An Empirical Evaluation of Evolutionary Algorithms for Unit Test Suite Generation

José Campos[a,*], Yan Ge[a], Nasser Albunian[a], Gordon Fraser[a,b], Marcelo Eler[c], Andrea Arcuri[d,e]

[a] *Department of Computer Science, The University of Sheffield, UK*
[b] *Chair of Software Engineering II, University of Passau, Germany*
[c] *University of São Paulo, Brazil*
[d] *Westerdals Oslo School of Arts, Communication and Technology, Norway*
[e] *University of Luxembourg, Luxembourg*

Abstract

Context: Evolutionary algorithms have been shown to be effective at generating unit test suites optimised for code coverage. While many specific 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 choice of evolutionary algorithm has to date seen less attention in the literature.

Objective: Since it is theoretically impossible to design an algorithm that is the best on all possible problems, a common approach in software engineering problems is to first try the most common algorithm, 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. The objective of this paper is to perform this analysis, in order to shed light on the influence of the search algorithm applied for unit test generation.

Method: We empirically evaluate thirteen different evolutionary algorithms and two random approaches on a selection of non-trivial open source classes. All algorithms are implemented in the EvoSuite test generation tool, which

*Corresponding author
Email addresses: jose.campos@sheffield.ac.uk (José Campos), yge5@sheffield.ac.uk (Yan Ge), nmalbunian1@sheffield.ac.uk (Nasser Albunian), gordon.fraser@uni-passau.de (Gordon Fraser), marceloeler@usp.br (Marcelo Eler), arcand@westerdals.no (Andrea Arcuri)

includes recent optimisations such as the use of an archive during the search and optimisation for multiple coverage criteria.

Results: Our study shows that the use of a test archive makes evolutionary algorithms clearly better than random testing, and it confirms that the DynaMOSA many-objective search algorithm is the most effective algorithm for unit test generation.

Conclusions: Our results show that the choice of algorithm can have a substantial influence on the performance of whole test suite optimisation. Although we can make a recommendation on which algorithm to use in practice, no algorithm is clearly superior in all cases, suggesting future work on improved search algorithms for unit test generation.

Keywords: Evolutionary algorithms, Test suite generation, Empirical study

---

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 [1]. Tools like EvoSuite [2] have been shown to be effective in achieving code coverage on different types of software [3].

Since the original introduction of whole test suite generation [4], many different optimisations 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 [5]. Optimisation now no longer focuses on individual coverage criteria, but combinations of multiple different coverage criteria [6, 7]. To cope with the resulting larger number of coverage goals, evolutionary search can be supported with archives [8] that keep track of useful solutions encountered throughout the search. To improve effectiveness, whole test suite optimisation has been re-

18 formulated as a many-objective optimisation problem [9]. In the context of these
19 developments, one aspect of whole test suite generation remains largely unex-
20 plored: What is the influence of the specific flavour of evolutionary algorithms
21 applied to evolve test suites?

22 In this paper, we aim to shed light on the influence of the different evolu-
23 tionary algorithms in whole test suite generation, to find out whether the choice
24 of algorithm is important, and which one should be used. By using a large set
25 of complex Java classes as case study, and the EvoSuite [2] search-based test
26 generation tool, we specifically investigate:

27 RQ1: Which archive-based single-objective evolutionary algorithm performs best?

28 RQ2: How does evolutionary search compare to random search and random
29 testing?

30 RQ3: Which archive-based many-objective evolutionary algorithm performs best?

31 RQ4: How does evolution of whole test suites compare to many-objective opti-
32 misation of test cases?

33 We investigate each of these questions in the light of individual and multi-
34 ple coverage criteria as optimisation objectives. This paper extends an earlier
35 study [10], where we compared seven evolutionary algorithms and two random
36 approaches. Our experiments now cover five additional algorithms, for a total
37 of 13 different evolutionary algorithms, and corroborate the original findings:
38 In most cases a simple (μ, λ) Evolutionary Algorithm (EA) is better than other,
39 more complex algorithms. In most cases, the variants of EAs and GAs are also
40 clearly better than random search and random testing, when a test archive is
41 used. This study also extends the previous study with experiments using many-
42 objective search algorithms using multiple criteria, and our experiments con-
43 firm that many-objective search, in particular the DynaMOSA algorithm [11],
44 achieves higher branch coverage, even in the case of optimisation for multiple
45 criteria, than all the other evaluated single/many-objective evolutionary algo-
46 rithms.

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 genetics-inspired operations such as *crossover*, which merges genetic material from at least two individuals to yield new offspring, *mutation*, which independently changes the elements of an individual with a low probability, and *selection* which chooses individuals for reproduction, preferring better, fitter individuals. 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 (e.g., Differential Evolution [12], PAES [13], Coral Reef Optimisation [14], etc.) 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. The length of a sequence of calls is variable, and there can be dependencies between statements. For example, one statement may define a variable used as a parameter for a call later in the call sequence. Standard types of statements in such sequences are definitions of primitive variables (e.g., integers or strings), calls to constructors to instantiate objects, and method calls on these objects.

Listing 1: Example of a test suite (with only a subset of test cases) automatically generated by EvoSuite [2] for `ArrayByteList` class of project `Apache Commons Collections`.

```
public class ArrayByteList_ESTest {
  @Test
  public void test0() throws Throwable {
    ArrayByteList arrayByteList0 = new ArrayByteList();
    arrayByteList0.ensureCapacity(550);
    assertEquals(0, arrayByteList0.size());
  }
```

```
  @Test
  public void test1() throws Throwable {
    ArrayByteList arrayByteList0 = new ArrayByteList();
    arrayByteList0.add((byte) (-113));
    arrayByteList0.add(0, (byte)0);
    byte byte0 = arrayByteList0.removeElementAt(0);
    assertEquals(1, arrayByteList0.size());
    assertEquals((byte)0, byte0);
  }
}
```

Crossover on test suites is based on exchanging test cases between sets [1]. Mutation adds/modifies tests to suites, and adds/removes/changes statements within tests. The mutations applied at test case level need to ensure that test cases remain valid (e.g., when adding a new call there need to be suitable parameter objects defined earlier in the sequence).

Although standard selection techniques are largely used (e.g., rank or tournament selection), the variable size representation (the number of statements in a test and number of test cases in a suite can vary) requires modification to avoid bloat [15]; this is typically achieved by ranking individuals with identical fitness based on their length, and then using rank selection.

Standard whole test suite optimisation algorithms use test suites as individuals, since they are targeting coverage of all goals at the same time. Existing many-objective algorithms, on the other hand, aim to optimise an individual test for each distinct coverage goal, and so the search representation in this case is test cases. In this case, the test case mutation operators used when test suites are mutated are still used and bloat control is also active during selection. Crossover, however, needs to ensure that sequences of calls remain valid (i.e., all dependency variables need to exist). This is typically achieved by using repair actions when attaching to subsequences.

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.

To provide a gradient to the search, most common fitness functions rely on the approach level and branch distance metrics [16, 17]. The approach level $\mathcal{A}(t, x)$ for a given test t on a coverage goal $x \in X$ (for any given set of coverage goals X) is the minimal number of control dependent edges in the control dependency graph between the target goal x and the control flow path represented by the test case t. That is, it estimates the approximation between the execution path of a given test input and the target. The branch distance $d(t, x)$ heuristically quantifies how far a branch (i.e., the control flow edge resulting from a true/false evaluation of an if-condition) is from being evaluated to true or to false. When optimising for individual coverage goals, the fitness function is usually a combination of approach level and branch distance. For example, for branch coverage the fitness function to minimize the approach level and branch distance between a test t and a branch coverage goal x is defined as:

$$f(t, x) = \mathcal{A}(t, x) + \nu(d(t, x)) \tag{1}$$

where ν is any normalizing function in the range $[0, 1]$ [18]. When evolving test suites, however, one does not target individual goals but *all* coverage goals. For example, for branch coverage the resulting fitness function aims to minimise the branch distance of *all* branches B in the program under test. Thus, the fitness function for a test suite T and a set of branches B is:

$$f_{BC}(T, B) = \sum_{b \in B} d(T, b) \tag{2}$$

131 where $d(T, b)$ is defined as:

$$d(T,b) = \begin{cases} 0 & \text{if branch } b \text{ has been covered,} \\ \nu(d_{min}(t \in T, b)) & \text{if the predicate has been} \\ & \text{executed at least twice,} \\ 1 & \text{otherwise.} \end{cases} \quad (3)$$

132 More recently, there is a trend to optimise for multiple coverage criteria at
133 the same time. Since coverage criteria usually do not represent conflicting goals,
134 it is possible to combine fitness functions with a weighted linear combination [6].
135 However, the increased number of coverage goals may affect the performance of
136 the EA. To counter these effects, it is possible to store tests for covered goals in
137 an archive [8], and then to dynamically adapt the fitness function to optimise
138 only for the remaining uncovered goals. That is, during fitness evaluation, if
139 a test case is found that newly covers a non-covered goal (e.g., branch, line,
140 etc.), the covering test case and the covered goal are added to an archive. The
141 fitness function is then optimised to only take into account the remaining goals.
142 Note that this optimisation is only performed at the end of an iteration, i.e.,
143 only after evaluating all individuals, and not during the evaluation of one test
144 suite or during the creation of a new population, as it would make fitness values
145 between individuals inconsistent. Once the search ends, the best individual of
146 the EA is no longer the best individual of the search population, but a test suite
147 composed by all the tests in the archive. Besides optimising fitness functions
148 to make use of the archive, search operators can also be adapted to make use
149 of the test archive; for example, new tests may be created by mutating tests in
150 the archive rather than randomly generating completely new tests.

151 *2.3. Random Search & Random Testing*

152 Random search is a baseline search strategy which does not use crossover,
153 mutation, or selection, but a simple replacement strategy [19]. Random search
154 consists of repeatedly sampling candidates from the search space; the previous

155 candidate is replaced if the fitness of the new sampled individual is better. Ran-
156 dom search can make use of a test archive by changing the sampling procedure
157 as indicated above. It has been shown that in unit test generation, due to the
158 flat fitness landscapes and often simple search problems, random search is often
159 as effective as EAs, and sometimes even better [5].

