An Empirical Evaluation of Evolutionary Algorithms for Test Suite Generation

José Campos[1], Yan Ge[1], Gordon Fraser[1], Marcelo Eler[2], and Andrea Arcuri[3]

[1] Department of Computer Science, The University of Sheffield, UK
[2] University of São Paulo, Brazil
[3] Westerdals Oslo ACT, Norway and University of Luxembourg, Luxembourg

Abstract. Evolutionary algorithms have been shown to be effective at generating unit test suites optimised for code coverage. While many aspects of these algorithms have been evaluated in detail (e.g., test length and different kinds of techniques aimed at improving performance, like seeding), the influence of the specific algorithms has to date seen less attention in the literature. As it is theoretically impossible to design an algorithm that is best on all possible problems, a common approach in software engineering problems is to first try a Genetic Algorithm, and only afterwards try to refine it or compare it with other algorithms to see if any of them is more suited for the addressed problem. This is particularly important in test generation, since recent work suggests that random search may in practice be equally effective, whereas the reformulation as a many-objective problem seems to be more effective. To shed light on the influence of the search algorithms, we empirically evaluate six different algorithms on a selection of non-trivial open source classes. Our study shows that the use of a test archive makes evolutionary algorithms clearly better than random testing, and it confirms that the many-objective search is the most effective.

1 Introduction

Search-based testing has been successfully applied to generating unit test suites optimised for code coverage on object-oriented classes. A popular approach is to use evolutionary algorithms where the individuals of the search population are whole test suites, and the optimisation goal is to find a test suite that achieves maximum code coverage [8]. Tools like EvoSuite [6] have been shown to be effective in achieving code coverage on different types of software [9].

Since the original introduction of whole test suite generation, many different techniques have been introduced to improve performance even further and to get a better understanding of the current limitations. For example, the insufficient guidance provided by basic coverage-based fitness functions has been shown to cause random search to often be equally effective as evolutionary algorithms [24]. Optimisation now no longer focuses on individual coverage criteria, but combinations of criteria [10,21]. To cope with the resulting larger number of coverage goals, evolutionary search can be supported with archives [22] that keep track of useful solutions encountered throughout the search. To improve effectiveness,

whole test suite optimisation has been re-formulated as a many-objective optimisation problem [19]. In the context of these developments, one aspect of whole test suite generation remains largely unexplored: What is the influence of the specific flavour of evolutionary algorithms applied to evolve test suites?

In this paper, we aim to shed light on the influence of the different evolutionary algorithms in whole test suite generation, to find out whether the choice of algorithm is important, and which one should be used. By using a large set of complex Java classes as case study, and the EvoSuite [6] search-based test generation tool, we specifically investigate:

RQ1: Which evolutionary algorithm works best when using a test archive for partial solutions?
RQ2: How does evolutionary search compare to random search and random testing?
RQ3: How does evolution of whole test suites compare to many-objective optimisation of test cases?

We investigate each of these questions in the light of individual and multiple coverage criteria as optimisation objectives, and we study the influence of the search budget. Our results show that in most cases a simple $\mu+\lambda$ Evolutionary Algorithm (EA) is better than other, more complex algorithms. In most cases, the variants of EAs and GAs are also clearly better than random search and random testing, when a test archive is used. Finally, we confirm that many-objective search achieves higher branch coverage, even in the case of optimisation for multiple criteria.

2 Evolutionary Algorithms for Test Suite Generation

Evolutionary Algorithms (EAs) are inspired by natural evolution, and have been successfully used to address many kinds of optimisation problems. In the context of EAs, a solution is encoded "genetically" as an individual ("chromosome"), and a set of individuals is called a population. The population is gradually optimised using genetic-inspired operations such as crossover, which merges genetic material from at least two individuals to yield new offspring, and mutation, which independently changes the elements of an individual with a low probability. While it is impossible to comprehensively cover all existing algorithms, in the following we discuss common variants of EAs for test suite optimisation. Expansion of the evaluation to less common algorithms will be future work.

2.1 Representation

For test suite generation, the individuals of a population are sets of test cases (test suites); each test case is a sequence of calls. Crossover on test suites is based on exchanging test cases [8]; mutation adds/modifies tests to suites, and adds/removes/changes statements within tests. While standard selection techniques are largely used, the variable size representation (number of statements in a test and number of test cases in a suite can vary) requires modification to avoid bloat [7]; this is typically achieved by ranking individuals with identical fitness based on their length, and then using rank selection.

2.2 Optimisation Goals and Archives

The selection of individuals is guided by fitness functions, such that individuals with good fitness values are more likely to survive and be involved in reproduction. In the context of test suite generation, the fitness functions are based on code coverage criteria such as statement or branch coverage. More recently, there is a trend to optimise for multiple coverage criteria at the same time. Since coverage criteria usually do not represent conflicting goals, it is possible to combine fitness functions with a weighted linear combination [21]. However, the increased number of coverage goals may affect the performance of the EA. To counter these effects, it is possible to store tests for covered goals in an archive [22], and then to dynamically adapt the fitness function to optimise only for the remaining uncovered goals. This, however, may again have effects on the underlying EA. Furthermore, search operators can be adapted to make use of the test archive; for example, new tests may be created by mutating tests in the archive rather than randomly generating completely new tests.

2.3 Random Search

Random search is a baseline search strategy which does not use crossover, mutation, or selection, but a simple replacement strategy [14]. Random search consists of repeatedly sampling candidates from the search space; the previous candidate is replaced if the fitness of the new sampled individual is better. Random search can make use of a test archive by changing the sampling procedure as indicated above. *Random testing* is a variant of random search in test generation which builds test suites incrementally. Test cases (rather than test suites) are sampled individually, and if a test improves coverage, it is retained in the test suite, otherwise it is discarded. It has been shown that in unit test generation, due to the flat fitness landscapes and often simple search problems, random search is often as effective as EAs, and sometimes even better [24].

