Does Diversity Improve the Test Suite Generation for Mobile Applications?

Thomas Vogel[(✉)], Chinh Tran, and Lars Grunske

Software Engineering Group, Humboldt-Universität zu Berlin, Berlin, Germany
{thomas.vogel,grunske}@informatik.hu-berlin.de, mail@chinhtran.de

Abstract. In search-based software engineering we often use popular heuristics with default configurations, which typically lead to suboptimal results, or we perform experiments to identify configurations on a trial-and-error basis, which may lead to better results for a specific problem. To obtain better results while avoiding trial-and-error experiments, a fitness landscape analysis is helpful in understanding the search problem, and making an informed decision about the heuristics. In this paper, we investigate the search problem of test suite generation for mobile applications (apps) using SAPIENZ whose heuristic is a default NSGA-II. We analyze the fitness landscape of SAPIENZ with respect to genotypic diversity and use the gained insights to adapt the heuristic of SAPIENZ. These adaptations result in SAPIENZ^{div} that aims for preserving the diversity of test suites during the search. To evaluate SAPIENZ^{div}, we perform a head-to-head comparison with SAPIENZ on 76 open-source apps.

Keywords: Fitness landscape analysis · Diversity · Test generation

1 Introduction

In search-based software engineering and particularly search-based testing, popular heuristics (*e.g.*,[17]) with best-practice configurations in terms of operators and parameters (*e.g.*,[7]) are often used. As this out-of-the-box usage typically leads to suboptimal results, costly trial-and-error experiments are performed to find a suitable configuration for a given problem, which leads to better results [4]. To obtain better results while avoiding trial-and-error experiments, fitness landscape analysis can be used [16,23]. The goal is to analytically understand the search problem, determine difficulties of the problem, and identify suitable configurations of heuristics that can cope with these difficulties (*cf.* [16,19]).

In this paper, we investigate the search problem of test suite generation for mobile applications (apps). We rely on SAPIENZ that uses a default NSGA-II to generate test suite for apps [17]. NSGA-II has been selected as it "is a widely-used multiobjective evolutionary search algorithm, popular in SBSE research" [17, p. 97], but without adapting it to the specific problem (instance). Thus, our goal is to analyze the fitness landscape of SAPIENZ and use the insights for adapting the heuristic of SAPIENZ. This should eventually yield better test results.

Our analysis focuses on the global topology of the landscape, especially how solutions (test suites) are spread in the search space and evolve over time. Thus, we are interested in the genotypic diversity of solutions, which is considered important for evolutionary search [30]. According to our analysis, SAPIENZ lacks diversity of solutions so that we extend it to SAPIENZ^{div} that integrates four diversity promoting mechanisms. Therefore, our contributions are the descriptive study analyzing the fitness landscape of SAPIENZ (Sect. 3), SAPIENZ^{div} (Sect. 4), and the empirical study with 76 apps evaluating SAPIENZ^{div} (Sect. 5).

2   Background: SAPIENZ and Fitness Landscape Analysis

SAPIENZ is a multi-objective search-based testing approach [17]. Using NSGA-II, it automatically generates test suites for end-to-end testing of Android apps. A test suite t consists of m test cases $\langle s_1, s_2, ..., s_m \rangle$, each of which is a sequence of up to n GUI-level events $\langle e_1, e_2, ..., e_n \rangle$ that exercise the app under test. The generation is guided by three objectives: (i) maximize fault revelation, (ii) maximize coverage, and (iii) minimize test sequence length. Having no oracle, SAPIENZ considers a crash of the app caused by a test as a fault. Coverage is measured at the code (statement coverage) or activity level (skin coverage). Given these objectives, the fitness function is the triple of the number of crashes found, coverage, and sequence length. To evaluate the fitness of a test suite, SAPIENZ executes the suite on the app under test deployed on an Android device or emulator.

A *fitness landscape analysis* can be used to better understand a search problem [16]. A fitness landscape is defined by three elements (*cf.* [28]): (1) A search space as a set X of potential solutions. (2) A fitness function $f_k : X \rightarrow \mathbb{R}$ for each of the k objectives. (3) A neighborhood relation $N : X \rightarrow 2^X$ that associates neighbor solutions to each solution (*e.g.*, using basic operators, or distances of solutions). Based on these three elements, various metrics have been proposed to analyze the landscape [16,23]. They characterize the landscape, for instance, in terms of the global topology (*i.e.*, how solutions and the fitness are distributed), local structure (*i.e.*, ruggedness and smoothness), and evolvability (*i.e.*, the ability to produce fitter solutions). The goal of analyzing the landscape is to determine difficulties of a search problem and identify suitable configurations of search algorithms that can cope with these difficulties (*cf.* [16,19]).

3   Fitness Landscape Analysis of SAPIENZ

3.1   Fitness Landscape of SAPIENZ

At first, we define the three elements of a fitness landscape (*cf.* Sect. 2) for SAPIENZ: (1) The search space is given by all possible test suites t according to the representation of test suites in Sect. 2. (2) The fitness function is given by the triple of the number of crashes found, coverage, and test sequence length (*cf.* Sect. 2). (3) As the neighborhood relation we define a genotypic distance metric for two test suites (see Algorithm 1). The distance of two test suites t_1 and t_2 is the sum of the distances between their ordered test sequences, which

is obtained by comparing all sequences s_i^{t1} of t_1 and s_i^{t2} t_2 by index i (lines 2–4). The distance of two sequences is the difference of their lengths (line 5) increased by 1 for each different event at index j (lines 6–9). Thus, the distance is based on the differences of ordered events between the ordered sequences of two test suites.

This metric is motivated by the basic mutation operator of SAPIENZ shuffling the order of test sequences within a suite, and the order of events within a sequence. It is common that the neighborhood relation is based on operators that make small changes to solutions [19].

Algorithm 1. $dist(t_1, t_2)$: compute distance between two test suites t_1 and t_2.

Input: Test suites t_1 and t_2, max. suite size $suite_{max}$, max. sequence length seq_{max}
Output: Distance between t_1 and t_2

```
1: distance ← 0;
2: for i ← 0 to suite_max do            ▷ iterate over all suite_max test sequences
3:     s_i^{t1} ← t_1[i];                         ▷ i^{th} test sequence of test suite t_1
4:     s_i^{t2} ← t_2[i];                         ▷ i^{th} test sequence of test suite t_2
5:     distance ← distance + abs(|s_i^{t1}| - |s_i^{t2}|); ▷ length difference as distance
6:     for j ← 0 to seq_max do               ▷ iterate over all seq_max events
7:         if |s_i^{t1}| ≤ j or |s1_i^{t2}| ≤ j then break;
8:         if s_i^{t1}[j] ≠ s_i^{t2}[j] then            ▷ event comparison by index j
9:             distance ← distance + 1;     ▷ events differ at index j in both seqs.
10: return distance;
```

3.2 Experimental Setup

To analyze the fitness landscape of SAPIENZ, we extended SAPIENZ with metrics that characterize the landscape. We then executed SAPIENZ on five apps, repeat each execution five times, and report mean values of the metrics for each app.[1]

The five apps we selected for the descriptive study are part of the 68 F-Droid benchmark apps [6] used to evaluate SAPIENZ [17]. We selected *aarddict*, *MunchLife*, and *passwordmanager* since SAPIENZ did not find any fault for these apps, and *hotdeath* and *k9mail*[2], for which SAPIENZ did find faults [17]. Thus, we consider apps for which SAPIENZ did and did not reveal crashes to obtain potentially different landscape characteristics that may present difficulties to SAPIENZ.