160 *Random testing* is a variant of random search in test generation which builds
161 a test suite incrementally. Test cases (rather than test suites) are sampled
162 individually, and if a test case improves the coverage of the test suite, it is
163 retained in the test suite, otherwise it is discarded. This incremental process
164 does not benefit from using an archive, because every sampled test case that
165 covers a goal that has not been covered is added to the test suite.

166 *2.4. Genetic Algorithms*

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

180 Many variants of the Standard GA have been proposed to improve effective-
181 ness. Specifically, we consider a *monotonic* version of the Standard GA (Al-
182 gorithm 2) which, after mutating and evaluating each offspring, only includes
183 either the best offspring or the best parent in the next population (whereas the
184 Standard GA includes both offspring in the next population regardless of their

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

```
1:  P ← GenerateRandomPopulation(p_s)
2:  PerformFitnessEvaluation(δ, P)
3:  while ¬C do
4:      N_P ← { } ∪ Elitism(P)
5:      while |N_P| < p_s do
6:          p_1, p_2 ← Selection(s_f, P)
7:          o_1, o_2 ← Crossover(c_f, c_p, p_1, p_2)
8:          Mutation(m_f, m_p, o_1)
9:          Mutation(m_f, m_p, o_2)
10:         N_P ← N_P ∪ {o_1, o_2}
11:     end while
12:     P ← N_P
13:     PerformFitnessEvaluation(δ, P)
14: end while
15: return P
```

185 fitness value). Another variation of the Standard GA is a *Steady State* GA (Al-
186 gorithm 3), which uses the same replacement strategy as the Monotonic GA,
187 but instead of creating a new population of offspring, the offspring replace the
188 parents from the current population immediately after the mutation phase.

189 A Breeder GA [20] (Algorithm 4) is a GA variant that does not aim to mimic
190 Darwinian evolutionary, but instead tries to mimic breeding mechanism, as used
191 for example in livestock. This is done by selecting a fixed percentage (e.g., 50%)
192 of the best individuals of the total population as gene pool, and then uniformly
193 sampling from this pool for reproduction (using standard crossover and muta-
194 tion) when generating a new population. In addition, the best q individuals
195 (e.g., 1) survive in terms of elitism.

196 The Cellular GA [21] differs from the Standard GA by considering a struc-
197 tured population which influences selection. For example, individuals can be set
198 in a toroidal d-dimensional grid where each individual takes a place per a grid
199 (i.e., cell) and belongs to an overlapped neighbourhood. The grid of individuals
200 can have different number of dimensions; common values are one-dimensional

Algorithm 2 Monotonic 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

```
1: P ← GenerateRandomPopulation(p_s)
2: PerformFitnessEvaluation(δ, P)
3: while ¬C do
4:     N_P ← { } ∪ Elitism(P)
5:     while |N_P| < p_s do
6:         p_1, p_2 ← Selection(s_f, P)
7:         o_1, o_2 ← Crossover(c_f, c_p, p_1, p_2)
8:         Mutation(m_f, m_p, o_1)
9:         Mutation(m_f, m_p, o_2)
10:        PerformFitnessEvaluation(δ, o_1)
11:        PerformFitnessEvaluation(δ, o_2)
12:        if Best(o_1, o_2) is better than Best(p_1, p_2) then
13:            N_P ← N_P ∪ {o_1, o_2}
14:        else
15:            N_P ← N_P ∪ {p_1, p_2}
16:        end if
17:    end while
18:    P ← N_P
19: end while
20: return P
```

201 (i.e., ring) or two-dimensional grids. In the case of a bi-dimensional grid, differ-
202 ent shapes (i.e., models) of a neighbourhood can be defined. For example, the
203 linear 5 model considers the individual itself and the individuals in its north,
204 south, east, and west positions as neighbours of the current one.

205 Each individual is only allowed to interact with its neighbours and therefore
206 the search operators are only applied on the individuals of one neighbourhood.
207 First, two parents p_1, p_2 are selected among the neighbours of one individual p
208 according to a selection criterion. Then, crossover is performed to create two
209 new individuals o_1, o_2, which are then evaluated. The best individual (o) among
210 the two new generated individuals is mutated and evaluated. Finally, if fitness
211 value of p is better than the fitness value of o, the former is included in the
212 next population, otherwise the later is included in the next population. Due

Algorithm 3 Steady-State 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

1: $P \leftarrow$ GenerateRandomPopulation(p_s)
2: PerformFitnessEvaluation(δ, P)
3: **while** $\neg C$ **do**
4: $\quad p_1, p_2 \leftarrow$ Selection(s_f, P)
5: $\quad o_1, o_2 \leftarrow$ Crossover(c_f, c_p, p_1, p_2)
6: $\quad$ Mutation(m_f, m_p, o_1)
7: $\quad$ Mutation(m_f, m_p, o_2)
8: $\quad$ PerformFitnessEvaluation(δ, o_1)
9: $\quad$ PerformFitnessEvaluation(δ, o_2)
10: $\quad$ **if** Best(o_1, o_2) is better than Best(p_1, p_2) **then**
11: $\qquad P \leftarrow P \setminus \{p_1, p_2\} \cup \{o_1, o_2\}$
12: $\quad$ **else**
13: $\qquad P \leftarrow P \setminus \{o_1, o_2\} \cup \{p_1, p_2\}$
14: $\quad$ **end if**
15: **end while**
16: **return** P

213 to the neighbourhood overlapping, the Cellular GA motivates slow diffusion of
214 solutions through the population and thus the exploration of the search space
215 and the exploitation inside each neighbourhood are promoted during the search.

216 The $1 + (\lambda, \lambda)$ GA (Algorithm 6), introduced by Doerr et al. [22], starts by
217 generating a random population of size 1. Then, mutation is used to create
218 λ different mutated versions of the current individual. Mutation is applied
219 with a high mutation probability, defined as $m_p = \frac{k}{n}$, where k is typically
220 greater than one, which allows, on average, more than one gene to be mutated
221 per chromosome. Then, uniform crossover is applied to the parent and best
222 generated mutant to create λ offspring. While a high mutation probability is
223 intended to support faster exploration of the search space, a uniform crossover
224 between the best individual among the λ mutants and the parent was suggested
225 to repair the defects caused by the aggressive mutation. Then all offspring are
226 evaluated and the best one is selected. If the best offspring is better than the
227 parent, the population of size one is replaced by the best offspring. $1 + (\lambda, \lambda)$

Algorithm 4 Breeder 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

```
1: P ← GenerateRandomPopulation(p_s)
2: PerformFitnessEvaluation(δ, P)
3: while ¬C do
4:     N_P ← { } ∪ Elitism(P)
5:     P' ← Truncate(P)
6:     while |N_P| < p_s do
7:         p_1 ← SelectRandom(P')
8:         p_2 ← SelectRandom(P')
9:         o_1, o_2 ← Crossover(c_f, c_p, p_1, p_2)
10:        Mutation(m_f, m_p, o_1)
11:        Mutation(m_f, m_p, o_2)
12:        o ← SelectRandom(o_1, o_2)
13:        N_P ← N_P ∪ {o}
14:    end while
15:    P ← N_P
16:    PerformFitnessEvaluation(δ, P)
17: end while
18: return P
```

228 GA could be very expensive for large values of λ, as fitness has to be evaluated
229 after mutation and after crossover.

230 *2.5. Evolution Strategies*

231 Evolution strategies, dating back to Rechenberg [23], primarily use mutation
232 and selection as search operators. Algorithm 7 shows a basic $(\mu+\lambda)$ Evolution-
233 ary Algorithm (EA), where a population of μ individuals is evolved by generating
234 λ individuals in each generation through mutation of the μ individuals in the
235 population. Among the different $(\mu+\lambda)$ EA versions, two common settings are
236 $(1+\lambda)$ EA and $(1+1)$ EA, where the population size is 1, and the number of
237 offspring is also limited to 1 for the $(1+1)$ EA. In the $(\mu+\lambda)$ EA, after the
238 mutation step the best μ individuals out of the previous generation and the
239 offspring are selected and kept as the new population. A variant of this is a

Algorithm 5 Cellular 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, Neighbourhood model n_m
Output: Population of optimised individuals P

```
1:  P ← GenerateRandomPopulation(p_s)
2:  PerformFitnessEvaluation(δ, P)
3:  while ¬C do
4:      N_P ← { }
5:      for all p ∈ P do
6:          N_B ← GetNeighbourhood(p, P, n_m)
7:          p_1, p_2 ← Selection(s_f, N_B)
8:          o_1, o_2 ← Crossover(c_f, c_p, p_1, p_2)
9:          PerformFitnessEvaluation(δ, o_1)
10:         PerformFitnessEvaluation(δ, o_2)
11:         o ← Best(o_1, o_2)
12:         Mutation(m_f, m_p, o)
13:         PerformFitnessEvaluation(δ, o)
14:         N_P ← N_P ∪ Best(o, p)
15:     end for
16:     P ← N_P
17: end while
18: return P
```

240 (μ, λ) EA (Algorithm 8), where the μ new individuals are only selected from
241 the offspring, and the parents are discarded.

242 *2.6. Chemical Reaction Optimisation (CRO)*

243 The Chemical Reaction Optimisation (CRO) [24] (Algorithm 9) is a meta-
244 heuristic algorithm which incorporates the best of a population-based algorithm
245 (e.g., as genetic algorithms) and the simulated annealing [25] local search. CRO
246 is inspired by the nature of chemical reactions, i.e., the process of transform-
247 ing a set of unstable molecules in a container (similar to a population in GAs)
248 to a set of stable molecules. The basic unit in CRO is a molecule (similar to
249 a chromosome in GAs) and it is characterised by its potential energy (corre-
250 sponding to the fitness value in GAs), its kinetic energy, and the number of
251 collisions that is has been involved in. To manipulate individuals and explore

Algorithm 6 $1 + (\lambda, \lambda)$ Genetic Algorithm

Input: Stopping condition C, Fitness function δ, Offspring size λ, Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p

Output: Best individual p

```
1: p ← GenerateRandomIndividual()
2: PerformFitnessEvaluation(δ, p)
3: while ¬C do
4:     M ← { }
5:     for i ← 1, λ do
6:         o ← Mutation(m_f, m_p, p)
7:         PerformFitnessEvaluation(δ, o)
8:         M ← M ∪ {o}
9:     end for
10:    p′ ← Best(M)
11:    O ← { }
12:    for i ← 1, λ/2 do
13:        o_1, o_2 ← Crossover(c_f, c_p, p, p′)
14:        PerformFitnessEvaluation(δ, o_1)
15:        PerformFitnessEvaluation(δ, o_2)
16:        O ← O ∪ {o_1, o_2}
17:    end for
18:    p′ ← Best(O)
19:    if p′ is better than p then
20:        p ← p′
21:    end if
22: end while
23: return p
```

252 the search space, CRO iteratively applies chemical reactions, which are similar
253 to the search operations in a GA.

