then, individuals with better fitness values are more likely to survive and propagate their genes to further generations. In the context of test suite generation, the fitness functions are typically based on structural coverage criteria such as statement coverage or branch coverage.

In many cases, it is desirable to optimise the generated test cases towards multiple (possibly conflicting) optimisation goals. A simple mechanism for combining multiple coverage goals is through a weighted linear combination [38]. However, a linear combination requires non-conflicting optimisation goals (e.g., high statement coverage and high mutation score [23]). For instance, a tester would like to obtain a test suite with higher statement coverage and fewer test case length, for debugging and maintenance purposes. It is easy to see that these objectives are conflicting: increasing test length might lead to higher statement coverage while decreasing test length might reduce the coverage. Multi-objective evolutionary algorithms are especially focused on targeting several (possibly conflicting) goals simultaneously.

2.3 Random Search

Random Search [28] is a simple approach for the test suite generation problem. It consists of repeatedly sampling candidates from the search space. Once the budget is exhausted, the fittest sampled individual is returned. Due to its simplicity, it is very useful as a baseline for studying the contributions of any proposed technique.

For unit test generation, it has been shown that Random Search performance is often as effective as other evolutionary algorithms, and it can also outperform them if the system under test is simple enough [41].

2.4 Genetic Algorithms

In this study we will use the Standard Genetic Algorithm (GA) as described by Campos et al. [14]. It starts by generating an initial random population of size p_s. The population is then evaluated using a fitness function δ. Each iteration (i.e., "generation") of the algorithm consists of building a new population and then evaluating each new individual in it. The new population is created by repeatedly choosing a pair of individuals from the current population and then, recombining them into two new individuals. The selection is done with a strategy s_f such as rank-based, elitism or tournament selection. The recombination is done with a crossover function c_f such as single-point or multiple-point, with probability c_p. Before inserting the offspring into the new population, mutation is applied independently on both, with probability m_p. This probability usually is $\frac{1}{n}$, where n is the number of genes in a chromosome. This ensures that, on average, at least one gene is mutated on each offspring, maintaining the diversity in the population.

Several variants of the Standard GA exist that strive to improve effectiveness. In particular, we consider the following alternatives:

- **Monotonic GA:** Similar to the Standard GA, but it only includes either the best offspring or the best parent in the next population. This ensures that achieved fitness value does not decrease as the population evolves.
- **Steady State GA:** This algorithm uses the same replacement strategy as the Monotonic GA, but instead of creating a new population in each generation, the offspring replaces parents in the current population immediately after they are mutated and evaluated.

2.5 Evolutionary Algorithms

The $(\mu + \lambda)$ Evolutionary Algorithm (EA) [39] is a mutation-based algorithm [43]. In this case, μ represents the size of parents and λ the size of the offpsring. For each individual in the current population, mutation is applied independently on each gene with probability $\frac{1}{n}$. After mutation, the best μ individuals are selected among a combined pool of parents and offspring to constitute the new population. Therefore, parents will be replaced only if a better offspring is found. A variant of this is a (μ, λ) EA, where the parents are discarded and the new μ individuals are only selected among the offspring.

3 THE SAPIENZ APPROACH

Sapienz [33] is a multi-objective Android test generation technique aiming at maximising code coverage and fault revelation, while minimising the length of fault-revealing test sequences.

In order to cope with the conflicting goals (i.e., maximising coverage while minimising test length), Sapienz employs the NSGA-II [17] multi-objective evolutionary algorithm, which is widely-used in search-based software engineering (SBSE) research [26]. This algorithm uses a fast non-dominated sorting with a selection operator which creates a mating pool by i) combining the parent and child populations, and ii) selecting the best N solutions according to fitness and spread. During the selection process, all objectives are combined using a Pareto-optimal [20] search-based approach. Formally, an individual x is said to be *dominated* by another individual y ($x \prec y$) according to a fitness function if and only if x is partially less than y:

$$\forall\, i = 1, \dots, n,\ f_i(x) \leq f_i(y) \quad \wedge \quad \exists\, i = 1, \dots, n : f_i(x) < f_i(y)$$

Then, a Pareto-optimal set consists of all the *non-dominated* individuals (belonging to all solutions S):

$$P^* \triangleq \{x \in S \mid \nexists y \in S, x \prec y\}$$

Therefore, a solution to the multi-objective optimisation problem is not a single point in the search space (as in WTS generation is), but a family of points. In practice, this means individuals with longer test sequences are not discarded when they are the only ones finding faults, nor where they are necessary to achieve higher code coverage. Thus, through its use of Pareto optimality, Sapienz progressively replaces the longer sequences with shorter test sequences when they are equally good.

3.1 Exploration strategy

In Sapienz one individual is a test suite. As mentioned previously, each individual consists of several chromosomes (test cases) and each chromosome contains multiple genes (test events), which consist of a random combination of *atomic* and *motif* genes. Event sequences in the test cases are generated and executed by a special component called MotifCore. This component combines random fuzzing and systematic exploration, corresponding to the two types of genes Sapienz supports: *atomic* genes and *motif* genes. The behaviour of each *motif* gene depends on the UI information available at the moment of its execution. These genes are used to perform common user patterns during the exploration, such as filling-in all text fields on the screen and clicking the actionable buttons.

4 EMPIRICAL STUDY

We would like to investigate how the evolutionary algorithm and the representation of individuals in Sapienz affect the overall performance of test generation. Thus, we pose the following research questions:

RQ1 (Representation) *What is the contribution of* motif *genes in* Sapienz *effectiveness?*

RQ2 (Algorithm) *What is the contribution of the NSGA-II evolutionary algorithm in* Sapienz *effectiveness?*

Table 1: Subjects used in empirical study.