We configured SAPIENZ as in [17]. The crossover and mutation rates are set to 0.7 and 0.3 respectively. The population and offspring size is 50. An individual (test suite) contains 5 test sequences, each constrained to 20–500 events. Instead of 100 generations [17], we observed in initial experiments that the search stagnates earlier so that we set the number of generation to 40 (stopping criterion).

3.3 Results

The results of our study provide an analysis of the fitness landscape of SAPIENZ with respect to the global topology, particularly the diversity of solutions, how the solutions are spread in the search space, and evolve over time. According to Smith *et al.* [27, p. 31], "No single measure or description can possibly characterize any high-dimensional heterogeneous search space". Thus, we selected 11 metrics from literature and implemented them in SAPIENZ, which characterize (1) the Pareto-optimal solutions, (2) the population, and (3) the connectedness

---

[1] All experiments were run on single 4.0 Ghz quad-core PC with 16 GB RAM, using 5 Android emulators (KitKat 4.4.2, API level 19) in parallel to test one app.

[2] We used ver. 5.207 of k9mail and not ver. 3.512 as in the 68 F-Droid apps benchmark.

of Pareto-optimal solutions, all with a focus on diversity. These metrics are computed after every generation so that we can analyze their development over time. In the following, we discuss these 11 metrics and the results of the fitness landscape analysis. The results are shown in Fig. 1 where the metrics (y-axis) are plotted over the 40 generations of the search (x-axis) for each of the five apps.

(1) Metrics for Pareto-Optimal Solutions

• *Proportion of Pareto-optimal solutions (ppos).* For a population P, *ppos* is the number of Pareto-optimal solutions P_{opt} divided by the population size: $ppos(P) = \frac{|P_{opt}|}{|P|}$. A high and especially strongly increasing *ppos* may indicate that the search based on Pareto dominance stagnates due to missing selection pressure [24]. A moderately increasing *ppos* may indicate a successful search.

For SAPIENZ and all apps (see Fig. 1(a)), *ppos* slightly fluctuates since a new solution can potentially dominate multiple previously non-dominated solutions. At the beginning of the search, *ppos* is low (0.0–0.1), shows no improvement in the first 15–20 generations, and then increases for all apps except of *password-manager*. Thus, the search seems to progress while the enormously increasing *ppos* for MunchLife and hotdeath might indicate a stagnation of the search.

• *Hypervolume (hv).* To further investigate the search progress, we compute the *hv* after each generation. The *hv* is the volume in the objective space covered by the Pareto-optimal solutions [10,31]. Thus, an increasing *hv* indicates that the search is able to find improved solutions, otherwise the *hv* and search stagnate.

Based on the objectives of SAPIENZ (max. crashes, max. coverage, and min. sequence length), we choose the nadir point (0 crashes, 0 coverage, and sequence length of 500) as the reference point for the *hv*. In Fig. 1(b), the evolution of the *hv* over time rather than the absolute numbers are relevant to analyze the search progress of SAPIENZ. While the *hv* increases during the first 25 generations, it stagnates afterwards for all apps; for k9mail already after 5 generations. For aarddict, MunchLife, and hotdeath the *hv* stagnates after the *ppos* drastically increases (*cf.* Fig. 1(a)), further indicating a stagnation of the search.

(2) Population-Based Metrics

• *Population diameter (diam).* The *diam* metrics measure the spread of all population members in the search space using a distance metric for individuals, in our case Algorithm 1. The maximum *diam* computes the largest distance between any two individuals of the population P: $maxdiam(P) = \max_{x_i,x_j \in P} dist(x_i, x_j)$ [5,20], showing the absolute spread of P. To respect outliers, we can compute the average *diam* as the average of all pairwise distances between all individuals [5]:

$$avgdiam(P) = \frac{\sum_{i=0}^{|P|} \sum_{j=0, j \neq i}^{|P|} dist(x_i, x_j)}{|P|(|P| - 1)} \qquad (1)$$

Additionally, we compute the minimum diameter to see how close individuals are in the search space, or even identical: $mindiam(P) = \min_{x_i,x_j \in P} dist(x_i, x_j)$.
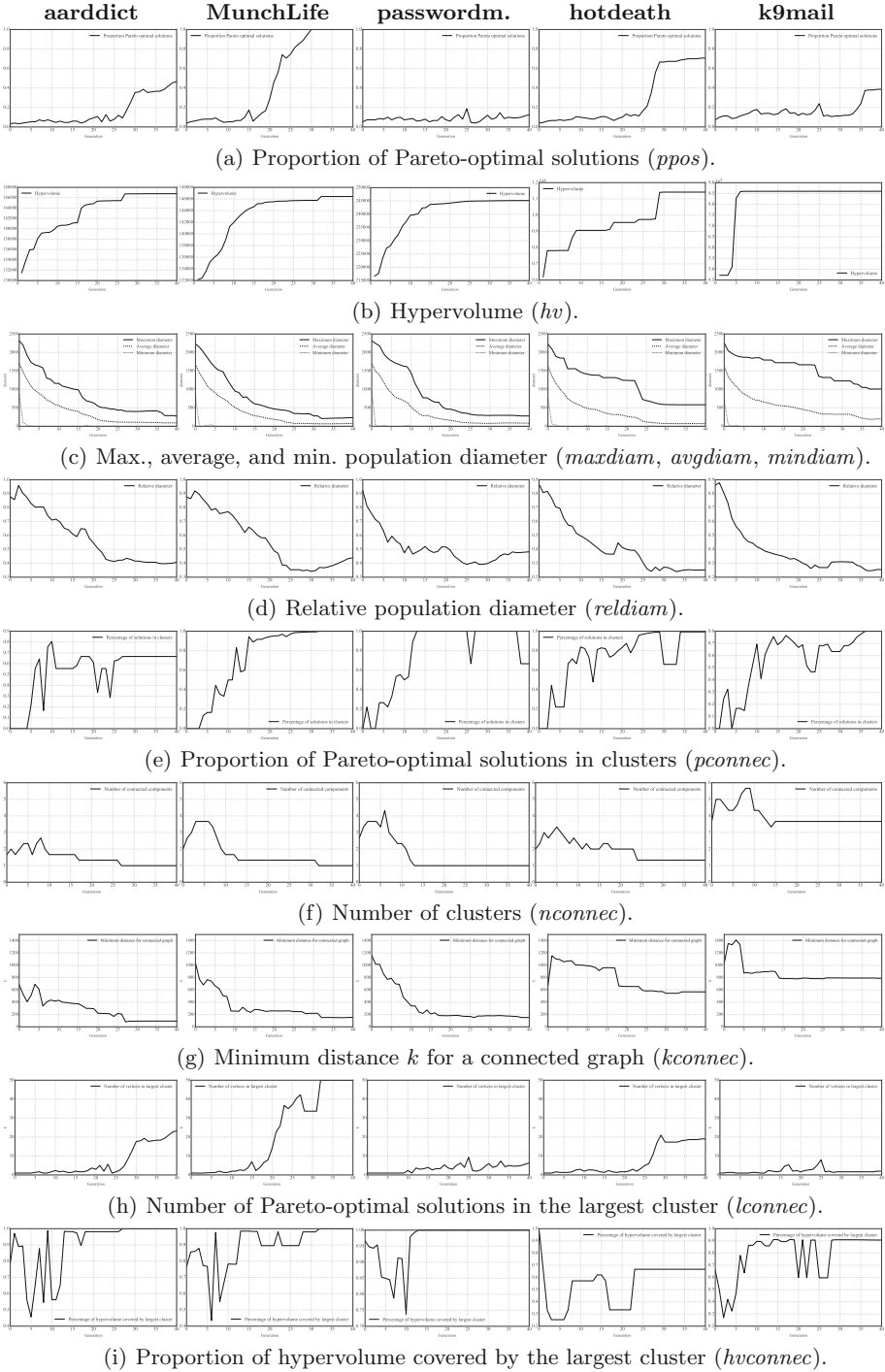
aarddict	MunchLife	passwordm.	hotdeath	k9mail