254 There are four types of reactions, each occurring in each iteration of CRO:
255 *on-wall ineffective collision* and *inter-molecular ineffective collision* are used
256 as local search operators, and on the other hand *decomposition* and *synthe-*
257 *sis* are used as global search operators. An on-wall ineffective collision occurs
258 when a molecule hits a wall of the container and stays as a single molecule. In
259 the process some of molecule's kinetic energy is transferred to the container.
260 An inter-molecular ineffective collision occurs when multiple molecules (typi-
261 cally two) collide with each other. Although this collision could be modelled

Algorithm 7 $(\mu + \lambda)$ Evolutionary Algorithm

Input: Stopping condition C, Fitness function δ, Population size μ, Offspring size λ, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P

```
1: P ← GenerateRandomPopulation(μ)
2: PerformFitnessEvaluation(δ, P)
3: while ¬C do
4:     O ← { }
5:     for all p ∈ P do
6:         for i ← 1, λ/μ do
7:             o ← Mutation(m_f, m_p, p)
8:             O ← O ∪ {o}
9:         end for
10:    end for
11:    PerformFitnessEvaluation(δ, O)
12:    P ← select best μ individuals from P ∪ O
13: end while
14: return P
```

Algorithm 8 (μ, λ) Evolutionary Algorithm

Input: Stopping condition C, Fitness function δ, Population size μ, Offspring size λ, Mutation function m_f, Mutation probability m_p
Output: Population of optimised individuals P

```
1: P ← GenerateRandomPopulation(μ)
2: PerformFitnessEvaluation(δ, P)
3: while ¬C do
4:     O ← { }
5:     for all p ∈ P do
6:         for i ← 1, λ/μ do
7:             o ← Mutation(m_f, m_p, p)
8:             O ← O ∪ {o}
9:         end for
10:    end for
11:    PerformFitnessEvaluation(δ, O)
12:    P ← select best μ individuals from O
13: end while
14: return P
```

262 as two independent on-wall ineffective collisions, the energy is handled in a dif-
263 ferent way, as molecules could exchange energy. A decomposition occurs when
264 a molecule hits a wall of a container, but rather than bouncing away as a sin-
265 gle molecule as in an inter-molecular ineffective collision, it breaks into several

Algorithm 9 Chemical Reaction Optimisation (CRO)

Input: Stopping condition C, Fitness function δ, Population size p_s (i.e., number of molecules), Crossover function c_f, Crossover probability c_p, Mutation function m_f, Mutation probability m_p, Collision rate c_r, Decomposition threshold d_t, Synthesis threshold s_t, Initial kinetic energy k_e, Kinetic energy loss rate k_r,

Output: Population of optimised molecules P

1: $P \leftarrow$ GenerateRandomPopulation(p_s, k_e)
2: PerformFitnessEvaluation(δ, P)
3: **while** $\neg C$ **do**
4: $r \leftarrow$ Random(0,1)
5: **if** $r > c_r$ **then**
6: $m \leftarrow$ Random(P)
7: **if** NumberCollisions(m) $> d_t$ **then**
8: Decomposition(δ, m_f, m_p, P, m)
9: **else**
10: OnwallIneffectiveCollision($\delta, m_f, m_p, k_r, P, m$)
11: **end if**
12: **else**
13: $m_1, m_2 \leftarrow$ Random(P)
14: **if** SynthesisThreshold(m_1) $\leq s_t$ **and**
15: SynthesisThreshold(m_2) $\leq s_t$ **then**
16: Synthesis($\delta, c_f, c_p, P, m_1, m_2$)
17: **else**
18: IntermolecularIneffectiveCollision($\delta, m_f, m_p, P, m_1, m_2$)
19: **end if**
20: **end if**
21: **end while**
22: **return** P

266 molecules (typically two). If the kinetic energy of the molecule is not enough
267 to create two new molecules, some energy from the container is added to the
268 newly generated molecules. On the other hand, a synthesis occurs when multi-
269 ple molecules (typically two) collide with each other and form a single molecule.
270 The kinetic energy of both molecules is joined and added to the new molecule.

271 CRO has more parameters to control than a common GA, in particular: the
272 rate at which molecules lose kinetic energy after a collision (*kinetic energy loss*
273 *rate*, a lower value would allow molecules to explore their local search space for
274 longer), the rate of molecular collisions (*molecular collision rate*, a higher value
275 would allow molecules to exchange information, i.e., energy more often), and the

276 *initial kinetic energy* of each molecule (a higher value would allow molecules to
277 explore their local search space for longer). There are two other parameters to
278 control the degree of diversity of the container (i.e., population of molecules):
279 a *decomposition threshold* to control whether a decomposition can be applied
280 to a molecule (only molecules that have not been involved in n collisions can
281 be decomposed), and a *synthesis threshold* to control whether a molecules can
282 be synthesised (a molecule can synthesised if its kinetic energy is lower than
283 a threshold). In this paper we used the values suggested by Lam and Li [26],
284 i.e., *kinetic energy loss rate* and *molecular collision rate* of 0.2, an *initial kinetic*
285 *energy* of 1000, a *decomposition threshold* of 500, and a *synthesis threshold* equal
286 to 10.

287 *2.7. Linearly Independent Path based Search (LIPS) Algorithm*

288 The Linearly Independent Path based Search (LIPS) algorithm [27] uses a
289 single-objective genetic algorithm to optimise one coverage target (i.e., a branch)
290 at a time. Algorithm 10 illustrates how LIPS works. As neither the pseudo-code
291 nor the source code of the original LIPS implementation are available, we refer
292 to the implementation proposed by Panichella et al. [28] and implemented on
293 EvoSuite.

294 Briefly, it starts by generating and evaluating a random test case i. If i
295 covers any branch goal, it is added to a pool of test cases (which keeps the best
296 test cases found by the search, similar to an archive). Then, the list of branches
297 not covered by test i is computed. For the next iteration of the algorithm, a
298 target goal is chosen from the list of uncovered goals (i.e., the last uncovered
299 goal of the path traversed by the last test case added to the pool of test cases),
300 and a population (which includes i) is randomly generated. In LIPS, every
301 target goal has an initial time limit to be covered equal to the total search
302 budget divided by the total number of targets. However, as the search evolves,
303 the time limit to satisfy each target is dynamically updated as branches are
304 covered during the search (as some branches could be easier/quicker to cover
305 than others). Within this time limit new offspring are generated based on

Algorithm 10 Linearly Independent Path based Search (LIPS) Algorithm

Input: Stopping condition C, Branch fitness function δ, Branch coverage goals B, 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 A

```
1:  A ← { }
2:  i ← GenerateRandomIndividual()
3:  PerformFitnessEvaluation(δ, i)
4:  U_B ← GetUncoveredBranches(B, i)
5:  b ← PopLast(U_B)
6:  UpdateOptimisedPopulation(A, i)
7:  P ← GenerateRandomPopulation(p_s − 1) ∪ {i}
8:  while ¬C and U_B ≠ Ø do
9:      N_P ← { } ∪ Elitism(P)
10:     while |N_P| < p_s do
11:         p_1, p_2 ← Selection(s_f, P)
12:         o_1, o_2 ← Crossover(c_f, c_p, p_1, p_2)
13:         Mutation(m_f, m_p, o_1)
14:         Mutation(m_f, m_p, o_2)
15:         N_P ← N_P ∪ {o_1, o_2}
16:     end while
17:     PerformFitnessEvaluation(δ, N_P)
18:     CollateralCoverage(U_B, N_P)
19:     UpdateUncoveredBranches(U_B, N_P)
20:     UpdateOptimisedPopulation(A, N_P)
21:     if b ∉ U_B or ¬HasBudgetLeftForBranchB(U_B, b) then
22:         b ← PopLast(U_B)
23:     end if
24:     P ← N_P
25: end while
26: return A
```

306 traditional selection, crossover, and mutation operators. Once the offspring is
307 generated it is then evaluated to assess whether it covers the target goal or any
308 other goal. If any offspring (i.e., test cases) cover the current target goal: 1)
309 the target goal is removed from the list of uncovered goals, 2) the new test case
310 is added to the final pool of test cases, and 3) a new uncovered target goal is
311 selected. If no offspring is able to cover the target goal within the allocated
312 time budget, no test case is added to the pool and a new uncovered target goal
313 is selected. Note that whether a new offspring covers the target goal or not, it

may by chance cover other goals ("collateral coverage"). In this case, all goals covered by the new offspring are removed from the list of uncovered goals and the test is added to the final pool of test cases. At the end of each iteration the current population seeds the next iteration of the algorithm as it may include individuals covering alternative branches of the uncovered target branch. The algorithm stops when all targets are covered or a stopping condition is met.

2.8. Many-Objective Sorting Algorithm

Unlike the single-objective optimisation on the test suite level described above, the Many-Objective Sorting Algorithm (MOSA) [9] regards each coverage goal as an independent optimisation objective. MOSA is a variant of NSGA-II [29], 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 11 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 [11] 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.