Subject	Description	Ver.	Date	LOC
Arity	Scientific calculator	1.27	2012-02-11	2,821
BookWorm	Book collection manager	1.0.18	2011-05-04	7,589
DroidSat	Satellite viewer	2.52	2015-01-11	15,149
FillUp	Calculate fuel mileage	1.7.2	2015-03-10	10,400
JustSit	Meditation timer	0.3.3	2012-07-26	728
Kanji	Character recognition	1.0	2012-10-30	200,154
L9Droid	Interactive fiction	0.6	2015-01-06	18,040
Maniana	User-friendly todo list	1.26	2013-06-28	20,263

In order to answer these questions, we conducted an empirical study. In the following subsection, we describe the experimental setup.

4.1 Experimental setup

For this study, we tried to mimic the experimental setup used in Study #2 presented by Mao et al. [33] as close as possible.

4.1.1 Selection of Subjects Under Test. We chose to use the 10 subjects already used in Study #2 of [33]. These subjects were randomly picked by the authors of Sapienz from the F-Droid[1] repository of open-source Android applications. Two subjects were preliminary discarded (`BabyCare` and `Hydrate`) due to missing source code (`BabyCare`) and dependencies on libraries that are no longer supported by the Android platform version used in our study (`Hydrate`). The remaining 8 subjects used in our study are shown in Table 1.

It is worth noticing that we have not used any of the 68 apps used in Study #1 nor any of the 1,000 Google Play apps used in Study #3 of [33] since those studies present no statistical analysis, as authors did not repeat any execution, and they alternatively used real devices and emulators to evaluate fitness. By doing this, we tried to avoid selection bias.

4.1.2 Measuring effectiveness. We opted to measure effectiveness using only statement coverage as other studies did [10, 11, 13, 37, 41, 47, 48]. Although this metric does not directly indicate the fault detection capability of each algorithm, several studies show evidence of a relationship between the former and statement coverage [21, 34].

4.1.3 Implementation. As we explained in Section 3, Sapienz [33] implements a multi-objective NSGA-II evolutionary algorithm, including a representation of individuals as sequences of both *atomic* and *motif* genes. We extended the latest publicly available version of Sapienz at GitHub [1], adding the algorithms described in Section 2.

However, as the authors stated, this implementation is regarded as “out-of-date and no longer supported”, with the latest activity in the version history recorded in May 2016. Due to this, we enhanced the latest version of Sapienz fixing some issues such as proper time budget management, handling of timeouts when issuing commands to emulators, recovery from an emulator crash.

Besides Sapienz, we have considered 8 algorithms that we have implemented in our extension of Sapienz. To be specific, Random Search (with and without *motif* genes), Standard GA, Monotonic GA, Steady State GA, $\mu + \lambda$ EA, (μ, λ) EA and NSGA-II (i.e., Sapienz without *motif* genes). All the implemented algorithms and the enhanced Sapienz implementation are publicly available on GitHub[2]. For this article, we considered a subset of the algorithms studied in [14]. We discarded many-objective search algorithm (i.e., MOSA [35] and DynaMOSA [36]) since these algorithms were originally designed to work with approach level and branch distance, which are not provided by EMMA tool [3].

It is worth noticing that the MotifCore component of Sapienz (that handles the *motif* genes) was not modified. In other words, the *atomic* and *motif* genes supported in this study are the same that were proposed by Mao et al. [33].

4.1.4 Parameters selection. The parameters were selected following the choices made in Study #2 of [33]. For the crossover function, the uniform crossover operator was used. For the mutation function, a combination of shuffling and one point crossover was used. The crossover and mutation probabilities were set to 0.7 and 0.3, respectively. The selection function used for NSGA-II algorithm was the same as the one depicted by their original authors [17]. The selection function used for the single-objective EAs was roulette selection. The maximum number of generations was set to 100, although none of the evolutionary algorithms accomplished this amount of generations. Population size for each generation was 50 individuals while individuals were composed of 5 test cases. The initial length of each test case was randomly selected between 20 and 500 events. All of these parameters were kept throughout all the executions. We opted to keep the parameters constant to ensure that comparison between the algorithms is fair; since tuning the parameters for each algorithm might change their effectiveness [10].

4.1.5 Experiment Procedure. All test cases were generated on Android KitKat (API 19) because it is the latest Android version supported by the Sapienz prototype publicly available at GitHub. All techniques are fully automated and no manual intervention was provided (e.g., logins) during the execution of the test generators. We executed all test generation algorithms for this study in the Microsoft Azure Cloud Computing Platform[3]. The type of virtual machine chosen was D16s_v3, which features 16 cores and 64GB of RAM. The operating system installed on these virtual machines was Ubuntu 14.04. On each 16 core machine, 16 Android emulators were launched.

Given the random nature of the algorithms, we decided to run 30 repetitions on each subject to gain statistical significance. For each of these executions, we set a maximum time budget of 2 hours. We conservatively doubled the original 1 hour time budget used in [33] to mitigate any emulator or hardware difference. This is by no means a threat to evolutionary approaches as more time budget allows more fitness evaluations (i.e., more generations).

Therefore, the total experimentation time was 9 $algorithms \times 8\ apps \times 30\ repetitions \times 2\ hours\ each = 4.320hs$ (i.e., 180 days of 16 core machines). If we consider the invested time for emulation, this represents $4.320hs \times 16\ emulators = 69.120hs$ (i.e., 2.880 days)

In the Sapienz prototype available at GitHub, the MotifCore component is used for both generating new random test sequences

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

[2]https://github.com/FlyingPumba/evolutiz

[3]https://azure.microsoft.com

and for executing a given test sequence. The latter is required for obtaining the statement coverage of a test suite. However, we found that this component has a known defect which hinders obtaining the correct fitness value while generating new test sequences. Therefore, it was required to re-execute the generated random tests to obtain their correct fitness value. Consequently, to avoid penalizing those approaches that heavily rely on random test generation (such as Random Search) we do not consume any time budget during random test case generation with the MotifCore component.

4.1.6 Experiment Analysis. Statement coverage for the generated test suites was obtained automatically using the EMMA tool [3]. For the single objective genetic algorithms (i.e., Standard GA, Monotonic GA, Steady State GA, $\mu + \lambda$ EA and (μ, λ) EA) we report the statement coverage achieved by the best individual in the last generation. For Random Search, we report the highest coverage achieved by any individual randomly sampled. For the multi-objective NSGA-II algorithm, since the solution is not a single individual but a set of individuals (i.e., the Pareto-optimal front), we report the highest statement coverage achieved by an individual in the Pareto-optimal front.