(a) Proportion of Pareto-optimal solutions (*ppos*).

(b) Hypervolume (*hv*).

(c) Max., average, and min. population diameter (*maxdiam*, *avgdiam*, *mindiam*).

(d) Relative population diameter (*reldiam*).

(e) Proportion of Pareto-optimal solutions in clusters (*pconnec*).

(f) Number of clusters (*nconnec*).

(g) Minimum distance k for a connected graph (*kconnec*).

(h) Number of Pareto-optimal solutions in the largest cluster (*lconnec*).

(i) Proportion of hypervolume covered by the largest cluster (*hvconnec*).

Fig. 1. Fitness landscape analysis results for SAPIENZ.

Concerning each plot for SAPIENZ and all apps (see Fig. 1(c)), the upper, middle, and lower curve are respectively *maxdiam*, *avgdiam*, and *mindiam*. For each curve, we see a clear trend that the metrics decrease over time, which is typical for genetic algorithms due to the crossover. However, the metrics drastically decrease for SAPIENZ in the first 25 generations. The *avgdiam* decreases from >1500 to eventually <200 for each app. The *maxdiam* decreases similarly but stays higher for hotdeath and k9mail than for the other apps. The development of the *avgdiam* and *maxdiam* indicates that all individuals are continuously getting closer to each other in the search space, thus becoming more similar. The population even contains identical solutions as indicated by *mindiam* reaching 0.

• *Relative population diameter (reldiam).* Bachelet [5] further proposes the relative population diameter, which is the *avgdiam* in proportion to the largest possible distance d: $reldiam(P) = \frac{avgdiam(P)}{d}$. This metric is indicative of the concentration of the population in the search space. A small *reldiam* indicates that the population members are grouped together in a region of the space [5].

For SAPIENZ, the largest possible distance d between two test suites is 2500, in which case they differ in all events (up to 500 for a test sequence) for all of their five individual test sequences. For $d = 2500$ and all apps (*cf.* Fig. 1(d)), *reldiam* starts at a high level of around 0.9 indicating that the solutions are spread in the search space. Then, it decreases in the first 25 generations to around 0.4 (aarddict, MunchLife, and passwordmanager), and below 0.3 (hotdeath and k9mail) indicating a grouping of the solutions in one or more regions of the search space.

(3) Metrics Based on the Connectedness of Pareto-Optimal Solutions

The following metrics analyze the *connectedness* and thus, clusters of Pareto-optimal solutions in the search space [9,22]. For this purpose, we consider a graph in which Pareto-optimal solutions are vertices. The edges connecting the vertices are labeled with weights δ, which are the number of moves a neighborhood operator has to make to reach one vertice from another [22]. This results in a graph of fully connected Pareto-optimal solutions. Introducing a limit k on δ and removing the edges whose weights δ are larger than k leads to varying sizes of connected components (clusters) in the graph. This graph can be analyzed by metrics to characterize the Pareto-optimal solutions in the search space [12,22].

In our case, the weights δ are determined by the distance metric for test suites based on the mutation operator of SAPIENZ (*cf.* Algorithm 1). We determined k experimentally to be 300 investigating values of 400, 300, 200, and 100. While a high value results in a single cluster of Pareto-optimal solutions, a low value results in a high number of singletons (*i.e.*, clusters with one solution). Thus, two test suites (vertices) are connected (neighbors) in the graph if they differ in less than 300 events across their test sequences as computed by Algorithm 1.

• *Proportion of Pareto-optimal solutions in clusters (pconnec).* This metric divides the number of vertices (Pareto-optimal solutions) that are members of clusters (excl. singletons) by the total number of vertices in the graph [22]. A high *pconnec* indicates a grouping of the Pareto-optimal solutions in the search space.

As shown in Fig. 1(e), *pconnec* is relatively low during the first generations before it increases for all apps. For MunchLife, passwordmanager, and hotdeath, *pconnec* reaches 1 meaning that all Pareto-optimal solutions are in clusters, while it converges around 0.7 and 0.8 for aarddict and k9mail respectively. This indicates that the Pareto-optimal solutions are grouped in the search space.

● *Number of clusters (nconnec).* We further analyze in how many areas of the search space (clusters) the Pareto-optimal solutions are grouped. Thus, *nconnec* counts the number of clusters in the graph [12, 22]. A high (low) *nconnec* indicates that the Pareto-optimal solutions are spread in many (few) areas of the search space.

Figure 1(f) plots *nconnec* for SAPIENZ and all apps. The y-axis of each plot denoting *nconnec* ranges from 0 to 6. Initially, the Pareto-optimal solutions are distributed in 2–4 clusters, then grouped in 1 cluster. An exception is k9mail for which there always exists more than 3 clusters. Except for k9mail, this indicates that the Pareto-optimal solutions are grouped in one area of the search space.

● *Minimum distance k for a connected graph (kconnec).* This metric identifies k so that all Pareto-optimal solutions are members of one cluster [12, 22]. Thus, *kconnec* quantifies the spread of all Pareto-optimal solutions in the search space.