2.4 Genetic Algorithms

The Genetic Algorithm (GA) is one of the most widely-used EAs in many domains because it can be easily implemented and obtains good results on average. Algorithm 1 illustrates a Standard GA. It starts by creating an initial random population of size p_n (Line 1). Then, a pair of individuals is selected from the population using a strategy s_f, such as rank-based, elitism or tournament selection (Line 6). Next, both selected individuals are recombined using crossover c_f (e.g., single point, multiple-point) with a probability of c_p to produce two new offspring o_1, o_2 (Line 7). Afterwards, mutation is applied on both offspring (Lines 8–9), independently changing the genes with a probability of m_p, which usually is equal to $\frac{1}{n}$, where n is the number of genes in a chromosome. The two mutated offspring are then included in the next population (Line 10). At the end of each iteration the fitness value of all individuals is computed (Line 13).

Many variants of the Standard GA have been proposed to improve effectiveness. Specifically, we consider a *monotonic* version of the Standard GA which, after mutating and evaluating each offspring, only includes either the best offspring or the best parent in the next population (whereas the Standard GA

Algorithm 1 Standard Genetic Algorithm

Input: Stopping condition C, Fitness function δ, Population size p_s, Selection function s_f, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P

$P \leftarrow$ GenerateRandomPopulation(p_s)
PerformFitnessEvaluation(δ, P)
while $\neg C$ **do**
    $N_P \leftarrow \{\}$
    while $|N_P| < p_s$ **do**
        $p_1, p_2 \leftarrow$ Selection(s_f, P)
        $o_1, o_2 \leftarrow$ Crossover(c_f, c_p, p_1, p_2)
        Mutation(m_f, m_p, o_1)
        Mutation(m_f, m_p, o_2)
        $N_P \leftarrow N_P \cup \{o_1, o_2\}$
    end while
    $P \leftarrow N_P$
    PerformFitnessEvaluation(δ, P)
end while
return P

includes both offspring in the next population regardless of their fitness value). Another variation of the Standard GA is a *Steady State* GA, which uses the same replacement strategy as the Monotonic GA, but instead of creating a new population of offspring, the offspring replace the parents from the current population immediately after the mutation phase.

The $1 + (\lambda, \lambda)$ GA, introduced by Doerr et al. [5], starts by generating a random population of size 1. Then, mutation is used to create λ different mutated versions of the current individual. Mutation is applied with a high mutation probability, defined as $m_p = \frac{k}{n}$, where k is typically greater than one, which allows, on average, more than one gene to be mutated per chromosome. Then, uniform crossover is applied to the parent and best generated mutant to create λ offspring. While a high mutation probability is intended to support faster exploration of the search space, a uniform crossover between the best individual among the λ mutants and the parent was suggested to repair the defects caused by the aggressive mutation. Then all offspring are evaluated and the best one is selected. If the best offspring is better than the parent, the population of size one is replaced by the best offspring. $1 + (\lambda, \lambda)$ GA could be very expensive for large values of λ, as fitness has to be evaluated after mutation and after crossover.

2.5 $\mu + \lambda$ Evolutionary Algorithm

The $\mu + \lambda$ Evolutionary Algorithm (EA) is a mutation-based algorithm [25]. As its name suggests, the number of parents and offspring are restricted to μ and λ, respectively. Each gene is mutated independently with probability $\frac{1}{n}$. After

mutation, the generated offspring are compared with each parent, aiming to preserve so-far best individual including parents; that is, parents are replaced once a better offspring is found. Among the different (μ+λ) EA versions, two common settings are (1+λ) EA and (1+1) EA, where the population size is 1, and the number of offspring is also limited to 1 for the (1+1) EA.

2.6 Many-Objective Sorting Algorithm

Unlike the single-objective optimisation on the test suite level described above, the Many-Objective Sorting Algorithm (MOSA) [19] regards each coverage goal as an independent optimisation objective. MOSA is a variant of NSGA-II [4], and uses a preference sorting criterion to reward the best tests for each non-covered target, regardless of their dominance relation with other tests in the population. MOSA also uses an archive to store the tests that cover new targets, which aiming to keep record on current best cases after each iteration.

Algorithm 2 illustrates how MOSA works. It starts with a random population of test cases. Then, and similar to typical EAs, the offspring are created by applying crossover and mutation (Line 6). Selection is based on the combined set of parents and offspring. This set is sorted (Line 9) based on a non-dominance relation and preference criterion. MOSA selects non-dominated individuals based on the resulting rank, starting from the lowest rank (F_0), until the population size is reached (Lines 11-14). In fewer than p_s individuals are selected, the individuals of the current rank (F_r) are sorted by crowding distance (Line 16-17), and the individuals with the largest distance are added. Finally, the archive that stores previously uncovered branches is updated in order to yield the final test suite (Line 18). In order to cope with the large numbers of goals resulting from the combination of multiple coverage criteria, the DynaMOSA [18] extension dynamically selects targets based on the dependencies between the uncovered targets and the newly covered targets. Both, MOSA and DynaMOSA, have been shown to result in higher coverage of some selected criteria than traditional GAs for whole test suite optimisation.

3 Empirical Study

In order to evaluate the influence of the evolutionary algorithm on test suite generation, we conducted an empirical study. In this section, we describe the experimental setup as well as results.