For the statistical analysis, we followed the same procedures as Campos et al. [13] for comparing different randomized algorithms over a set of subjects. Specifically, we apply the Friedman test [24] with significance level $\alpha = 0.05$ to compare the 9 algorithms on the 8 subject Android applications presented in Table 1. Each algorithm has 8 data points representing the average of achieved statement coverage over the 30 independent repetitions for a given subject application.

The Friedman test is a non-parametric test for multiple-problem analysis and it departs from the traditional tests for significance (e.g., the Wilcoxon test) since it computes the ranking between algorithms over multiple independent problems, i.e., Android applications in our case. A significant $p - value$ indicates that the null hypothesis (i.e., no algorithm in the tournament performs significantly different from the others) has to be rejected in favour of the alternative one (i.e., the performance of algorithms is significantly different from each other). If the null hypothesis is rejected, we use the post hoc Conover's test for pairwise multiple comparisons. Such a test is used to detect pairs of algorithms that are significantly different. Finally, $p - values$ obtained with the post hoc test are adjusted with the Holm-Bonferroni procedure to correct the statistical significance level ($\alpha = 0.05$) in case of multiple comparisons.

In the cases where we want to obtain a more detailed comparison between two algorithms for a given subject, we use the Wilcoxon-Mann-Whitney U-test to determine whether there is a statistically significant difference and the Vargha-Delaney A_{12} effect size to measure this difference (if any).

4.2 Results

Table 2 summarises the results of the experiment described in the previous section. We report the overall statement coverage and the rank of each algorithm, procured by the Friedman test based on their average performance. Table 2 also reports the standard deviation and confidence intervals (CI) using bootstrapping at 95% significance level of the statement coverage achieved.

Among all the algorithms evaluated, NSGA-II + *motif* genes (i.e., Sapienz) is the one that achieves the highest overall statement coverage (47%) and CI. The $p - value$ obtained from the Friedman is 1.97e-09 . This means that we can reject the null hypothesis of the Friedman test (i.e., there is at least one algorithm that differs from the rest). Table 3 shows the rankings achieved by each algorithm for every application. For example, for subject `Arity` NSGA-II achieved the best statement coverage, while (μ, λ) EA performed the worst in terms of that metric.

Table 4 shows the $p - values$ obtained by the post hoc Conover's test for pairwise comparison. These $p - values$ indicate whether there is statistical significance or not for each pair of algorithms. For example, although Table 2 shows that $(\mu + \lambda)$ EA achieved higher ranking than Monotonic GA, Table 4 indicates that the $p - value$ obtained for the post hoc Conover's test is far greater than 0.05. Thus, there is not enough evidence to support with statistical confidence that the average for $(\mu + \lambda)$ EA is different from Monotonic GA.

Figure 2 shows visually the overall statement coverage achieved by each algorithm.

RQ1: What is the contribution of *motif* genes in Sapienz effectiveness?

To answer this question, we conduct a pairwise tournament between NSGA-II with *motif* genes (i.e., Sapienz) and NSGA-II. Furthermore, to understand whether *motif* genes contribute to major gains (even without using an evolutionary algorithm), we also add to the pairwise tournament Random Search and Random Search with *motif* genes.

Table 5 summarises the results of the pairwise tournament. Given a particular subject, Algorithm X is considered to be better than Algorithm Y for that subject if the result of the Wilcoxon-Mann-Whitney U-test gives a $p - value < 0.05$ (i.e., we can say with statistical confidence that Algorithm X is different from Algorithm Y) and the Vargha-Delaney A_{12} effect size is greater than 0.5. Colloquially, this means that Algorithm X performs significantly better on a higher number of comparisons than Algorithm Y. If the $p - value$ of the U-test is greater or equal than 0.05, we can not conclude that Algorithm X is neither better nor worse than Algorithm Y. The positions in the tournament are decided by ranking the differences between the "Better than" and "Worse than" columns. For example, Sapienz has a difference of $16 - 0 = 16$, while Random Search with *motif* genes has a difference of $10 - 3 = 7$, which means the former should be higher in the tournament than the latter.

We can see that the first position is assigned to Sapienz with a significantly better performance in 16 out of 24 comparisons and an average effect size of 0.86. Furthermore, in the remaining 8 of the 24 comparisons, Sapienz is not significantly worse. Surprisingly, the second position goes to Random Search with *motif* genes. This algorithm has a significantly better performance in 10 out of 24 comparisons and an average effect size of 0.90.

Overall, we can see in Table 5 a clear improvement of statement coverage on those algorithms that include *motif* genes over the vanilla version of those algorithms. In terms of statistical significance, Table 4 shows there is enough statistical evidence to hold that there is a difference between both: NSGA-II and NSGA-II with

Table 2: Overall coverage, standard deviation, and the rank of each algorithm based on their average performance, which is statistically significant according to the Friedman test (p-value is < 0.0001, full data is available on Table 3). For averaged coverage values we also report confidence intervals (CI) using bootstrapping at 95% significance level.

Algorithm	Ranking Mean	Ranking SD	Overall Coverage Mean	Coverage SD	CI
NSGA-II + MG (Sapienz)	1.25	0.71	47.87	17.22	[45.68, 50.06]
Random Search + MG	2.12	0.35	46.95	17.65	[44.71, 49.23]
NSGA-II	3.00	1.41	44.07	18.71	[41.70, 46.47]
Random Search	4.19	0.37	43.78	18.57	[41.46, 46.16]
$(\mu + \lambda)$ EA	5.12	1.36	41.79	17.79	[39.47, 44.06]
Monotonic GA	5.56	0.82	40.70	17.62	[38.46, 42.91]
(μ, λ) EA	7.62	1.06	37.23	17.85	[35.00, 39.51]
Standard GA	8.00	0.76	34.49	19.43	[32.05, 36.96]
Steady State GA	8.12	0.83	33.23	19.80	[30.70, 35.74]

Table 3: Full ranking of algorithms for each subject.