For SAPIENZ, Fig. 1(g) plots *kconnec* (ranging from 0 to 1400) over the generations. Similarly to the *diam* metrics (*cf.* Fig. 1(c)), *kconnec* decreases, moderately for hotdeath (from initially ≈700 to ≈600) and k9mail (≈1000 → ≈800), and drastically for passwordmanager (≈1200 → ≈200), MunchLife (≈1000 → ≈200), and aarddict (≈600 → ≈100). This indicates that *all* Pareto-optimal solutions are getting closer in the search space as the spread of the cluster is decreasing.

● *Number of Pareto-optimal solutions in the largest cluster (lconnec).* It determines the size of the largest cluster by the number of members [12], showing how many Pareto-optimal solutions are in the most dense area of the search space.

Figure 1(h) plots *lconnec* (ranging from 0 to 50 given the population size of 50) over the generations. *lconnec* increases after 15–30 generations to 20 (aarddict and hotdeath) or even 50 (MunchLife) solutions. This indicates that the largest cluster is indeed large so that many Pareto-optimal solutions are grouped in one area of the search space. In contrast, *lconnec* stays always below 10 indicating smaller largest clusters for passwordmanager and k9mail than for the other apps.

● *Proportion of hypervolume covered by the largest cluster (hvconnec).* Besides *lconnec*, we compute the relative size of the largest cluster in terms of hypervolume (*hv*). Thus, *hvconnec* is the proportion of the overall *hv* covered by the Pareto-optimal solutions in the largest cluster. It quantifies how this cluster in the search space dominates in the objective space and contributes to the *hv*.

For SAPIENZ (*cf.* Fig. 1(i)), *hvconnec* varies a lot during the first 10 generations, then stabilizes at a high level for all apps. For aarddict, MunchLife, and passwordmanager, the largest clusters covers 100% of the *hv* since there is only 1 cluster left (*cf. nconnec* in Fig. 1(f)). For hotdeath, *hvconnec* is close to 70% indicating that there is 1 other cluster covering 30% of the *hv* (*cf. nconnec*). For k9mail, *hvconnec* is around 90% indicating that the other 2–3 clusters

(*cf. nconnec*) cover only 10% of the *hv*. This indicates that the largest cluster covers the largest proportion of the *hv*, and thus contributes most to the Pareto front.

3.4   Discussion

The results characterizing the fitness landscape of Sapienz reveal insights about how Sapienz manages the search problem of generating test suites for apps.

Firstly, the development of the proportion of Pareto-optimal solutions (*cf.* Fig. 1(a)) and hypervolume (*cf.* Fig. 1(b)) indicates a stagnation of the search after 25 generations. The drastically increasing proportion of Pareto-optimal solutions in some cases may indicate a problem of *dominance resistance*, *i.e.*, the search cannot produce new solutions that dominate the current, poorly performing but locally non-dominated solutions [24]. In other cases, the proportion remains low, *i.e.*, the search cannot find many non-dominated solutions.

Secondly, the development of the population diameters (*cf.* Fig. 1(c)) indicate a decreasing diversity of *all* solutions during the search. The development of the relative population diameter (*cf.* Fig. 1(d)) witnesses this observation and indicates that the population members are concentrated in the search space [5]. The minimum diameter (*cf.* Fig. 1(c)) even indicates that the population contains duplicates of solutions, which reduces the genetic variation in the population.

Thirdly, the development of the proportion of Pareto-optimal solutions in clusters (*cf.* Fig. 1(e)) indicates a grouping of these solutions in the search space, mostly in one cluster (*cf.* Fig. 1(f)). Another indicator for the decreasing diversity of the Pareto-optimal solutions is the decreasing minimum distance k required to form one cluster of all these solutions (*cf.* Fig. 1(g)). Additionally, the largest cluster is often indeed large in terms of number of Pareto-optimal solutions (*cf.* Fig. 1(h)), and hypervolume covered by these solutions (*cf.* Fig. 1(i)). Even if there exist multiple clusters of Pareto-optimal solutions, the largest cluster still contributes most to the overall hypervolume and thus, to the Pareto front.

In summary, the fitness landscape analysis of Sapienz indicates a stagnation of the search while the diversity of all solutions decreases in the search space.

4   Sapienz^{div}

Given the fitness landscape analysis results, Sapienz suffers from a decreasing diversity of solutions in the search space over time. It is known that the performance of genetic algorithms is influenced by diversity [21,30]. A low diversity may lead the search to a local optimum that cannot be escaped easily [30]. Thus, diversity is important to address dominance resistance so that the search can produce new solutions that dominate poorly performing, locally non-dominated solutions [24]. Moreover, Shir *et al.* [26, p. 95] report that promoting diversity in the search space does not hamper "the convergence to a precise and diverse Pareto front approximation in the objective space of the original algorithm".

Therefore, we extended SAPIENZ to SAPIENZ^{div} by integrating mechanisms into the search algorithm that promote the diversity of the population in the search space.[3] We developed four mechanisms that extend the SAPIENZ algorithm at different steps: at the initialization, before and after the variation, and at the selection. Algorithm 2 shows the extended search algorithm of SAPIENZ^{div} and highlights the novel mechanisms in blue. We now discuss these mechanisms.

Diverse initial population. As the initial population may effect the results of the search [13], we assume that a diverse initial population could be a better start for the exploration. Thus, we extend the generation of the initial population P_{init} to promote diversity. Instead of generating $|P| = size_{pop}$ solutions, we generate $size_{init}$ solutions where $size_{init} > size_{pop}$ (line 7 in Algorithm 2). Then, we select those $size_{pop}$ solutions from P_{init} that are most distant from each other using Algorithm 1, to form the first population P (line 8).

Adaptive diversity control. This mechanism dynamically controls the diversity if the population members are becoming too close in the search space relative to the initial population. It further makes the algorithm adaptive as it uses feedback of the search to adapt the search (*cf.* [30]).