3.1 Experimental Setup

Selection of Classes Under Test: A key factor of studying evolutionary algorithms on automatic test generation is the selection of classes under test. As many open source classes, for example contained in the SF110 [9] corpus, are trivially simple [24] and would not reveal differences between algorithms, we used the selection of non-trivial classes from the DynaMOSA study [17]. This is a corpus of 117 open-source Java projects and 346 classes, selected from four different benchmarks. The complexity of classes ranges from 14 statements and 2 branches to 16,624 statements and 7,938 branches. The average number of statements is 1,109, and the average number of branches is 259.

Algorithm 2 Many-Objective Sorting Algorithm (MOSA)

Input: Stopping condition C, Fitness function δ, Population size p_s, Crossover function c_f, Crossover probability c_p, Mutation probability m_p
Output: Archive of optimised individuals A

$p \leftarrow 0$
$N_p \leftarrow$ GenerateRandomPopulation(p_s)
PerformFitnessEvaluation(δ, N_p)
$A \leftarrow \{\}$
while $\neg C$ **do**
    $N_o \leftarrow$ GenerateOffspring(c_f, c_p, m_p, N_p)
    $R_t \leftarrow P \cup N_o$
    $r \leftarrow 0$
    $F_r \leftarrow$ PreferenceSorting(R_t)
    $N_{p+1} \leftarrow \{\}$
    while $|N_{p+1}| + |F_r| \leq p_s$ **do**
        CalculateCrowdingDistance(F_r)
        $N_{p+1} \leftarrow N_{p+1} \cup F_r$
        $r \leftarrow r + 1$
    end while
    DistanceCrowdingSort(F_r)
    $N_{p+1} \leftarrow N_{p+1} \cup F_r$ with size $p_s - |N_{p+1}|$
    UpdateArchive(A, N_{p+1})
    $p \leftarrow p + 1$
end while
return A

Unit Test Generation Tool: We used EvoSuite [6], which provides search algorithms to evolve coverage-optimised test suites. By default, EvoSuite uses a Monotonic GA described in Section 2.4. It also provides a Standard and Steady State GA, Random search, Random testing and, more recently, MOSA and DynaMOSA. For this study, we added the 1+(λ, λ) GA and the $\mu + \lambda$ EA. All evolutionary algorithms use a test archive.

Experiment Procedure: We performed two experiments to assess the performance of six evolutionary algorithms (described in Section 2). First, we conducted a tuning study to select the best population size (μ) of four algorithms, number of mutations (λ) of $1 + (\lambda, \lambda)$ GA, and population size (μ) and number of mutations (λ) of $\mu + \lambda$ EA, since the performance of each EA can be influenced by the parameters used [1]. Random-based approaches do not require any tuning. Then, we conducted a larger study to perform the comparison.

For both experiments we have four configurations: two search budgets, EvoSuite's default search budget (i.e., a small search budget) of 1 minute, and a larger search budget of 10 minutes to study the effect of the search budget on the coverage of resulting test suites; single-criterion optimisation (branch coverage)

and multiple-criteria optimisation[4] (i.e., line, branch, exception, weak-mutation, output, method, method-no-exception, and cbranch) [21]. Due to the randomness of EAs, we repeated the one minute experiments 30 times, and the 10 minutes experiments 10 times.

For the tuning study, we randomly selected 10% (i.e., 34) of DynaMOSA's study classes [17][5] (with 15 to 1,707 branches, 227 on average) from 30 Java projects. This resulted in a total of 79,200 (59, 400 one minute configurations, and 19, 800 ten minutes configurations) calls to EvoSuite and more than 175 days of CPU-time overall. For the second experiment, we used the remaining 312 classes[6] (346 total - 34 used to tune each EA) from the DynaMOSA study [17]. Besides the tuned μ and λ parameters, we used EvoSuite's default parameters [1].

Experiment Analysis: For any test suite generated by EvoSuite on any experimental configuration we measure the coverage achieved on eight criteria, alongside other metrics, such as the number of generated test cases, the length of generated test suites, number of iterations of each EA, number of fitness evaluations. As described by Arcuri et al. [1] "easy" branches are always covered independently of the parameter settings used, and several others are just infeasible. Therefore, rather than using raw coverage values, we use relative coverage [1]: Given the coverage of a class c in a run r, $c(r)$, the best and worst coverage of c in any run, $max(c)$ and $min(c)$ respectively, a *relative coverage* (r_c) can be defined as $\frac{c(r) - min(c)}{max(c) - min(c)}$. If the best and worst coverage of c is equal, i.e., $max(c) == min(c)$, then r_c is 1 (if range of $c(r)$ is between 0 and 1) or 100 (if range of $c(r)$ is between 0 and 100). In order to statistically compare the performance of each EA we use the Vargha-Delaney $\hat{A}_{12}$ effect size, and the Wilcoxon-Mann-Whitney U-test with a 95% confidence level. Besides the Vargha-Delaney effect size we also consider a *relative average improvement*. Given two sets of coverage values, configuration A and configuration B, a *relative average improvement* is defined as $\frac{mean(A) - mean(B)}{mean(B)}$.

Threats to Validity: The results reported in this paper are limited to the number and type of EAs used in the experiments. However, we believe these are representative of state-of-art algorithms. Although we used a large number of different subjects (346 complex classes from 117 open-source Java projects), also used by a previous study [17] on test generation, our results may not generalise to other subjects. The range of parameters used in the tuning experiments was limited to only 4 values per EA. Although common or reported as best values, different values might influence the performance of each EA. The two search budgets used in the tuning experiments and in the empirical study are based on EvoSuite's defaults (1 minute), and used by previous studies to assess the performance of EAs with a larger search budget (10 minutes) [21].