	$(\mu + \lambda)$ EA	(μ, λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG (Sapienz)	Random Search	Random Search + MG	Standard GA	Steady State GA
Arity	5.00	9.00	6.00	1.00	3.00	4.00	2.00	8.00	7.00
BookWorm	3.00	8.00	4.00	6.00	1.00	5.00	2.00	7.00	9.00
DroidSat	8.00	6.00	4.50	2.00	1.00	4.50	3.00	8.00	8.00
FillUp	5.00	7.00	6.00	3.00	1.00	4.00	2.00	8.00	9.00
JustSit	5.00	7.00	6.00	3.00	1.00	4.00	2.00	9.00	8.00
Kanji	5.00	8.00	6.00	3.00	1.00	4.00	2.00	7.00	9.00
L9Droid	5.00	7.00	6.00	3.00	1.00	4.00	2.00	9.00	8.00
Maniana	5.00	9.00	6.00	3.00	1.00	4.00	2.00	8.00	7.00
Mean	5.12	7.62	5.56	3.00	1.25	4.19	2.12	8.00	8.12

Table 4: Results of the post hoc Conover's test for pairwise analysis. A $p - value$ less than 0.05 for algorithms X and Y means there is enough evidence to claim they are different with statistically significance.

	$(\mu + \lambda)$ EA	(μ, λ) EA	Monotonic GA	NSGA-II	NSGA-II + MG (Sapienz)	Random Search	Random Search + MG	Standard GA
(μ, λ) EA	< 0.05	-	-	-	-	-	-	-
Monotonic GA	1.000	< 0.05	-	-	-	-	-	-
NSGA-II	< 0.05	< 0.05	< 0.05	-	-	-	-	-
NSGA-II + MG (Sapienz)	< 0.05	< 0.05	< 0.05	< 0.05	-	-	-	-
Random Search	0.410	< 0.05	0.058	0.141	< 0.05	-	-	-
Random Search + MG	< 0.05	< 0.05	< 0.05	0.462	0.462	< 0.05	-	-
Standard GA	< 0.05	1.000	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	-
Steady State GA	< 0.05	1.000	< 0.05	< 0.05	< 0.05	< 0.05	< 0.05	1.000

motif genes (i.e., Sapienz) and between Random Search and Random Search with *motif* genes. We conjecture that this increment is due to *motif* genes using a more compact representation than *atomic* genes. In other words, a complex user pattern can be represented either with a sequence of *N atomic* events or one *motif* gene. This means that, as long as that particular gene keeps propagating across generations, the pattern will survive in the population. On the other hand, if that same pattern was sprayed out into several dozens of events, it would be easier for crossover and mutation operators to break it and lose its benefits. In summary, a more compact representation of test cases might help to trim the search space and achieve higher statement coverage.

RQ1: *Motif genes have a **significant** impact on* Sapienz *effectiveness. In fact, both NSGA-II and Random Search improve their effectiveness when test cases include motif genes.*

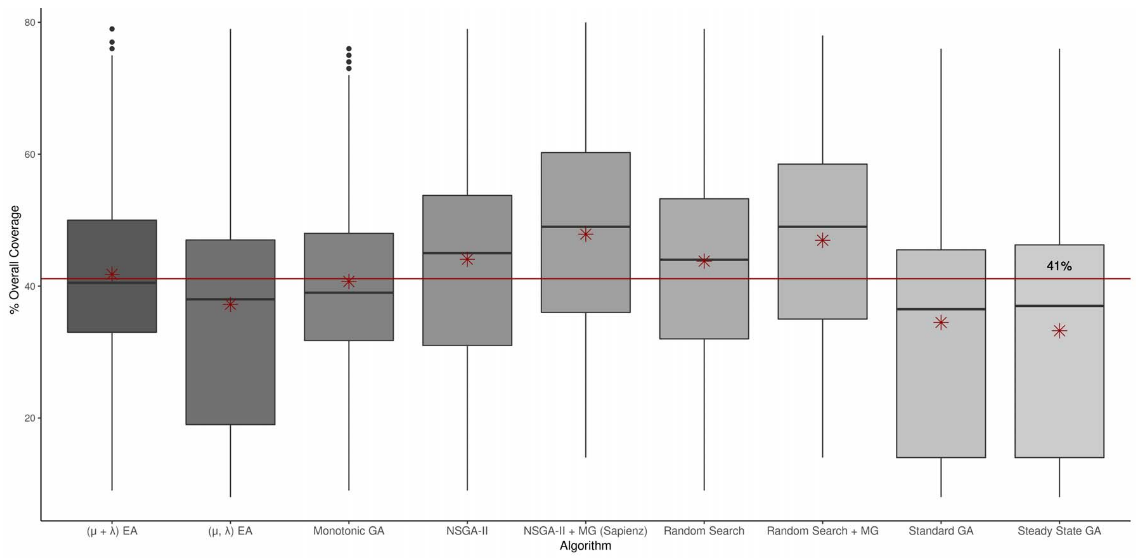

Figure 2: Overall coverage achieved by each algorithm. Middle line of each boxplot marks the median, black circles represent outliers, ⋆ symbol shows the mean, and the red line represents the mean of all coverages (41%).

Table 5: Pairwise comparison of algorithms with and without *Motif Genes*. "Better than" and "Worse than" give the number of comparisons for which the best EA is statistically significantly (i.e., $p - value$ of Wilcoxon-Mann-Whitney U-test less than 0.05) better and worse, respectively. Columns $\widehat{A}_{12}$ give the average effect size.

Algorithm	Tournament position	Overall Coverage Mean	Better than	Better $\widehat{A}_{12}$	Worse than	Worse $\widehat{A}_{12}$
NSGA-II + MG (SAPIENZ)	1.00	47.87	16/24	0.86	0/24	-
NSGA-II	3.00	44.07	1/24	0.66	10/24	0.07
Random Search + MG	2.00	46.95	10/24	0.90	3/24	0.32
Random Search	4.00	43.78	0/24	-	14/24	0.14

RQ2: What is the contribution of the NSGA-II evolutionary algorithm in SAPIENZ effectiveness?