To quantify the diversity div_{pop} of population P, we use the average population diameter (*avgdiam*) defined in Eq. 1. At the beginning of each generation, div_{pop} is calculated (line 13) and compared to the diversity of the initial population div_{init} (line 14) calculated once in

---

Algorithm 2. Overall algorithm of SAPIENZ^{div}

Input: AUT A, crossover probability p, mutation probability q, max. generation g_{max}, population size $size_{pop}$, offspring size $size_{off}$, size of the large initial population $size_{init}$, diversity threshold div_{limit}, number of diverse solutions to include n_{div}
Output: UI model M, Pareto front PF, test reports C
1: $M \leftarrow K_0; PF \leftarrow \emptyset; C \leftarrow \emptyset;$ ▷ initialization
2: generation $g \leftarrow 0;$
3: boot up devices $D;$ ▷ prepare devices/emulators that will run the app
4: inject MOTIFCORE into $D;$ ▷ install SAPIENZ component for hybrid exploration
5: static analysis on $A;$ ▷ for seeding strings to be used for text fields of A
6: instrument and install $A;$ ▷ app under test is instrumented and installed on D
7: initialize population P_{init} of size $size_{init};$ ▷ large initial population
8: $P = selectMostDistant(P_{init}, size_{pop});$ ▷ select $size_{pop}$ most distant individuals
9: evaluate P with MOTIFCORE and update $(M, PF, C);$
10: $div_{init} = calculateDiversity(P);$ ▷ diversity of the initial population (Eq. 1)
11: **while** $g < g_{max}$ **do**
12:     $g \leftarrow g + 1;$
13:     $div_{pop} = calculateDiversity(P);$ ▷ diversity of the current population (Eq. 1)
14:     **if** $div_{pop} \leq div_{limit} \times div_{init}$ **then**   ▷ check decrease of diversity
15:         $Q \leftarrow$ generate offspring of size $size_{off};$ ▷ ≈ generate a population
16:         evaluate Q with MOTIFCORE and update $(M, PF, C);$
17:         $P = selectMostDistant(P \cup Q, |P|);$ ▷ selection based on distance
18:     **else**
19:         $Q \leftarrow wholeTestSuiteVariation(P, p, q);$ ▷ create offspring
20:         evaluate Q with MOTIFCORE and update $(M, PF, C);$
21:         $PQ \leftarrow removeDuplicates(P \cup Q);$ ▷ duplicate elimination
22:         $\mathcal{F} \leftarrow sortNonDominated(PQ, |P|);$
23:         $P' \leftarrow \emptyset;$ ▷ non-dominated individuals
24:         **for each** front F in $\mathcal{F}$ **do**
25:             **if** $|P'| \geq |P|$ **then** break;
26:             $assignCrowdingDistance(F);$
27:             **for each** individual f in F **do**
28:                 $P' \leftarrow P' \cup f;$
29:         $P' \leftarrow sorted(P', \prec_c);$
30:         $P \leftarrow P'[0 : (size_{pop} - n_{div})];$ ▷ take best $(size_{pop} - n_{div})$ solution from P'
31:         $P_{div} = selectMostDistant(PQ, n_{div});$ ▷ select n_{div} most distant solutions
32:         $P = P \cup P_{div};$ ▷ next population
33: **return** $(M, PF, C);$

---

line 10. The comparison checks whether div_{pop} has decreased to less than $div_{limit} \times div_{init}$. For example, the condition is satisfied for the given threshold $div_{limit} = 0.4$ if div_{pop} has decreased to less than 40% of div_{init}.

---

[3] SAPIENZ^{div} is available at: https://github.com/thomas-vogel/sapienzdiv-ssbse19.

In this case, the offspring Q is obtained by generating new solutions using the original SAPIENZ method to initialize a population (line 15). The next population is formed by selecting the $|P|$ most distant individuals from the current population P and offspring Q (line 17). In the other case, the variation operators (crossover and mutation) of SAPIENZ are applied to obtain the offspring (line 19) followed by the selection. Thus, this mechanism promotes diversity by inserting new individuals to the population, having an effect of restarting the search.

Duplicate elimination. The fitness landscape analysis found duplicated test suites in the population. Eliminating duplicates is one technique to maintain diversity and improve search performance [25,30]. Thus, we remove duplicates after reproduction and before selection in the current population and offspring (line 21). Duplicated test suites are identified by a distance of 0 computed by Algorithm 1.

Hybrid selection. To promote diversity in the search space, the selection is extended by dividing it in two parts: (1) The non-dominated sorting of NSGA-II is performed as in SAPIENZ (lines 22–29 in Algorithm 2) to obtain the solutions P' sorted by domination rank and crowding distance. (2) From P', the best $(size_{pop} - n_{div})$ solutions form the next population P where $size_{pop}$ is the size of P and n_{div} the configurable number of diverse solutions to be included in P (line 30). These n_{div} diverse solutions P_{div} are selected as the most distant solutions from the current population and offspring PQ (line 31) using the distance metric of Algorithm 1. Finally, P_{div} is added to the next population P (line 32).

While the NSGA-II sorting considers the diversity of solutions in the objective space (crowding distance), the selection of SAPIENZ^{div} also considers the diversity of solutions in the search space, which makes the selection hybrid.

5   Evaluation

We evaluate SAPIENZ^{div} in a head-to-head comparison with SAPIENZ to investigate the benefits of the diversity-promoting mechanisms. Our evaluation targets five research questions (RQ) with two empirical studies similarly to [17]:

RQ1. How does the coverage achieved by SAPIENZ^{div} compare to SAPIENZ?

RQ2. How do the faults found by SAPIENZ^{div} compare to SAPIENZ?

RQ3. How does SAPIENZ^{div} compare to SAPIENZ concerning the length of their fault-revealing test sequences?

RQ4. How does the runtime overhead of SAPIENZ^{div} compare to SAPIENZ?

RQ5. How does the performance of SAPIENZ^{div} compare to the performance of SAPIENZ with inferential statistical testing?

5.1 Experimental Setup

We conduct two empirical studies, Study 1 to answer RQ1–4, and Study 2 to answer RQ5. The execution of both studies was distributed on eight servers[4] while each server runs one approach to test one app at a time using 10 Android emulators (Android KitKat version, API 19). We configured SAPIENZ and SAPIENZ^{div} as in the experiment for the fitness landscape analysis (*cf.* Sect. 3.2) and in [17]. The only difference is that we test each app for 10 generations in contrast to Mao *et al.* [17] who test each app for one hour, since we were not in full control of the servers running in the cloud. However, we still report the execution times of both approaches (RQ4). Moreover, we configured the novel parameters of SAPIENZ^{div} as follows: $size_{init} = 100$, $div_{limit} = 0.5$, and $n_{div} = 15$. For Study 1 we perform one run to test each app over 10 generations by each approach. For Study 2 we perform 20 repetitions of such runs for each app and approach.

5.2 Results

Study 1. In this study we use 66 of the 68 F-Droid benchmark apps[5] provided by Choudhary *et al.* [6] and used to evaluate SAPIENZ [17]. The results on each app are shown in Table 1 where **S** refers to SAPIENZ, **Sd** to SAPIENZ^{div}, **Coverage** to the final statement coverage achieved, #**Crashes** to the number of revealed unique crashes, **Length** to the average length of the minimal fault-revealing test sequences (or '–' if no fault has been found), and **Time (min)** to the execution time in minutes of each approach to test the app over 10 generations.

RQ1. SAPIENZ achieves a higher final coverage for 15 apps, SAPIENZ^{div} for 24 apps, and both achieve the same coverage for 27 apps. Figure 2 shows that a similar coverage is achieved by both approaches on the 66 apps, in average 45.05 by SAPIENZ and 45.67 by SAPIENZ^{div}, providing initial evidence that SAPIENZ^{div} and SAPIENZ perform similarly with respect to coverage.