Algorithm 11 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

```
1:  p ← 0
2:  N_p ← GenerateRandomPopulation(p_s)
3:  PerformFitnessEvaluation(δ, N_p)
4:  A ← { }
5:  while ¬C do
6:      N_o ← GenerateOffspring(c_f, c_p, m_p, N_p)
7:      R_t ← N_p ∪ N_o
8:      r ← 0
9:      F_r ← PreferenceSorting(R_t)
10:     N_{p+1} ← { }
11:     while |N_{p+1}| + |F_r| ≤ p_s do
12:         CalculateCrowdingDistance(F_r)
13:         N_{p+1} ← N_{p+1} ∪ F_r
14:         r ← r + 1
15:     end while
16:     DistanceCrowdingSort(F_r)
17:     N_{p+1} ← N_{p+1} ∪ F_r with size p_s − |N_{p+1}|
18:     UpdateArchive(A, N_{p+1})
19:     p ← p + 1
20: end while
21: return A
```

344 *2.9. Many Independent Objective (MIO) Algorithm*

345 The Many Independent Objective (MIO) Algorithm [30] is a search algorithm
346 that is tailored for test suite generation. Its main motivation is to tackle cases
347 when there is a large number of testing targets, and comparatively little available
348 search budget. This is mainly the case for system testing, but could also happen
349 for unit testing of large classes with test criteria like mutation testing (which
350 typically results in many test targets).

351 A high level pseudo-code of how MIO works is listed in Figure 12. MIO
352 evolves individual test cases, which are stored in an archive. At the end of
353 search, a test suite is composed of the tests in the archive. In MIO, testing
354 targets are sought independently, and a population of test cases is kept for
355 each testing target. Once a target is covered, its best solution is saved in the

Algorithm 12 Many Independent Objective (MIO) Algorithm

Input: Stopping condition C, Fitness function δ, Population size N, Mutation function m_f, Mutation probability m_p, Probability of random sampling R, Start of focused search F
Output: Archive of optimised individuals A

```
1:  Z ← SetOfEmptyPopulations()
2:  A ← { }
3:  while ¬C do
4:      if R > Random(0,1) then
5:          p ← GenerateRandomIndividual()
6:      else
7:          p ← SampleIndividual(Z)
8:          p ← Mutation(m_f, m_p, p)
9:      end if
10:     for all t ∈ ReachedTargets(p) do
11:         if IsTargetCovered(t) then
12:             UpdateArchive(A, p)
13:             Z ← Z \ {Z_t}
14:         else
15:             Z_t ← Z_t ∪ {p}
16:             if |Z_t| > N then
17:                 RemoveWorstTest(Z_t, δ)
18:             end if
19:         end if
20:     end for
21:     UpdateParameters(F, R, N)
22: end while
23: return A
```

356 archive, and the population is deleted. To avoid memory problems, the number
357 of populations is dynamic: MIO only holds populations for targets that are
358 reached and not fully covered yet.

359 At the beginning of the search, all populations are empty, and a random
360 test case is generated. This test is added to all the populations of the targets
361 reached by its execution. At each iteration, like in a (1+1) EA, a test case is
362 sampled and mutated. The resulting offspring is copied and added to all the
363 populations of targets reached by the offspring execution. When a population
364 size reaches a certain threshold N, adding a new offspring will be followed by
365 removing the worst test case in that population, where the fitness value is only

366 based on that single target the population is for. In other words, a population
367 will not increase in size more than N.

368 The sampling of which offspring to generate is done in two ways: with prob-
369 ability P, it is created at random, whereas with $1 - P$ it is sampled from one
370 of the populations. When a population to sample from is chosen, the actual
371 test in the population to copy and mutate is chosen randomly with uniform
372 probability.

373 To handle the tradeoff between exploration and exploitation of the search
374 landscape, MIO employs a dynamic parameter control. For example, give a
375 starting value for R (e.g., $R = 0.5$), this value is decreased linearly over time
376 until it reaches $R = 0$, when a more focused search starts. Similarly, N decreases
377 down to $N = 1$. In other words, at the beginning of the search, MIO is similar
378 to random search, but, with the passing of iterations, it becomes closer and
379 closer to a focused (1+1) EA. When the focused search starts is controlled by
380 a parameter F, which represents the amount of search budget consumed before
381 starting the focused search.

382 To handle possible issues with infeasible targets, the choice of which popu-
383 lation to sample from is not at random. MIO keeps track of how often there are
384 improvements in fitness value for the different testing targets that are not yet
385 covered. Populations for testing targets with recent fitness improvements are
386 more likely to be sampled from compared to populations for targets whose best
387 fitness value has been stagnating (which would happen for infeasible targets).

388 **3. Empirical Study**

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

392 *3.1. Experimental Setup*

393 *3.1.1. Selection of Classes Under Test*

394 A key factor of studying evolutionary algorithms on automatic test gener-
395 ation is the selection of classes under test. As many open source classes, for
396 example contained in the SF110 [3] corpus, are trivially simple [5] and would
397 not reveal differences between algorithms, we used the selection of non-trivial
398 classes from the DynaMOSA study [11]. This is a corpus of 117 open-source
399 Java projects and 346 classes, selected from four different benchmarks. The
400 complexity of classes ranges from 14 statements and 2 branches to 16,624 state-
401 ments and 7,938 branches. The average number of statements is 1,109, and the
402 average number of branches is 259.

403 *3.1.2. Unit Test Generation Tool*

404 We used EvoSuite [2], which provides search algorithms to evolve coverage-
405 optimised test suites, and allows an unbiased comparison of the algorithms as
406 the underlying implementation of the tool is the same across all algorithms.
407 By default, EvoSuite uses a Monotonic GA described in Section 2.4. It also
408 provides a Standard and Steady State GA, Random search, Random testing
409 and, more recently, MOSA, DynaMOSA, and LIPS. For this study, we extended
410 EvoSuite with seven algorithms: the 1+(λ, λ) GA, ($\mu + \lambda$) EA, (μ, λ) EA,
411 Breeder GA, Cellular GA, CRO, and MIO. All evolutionary algorithms use a
412 test archive.

413 *3.1.3. Experiment Procedure*

414 We performed two experiments to assess the performance of the 13 selected
415 evolutionary algorithms (described in Section 2). First, we conducted a tun-
416 ing study to select the best population size (μ) of nine algorithms, number of
417 mutations (λ) of $1 + (\lambda, \lambda)$ GA, population size (μ) and number of mutations
418 (λ) of ($\mu + \lambda$) EA and (μ, λ) EA, and the amount of search budget consumed
419 before starting MIO's focused search, as the performance of each EA can be in-

fluenced by the parameters used [31]. Random-based approaches do not require any tuning. Then, we conducted a larger study to perform the comparison.

For both experiments we have two configurations: 1) single-criterion optimisation (i.e., branch coverage optimisation), and 2) multiple-criteria optimisation[1] (i.e., line, branch, exception, weak-mutation, output, method, method-no-exception, and context-dependent branch coverage) [6] to study the effect of the number of coverage criteria on the coverage of resulting test suites. For both configurations we used EvoSuite's default search budget of 1 minute. Due to the randomness of EAs, we repeated the experiments 30 times.

For the tuning study, we randomly selected 10% (i.e., 34) of DynaMOSA's study classes [11] (with 15 to 1,707 branches, 227 on average) from 30 Java projects. This resulted in a total of 25,500 (13,260 single-criterion configurations, and 12,240 multiple-criteria configurations; the number of multiple-criteria configurations is lower because LIPS only supports single criteria) calls to EvoSuite and more than 17 days of CPU-time overall. For the second experiment, we used the remaining 308 classes (346 total - 34 used to tune each EA - 4 discarded due to crashes of EvoSuite) from the DynaMOSA study [11]. Besides the tuned μ, λ parameters, and MIO's exploitation starting point, we used EvoSuite's default parameters [31].

3.1.4. Experiment Analysis

For each 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 in terms of statements, number of iterations of each EA, number of fitness evaluations, mutation score of the generated test suites, etc. As described by Arcuri and Fraser [31] "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 [31]: Given the coverage

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

448 of a class c in a run r, $cov(c, r)$, the best and worst coverage of c in any run,
449 $max(cov(c))$ and $min(cov(c))$ respectively, a *relative coverage*, $\delta(c, r)$, can be
450 defined as

$$\delta(c, r) = \frac{cov(c, r) \ - \ min(cov(c))}{max(cov(c)) \ - \ min(cov(c))}$$

451 If the best and worst coverage of c is equal, i.e., $max(cov(c)) == min(cov(c))$,
452 then $\delta(c, r)$ is 1 (if range of $cov(c, r)$ is between 0 and 1) or 100 (if range of
453 $cov(c, r)$ is between 0 and 100). Given a set of runs R, the average relative
454 coverage of a class c is defined as

$$\Delta(c) = \frac{1}{|R|} \sum_{r \in R} \delta(c, r)$$

455 Thus, the coverage achieved by an algorithm A can be defined as

$$cov_A = \frac{1}{|C|} \sum_{c \in C} \Delta(c)$$

456 where C represents the set of classes. This way, the coverage of a trivial small
457 class would be as important as the coverage of a large (perhaps more complex)
458 class. For each averaged coverage value we compute common statistics such as
459 standard deviation (σ), and confidence intervals ("CI") using bootstrapping at
460 95% confidence level. In order to statistically compare the performance of each
461 EA we use the Vargha-Delaney $\hat{A}_{12}$ effect size, the Wilcoxon-Mann-Whitney
462 U-test with a 95% confidence level, and the Friedman test. Note that we do
463 not perform any p-value adjustments in this study, e.g., Bonferroni, as the use
464 of such adjustments has been discouraged [32] due to substantial reduction in
465 the statistical power of rejecting an incorrect null hypothesis [33], and therefore
466 increasing the likelihood of Type II errors.

467 *3.1.5. Threats to Validity*

468 Threats to *internal validity* might result from how the empirical study was
469 carried out. We thoroughly tested the experiment framework and test genera-
470 tion tool in order to reduce the chances of having faults, but it is well-known

471 that testing alone cannot prove the absence of defects. Since the randomised
472 algorithms underlying our study are affected by chance, we ran each experiment
473 30 times and followed rigorous statistical procedures to evaluate the results. To
474 avoid possible confounding factors when comparing different algorithms, they
475 were all implemented in the same tool. Furthermore, we used the same de-
476 fault values for all relevant parameters, and tuned the algorithm-specific ones.
477 It is nevertheless possible that different parameter values might influence the
478 performance of each EA.