[4] At the time of writing this paper, DynaMOSA did not support all the criteria used by EvoSuite.

[5] Class `com.yahoo.platform.yui.compressor.YUICompressor` was excluded from tuning experiments due to a bug in EvoSuite.

[6] Nine classes were discarded from the second experiment due to crashes of EvoSuite.

Table 1: Best population / λ size of each EA per search budget, and single and multiple criteria optimisation. "Br. Cov." column reports the branch coverage per EA, and column "Over. Cov.", the overall coverage of a multiple-criteria optimisation.

	Single-criteria					Multiple-criteria					
		Br.	Avg.	Better	Worse		Br.	Over.	Avg.	Better	Worse
Algorithm	$\|P\|$	Cov.	$\hat{A}_{12}$	$\hat{A}_{12}$	$\hat{A}_{12}$	$\|P\|$	Cov.	Cov.	$\hat{A}_{12}$	$\hat{A}_{12}$	$\hat{A}_{12}$
Search budget of 60 seconds											
Standard GA	10	0.83	0.52	0.75	0.24	100	0.78	0.88	0.52	0.75	0.23
Monotonic GA	25	0.83	0.52	0.76	0.32	100	0.78	0.88	0.52	0.77	0.21
Steady-State GA	100	0.81	0.50	0.72	0.32	100	0.74	0.86	0.53	0.75	0.27
$1 + (\lambda, \lambda)$ GA	50	0.57	0.58	0.70	N/A	50	0.65	0.81	0.53	0.69	0.33
$\mu + \lambda$ EA	1+7	0.84	0.55	0.74	0.21	1+7	0.79	0.89	0.56	0.76	0.28
MOSA	100	0.84	0.51	0.79	0.32	25	0.81	0.62	0.54	0.70	0.21
DynaMOSA	25	0.84	0.51	0.68	0.28	—	—	—	—	—	—
Search budget of 600 seconds											
Standard GA	100	0.86	0.50	0.84	0.21	25	0.84	0.93	0.51	0.76	0.23
Monotonic GA	100	0.87	0.53	0.83	0.22	25	0.84	0.92	0.52	0.80	0.24
Steady-State GA	10	0.85	0.51	0.80	0.23	25	0.79	0.90	0.51	0.79	0.26
$1 + (\lambda, \lambda)$ GA	50	0.57	0.57	0.83	N/A	8	0.75	0.81	0.53	0.85	0.19
$\mu + \lambda$ EA	50+50	0.85	0.49	0.84	0.12	1+1	0.85	0.92	0.53	0.86	0.22
MOSA	50	0.86	0.53	0.88	0.18	10	0.87	0.68	0.54	0.86	0.12
DynaMOSA	25	0.85	0.50	0.83	0.19	—	—	—	—	—	—

A *N/A* worse effect size means there is no other configuration that achieved a significantly higher coverage than the best configuration.

3.2 Parameter Tuning

The execution of an EA requires a number of parameters to be set. As there is not a single best configuration setting to solve all problems [28] in which an EA could be applied, a possible alternative is to tune EA's parameters for a specific problem at hand to find the "best" ones. We largely rely on a previous tuning study [1] in which default values were determined for most parameters of EvoSuite. However, the main distinguishing factor between the algorithms we are considering in this study are μ (i.e., the population size) and λ (i.e., the number of mutations). In particular, we selected common values used in previous studies and reported to be the best for each EA:

- Population size of 10, 25, 50, and 100 for Standard GA, Monotonic GA, SteadyState GA, MOSA, and DynaMOSA.
- λ size of 1, 8 [5], 25, and 50 for $1 + (\lambda, \lambda)$ GA.
- μ size of 1, 7 [13], 25, and 50, and λ size of 1, 7, 25, and 50 for $\mu + \lambda$ EA.

Thus, for Standard GA, Monotonic GA, SteadyState GA, MOSA, DynaMOSA, and $1 + (\lambda, \lambda)$ GA there are 4 different configurations; for $\mu + \lambda$, and as λ must be divisible by μ, there are 8 different configurations (i.e., $1 + 1$, $1 + 7$, $1 + 25$, $1 + 50$, $7 + 7$, $25 + 25$, $25 + 50$, $50 + 50$); i.e., a total of 32 different configurations.

To identify the best population size of each EA, we performed a pairwise comparison of the coverage achieved by using any population size. The population size that achieved a significantly higher coverage more often was selected as the

Table 2: Pairwise comparison of all evolutionary algorithms. "Better than" and "Worse than" give the number of comparisons for which the best EA is statistically significantly (i.e., *p-value*< 0.05) better and worse, respectively. Columns $\hat{A}_{12}$ give the average effect size.