To answer this question, we conduct a pairwise tournament among NSGA-II and the evolutionary algorithms presented in Section 2. As a baseline for comparison, we also included Random Search to the tournament.

Table 6 summarises the results of the pairwise tournament. Among all the evolutionary algorithms evaluated, NSGA-II is the one that achieves the highest overall statement coverage (44%). It is also significantly better than the other algorithms in 36 out of 48 comparisons, and only worst in 2 out of 48. An averaged $\widehat{A}_{12}$ of 0.88 means that in the comparisons for which NSGA-II is significantly better than another algorithm, it obtains a highest statement coverage in 88% of the repetitions.

This result is consistent with other studies such as the one performed by Campos et al. [13] for JAVA unit test case generation in which multi-objective algorithms such as MOSA (a variation of NSGA-II) and DynaMOSA (a latter variation of MOSA) showed higher coverage over single objective search-algorithms.

It is worth noticing that Random Search obtained the second-best place in this pairwise tournament. What is more, the $p - value$ obtained from the Conover's post hoc test when comparing NSGA-II vs Random Search is higher than 0.05. This means that there is not enough evidence to reject the null hypothesis (i.e., that NSGA-II is different from Random Search). To better understand what is the magnitude of this difference between Random Search and evolutionary algorithms, we conducted a more detailed comparison and then calculated the average effect size. Table 7 shows the results of this comparison. Figure 3 shows the results visually. As we can see, NSGA-II is the only algorithm that achieves an average effect size higher than 0.5, but there is no statistical significance.

In other words, Random Search is at least as good as NSGA-II, Monotonic GA and $(\mu + \lambda)$ EA. For all the other evolutionary algorithms, Random Search is better with statistical significance. In summary, this means that EAs are not contributing substantially to gain better statement coverage in ANDROID test generation. This result is also consistent with the study presented by Sell et al. [40] for ANDROID test generation in which single-objective and multi-objective algorithms do not perform better than random algorithms, and sometimes they even perform slightly worse.

In order to optimise a population towards a given objective, evolutionary algorithms require to evolve as many generations as

Table 6: Pairwise comparison of evolutionary algorithms and Random Search. "Better than" and "Worse than" give the number of comparisons for which the best EA is statistically significantly (i.e., $p-value$ of Wilcoxon-Mann-Whitney U-test less than 0.05) better and worse, respectively. Columns $\widehat{A}_{12}$ give the average effect size.

Algorithm	Tournament position	Overall Coverage Mean	Better than	$\widehat{A}_{12}$	Worse than	$\widehat{A}_{12}$
NSGA-II	1.00	44.07	36/48	0.88	2/48	0.31
Random Search	2.00	43.78	33/48	0.90	1/48	0.34
Standard GA	6.50	34.49	1/48	0.73	30/48	0.15
Monotonic GA	4.00	40.70	17/48	0.78	14/48	0.14
Steady State GA	6.50	33.23	1/48	0.68	30/48	0.09
$(\mu + \lambda)$ EA	3.00	41.79	24/48	0.79	13/48	0.21
(μ, λ) EA	5.00	37.23	5/48	0.83	27/48	0.18

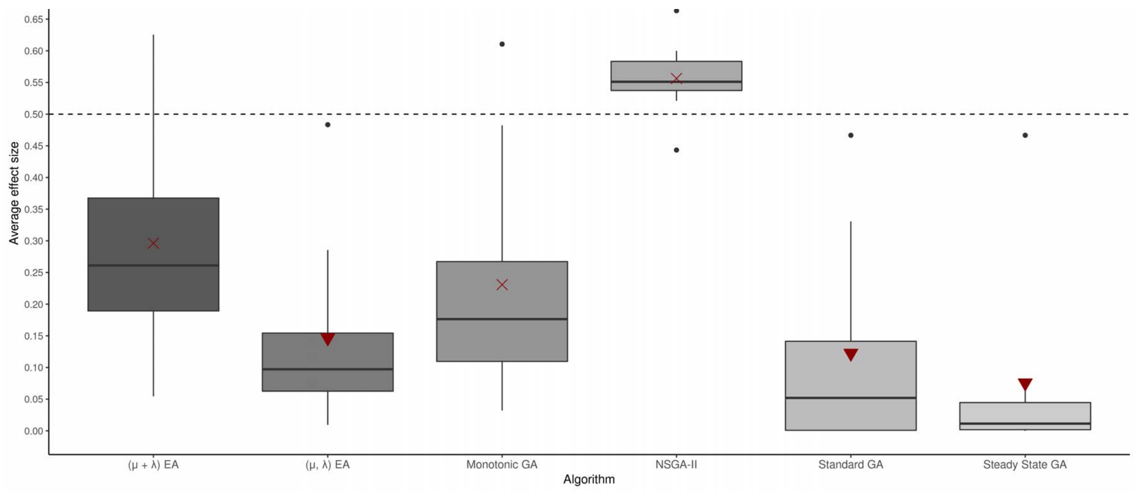

Figure 3: Effect size $\widehat{A}_{12}$ of EA X vs Random Search. Middle line of each boxplot marks the median, black circles represent the outliers, ▲ represents the mean of a significant effect size greater than 0.5 (i.e., EA X performs significantly better than Random Search), ▼ the mean of a significant effect size lower than 0.5 (i.e., EA X performs significantly worse than Random Search), × the mean of a no significant effect size.

Table 7: Comparison of evolutionary algorithms and Random Search. Statistically significant effect sizes are shown in bold.