479 We measured the success of different EAs using code coverage. While higher
480 coverage is a desirable goal for test generation, there is an ongoing debate on
481 how code coverage correlates to fault finding potential, and so there is a threat
482 to *construct validity* resulting from how we measure test suite quality. However,
483 code coverage is nevertheless sufficient to compare the effectiveness of different
484 optimisation algorithms at achieving their optimisation goal.

485 As with any empirical study, there are threats to *external validity* regarding
486 the generalisation to other types of software. The results reported in this paper
487 are limited to the number and type of EAs used in the experiments. However,
488 we believe these are representative of state-of-art algorithms, and are sufficient
489 in order to demonstrate the influence of each algorithm on the problem, and of
490 the choice of algorithm on the problem in general. We used 346 complex classes
491 from 117 open-source Java projects in our experiments. While this resulted in a
492 substantial computational effort, our results may not generalise to other classes.
493 However, we specifically chose classes that are complex, as also used in previous
494 studies [11] on test generation.

495 *3.2. Parameter Tuning*

496 The execution of an EA requires a number of parameters to be set. As there
497 is not a single best configuration setting to solve all problems [34] in which
498 an EA could be applied, a possible alternative is to tune EA's parameters for a
499 specific problem at hand to find the "best" ones. Our experimental setup largely
500 relies on two previous tuning studies: 1) Arcuri and Fraser [31] determined the

best values for most parameters of EvoSuite, such as crossover rate, elitism rate, selection function, etc.; and 2) Shamshiri et al. [35] determined the best values for CRO in the context of search-based test generation, for instance, the best potential energy value, or the best number of collisions allowed, etc. Both studies performed a similar tuning study as the one defined and reported in this paper to identify the best parameters. Note that, although neither Breeder GA, Cellular GA, $1 + (\lambda, \lambda)$ GA, $(\mu + \lambda)$ EA, and (μ, λ) EA have been evaluated in the context of unit test generation, none of the algorithms except Cellular GA require any new parameters. For the Cellular GA we use the best parameter (i.e., neighbourhood model) that has been reported by previous work [21]. The main distinguishing factors between the algorithms we are considering in this study are μ (i.e., the population size) and λ (i.e., the number of mutations), or F which represents the amount of search budget consumed before starting the focused search in MIO. 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, Breeder GA, Cellular GA, CRO, MOSA, DynaMOSA, and LIPS.
- λ size of 1, 8 [22], 25, and 50 for $1 + (\lambda, \lambda)$ GA.
- μ size of 1, 7 [36], 25, and 50, and λ size of 1, 7, 25, and 50 for $(\mu + \lambda)$ EA and (μ, λ) EA.
- F of 0.00, 0.25, 0.50, 0.75, 1.00 for MIO.

Thus, for Standard GA, Monotonic GA, SteadyState GA, Breeder GA, Cellular GA, CRO, MOSA, DynaMOSA, LIPS, and $1 + (\lambda, \lambda)$ GA there are 4 different configurations; for $(\mu + \lambda)$ EA and (μ, λ) EA, and as λ must be divisible by μ, there are 8 different configurations (i.e., $1+1$, $1+7$, $1+25$, $1+50$, $7+7$, $25+25$, $25+50$, $50+50$); for MIO there are 5 different configurations, i.e., a total of 61 different configurations.

Algorithm	X	Branch Cov.	Overall Cov.	Avg. $\hat{A}_{12}$	Better $\hat{A}_{12}$	Worse $\hat{A}_{12}$
Search budget of 60 seconds – Single-criteria						
Standard GA	10	0.74	—	0.53	0.76	0.31
Monotonic GA	25	0.75	—	0.54	0.73	0.32
Steady-State GA	10	0.70	—	0.54	0.73	0.32
$1 + (\lambda, \lambda)$ GA	8	0.61	—	0.53	0.69	0.30
$(\mu + \lambda)$ EA	7+7	0.74	—	0.52	0.78	0.26
(μ, λ) EA	1,7	0.76	—	0.65	0.83	0.28
Breeder GA	10	0.67	—	0.51	0.73	0.23
Cellular GA	100	0.60	—	0.52	0.77	0.26
CRO	10	0.70	—	0.51	0.73	0.26
MOSA	10	0.74	—	0.53	0.72	0.24
DynaMOSA	10	0.75	—	0.55	0.73	0.16
LIPS	100	0.58	—	0.54	0.72	0.31
MIO	1.00	0.68	—	0.52	0.72	0.34
Search budget of 60 seconds – Multiple-criteria						
Standard GA	100	0.64	0.66	0.52	0.74	0.23
Monotonic GA	100	0.63	0.65	0.53	0.76	0.22
Steady-State GA	100	0.58	0.61	0.53	0.77	0.23
$1 + (\lambda, \lambda)$ GA	50	0.49	0.51	0.60	0.77	0.31
$(\mu + \lambda)$ EA	50+50	0.67	0.69	0.55	0.77	0.21
(μ, λ) EA	25,50	0.68	0.70	0.61	0.81	0.25
Breeder GA	100	0.61	0.63	0.57	0.75	0.23
Cellular GA	100	0.57	0.60	0.62	0.79	0.25
CRO	100	0.62	0.64	0.49	0.73	0.23
MOSA	25	0.73	0.73	0.58	0.77	0.29
DynaMOSA	10	0.77	0.73	0.55	0.72	0.20
LIPS	—	—	—	—	—	—
MIO	0.25	0.67	0.66	0.54	0.71	0.28

Table 1: Best parameter (X, i.e., μ, $\mu + \lambda$, or F) of each EA for single and multiple criteria optimisation. "Branch Coverage" column reports the branch coverage per EA, and column "Overall Coverage", the overall coverage of a multiple-criteria optimisation, "Avg. $\hat{A}_{12}$" represents the average effect size of the best parameter value when compared to all possible parameter values, "Better $\hat{A}_{12}$" the effect size of all pairwise comparisons in which the best parameter was significantly better, and "Worse $\hat{A}_{12}$" the effect size of pairwise all comparisons in which the best parameter was significantly worse.

529 To identify the best parameter of each EA, we performed a pairwise com-
530 parison of the coverage achieved by using any μ (population size), $\mu + \lambda$, or F.
531 The parameter for which an EA achieved a significantly higher coverage more
532 often was selected as the best. Table 1 shows the best parameter per EA. For
533 single and multiple-criteria the best population size is shared by several EAs,
534 for instance, Standard GA, Steady-State GA, Breeder GA, and CRO share the
535 same value (10 for single-criteria, and 100 for multiple-criteria). The best pop-
536 ulation size for MOSA and DynaMOSA is the same for single-criteria (i.e., 10),

537 but different for multiple-criteria (25 for MOSA, and 10 for DynaMOSA). The
538 best F value for MIO is 1.0 for single-criteria, and 0.25 for multiple-criteria,
539 i.e., for a smaller number of coverage goals MIO works best without focusing
540 the search, and for a larger number of coverage goals (multiple-criteria scenario)
541 MIO works best if the focus search is enabled once 25% of the search budget
542 has been consumed. Table 1 also reports the average effect size of the best
543 parameter value when compared to all possible parameter values; and the ef-
544 fect size of pairwise comparisons in which the best parameter was significantly
545 better/worse.

546 ## 4. Experiment Results

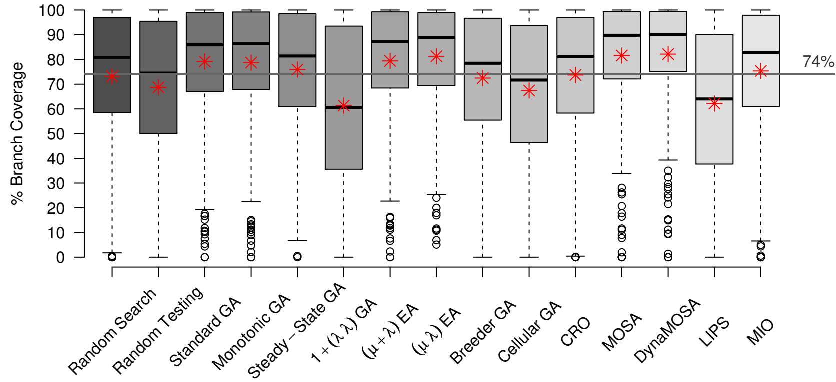
547 Table 2 summarises the results of the main experiment described in the pre-
548 vious section. For each algorithm we report the branch coverage achieved for
549 single and multiple criteria, the overall coverage for multiple criteria, the muta-
550 tion score, the number of generated test cases, and the rank of each algorithm
551 based on their average performance. Table 2 also reports the standard deviation
552 and confidence intervals (CI) using bootstrapping at 95% significance level of
553 the coverage achieved (either branch or overall coverage).