RQ2. To report about the found faults, we count the total crashes, out of which we also identify the unique crashes (*i.e.*, their stack traces are different from the traces of the other crashes of the app). Moreover, we exclude faults caused by the Android system (*e.g.*, native crashes) and test harness (*e.g.*, code instrumentation).

As shown in Table 2, SAPIENZ^{div} revealed more total (6941 vs 5974) and unique (141 vs 119) crashes, and found faults in more apps (46 vs 43) than SAPIENZ. Moreover, it found 51 unique crashes undetected by SAPIENZ, SAPIENZ found 29 unique crashes undetected by SAPIENZ^{div}, and both found the same 90 unique crashes. The results for the 66 apps provide initial evidence that SAPIENZ^{div} can outperform SAPIENZ in revealing crashes.

---

[4] For each server: 2×Intel(R) Xeon(R) CPU E5-2620 @ 2.00 GHz, with 64 GB RAM.
[5] We exclude *aGrep* and *frozenbubble* as SAPIENZ/SAPIENZ^{div} cannot start these apps.

RQ3. Considering the minimal fault-revealing test sequences (*i.e.*, the shortest of all sequences causing the same crash), their mean length is 244 for Sapienz^{div} and 209 for Sapienz on the 66 apps (*cf.* Table 2). This provides initial evidence that Sapienz^{div} produces longer fault-revealing sequences than Sapienz.

RQ4. Considering the mean execution time of testing one app over 10 generation, Sapienz^{div} takes 118 and Sapienz 101 min for the 66 apps. Figure 3 shows that the diversity-promoting mechanisms of Sapienz^{div} cause a noticeable runtime overhead compared to Sapienz. This provides initial evidence about the cost of promoting diversity at which an improved fault detection can be obtained.

Study 2. In this study we use the same 10 F-Droid apps as in the statistical analysis in [17]. Assuming no Gaussian distribution of the results, we use the Kruskal-Wallis test to assess the statistical significance ($p<0.05$) and the Vargha-Delaney effect size $\hat{A}_{12}$ to characterize small, medium, and large differences between Sapienz^{div} and Sapienz ($\hat{A}_{12} > 0.56$, 0.64, and 0.71 respectively).

RQ5. The results are presented by boxplots in Fig. 4 for each of the 10 apps and concern: coverage, #crashes,

Table 1. Results on the 66 benchmark apps.

Subject	Coverage		#Crashes		Length		Time (min)	
	S	**Sd**	**S**	**Sd**	**S**	**Sd**	**S**	**Sd**
a2dp	33	32	4	3	315	250	95	117
aarddict	14	14	1	1	103	454	69	74
aLogCat	66	67	0	2	–	232	125	140
Amazed	69	69	2	1	193	69	67	78
AnyCut	64	64	2	0	244	–	80	105
baterrydog	65	65	1	1	26	155	82	91
swiftp	13	13	0	0	–	–	88	105
Book-Catalogue	19	24	2	4	273	223	86	98
bites	33	35	1	1	76	39	78	91
battery	79	79	9	10	251	230	109	122
addi	19	18	1	1	39	31	87	133
alarmclock	62	62	6	9	133	279	143	163
manpages	69	69	0	0	–	–	81	92
mileage	34	33	5	6	252	286	100	114
autoanswer	16	16	0	0	–	–	78	90
hndroid	15	16	1	1	27	53	97	111
multismssender	57	54	0	0	–	–	88	102
worldclock	90	91	2	1	266	169	109	132
Nectroid	54	54	1	1	261	243	112	136
acal	21	20	7	7	222	187	140	160
jamendo	32	38	8	5	248	266	91	105
aka	45	44	8	9	234	226	140	171
yahtzee	47	47	1	1	356	215	79	86
aagtl	17	17	5	4	170	123	84	111
CountdownTimer	61	62	0	0	–	–	108	143
sanity	13	13	2	3	236	192	154	149
dalvik-explorer	69	69	2	4	148	272	143	162
Mirrored	42	44	10	9	114	179	219	245
dialer2	41	41	2	0	223	–	123	129
DivideAndConquer	79	81	3	3	75	55	90	94
fileexplorer	50	50	0	0	–	–	142	153
gestures	52	52	0	0	–	–	62	69
hotdeath	61	67	2	2	312	360	80	95
adsdroid	38	34	2	4	210	211	107	161
myLock	31	30	0	0	–	–	87	101
lockpatterngenerator	76	76	0	0	–	–	80	94
mnv	29	32	5	6	222	315	118	131
k9mail	5	6	1	2	445	412	93	113
LolcatBuilder	29	28	0	0	–	–	88	101
MunchLife	67	67	0	0	–	–	72	80
MyExpenses	45	41	2	3	359	309	115	133
LNM	57	58	1	1	292	209	104	120
netcounter	59	61	0	1	–	256	95	106
bomber	72	71	0	0	–	–	63	72
fantastischmemo	25	28	3	6	325	275	86	96
blokish	49	62	2	2	197	204	75	86
zooborns	36	36	0	0	–	–	86	95
importcontacts	41	41	0	1	–	462	94	106
wikipedia	26	31	1	3	95	373	69	88
PasswordMaker	50	49	1	2	86	216	103	112
passwordmanager	15	13	1	1	185	354	121	136
Photostream	30	31	2	3	195	161	143	192
QuickSettings	44	41	0	1	–	307	96	130
RandomMusicPlayer	58	59	0	0	–	–	97	113
Ringdroid	40	23	2	4	126	208	280	188
soundboard	53	53	0	0	–	–	61	67
SpriteMethodTest	59	73	0	0	–	–	63	74
SpriteText	60	60	1	2	116	448	93	101
SyncMyPix	19	19	0	2	–	402	97	143
tippy	70	72	1	1	384	459	84	105
tomdroid	50	52	1	1	152	90	93	111
Translate	48	48	0	0	–	–	82	99
Triangle	79	79	1	0	235	–	93	89
weight-chart	47	49	3	4	171	283	88	109
whohasmystuff	60	66	0	1	–	466	118	139
Wordpress	5	5	1	1	244	223	104	224

Fig. 2. Coverage.

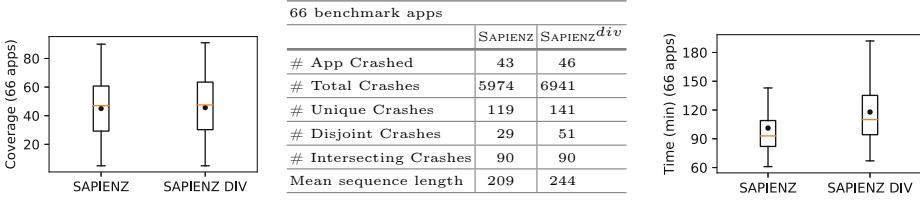| 66 benchmark apps		
	SAPIENZ	SAPIENZ^{div}
# App Crashed	43	46
# Total Crashes	5974	6941
# Unique Crashes	119	141
# Disjoint Crashes	29	51
# Intersecting Crashes	90	90
Mean sequence length	209	244

Table 2. Crashes and seq. length.