Algorithm	Overall Coverage Mean	Random Search $\widehat{A}_{12}$	Random Search p
NSGA-II	44.07	0.56	0.141
Random Search	43.78	-	-
Standard GA	34.49	**0.12**	$< \mathbf{0.05}$
Monotonic GA	40.70	0.23	0.058
Steady State GA	33.23	**0.08**	$< \mathbf{0.05}$
$(\mu + \lambda)$ EA	41.79	0.30	0.410
(μ, λ) EA	37.23	**0.15**	$< \mathbf{0.05}$

possible. The cost of a fitness evaluation directly affects the number of generations the EA can achieve. In particular, for ANDROID test generation, in order to obtain statement coverage for a given individual, evolutionary approaches need to: push the test case to a device/emulator, start the application, run test case, gather fitness information and pull it from device/emulator. In [40], Sell et al. suggested that high execution costs hamper any meaningful evolution for search algorithms. In our study, we observed that the fitness evaluation might take up to 60 seconds for a test case, depending on its length. Overall, this resulted in approximately 30 generations for each EA on average. Having a population of 50 individuals, the maximum number of fitness evaluations achieved within the time budget of 2 hours was on average $50 \times 30 = 1.500$ fitness evaluations. Similar execution times can also be found in the work of Vogel et al. [45], where authors report execution times of 101 minutes on average, and up to 5 hours, for running 10 generations of SAPIENZ on one app (using 10 ANDROID emulators).

Finally, Table 4 indicates that there is not enough evidence to hold with statistical significance that NSGA-II with *motif* genes (i.e., SAPIENZ) is different from Random Search with *motif* genes.

RQ2: *NSGA-II evolutionary algorithm has a* ***marginal*** *impact on* SAPIENZ *effectiveness. Although, NSGA-II is better than the other evolutionary algorithms considered, Random Search is at least as good as NSGA-II.*

4.3 Threats to Validity:

Threats to internal validity might result from how the empirical study was carried out. Since all the studied algorithms are affected by non-determinism, we ran 30 repetitions of each experiment with different random seeds and followed rigorous statistical procedures to evaluate the results. To avoid possible confounding factors when comparing different algorithms, they were all implemented on the same tool. Since parameter tuning can affect the performance of algorithms, we used the same default values for all parameters across experimentation. These values were chosen based on the paper presenting SAPIENZ [33]. We used roulette selection as a selection function for the single-objective EAs. Although the rank selection function is preferred to avoid premature convergence [11], the average number of generations performed by the single-objective EAs in our study was 30, which mitigates this possible threat.

Another possible threat to internal validity might come from the fact that we used for our experiments a version of SAPIENZ that might be different from the one that is currently under development at the industrial setting (i.e., Facebook). We chose to use that version (although marked as "out-of-date and no longer supported" by their authors) because it is nevertheless the latest publicly available version used for evaluation by Mao et al. [33].

We measured the success of each algorithm in terms of statement coverage. While higher coverage is a desirable goal for test generation, it is only a proxy for the more important goal of fault detection. Therefore, there is a threat to construct validity caused by how we determine which algorithm is better. However, we believe that this test adequacy criterion is still a reasonable indicator of the effectiveness of different search-based algorithms.

Threats to external validity come from the fact that, due to the very large number of experiments, we only used 8 subjects as case studies, which still took a long time even when using a cluster of computers. To avoid selection bias, we explicitly decided to include *only* those apps that have been previously used in a statistical analysis of SAPIENZ (i.e., Study #2 in Mao et al. [33]). Instead of including new evaluation subjects, we opted to favour a larger number of repetitions (30 per combination of subject & algorithm) to gain better statistical significance. Nevertheless, it is important to note that another selection of subjects might result in different conclusions.

For the selection of algorithms, we considered the algorithms studied in [14]. Some of the multi-objective algorithms (e.g., MOSA and DynaMOSA) had to be excluded from the study since they were not designed to work exclusively with the statement coverage provided by EMMA. Although we included one multi-objective algorithm (i.e., NSGA-II), including further multi-objective algorithms might also result in different conclusions. In future work, we plan to compare the mentioned algorithms as well as other ones such as SPEA-2 [29], NSGA-III [18] and MIO [9].

5 RELATED WORK

Although several studies exist proposing new techniques and tools for automatic ANDROID test generation [5, 7, 25, 31–33], to the best of our knowledge, none of these works performs a thorough analysis of the specific features contributing to the increment or decrement of their effectiveness. In particular, Mao et al. perform a study [33] that shows statistical evidence that SAPIENZ outperforms both MONKEY [5] and DYNODROID [31], but they do not deepen into the causes of the observed gains.

Choudhary et al. [16] compare several test generation tools for ANDROID on a considerable number of open source applications. The tools considered in their study are: MONKEY [5] and DYNODROID [31], EVODROID [32], GUIRIPPER [7], PUMA [25], A3E-DEPTH-FIRST [12], SWIFTHAND [15], JPF-ANDROID [44], and ACTEVE [8]. Although it is an impressive amount of empirical work they do not focus on the specific contributions of the underlying algorithm used by each of the techniques.

Wang et al. [46] also compare several state-of-the-art techniques on industrial applications. The tools evaluated in their study are: MONKEY [5], WCTester [47, 48], SAPIENZ, Stoat [42], DroidBot [30] and A3E-DEPTH-FIRST. The study does not achieve statistical confidence: they only use a few repetitions to compensate for the random nature of algorithms used by the tools.

Campos et al. [13] conducted an empirical study comparing multiple evolutionary algorithms (including some multi-objective) and two random approaches for JAVA unit test generation. The study was applied to a selection of non-trivial open-source classes. They show that the choice of algorithm can have a substantial influence on the performance of Whole Test Suite optimisation. Panichella et al. [37] also performed an empirical study with different evolutionary algorithms for JAVA unit test generation and confirmed several of the findings in [13]. Our work also analyses evolutionary and random algorithms but in the context of ANDROID apps, paying special attention to the effect of using *motif* genes.

Sell et al. [40], also present a study comparing different algorithms for ANDROID test generation, but these algorithms are evaluated on the testing tool MATE [19]. As we have stated before, MATE uses a widget-based representation of individuals, while SAPIENZ does not. This means that evolutionary operators such as crossover and mutation are different between both tools, and might influence results obtained. What is more, some classic genetic and evolutionary algorithms in our study are not included in the work by Sell et al. [40], namely: Monotonic GA, Steady-State GA, $(\mu + \lambda)$ EA, (μ, λ) EA. Finally, the work by Sell et al. [40] uses mainly test cases instead of test suites and does not study the effect of *motif* genes.