554 On one hand, MOSA and DynaMOSA achieve the highest coverage on aver-
555 age (82%) for single criteria. Although the CI of both algorithms overlap ([80%,
556 85%] vs. [79%, 84%]), DynaMOSA is ranked first. According to the Friedman
557 test, the ranking reported in Table 2 is statistically significant for both sin-
558 gle and multiple criteria, i.e., p-value is < 0.0001 for single criteria, and 0 for
559 multiple criteria (full data is available on the accompanying website [37]). For
560 multiple criteria, DynaMOSA achieves the highest overall coverage (86%) and
561 CI among all algorithms. On the other hand, the $1 + (\lambda, \lambda)$ EA achieves the
562 lowest branch coverage (61%) for single criteria, and Random testing achieves
563 the lowest overall coverage (45%) for multiple criteria, thus it is ranked as the
564 worst algorithm. There are a few algorithms that perform similarly, for instance,
565 Standard GA, Monotonic GA, and $(\mu+\lambda)$ EA achieve the same branch coverage
566 for single criteria (79%); and $(\mu+\lambda)$ EA, and (μ, λ) EA achieve the same overall

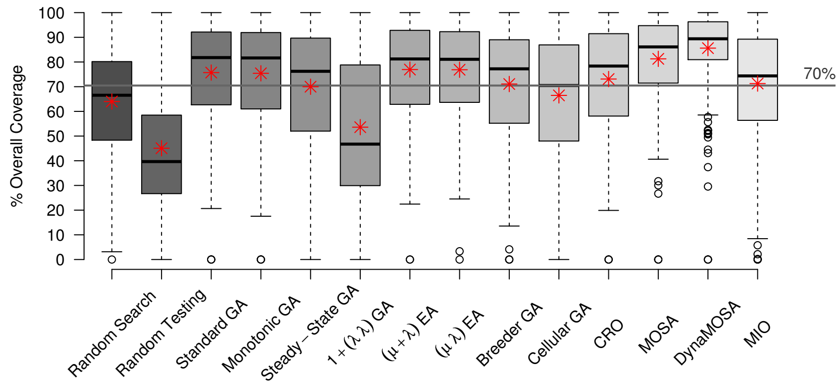
Algorithm	Branch Cov.	σ	CI	Overall Cov.	σ	CI	Mut. Score	#T	R
Search budget of 60 seconds – Single-criteria									
Random Search	0.73	0.07	[0.70, 0.76]	—	—	—	0.44	28	8.3
Random Testing	0.69	0.08	[0.66, 0.72]	—	—	—	0.43	25	10.5
Standard GA	0.79	0.09	[0.77, 0.82]	—	—	—	0.46	27	6.3
Monotonic GA	0.79	0.08	[0.76, 0.81]	—	—	—	0.45	27	6.2
Steady-State GA	0.76	0.08	[0.73, 0.79]	—	—	—	0.44	27	8.1
$1 + (\lambda, \lambda)$ GA	0.61	0.13	[0.58, 0.65]	—	—	—	0.41	18	11.0
$(\mu + \lambda)$ EA	0.79	0.08	[0.77, 0.82]	—	—	—	0.46	28	5.9
(μ, λ) EA	0.81	0.09	[0.79, 0.83]	—	—	—	0.46	28	5.1
Breeder GA	0.72	0.10	[0.70, 0.76]	—	—	—	0.44	25	9.5
Cellular GA	0.67	0.09	[0.64, 0.71]	—	—	—	0.43	26	10.8
CRO	0.74	0.10	[0.71, 0.77]	—	—	—	0.44	26	8.6
MOSA	0.82	0.08	[0.79, 0.84]	—	—	—	0.47	29	5.1
DynaMOSA	0.82	0.08	[0.80, 0.85]	—	—	—	0.47	30	4.8
LIPS	0.62	0.11	[0.59, 0.66]	—	—	—	0.42	23	11.9
MIO	0.75	0.09	[0.72, 0.78]	—	—	—	0.44	27	7.9
Search budget of 60 seconds – Multiple-criteria									
Random Search	0.65	0.10	[0.62, 0.67]	0.64	0.10	[0.62, 0.66]	0.44	31	9.1
Random Testing	0.55	0.09	[0.52, 0.59]	0.45	0.12	[0.42, 0.48]	0.41	28	12.3
Standard GA	0.71	0.08	[0.68, 0.74]	0.76	0.07	[0.73, 0.78]	0.46	42	5.6
Monotonic GA	0.71	0.08	[0.68, 0.74]	0.75	0.08	[0.73, 0.78]	0.46	41	6.1
Steady-State GA	0.65	0.08	[0.61, 0.68]	0.70	0.07	[0.67, 0.73]	0.45	39	8.9
$1 + (\lambda, \lambda)$ GA	0.48	0.13	[0.45, 0.52]	0.54	0.12	[0.51, 0.57]	0.40	26	11.3
$(\mu + \lambda)$ EA	0.72	0.08	[0.69, 0.75]	0.77	0.08	[0.75, 0.79]	0.46	42	5.3
(μ, λ) EA	0.72	0.09	[0.69, 0.75]	0.77	0.08	[0.75, 0.79]	0.47	40	5.6
Breeder GA	0.66	0.09	[0.63, 0.69]	0.71	0.08	[0.69, 0.74]	0.45	39	8.2
Cellular GA	0.61	0.09	[0.58, 0.64]	0.66	0.08	[0.63, 0.69]	0.44	39	10.2
CRO	0.69	0.09	[0.65, 0.72]	0.73	0.08	[0.71, 0.75]	0.46	40	7.1
MOSA	0.79	0.09	[0.76, 0.81]	0.81	0.08	[0.79, 0.83]	0.49	44	4.3
DynaMOSA	0.84	0.08	[0.82, 0.86]	0.86	0.07	[0.84, 0.87]	0.51	48	3.2
LIPS	—	—	—	—	—	—	—	—	—
MIO	0.68	0.10	[0.65, 0.71]	0.71	0.09	[0.69, 0.74]	0.45	37	7.9

Table 2: For each algorithm, we report several statistics on the obtained results, such as branch and overall coverage, standard deviation (σ), mutation score, number of generated test cases (#T), and the rank of each algorithm based on their average performance (R), which is statistically significant for both single and multiple criteria according to the Friedman test (p-value is < 0.0001 for single criteria, and 0 for multiple criteria, full data is available on the accompanying website [37]). For averaged coverage values we also report confidence intervals (CI) using bootstrapping at 95% significance level.

567 coverage for multiple criteria (77%). To make these quantitative results more
568 accessible, Figure 1 shows the coverage distribution achieved by each algorithm.
569 It also shows the median and the mean per algorithm, and the mean of all al-
570 gorithms. For single criteria the average coverage among all algorithms is 74%,
571 which means 7 algorithms (i.e., Random search and Random testing, $1 + (\lambda, \lambda)$



(a) Single criteria.



(b) Multiple criteria.

Figure 1: Coverage achieved by each algorithm. Middle line of each boxplot marks the median, white circles represent outliers, * symbol signifies the mean, and the grey line represents the mean of all coverages.

572 EA, Breeder GA, Cellular GA, and LIPS) out of 15 perform below the average.
573 On the other hand, for multiple criteria only 4 algorithms perform below the
574 average (i.e., Random search and testing, $1 + (\lambda, \lambda)$ EA, and Cellular GA).

575 In terms of mutation score and number of generated test cases, all algorithms
576 performed similarly. For instance, MOSA and DynaMOSA generated 29 and 30
577 test cases for single criteria, respectively, and both sets of test cases achieve the
578 same mutation score (47%). The algorithm that generated the lowest number
579 of test cases (18) and achieved the lowest mutation score (41%) is the $1+(\lambda, \lambda)$
580 EA. Besides these three EAs, the range of mutation scores for single criteria

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	4	0.79	—	0.58	741 / 2464	0.82	210 / 2464	0.26
Monotonic GA	3	0.79	—	0.58	733 / 2464	0.81	189 / 2464	0.26
Steady-State GA	5	0.76	—	0.50	536 / 2464	0.82	599 / 2464	0.21
$1 + (\lambda, \lambda)$ GA	8	0.61	—	0.33	189 / 2464	0.76	1218 / 2464	0.12
$(\mu + \lambda)$ EA	2	0.79	—	0.60	815 / 2464	0.81	128 / 2464	0.28
(μ, λ) EA	1	0.81	—	0.63	1028 / 2464	0.81	60 / 2464	0.30
Breeder GA	7	0.72	—	0.44	322 / 2464	0.82	846 / 2464	0.21
Cellular GA	9	0.67	—	0.35	210 / 2464	0.82	1256 / 2464	0.17
CRO	6	0.74	—	0.50	460 / 2464	0.82	528 / 2464	0.23
Search budget of 60 seconds – Multiple-criteria								
Standard GA	2	0.71	0.76	0.62	961 / 2464	0.82	130 / 2464	0.26
Monotonic GA	4	0.71	0.75	0.60	892 / 2464	0.81	188 / 2464	0.26
Steady-State GA	7	0.65	0.70	0.44	371 / 2464	0.85	893 / 2464	0.21
$1 + (\lambda, \lambda)$ GA	9	0.48	0.54	0.22	120 / 2464	0.76	1724 / 2464	0.08
$(\mu + \lambda)$ EA	1	0.72	0.77	0.64	1066 / 2464	0.83	106 / 2464	0.27
(μ, λ) EA	3	0.72	0.77	0.62	1012 / 2464	0.83	216 / 2464	0.24
Breeder GA	6	0.66	0.71	0.47	411 / 2464	0.84	733 / 2464	0.23
Cellular GA	8	0.61	0.66	0.37	223 / 2464	0.88	1207 / 2464	0.18
CRO	5	0.69	0.73	0.53	601 / 2464	0.82	460 / 2464	0.22

Table 3: X 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.

581 is only [42%, 46%], and the number of test cases is in the range of [23, 28].
582 Note that for both, single and multiple criteria, the EA that generated more
583 test cases is the one that achieved the highest coverage (either branch or overall
584 coverage) and mutation score.

585 Although DynaMOSA achieved the highest coverage and mutation score
586 among all algorithms, and is ranked first for both single and multiple criteria,
587 it is not clear whether it performs consistently better than any other algorithm
588 across all classes under test. In the following sections we perform further anal-
589 yses to address this issue and answer our research questions.