Fig. 3. Time (min).

sequence length, and time (*cf.* Study 1). The $\hat{A}_{12}$ effect size for these concerns are shown in Table 3, which compares SAPIENZ^{div} and SAPIENZ (Sd-S) and emphasizes statistically significant results in bold. SAPIENZ significantly outperforms SAPIENZ^{div} with large effect size on all apps for execution time. The remaining results are inconclusive. SAPIENZ^{div} significantly outperforms SAPIENZ with large effect size on only 3/10 apps for coverage, 2/10 for #crashes, and almost 1/10 for length. The remaining results are not statistically significant or do not indicate large differences.
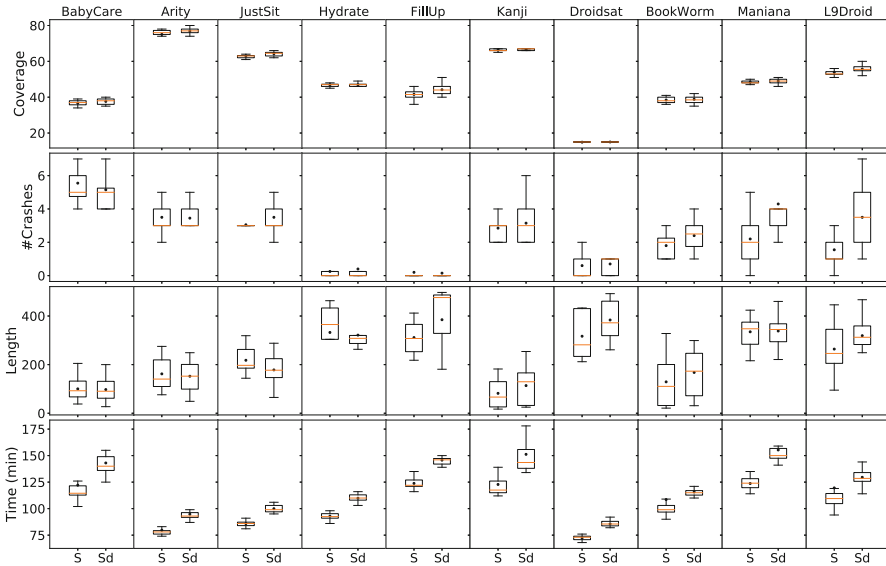

Fig. 4. Performance comparison on 10 apps for SAPIENZ^{div} (Sd) and SAPIENZ (S).

Table 3. Vargha-Delaney effect size (statistically significant results in bold).

App	Ver.	Coverage Sd-S	#Crashes Sd-S	Length Sd-S	Time Sd-S
BabyCare	1.5	0.66	0.46	0.52	**0.15**
Arity	1.27	0.67	0.49	0.54	**0.05**
JustSit	0.3.3	**0.75**	**0.66**	**0.70**	**0.00**
Hydrate	1.5	0.52	0.52	0.64	**0.00**
FillUp	1.7.2	**0.77**	0.47	0.33	**0.00**
Kanji	1.0	0.66	0.56	0.38	**0.09**
Droidsat	2.52	0.55	0.60	0.26	**0.00**
BookWorm	1.0.18	0.58	0.66	0.36	**0.05**
Maniana	1.26	0.66	**0.82**	0.49	**0.00**
L9Droid	0.6	**0.75**	**0.81**	0.32	**0.11**

5.3 Discussion

Study 1 provided initial evidence that SAPIENZ^{div} can find more faults than SAPIENZ while achieving a similar coverage but using longer sequences. Especially, the fault revelation capabilities of SAPIENZ^{div} seemed promising, however, we could not confirm them by the statistical analysis in Study 2. The results of Study 2 are inconclusive in differentiating both approaches by their performance. Potentially, the diversity promotion of SAPIENZ^{div} does not results in the desired effect in the first 10 generations we considered in the studies. In contrast, it might show a stronger effect at later stages since we observed in the fitness landscape analysis that the search of SAPIENZ stagnates after 25 generations.

6 Threats to Validity

Internal validity. A threat to the internal validity is a bias in the selection of the apps we took from [6,17] although the 10 apps for Study 2 were selected by an "unbiased random sampling" [17, p. 103]. We further use the default configuration of SAPIENZ and SAPIENZ^{div} without tuning the parameters to reduce the threat of overfitting to the given apps. Finally, the correctness of the diversity-promoting mechanisms is a threat that we addressed by computing the fitness landscape analysis metrics with SAPIENZ^{div} to confirm the improved diversity.

External validity. As we used 5 (for analyzing the fitness landscape) and 76 Android apps (for evaluating SAPIENZ^{div}) out of over 2.500 apps on F-Droid and millions on Google Play, we cannot generalize our findings although we rely on the well-accepted "68 F-Droid benchmark apps" [6].

7 Related Work

Related work exists in two main areas: approaches on test case generation for apps, and approaches on diversity in search-based software testing (SBST).

Test case generation for apps. Such approaches use random, model-based, or systematic exploration strategies for the generation. Random strategies implement UI-guided test input generators where events on the GUI are selected randomly [3]. Dynodroid [14] extends the random selection using weights and frequencies of events. Model-based strategies such as PUMA [8], DroidBot [11], MobiGUITAR [2], and Stoat [29] apply model-based testing to apps. Systematic exploration strategies range from full-scale symbolic execution [18] to evolutionary algorithms [15,17]. All of these approaches do not explicitly manage diversity, except of Stoat [29] encoding diversity of sequences into the objective function.

Diversity in SBST. Diversity of solutions has been researched for test case selection and generation. For the former, promoting diversity can significantly improve the performance of state-of-the-art multi-objective genetic algorithms [21]. For the latter, promoting diversity results in increased lengths of tests without improved coverage [1], matching our observation. Both approaches witness that diversity promotion is crucial and its realization "requires some care" [24, p. 782].

8   Conclusions and Future Work

In this paper, we reported on our descriptive study analyzing the fitness landscape of SAPIENZ indicating a lack of diversity during the search. Therefore, we proposed SAPIENZ^{div} that integrates four mechanisms to promote diversity. The results of the first empirical study on the 68 F-Droid benchmark apps were promising for SAPIENZ^{div} but they could not be confirmed statistically by the inconclusive results of the second study with 10 further apps. As future work, we plan to extend the evaluation to more generations to see the effect of SAPIENZ^{div} when the search of SAPIENZ stagnates. Moreover, we plan to identify diversity-promoting mechanisms that quickly yield benefits in the first few generations.