Algorithm	Tourn. Position	Branch Cov.	Overall Cov.	$\hat{A}_{12}$	Better than	$\hat{A}_{12}$	Worse than	$\hat{A}_{12}$
Search budget of 60 seconds – Single-criteria								
Standard GA	3	0.80	—	0.52	223 / 1212	0.79	149 / 1212	0.25
Monotonic GA	2	0.82	—	0.56	299 / 1212	0.78	57 / 1212	0.27
Steady-State GA	4	0.77	—	0.42	112 / 1212	0.76	401 / 1212	0.19
$1 + (\lambda, \lambda)$ GA	5	0.74	—	0.40	53 / 1212	0.73	432 / 1212	0.22
$\mu + \lambda$ EA	1	0.83	—	0.60	387 / 1212	0.79	35 / 1212	0.26
Search budget of 600 seconds – Single-criteria								
Standard GA	3	0.87	—	0.52	129 / 1212	0.87	96 / 1212	0.16
Monotonic GA	2	0.89	—	0.57	192 / 1212	0.89	20 / 1212	0.16
Steady-State GA	4	0.86	—	0.44	50 / 1212	0.80	217 / 1212	0.10
$1 + (\lambda, \lambda)$ GA	5	0.77	—	0.39	14 / 1212	0.82	258 / 1212	0.13
$\mu + \lambda$ EA	1	0.90	—	0.59	224 / 1212	0.88	18 / 1212	0.19
Search budget of 60 seconds – Multiple-criteria								
Standard GA	2	0.77	0.79	0.62	473 / 1212	0.85	98 / 1212	0.20
Monotonic GA	1	0.78	0.80	0.62	470 / 1212	0.85	95 / 1212	0.21
Steady-State GA	4	0.72	0.76	0.43	233 / 1212	0.88	503 / 1212	0.19
$1 + (\lambda, \lambda)$ GA	5	0.53	0.70	0.25	140 / 1212	0.86	896 / 1212	0.10
$\mu + \lambda$ EA	3	0.77	0.79	0.59	493 / 1212	0.84	217 / 1212	0.19
Search budget of 600 seconds – Multiple-criteria								
Standard GA	2	0.84	0.85	0.59	357 / 1212	0.93	112 / 1212	0.11
Monotonic GA	3	0.85	0.85	0.58	345 / 1212	0.93	125 / 1212	0.13
Steady-State GA	5	0.72	0.79	0.33	118 / 1212	0.94	566 / 1212	0.08
$1 + (\lambda, \lambda)$ GA	4	0.62	0.75	0.35	254 / 1212	0.91	623 / 1212	0.05
$\mu + \lambda$ EA	1	0.87	0.86	0.64	437 / 1212	0.93	85 / 1212	0.09

best. Table 1 shows that, for a search budget of 60 seconds and single-criteria, the best population size is different for almost all EAs (e.g., Standard GA works best with a population size of 10, and MOSA with a population size of 100). For a search budget of 600 seconds and multiple-criteria several EAs share the same population size, for example, the best value for Standard GA, Monotonic GA and Steady-State GA on multiple-criteria is 25. Table 1 also reports the average effect size of the best parameter value when compared to all possible parameter values; and the effect size of pairwise comparisons in which the best parameter was significantly better/worse.

3.3 RQ1 – Which evolutionary algorithm works best when using a test archive for partial solutions?

Table 2 summarises the results of a pairwise tournament of all EAs. An EA X is considered to be better than an EA Y if it performs significantly better on a higher number of comparisons. For example, for a search budget of 60 seconds and single-criteria, $1 + (\lambda, \lambda)$ was statistically significantly better than on 53

comparisons, while it was statistically significantly worse on 432 comparisons out of 1,212 – which make it the worst EA. On the other hand, $\mu + \lambda$ was the one with more positive comparisons (387) and the least negative comparisons (just 35) – thus, being the best EA for a search budget of 60 seconds and single-criteria, and for a search budget of 600 seconds on single and multiple-criteria. While it is ranked only third for 60 seconds search budget and multiple-criteria, the coverage is only slightly lower compared to the higher ranked algorithms (0.79 vs. 0.80), with an $\hat{A}_{12}$ effect size of 0.59 averaged over all comparisons.

RQ1: In 3 out of 4 configurations, $\mu + \lambda$ EA is better than the other considered evolutionary algorithms.

3.4 RQ2 – How does evolutionary search compare to random search and random testing?

Table 3 compares the results of each EA with the two random-based techniques, Random search and Random testing. On one hand, Random search performs better than Random testing on single-criteria. However, the overall coverage in the multiple-criteria case is higher for Random testing than Random search. Our conjecture is that, in the multiple-criteria scenario, there are many more trivial coverage goals where the fitness function provides no guidance (thus benefiting Random testing); in contrast, branch coverage goals seem to benefit from the the test archive when generating new individuals (thus benefiting Random search).

On average, EAs achieve higher coverage (either branch-coverage on single-criteria or overall coverage on multiple-criteria) than Random search and Random testing. For instance, for a search budget of 600 seconds and single-criteria, Random search covers 80% of all branches on average and $\mu + \lambda$ EA covers 90% (a relative improvement of +36.5%). This result is different to the earlier study by Shamshiri et al. [24], where random testing achieved similar, and sometimes higher coverage. Our conjecture is that the better performance of the EAs in our evaluation is due to (1) the use of the test archive, and (2) the use of more complex classes in the experiment.

RQ2: Evolutionary algorithms (in particular $\mu + \lambda$ EA) perform better than random search and random testing.

3.5 RQ3 – How does evolution of whole test suites compare to many-objective optimisation of test cases?

Table 4 compares each EA with the many-objective optimisation techniques MOSA and DynaMOSA. Our results confirm and enhance previous studies [17, 19] by evaluating four different EAs (i.e., Standard GA, Steady-State GA, $1+(\lambda, \lambda)$ GA, and $\mu + \lambda$ EA) in addition to Monotonic GA, and show that MOSA and DynaMOSA perform better at optimising test cases than any EA at optimising test suites for single criteria. Although $\mu+\lambda$ achieves a marginally higher average coverage on single criteria (600 seconds) with a relative improvement of +1.6%, it is still slightly worse than MOSA with an average effect size of 0.49.

Table 3: Comparison of evolutionary algorithms and two random-based approaches: Random search and Random testing.