590 *4.1. RQ1 – Which archive-based single-objective evolutionary algorithm per-* 591 *forms best?*

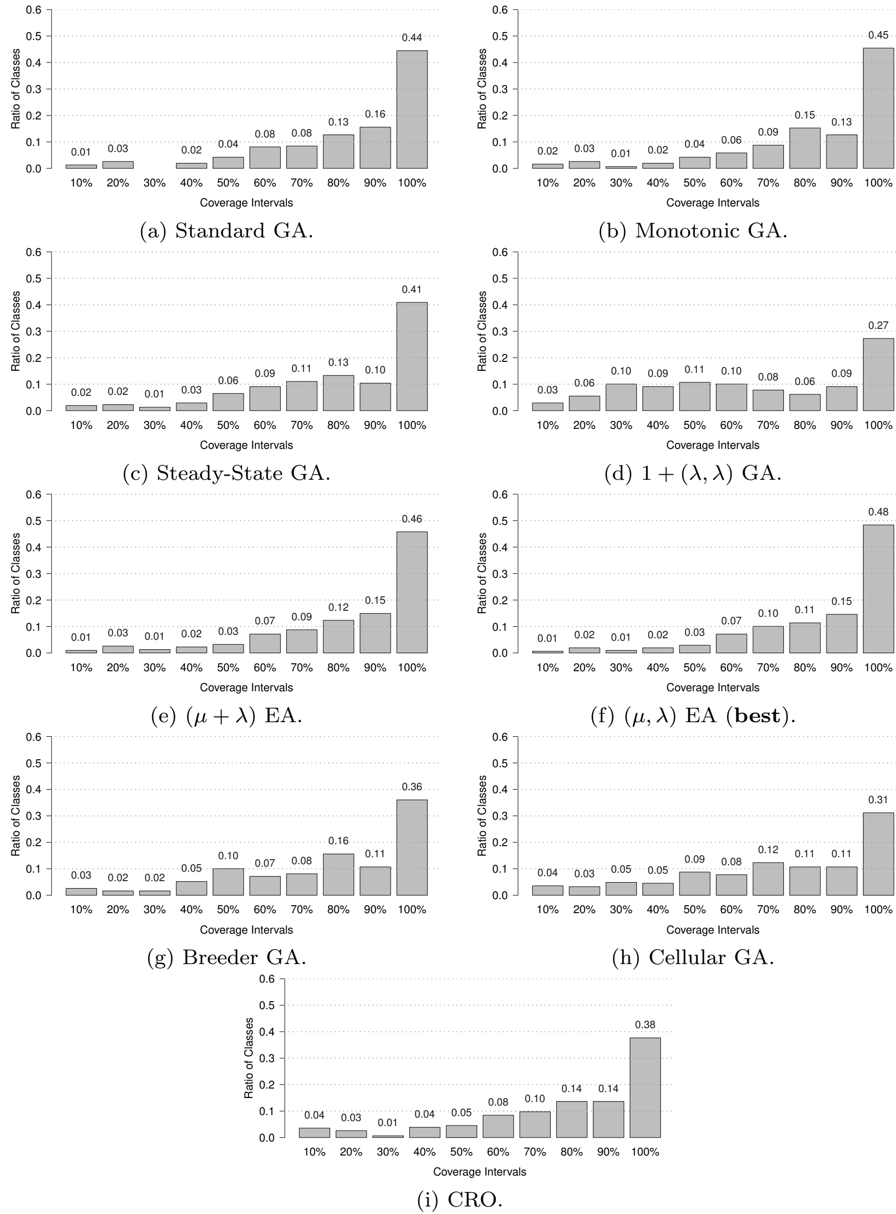
592 Table 3 summarises the results of a pairwise tournament of all EAs. An EA
593 X is considered to be better than an EA Y if it performs significantly better
594 on a higher number of comparisons. For example, the (μ, λ) EA was the one

with more positive comparisons (1028) and the least negative comparisons (just 60) – thus, being the best EA for single criteria. While it is ranked third for multiple criteria, it achieved the same branch and overall coverage (72% and 77%, respectively) as the first ranked EA, i.e., $(\mu + \lambda)$ EA, with an $\hat{A}_{12}$ effect size of 62% averaged over all comparisons.

Figures 2 and 3 illustrate these results visually by showing the proportion of classes per coverage interval for single and multiple criterion respectively. For example, (μ, λ) EA achieved a branch coverage between 91% and 100% for 63% of all classes under test (see Figure 2f), and an overall coverage between 91% and 100% for 52% of all classes under test (see Figure 3f). As expected, the best EA for single and multiple criteria is the one with the highest ratio of classes within the coverage interval $]90\%, 100\%]$.

Surprisingly, despite its reported good performance [22] the $1+(\lambda, \lambda)$ EA was statistically significantly better only on 120 comparisons for multiple criteria, while it was statistically significantly worse on 1,724 comparisons out of 2,464 – which make it the worst EA in our comparison. A recent study has shown that due to the presence of many plateaus in the landscape of a test generation problem (and not the number of local optima), crossover has little or no impact on the search [38]. Thus, our conjecture is that the worst performance of the $1 + (\lambda, \lambda)$ EA in our evaluation is due to the fact the only individual in the population heavily relies on the outcome of λ crossover operations, which may or may not perform successfully (i.e., generate an individual that is better than the single one in the population). Another EA that performed poorly is the Cellular GA. To the best of our knowledge, this is the first time a Cellular GA has been applied to automatic software test generation and therefore it has not been studied in detail, for instance, the question which neighbourhood model works best for this particular problem still remains.

> *RQ1: For a small number of coverage goals a (μ, λ) EA is better than the other considered evolutionary algorithms, for a large number of coverage goals a $(\mu + \lambda)$ EA performed better.*



(a) Standard GA.

(b) Monotonic GA.

(c) Steady-State GA.

(d) $1 + (\lambda, \lambda)$ GA.

(e) $(\mu + \lambda)$ EA.

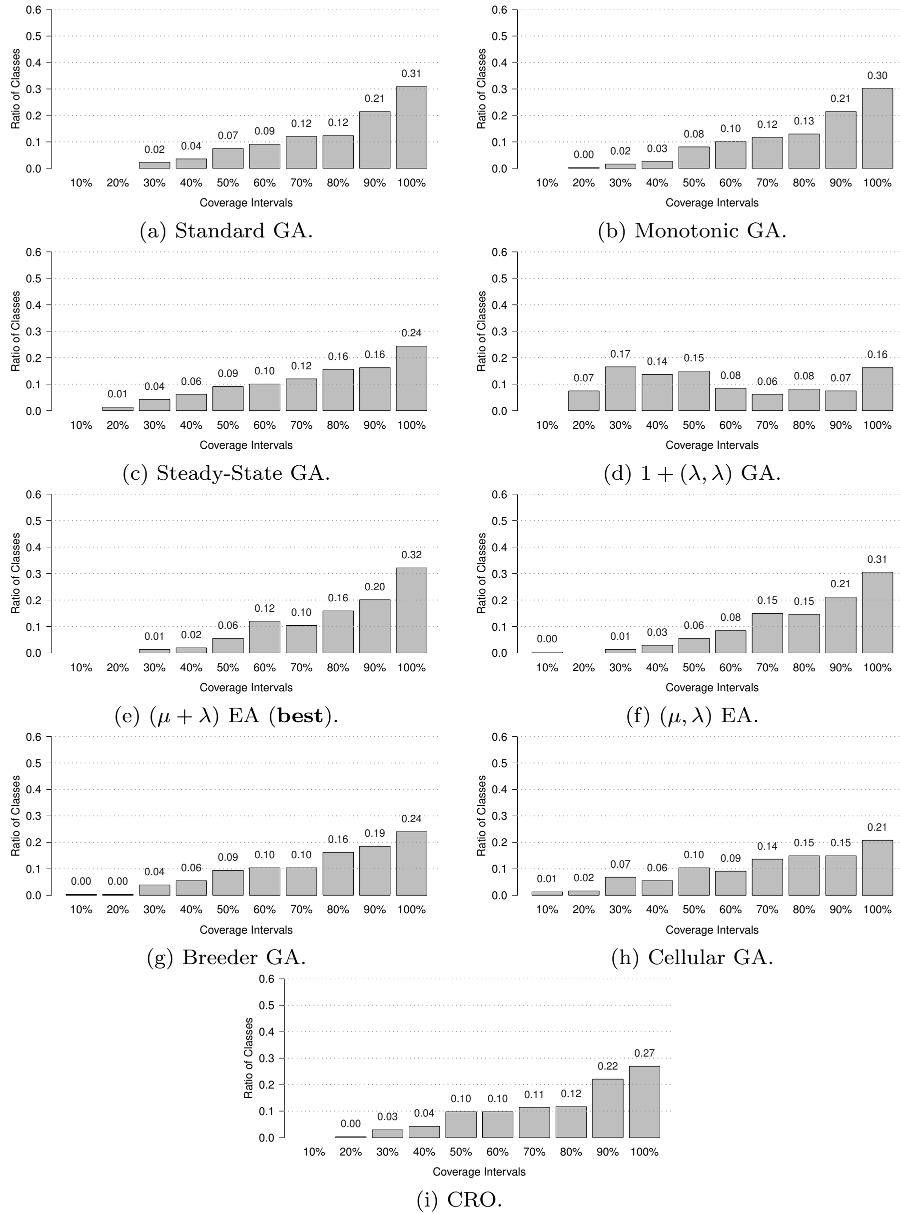
(f) (μ, λ) EA (**best**).

(g) Breeder GA.

(h) Cellular GA.

(i) CRO.

Figure 2: Proportion of classes that have an average branch coverage (averaged out of 30 runs on all their classes) within each 10% branch coverage interval. X-labels show the upper limit (inclusive). For example, the group 30% represents all the classes with an average branch coverage greater than 20% and lower than or equal to 30%.



(a) Standard GA.

(b) Monotonic GA.

(c) Steady-State GA.

(d) $1 + (\lambda, \lambda)$ GA.

(e) $(\mu + \lambda)$ EA (**best**).

(f) (μ, λ) EA.

(g) Breeder GA.

(h) Cellular GA.

(i) CRO.

Figure 3: Proportion of classes that have an average overall coverage (averaged out of 30 runs on all their classes) within each 10% overall coverage interval. X-labels show the upper limit (inclusive). For example, the group 30% represents all the classes with an average overall coverage greater than 20% and lower than or equal to 30%.