6 CONCLUSIONS

In this work, we aimed to deepen into how the main features of SAPIENZ (namely, the NSGA-II algorithm and the representation of individuals using *motif* genes) impact over effectiveness by conducting an extensive empirical study using experimental subject previously used in the related literature.

Our study shows that, for the case of ANDROID test generation, the multi-objective NSGA-II algorithm outperforms other evolutionary algorithms. However, we also found out that NSGA-II is not distinguishable with statistical confidence from Random Search, which casts doubts about the actual effectiveness of multi-objective evolutionary algorithms for ANDROID test generation. In terms of the impact of *motif* genes, our experimental results provide evidence showing that NSGA-II and Random Search performed statistically better when *motif* genes were included. Both findings suggest that the SAPIENZ's improvement on effectiveness is more attributable to adding *motif* genes rather than to the use of a particular choice of multi-objective evolutionary algorithm. Therefore, intra-tool comparisons (as the ones performed in this article and in [40]) should be preferable over cross-tool comparisons (as the one performed by Mao et. al. [33]) whenever possible. In other words, different techniques should be compared using the same tool, aiming to avoid conflating factors behind changes in test suite effectiveness.

As further work, we plan to conduct a large cross-tool empirical study including several ANDROID test generators to assess how sensitive algorithms are to implementation details.

ACKNOWLEDGMENTS

This work is partially funded by UBACYT-2018 20020170200249BA, UBACYT-2018 20020170100419BA, PICT-2015-2741, PICT-2018-3835 and Microsoft Research Azure Grant.

REFERENCES

[1] 2016. Rhapsod/sapienz: A Prototype of Sapienz (Out-of-date and no longer supported). https://github.com/Rhapsod/sapienz.

[2] 2018. Global Digital Future in Focus 2018 - Comscore, Inc. https://www.comscore.com/Insights/Presentations-and-Whitepapers/2018/Global-Digital-Future-in-Focus-2018.

[3] 2020. EMMA: a free Java code coverage tool. http://emma.sourceforge.net/.

[4] 2020. Number of Android applications on the Google Play store | AppBrain. https://www.appbrain.com/stats/number-of-android-apps.

[5] 2020. UI/Application Exerciser Monkey. https://developer.android.com/studio/test/monkey.html.

[6] Nadia Alshahwan, Xinbo Gao, Mark Harman, Yue Jia, Ke Mao, Alexander Mols, Taijin Tei, and Ilya Zorin. 2018. Deploying Search Based Software Engineering with Sapienz at Facebook. In *SSBSE (LNCS)*. Springer.

[7] Domenico Amalfitano, Anna Rita Fasolino, Porfirio Tramontana, Salvatore De Carmine, and Atif M. Memon. 2012. Using GUI ripping for automated testing of Android applications. In *ASE*. ACM, 258–261.

[8] Saswat Anand, Mayur Naik, Mary Jean Harrold, and Hongseok Yang. 2012. Automated concolic testing of smartphone apps. In *SIGSOFT FSE*. ACM, 59.

[9] Andrea Arcuri. 2017. Many Independent Objective (MIO) Algorithm for Test Suite Generation. In *SSBSE (Lecture Notes in Computer Science)*, Vol. 10452. Springer, 3–17.

[10] Andrea Arcuri and Gordon Fraser. 2011. On Parameter Tuning in Search Based Software Engineering. In *SSBSE (Lecture Notes in Computer Science)*, Vol. 6956. Springer, 33–47.

[11] Andrea Arcuri and Gordon Fraser. 2013. Parameter tuning or default values? An empirical investigation in search-based software engineering. *Empirical Software Engineering* 18, 3 (2013), 594–623.

[12] Tanzirul Azim and Iulian Neamtiu. 2013. Targeted and depth-first exploration for systematic testing of android apps. In *OOPSLA*. ACM, 641–660.

[13] José Campos, Yan Ge, Nasser Albunian, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2018. An empirical evaluation of evolutionary algorithms for unit test suite generation. *Information & Software Technology* 104 (2018), 207–235.

[14] José Campos, Yan Ge, Gordon Fraser, Marcelo Eler, and Andrea Arcuri. 2017. An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation. In *SSBSE (Lecture Notes in Computer Science)*, Vol. 10452. Springer, 33–48.

[15] Wontae Choi, George C. Necula, and Koushik Sen. 2013. Guided GUI testing of android apps with minimal restart and approximate learning. In *OOPSLA*. ACM, 623–640.

[16] Shauvik Roy Choudhary, Alessandra Gorla, and Alessandro Orso. 2015. Automated Test Input Generation for Android: Are We There Yet?. In *ASE*. IEEE.

[17] Kalyanmoy Deb, Samir Agrawal, Amrit Pratap, and T. Meyarivan. 2002. A fast and elitist multiobjective genetic algorithm: NSGA-II. *IEEE Trans. Evolutionary Computation* 6, 2 (2002), 182–197.

[18] Kalyanmoy Deb and Himanshu Jain. 2014. An Evolutionary Many-Objective Optimization Algorithm Using Reference-Point-Based Nondominated Sorting Approach, Part I: Solving Problems With Box Constraints. *IEEE Trans. Evolutionary Computation* 18, 4 (2014), 577–601.

[19] Marcelo Medeiros Eler, José Miguel Rojas, Yan Ge, and Gordon Fraser. 2018. Automated Accessibility Testing of Mobile Apps. In *ICST*. IEEE Computer Society.

[20] Carlos M. Fonseca and Peter J. Fleming. 1993. Genetic Algorithms for Multiobjective Optimization: Formulation, Discussion and Generalization. In *ICGA*. Morgan Kaufmann, 416–423.

[21] Phyllis G. Frankl and Oleg Iakounenko. 1998. Further Empirical Studies of Test Effectiveness. In *SIGSOFT FSE*. ACM, 153–162.