References

1. Albunian, N.M.: Diversity in search-based unit test suite generation. In: Menzies, T., Petke, J. (eds.) SSBSE 2017. LNCS, vol. 10452, pp. 183–189. Springer, Cham (2017). https://doi.org/10.1007/978-3-319-66299-2_17
2. Amalfitano, D., Fasolino, A.R., Tramontana, P., Ta, B.D., Memon, A.: Mobiguitar: automated model-based testing of mobile apps. IEEE Softw. **32**(5), 53–59 (2015)
3. Android: Ui/application exerciser monkey (2017)
4. Arcuri, A., Fraser, G.: Parameter tuning or default values? an empirical investigation in search-based software engineering. Emp. Softw. Eng. **18**(3), 594–623 (2013)
5. Bachelet, V.: Métaheuristiques Parallèles Hybrides: Application au Problème D'affectation Quadratique. Ph.D. thesis, Université Lille-I (1999)

6. Choudhary, S.R., Gorla, A., Orso, A.: Automated test input generation for android: are we there yet? In: Proceedings of ASE 2015, pp. 429–440. IEEE (2015)
7. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Trans. Softw. Eng. **39**(2), 276–291 (2013)
8. Hao, S., Liu, B., Nath, S., Halfond, W.G., Govindan, R.: Puma: programmable ui-automation for large-scale dynamic analysis of mobile apps. In: Proceedings of MobiSys 2014, pp. 204–217. ACM (2014)
9. Isermann, H.: The enumeration of the set of all efficient solutions for a linear multiple objective program. Oper. Res. Q. **28**(3), 711–725 (1977)
10. Li, M., Yao, X.: Quality evaluation of solution sets in multiobjective optimisation: a survey. ACM Comput. Surv. **52**(2), 26:1–26:38 (2019)
11. Li, Y., Yang, Z., Guo, Y., Chen, X.: Droidbot: a lightweight ui-guided test input generator for android. In: Proceedings of ICSE 2017 Companion, pp. 23–26. IEEE (2017)
12. Liefooghe, A., Verel, S., Aguirre, H., Tanaka, K.: What makes an instance difficult for black-box 0–1 evolutionary multiobjective optimizers? In: Legrand, P., Corsini, M.-M., Hao, J.-K., Monmarché, N., Lutton, E., Schoenauer, M. (eds.) EA 2013. LNCS, vol. 8752, pp. 3–15. Springer, Cham (2014). https://doi.org/10.1007/978-3-319-11683-9_1
13. Maaranen, H., Miettinen, K., Penttinen, A.: On initial populations of a genetic algorithm for continuous optimization problems. J. Global Optim. **37**(3), 405–436 (2006)
14. Machiry, A., Tahiliani, R., Naik, M.: Dynodroid: an input generation system for android apps. In: Proceedings of ESEC/FSE 2013. pp. 599–609. ACM (2013)
15. Mahmood, R., Mirzaei, N., Malek, S.: Evodroid: segmented evolutionary testing of android apps. In: Proceedings of FSE 2014. pp. 599–609. ACM (2014)
16. Malan, K.M., Engelbrecht, A.P.: A survey of techniques for characterising fitness landscapes and some possible ways forward. Inf. Sci. **241**, 148–163 (2013)
17. Mao, K., Harman, M., Jia, Y.: Sapienz: multi-objective automated testing for android applications. In: Proceedings of ISSTA 2016. pp. 94–105. ACM (2016)
18. Mirzaei, N., Malek, S., Păsăreanu, C.S., Esfahani, N., Mahmood, R.: Testing android apps through symbolic execution. Softw. Eng. Notes **37**(6), 1–5 (2012)
19. Moser, I., Gheorghita, M., Aleti, A.: Identifying features of fitness landscapes and relating them to problem difficulty. Evol. Comp. **25**(3), 407–437 (2017)
20. Olorunda, O., Engelbrecht, A.P.: Measuring exploration/exploitation in particle swarms using swarm diversity. In: Proceedings of CEC 2008. pp. 1128–1134. IEEE (2008)
21. Panichella, A., Oliveto, R., Penta, M.D., Lucia, A.D.: Improving multi-objective test case selection by injecting diversity in genetic algorithms. IEEE Trans. Software Eng. **41**(4), 358–383 (2015)
22. Paquete, L., Stützle, T.: Clusters of non-dominated solutions in multiobjective combinatorial optimization: an experimental analysis. Multiobjective Programming and Goal Programming. Lecture Notes in Economics and Mathematical Systems, vol. 618, pp. 69–77. Springer, Berlin (2009). https://doi.org/10.1007/978-3-540-85646-7_7
23. Pitzer, E., Affenzeller, M.: A comprehensive survey on fitness landscape analysis. Recent Advances in Intelligent Engineering. Studies in Computational Intelligence, vol. 378, pp. 161–191. Springer, Berlin (2012). https://doi.org/10.1007/978-3-642-23229-9_8
24. Purshouse, R.C., Fleming, P.J.: On the evolutionary optimization of many conflicting objectives. IEEE Trans. Evolut. Comp. **11**(6), 770–784 (2007)

25. Ronald, S.: Duplicate genotypes in a genetic algorithm. In: Proceedings of ICEC 1998. pp. 793–798. IEEE (1998)
26. Shir, O.M., Preuss, M., Naujoks, B., Emmerich, M.: Enhancing decision space diversity in evolutionary multiobjective algorithms. In: Ehrgott, M., Fonseca, C.M., Gandibleux, X., Hao, J.-K., Sevaux, M. (eds.) EMO 2009. LNCS, vol. 5467, pp. 95–109. Springer, Heidelberg (2009). https://doi.org/10.1007/978-3-642-01020-0_12
27. Smith, T., Husbands, P., Layzell, P.J., O'Shea, M.: Fitness landscapes and evolvability. Evol. Comput. **10**(1), 1–34 (2002)
28. Stadler, P.F.: Fitness landscapes. Biological Evolution and Statistical Physics. Lecture Notes in Physics, vol. 585, pp. 183–204. Springer, Berlin (2002). https://doi.org/10.1007/3-540-45692-9_10
29. Su, T., Meng, G., Chen, Y., Wu, K., et al.: Guided, stochastic model-based gui testing of android apps. In: Proceedings of ESEC/FSE 2017. pp. 245–256. ACM (2017)
30. Črepinšek, M., Liu, S.H., Mernik, M.: Exploration and exploitation in evolutionary algorithms: a survey. ACM Comput. Surv. **45**(3), 35:1–35:33 (2013)
31. Wang, S., Ali, S., Yue, T., Li, Y., Liaaen, M.: A practical guide to select quality indicators for assessing pareto-based search algorithms in search-based software engineering. In: Proceedings of ICSE 2016. pp. 631–642. ACM (2016)