Algorithm	Branch Cov.	Overall Cov.	EA vs. Random search $\hat{A}_{12}$	p	Rel. Impr.	EA vs. Random testing $\hat{A}_{12}$	p	Rel. Impr.
Search budget of 60 seconds – Single-criteria								
Random search	0.78	—	—	—	—	—	—	—
Random testing	0.72	—	—	—	—	—	—	—
Standard GA	0.80	—	0.62	0.26	+15.9%	0.68	0.22	+62.4%
Monotonic GA	0.82	—	0.66	0.23	+21.9%	0.71	0.20	+68.9%
Steady-State GA	0.77	—	0.51	0.27	+2.9%	0.60	0.28	+37.8%
$1 + (\lambda, \lambda)$ GA	0.74	—	0.50	0.32	+1.5%	0.58	0.34	+36.1%
$\mu + \lambda$ EA	0.83	—	0.69	0.22	+23.5%	0.73	0.19	+71.8%
Search budget of 600 seconds – Single-criteria								
Random search	0.80	—	—	—	—	—	—	—
Random testing	0.73	—	—	—	—	—	—	—
Standard GA	0.87	—	0.69	0.19	+29.0%	0.73	0.16	+116.0%
Monotonic GA	0.89	—	0.73	0.16	+35.2%	0.76	0.14	+122.0%
Steady-State GA	0.86	—	0.63	0.22	+20.9%	0.71	0.19	+97.3%
$1 + (\lambda, \lambda)$ GA	0.77	—	0.57	0.39	+8.4%	0.63	0.38	+63.6%
$\mu + \lambda$ EA	0.90	—	0.74	0.16	+36.5%	0.76	0.12	+128.7%
Search budget of 60 seconds – Multiple-criteria								
Random search	0.76	0.65	—	—	—	—	—	—
Random testing	0.71	0.67	—	—	—	—	—	—
Standard GA	0.77	0.79	0.79	0.20	+36.2%	0.84	0.19	+26.7%
Monotonic GA	0.78	0.80	0.80	0.21	+37.6%	0.84	0.18	+28.5%
Steady-State GA	0.72	0.76	0.72	0.23	+29.6%	0.78	0.24	+18.8%
$1 + (\lambda, \lambda)$ GA	0.53	0.70	0.62	0.26	+20.1%	0.62	0.39	+9.7%
$\mu + \lambda$ EA	0.77	0.79	0.76	0.21	+35.9%	0.83	0.20	+25.8%
Search budget of 600 seconds – Multiple-criteria								
Random search	0.70	0.65	—	—	—	—	—	—
Random testing	0.72	0.74	—	—	—	—	—	—
Standard GA	0.84	0.85	0.88	0.17	+64.0%	0.83	0.20	+28.0%
Monotonic GA	0.85	0.85	0.88	0.18	+64.8%	0.83	0.20	+28.7%
Steady-State GA	0.72	0.79	0.79	0.23	+51.4%	0.71	0.29	+17.6%
$1 + (\lambda, \lambda)$ GA	0.62	0.75	0.79	0.30	+49.1%	0.72	0.40	+14.0%
$\mu + \lambda$ EA	0.87	0.86	0.88	0.15	+66.1%	0.84	0.18	+30.6%

In the multiple-criteria scenario (in which we can only compare to MOSA), MOSA performs better than any other EA at optimising branch coverage, but the overall coverage is substantially lower compared to all other EAs. On the one hand, the lower overall coverage is expected since MOSA is not efficient for very large sets of coverage goals (this is what DynaMOSA addresses). However, the fact that branch coverage is nevertheless higher is interesting. A possible conjecture is that this is due to MOSA's slightly different fitness function for branch coverage [19], which includes the approach level (whereas whole test suite optimisation considers only branch distances).

RQ3: MOSA improves over EAs for individual criteria; for multiple-criteria it achieves higher branch coverage even though overall coverage is lower.

Table 4: Comparison of evolutionary algorithms on whole test suites optimisation and many-objective optimisation algorithms of test cases.

Algorithm	Branch Cov.	Overall Cov.	EA vs. MOSA $\hat{A}_{12}$	EA vs. MOSA p	EA vs. MOSA Rel. Impr.	EA vs. DynaMOSA $\hat{A}_{12}$	EA vs. DynaMOSA p	EA vs. DynaMOSA Rel. Impr.
Search budget of 60 seconds – Single-criteria								
MOSA	0.84	—	—	—	—	—	—	—
DynaMOSA	0.85	—	—	—	—	—	—	—
Standard GA	0.80	—	0.39	0.27	-3.6%	0.37	0.28	-6.0%
Monotonic GA	0.82	—	0.43	0.26	-0.4%	0.41	0.28	-2.3%
Steady-State GA	0.77	—	0.30	0.19	-9.7%	0.28	0.19	-10.7%
$1 + (\lambda, \lambda)$ GA	0.74	—	0.31	0.26	-12.5%	0.29	0.25	-14.3%
$\mu + \lambda$ EA	0.83	—	0.46	0.28	+0.8%	0.44	0.29	-1.5%
Search budget of 600 seconds – Single-criteria								
MOSA	0.90	—	—	—	—	—	—	—
DynaMOSA	0.91	—	—	—	—	—	—	—
Standard GA	0.87	—	0.42	0.24	-3.2%	0.40	0.23	-4.6%
Monotonic GA	0.89	—	0.47	0.24	+0.2%	0.44	0.23	-1.4%
Steady-State GA	0.86	—	0.38	0.22	-3.5%	0.36	0.21	-5.1%
$1 + (\lambda, \lambda)$ GA	0.77	—	0.34	0.37	-14.3%	0.33	0.35	-15.6%
$\mu + \lambda$ EA	0.90	—	0.49	0.22	+1.6%	0.47	0.23	-0.7%
Search budget of 60 seconds – Multiple-criteria								
MOSA	0.80	0.58	—	—	—	—	—	—
DynaMOSA	—	—	—	—	—	—	—	—
Standard GA	0.77	0.79	0.71	0.18	+8737.7%	—	—	—
Monotonic GA	0.78	0.80	0.71	0.17	+9069.9%	—	—	—
Steady-State GA	0.72	0.76	0.63	0.17	+9058.6%	—	—	—
$1 + (\lambda, \lambda)$ GA	0.53	0.70	0.59	0.21	+7941.9%	—	—	—
$\mu + \lambda$ EA	0.77	0.79	0.70	0.17	+9071.2%	—	—	—
Search budget of 600 seconds – Multiple-criteria								
MOSA	0.87	0.71	—	—	—	—	—	—
DynaMOSA	—	—	—	—	—	—	—	—
Standard GA	0.84	0.85	0.64	0.19	+772.4%	—	—	—
Monotonic GA	0.85	0.85	0.64	0.20	+773.4%	—	—	—
Steady-State GA	0.72	0.79	0.52	0.19	+694.6%	—	—	—
$1 + (\lambda, \lambda)$ GA	0.62	0.75	0.56	0.27	+632.7%	—	—	—
$\mu + \lambda$ EA	0.87	0.86	0.67	0.18	+769.5%	—	—	—