[22] Gordon Fraser and Andrea Arcuri. 2013. Whole Test Suite Generation. *IEEE Trans. Software Eng.* 39, 2 (2013), 276–291.

[23] Gordon Fraser and Andrea Arcuri. 2015. Achieving scalable mutation-based generation of whole test suites. *Empirical Software Engineering* 20, 3 (2015), 783–812.

[24] Salvador García, Daniel Molina, Manuel Lozano, and Francisco Herrera. 2009. A study on the use of non-parametric tests for analyzing the evolutionary algorithms' behaviour: a case study on the CEC'2005 Special Session on Real Parameter Optimization. *J. Heuristics* 15, 6 (2009), 617–644.

[25] Shuai Hao, Bin Liu, Suman Nath, William G. J. Halfond, and Ramesh Govindan. 2014. PUMA: programmable UI-automation for large-scale dynamic analysis of mobile apps. In *MobiSys*. ACM, 204–217.

[26] Mark Harman, S. Afshin Mansouri, and Yuanyuan Zhang. 2012. Search-based software engineering: Trends, techniques and applications. *ACM Comput. Surv.* 45, 1 (2012), 11:1–11:61.

[27] Mona Erfani Joorabchi, Ali Mesbah, and Philippe Kruchten. 2013. Real Challenges in Mobile App Development. In *ESEM*. IEEE Computer Society, 15–24.

[28] Dean C. Karnopp. 1963. Random search techniques for optimization problems. *Automatica* 1, 2-3 (1963), 111–121.

[29] Mifa Kim, Tomoyuki Hiroyasu, Mitsunori Miki, and Shinya Watanabe. 2004. SPEA2+: Improving the Performance of the Strength Pareto Evolutionary Algorithm 2. In *PPSN (Lecture Notes in Computer Science)*, Vol. 3242. Springer, 742–751.

[30] Yuanchun Li, Ziyue Yang, Yao Guo, and Xiangqun Chen. 2017. DroidBot: a lightweight UI-guided test input generator for Android. In *ICSE (Companion Volume)*. IEEE Computer Society, 23–26.

[31] Aravind Machiry, Rohan Tahiliani, and Mayur Naik. 2013. Dynodroid: an input generation system for Android apps. In *ESEC/SIGSOFT FSE*. ACM, 224–234.

[32] Riyadh Mahmood, Nariman Mirzaei, and Sam Malek. 2014. EvoDroid: segmented evolutionary testing of Android apps. In *SIGSOFT FSE*. ACM, 599–609.

[33] Ke Mao, Mark Harman, and Yue Jia. 2016. Sapienz: multi-objective automated testing for Android applications. In *ISSTA*. ACM.

[34] Akbar Siami Namin and James H. Andrews. 2009. The influence of size and coverage on test suite effectiveness. In *ISSTA*. ACM, 57–68.

[35] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2015. Reformulating Branch Coverage as a Many-Objective Optimization Problem. In *ICST*. IEEE Computer Society, 1–10.

[36] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. *IEEE Trans. Software Eng.* 44, 2 (2018), 122–158.

[37] Annibale Panichella, Fitsum Meshesha Kifetew, and Paolo Tonella. 2018. A large scale empirical comparison of state-of-the-art search-based test case generators. *Information & Software Technology* 104 (2018), 236–256.

[38] José Miguel Rojas, José Campos, Mattia Vivanti, Gordon Fraser, and Andrea Arcuri. 2015. Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In *SSBSE (Lecture Notes in Computer Science)*, Vol. 9275. Springer, 93–108.

[39] Omur Sahin and Bahriye Akay. 2016. Comparisons of metaheuristic algorithms and fitness functions on software test data generation. *Appl. Soft Comput.* 49 (2016), 1202–1214.

[40] Leon Sell, Michael Auer, Christoph Frädrich, Michael Gruber, Philemon Werli, and Gordon Fraser. 2019. An Empirical Evaluation of Search Algorithms for App Testing. In *ICTSS (LNCS)*, Vol. 11812. Springer.

[41] Sina Shamshiri, José Miguel Rojas, Gordon Fraser, and Phil McMinn. 2015. Random or Genetic Algorithm Search for Object-Oriented Test Suite Generation?. In *GECCO*. ACM, 1367–1374.

[42] Ting Su, Guozhu Meng, Yuting Chen, Ke Wu, Weiming Yang, Yao Yao, Geguang Pu, Yang Liu, and Zhendong Su. 2017. Guided, stochastic model-based GUI testing of Android apps. In *ESEC/SIGSOFT FSE*. ACM.

[43] Aram Ter-Sarkisov and Stephen R. Marsland. 2011. Convergence Properties of $(\mu + \lambda)$ Evolutionary Algorithms. In *AAAI*. AAAI Press.

[44] Heila van der Merwe, Brink van der Merwe, and Willem Visser. 2012. Verifying android applications using Java PathFinder. *ACM SIGSOFT Software Engineering Notes* 37, 6 (2012), 1–5.

[45] Thomas Vogel, Chinh Tran, and Lars Grunske. 2019. Does Diversity Improve the Test Suite Generation for Mobile Applications?. In *SSBSE (Lecture Notes in Computer Science)*, Vol. 11664. Springer, 58–74.

[46] Wenyu Wang, Dengfeng Li, Wei Yang, Yurui Cao, Zhenwen Zhang, Yuetang Deng, and Tao Xie. 2018. An empirical study of Android test generation tools in industrial cases. In *ASE*. ACM, 738–748.

[47] Xia Zeng, Dengfeng Li, Wujie Zheng, Fan Xia, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2016. Automated test input generation for Android: are we really there yet in an industrial case?. In *SIGSOFT FSE*. ACM, 987–992.

[48] Haibing Zheng, Dengfeng Li, Beihai Liang, Xia Zeng, Wujie Zheng, Yuetang Deng, Wing Lam, Wei Yang, and Tao Xie. 2017. Automated Test Input Generation for Android: Towards Getting There in an Industrial Case. In *ICSE-SEIP*. IEEE Computer Society, 253–262.