4 Related Work

Although a common approach in search-based testing is to use genetic algorithms, numerous other algorithms have been proposed in the domain of nature-inspired algorithms, as no algorithm can be best on all domains [28]. Many researchers compared evolutionary algorithms to solve problems in domains outside software engineering [2, 27, 29]. Within search-based software engineering, comparative studies have been conducted in several domains such as discovery of software architectures [20], pairwise testing of software product lines [15], or finding subtle higher order mutants [16].

In the context of test data generation, Harman and McMinn [12] empirically compared GA, Random testing and Hill Climbing for structural test data generation. While their results indicate that sophisticated evolutionary algorithms can often be outperformed by simpler search techniques, there are more complex scenarios, for which evolutionary algorithms are better suited. Ghani et al. [11] compared Simulated Annealing (SA) and GA for the test data generation for Matlab Simulink models, and their results show that GA performed slightly better than SA. Sahin and Akay [23] evaluated Particle Swarm Optimisation (PSO), Differential Evolution (DE), Artificial Bee Colony, Firefly Algorithm and Random search algorithms on software test data generation benchmark problems, and concluded that some algorithms performs better than others depending on the characteristics of the problem. Varshney and Mehrotra [26] proposed a DE-based approach to generate test data that cover data-flow coverage criteria, and compared the proposed approach to Random search, GA and PSO with respect to number of generations and average percentage coverage. Their results show that the proposed DE-based approach is comparable to PSO and has better performance than Random search and GA. In contrast to these studies, we consider unit test generation, which arguably is a more complex scenario than test data generation, and in particular local search algorithms are rarely applied.

Although often newly proposed algorithms are compared to random search as a baseline (usually showing clear improvements), there are some studies that show that random search can actually be very efficient for test generation. In particular, Shamshiri et al. [24] compared GA against Random search for generating test suites, and found almost no difference between the coverage achieved by evolutionary search compared to random search. They observed that GAs covers more branches when standard fitness functions provide guidance, but most branches of the analyzed projects provided no such guidance. Similarly, Sahin and Akay [23] showed that Random search is effective on simple problems.

To the best of our knowledge, no study has been conducted to evaluate several different evolutionary algorithms in a whole test suite generation context and considering a large number of complex classes. As can be seen from this overview of comparative studies, it is far from obvious what the best algorithm is, since there are large variations between different search problems.

5 Conclusions

Although evolutionary algorithms are commonly applied for whole test suite generation, there is a lack of evidence on the influence of different algorithms. Our study yielded the following key results:

- The choice of algorithm can have a substantial influence on the performance of whole test suite optimisation, hence tuning is important. While EvoSuite provides tuned default values, these values may not be optimal for different flavours of evolutionary algorithms.
- EvoSuite's default algorithm, a Monotonic GA, is an appropriate choice for EvoSuite's default configuration (60 seconds search budget, multiple

criteria). However, for other search budgets and optimisation goals, other algorithms such as a $\mu + \lambda$ EA may be a better choice.
- Although previous studies showed little benefit of using a GA over random testing, our study shows that on complex classes and with a test archive, evolutionary algorithms are superior to random testing and random search.
- The Many Objective Sorting Algorithm (MOSA) is superior to whole test suite optimisation; it would be desirable to extend EvoSuite so that DynaMOSA supports all coverage criteria.

It would be of interest to extend our experiments to further search algorithms. In particular, the use of other non-functional attributes such as readability [3] suggests the exploration of multi-objective algorithms. Considering the variation of results with respect to different configurations and classes under test, it would also be of interest to use these insights to develop hyper-heuristics that select and adapt the optimal algorithm to the specific problem at hand.

Acknowledgments. This work is supported by EPSRC project EP/N023978/1, São Paulo Research Foundation (FAPESP) grant 2015/26044-0, and the National Research Fund, Luxembourg (FNR/P10/03).

References

1. Arcuri, A., Fraser, G.: Parameter tuning or default values? an empirical investigation in search-based software engineering. Empirical Software Engineering 18(3), 594–623 (2013)
2. Basak, A., Lohn, J.: A comparison of evolutionary algorithms on a set of antenna design benchmarks. In: de la Fraga, L.G. (ed.) 2013 IEEE Conference on Evolutionary Computation. vol. 1, pp. 598–604. Cancun, Mexico (June 20-23 2013)
3. Daka, E., Campos, J., Fraser, G., Dorn, J., Weimer, W.: Modeling Readability to Improve Unit Tests. In: Proc. of the Joint Meeting on Foundations of Software Engineering. pp. 107–118. ESEC/FSE 2015, ACM, New York, NY, USA (2015)
4. Deb, K., Agrawal, S., Pratap, A., Meyarivan, T.: A fast elitist non-dominated sorting genetic algorithm for multi-objective optimization: NSGA-II. In: International Conference on Parallel Problem Solving From Nature. pp. 849–858. Springer (2000)
5. Doerr, B., Doerr, C., Ebel, F.: From black-box complexity to designing new genetic algorithms. Theoretical Computer Science 567, 87–104 (2015)
6. Fraser, G., Arcuri, A.: EvoSuite: Automatic Test Suite Generation for Object-oriented Software. In: Proc. ESEC/FSE. pp. 416–419. ACM (2011)
7. Fraser, G., Arcuri, A.: Handling test length bloat. Software Testing, Verification and Reliability 23(7), 553–582 (2013)
8. Fraser, G., Arcuri, A.: Whole test suite generation. IEEE Transactions on Software Engineering 39(2), 276–291 (2013)
9. Fraser, G., Arcuri, A.: A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite. ACM Transactions on Software Engineering and Methodology (TOSEM) 24(2), 8:1–8:42 (Dec 2014)
10. Gay, G.: The fitness function for the job: Search-based generation of test suites that detect real faults. In: Software Testing, Verification and Validation (ICST), 2017 IEEE 10th International Conference on. IEEE (2017)
11. Ghani, K., Clark, J.A., Zhan, Y.: Comparing algorithms for search-based test data generation of matlab simulink models. In: 2009 IEEE Congress on Evolutionary Computation. pp. 2940–2947 (May 2009)

12. Harman, M., McMinn, P.: A theoretical & empirical analysis of evolutionary testing and hill climbing for structural test data generation. In: Proceedings of the International Symposium on Software Testing and Analysis. pp. 73–83. ACM (2007)
13. Jansen, T., De Jong, K.A., Wegener, I.: On the choice of the offspring population size in evolutionary algorithms. Evolutionary Computation 13(4), 413–440 (2005)
14. Karnopp, D.C.: Random search techniques for optimization problems. Automatica 1(2-3), 111–121 (1963)
15. Lopez-Herrejon, R.E., Ferrer, J., Chicano, F., Egyed, A., Alba, E.: Comparative analysis of classical multi-objective evolutionary algorithms and seeding strategies for pairwise testing of software product lines. In: Proceedings of the IEEE Congress on Evolutionary Computation, CEC. pp. 387–396 (2014)
16. Omar, E., Ghosh, S., Whitley, D.: Comparing search techniques for finding subtle higher order mutants. In: Proceedings of the Conference on Genetic and Evolutionary Computation. pp. 1271–1278. GECCO '14, ACM (2014)
17. Panichella, A., Kifetew, F., Tonella, P.: Automated Test Case Generation as a Many-Objective Optimisation Problem with Dynamic Selection of the Targets. IEEE Transactions on Software Engineering PP(99), 1–1 (2017)
18. Panichella, A., Kifetew, F., Tonella, P.: Automated test case generation as a many-objective optimisation problem with dynamic selection of the targets. IEEE Transactions on Software Engineering (2017)
19. Panichella, A., Kifetew, F.M., Tonella, P.: Reformulating branch coverage as a many-objective optimization problem. In: Software Testing, Verification and Validation (ICST), 2015 IEEE 8th International Conference on. pp. 1–10. IEEE (2015)
20. Ramírez, A., Romero, J.R., Ventura, S.: A comparative study of many-objective evolutionary algorithms for the discovery of software architectures. Empirical Softw. Engg. 21(6), 2546–2600 (dec 2016)
21. Rojas, J.M., Campos, J., Vivanti, M., Fraser, G., Arcuri, A.: Combining Multiple Coverage Criteria in Search-Based Unit Test Generation. In: Barros, M., Labiche, Y. (eds.) Search-Based Software Engineering (SSBSE), Lecture Notes in Computer Science, vol. 9275, pp. 93–108. Springer International Publishing (2015)
22. Rojas, J.M., Vivanti, M., Arcuri, A., Fraser, G.: A detailed investigation of the effectiveness of whole test suite generation. Empirical Software Engineering (2016)
23. Sahin, O., Akay, B.: Comparisons of metaheuristic algorithms and fitness functions on software test data generation. Applied Soft Computing 49, 1202 – 1214 (2016)
24. Shamshiri, S., Rojas, J.M., Fraser, G., McMinn, P.: Random or genetic algorithm search for object-oriented test suite generation? In: Proceedings of the Conference on Genetic and Evolutionary Computation. pp. 1367–1374. ACM (2015)
25. Ter-Sarkisov, A., Marsland, S.R.: Convergence properties of (μ+ λ) evolutionary algorithms. In: AAAI (2011)
26. Varshney, S., Mehrotra, M.: A differential evolution based approach to generate test data for data-flow coverage. In: 2016 International Conference on Computing, Communication and Automation (ICCCA). pp. 796–801 (April 2016)
27. Wolfram, M., Marten, A.K., Westermann, D.: A comparative study of evolutionary algorithms for phase shifting transformer setting optimization. In: 2016 IEEE International Energy Conference (ENERGYCON). pp. 1–6 (April 2016)
28. Wolpert, D.H., Macready, W.G.: No free lunch theorems for optimization. IEEE transactions on evolutionary computation 1(1), 67–82 (1997)
29. Zitzler, E., Deb, K., Thiele, L.: Comparison of multiobjective evolutionary algorithms: Empirical results. Evolutionary computation 8(2), 173–195 (2000)