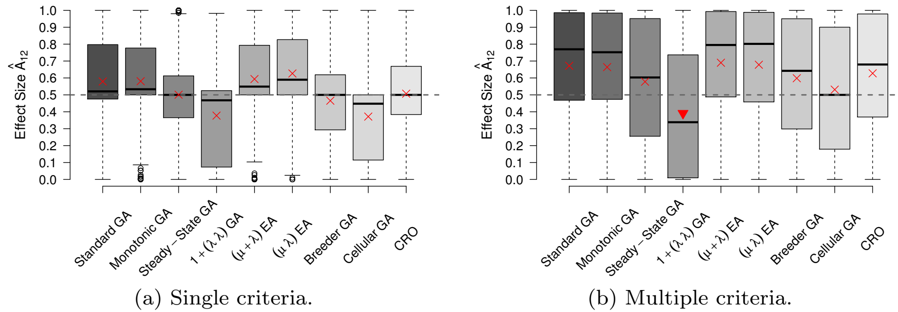
Algorithm	Branch Cov.	Overall Cov.	vs. Random Search $\hat{A}_{12}$	p	vs. Random Testing $\hat{A}_{12}$	p
Search budget of 60 seconds – Single-criteria						
Random Search	0.73	—	—	—	0.64	0.16
Random Testing	0.69	—	0.36	0.16	—	—
Standard GA	0.79	—	0.58	0.12	0.70	0.09
Monotonic GA	0.79	—	0.58	0.13	0.70	0.07
Steady-State GA	0.76	—	0.50	0.16	0.63	0.12
$1 + (\lambda, \lambda)$ GA	0.61	—	0.38	0.11	0.42	0.11
$(\mu + \lambda)$ EA	0.79	—	0.59	0.12	0.71	0.08
(μ, λ) EA	0.81	—	0.63	0.11	0.73	0.07
Breeder GA	0.72	—	0.47	0.12	0.56	0.14
Cellular GA	0.67	—	0.37	0.11	0.48	0.11
CRO	0.74	—	0.51	0.13	0.63	0.11
Search budget of 60 seconds – Multiple-criteria						
Random Search	0.65	0.64	—	—	0.75	0.05
Random Testing	0.55	0.45	0.25	0.05	—	—
Standard GA	0.71	0.76	0.67	0.07	**0.87**	**0.02**
Monotonic GA	0.71	0.75	0.66	0.07	**0.87**	**0.02**
Steady-State GA	0.65	0.70	0.58	0.06	**0.81**	**0.03**
$1 + (\lambda, \lambda)$ GA	0.48	0.54	**0.39**	**0.05**	0.56	0.15
$(\mu + \lambda)$ EA	0.72	0.77	0.69	0.06	**0.89**	**0.03**
(μ, λ) EA	0.72	0.77	0.68	0.06	**0.89**	**0.03**
Breeder GA	0.66	0.71	0.60	0.07	**0.83**	**0.03**
Cellular GA	0.61	0.66	0.53	0.07	**0.78**	**0.05**
CRO	0.69	0.73	0.63	0.07	**0.84**	**0.03**

Table 4: Comparison of evolutionary algorithms and two random-based approaches: Random search and Random testing. Statistically significant effect sizes are shown in bold.

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

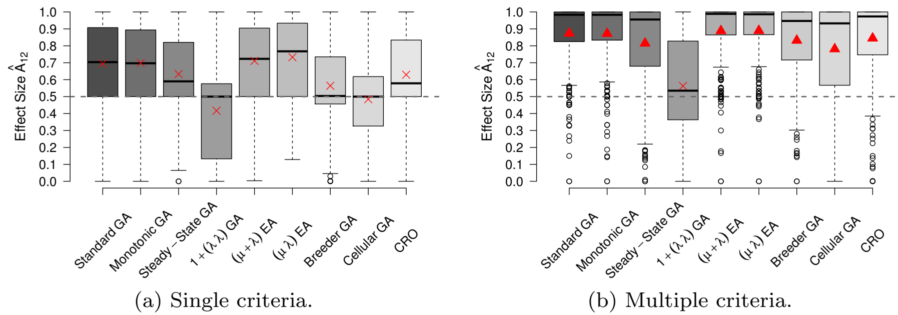
Table 4 compares the results of each EA with the two random-based techniques considered in this study: Random search and Random testing. Both random approaches are hardly affected by the number of coverage goals. For instance, Random testing covers 69% of all branch goals for single criteria, where for multiple criteria it only covers 45% of all goals (55% of all branch goals). The % of goals covered by Random search decreases from 73% (single criteria) to 64% (multiple criteria).

As we can see in Figure 5, for single criteria all EAs but $1 + (\lambda, \lambda)$ EA and Cellular GA achieve higher branch coverage than Random testing. For multiple criteria, all EAs achieve higher overall coverage than Random testing, most of



(a) Single criteria. (b) Multiple criteria.

Figure 4: Effect size $\hat{A}_{12}$ of EA X vs. Random search. Middle line of each boxplot marks the median, white circles represent the outliers, ▲ represents the mean of a significant effect size greater than 0.5 (i.e., EA X performs significantly better than Random search), ▼ the mean of a significant effect size lower than 0.5 (i.e., EA X performs significantly worse than Random search), × the mean of a no significant effect size.



(a) Single criteria. (b) Multiple criteria.

Figure 5: Effect size $\hat{A}_{12}$ of EA X vs. Random testing. Please refer to Figure 4 for an explanation of each symbol.

them significantly higher overall coverage. For example, Random testing covers 45% of all coverage goals for multiple criteria where $\mu + \lambda$ EA covers 77% (an effect size $\hat{A}_{12}$ of 0.89 and a p-value of 0.03). When compared to Random search (see Figure 4), six out of nine EAs performed better for single criteria (i.e., Standard GA, Monotonic GA, Steady-State GA, $(\mu + \lambda)$ EA, (μ, λ) EA), and CRO; and all EAs but $1 + (\lambda, \lambda)$ EA performed better than Random search for multiple criteria. This result is different to the earlier study by Shamshiri et al. [5], where random achieved similar, and sometimes higher coverage than a genetic algorithm. Our conjecture is that the better performance of some EAs

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								
MOSA	2	0.82	—	0.63	370 / 924	0.86	51 / 924	0.23
DynaMOSA	1	0.82	—	0.66	391 / 924	0.87	31 / 924	0.26
LIPS	4	0.62	—	0.24	40 / 924	0.83	614 / 924	0.09
MIO	3	0.75	—	0.48	196 / 924	0.89	301 / 924	0.18
Search budget of 60 seconds – Multiple-criteria								
MOSA	2	0.79	0.81	0.55	212 / 616	0.85	140 / 616	0.21
DynaMOSA	1	0.84	0.86	0.71	352 / 616	0.85	15 / 616	0.20
LIPS	—	—	—	—	—	—	—	—
MIO	3	0.68	0.71	0.25	16 / 616	0.80	425 / 616	0.13

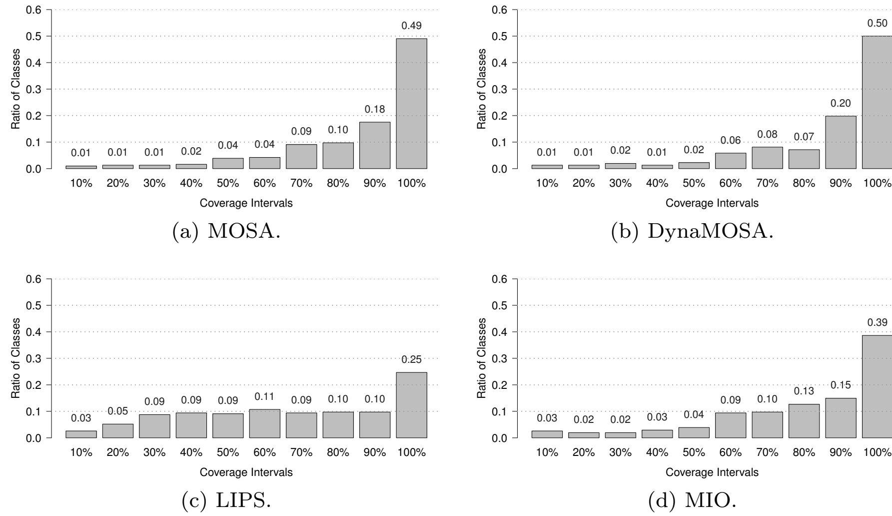
Table 5: Pairwise comparison of all many objective algorithms. "Better than" and "Worse than" give the number of comparisons for which the best EA is statistically significantly (i.e., *p-value* < 0.05) better and worse, respectively. Columns $\hat{A}_{12}$ give the average effect size.

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 (μ, λ) EA) perform better than random search and statistically better than random testing.*

4.3. RQ3 – Which archive-based many-objective evolutionary algorithm performs best?

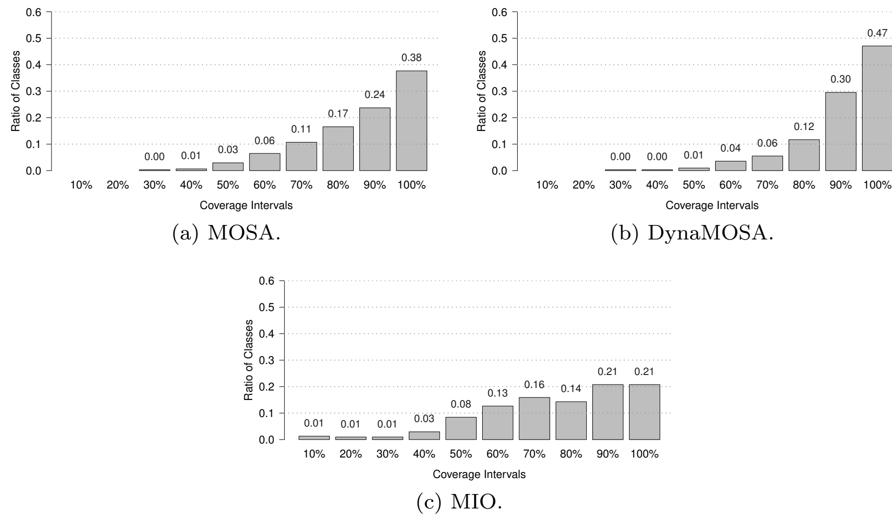
Table 5 summarises the results of a pairwise tournament of all many objective algorithms, i.e., MOSA, DynaMOSA, LIPS, and MIO. For both single and multiple criteria configurations, DynaMOSA is ranked first (e.g., it was statistically significantly better on 391 comparisons and significantly worse on only 31 out of 924 comparisons), MOSA is second, followed by MIO and then LIPS. As we discussed in RQ1, the most effective algorithm (i.e., the one with more positive comparisons) is the one with the highest ratio of classes with a coverage between $]90\%, 100\%]$. For DynaMOSA, 70% of all classes fall into the $]90\%, 100\%]$ interval, while for MOSA this number is lower at 67%, for MIO at 54%, and LIPS only managed to achieve coverage in this interval for 35% of classes (see Figure 6). For the multiple criteria configuration, for DynaMOSA 77% of all classes under test fall into the $]90\%, 100\%]$ interval, for MOSA it is 62%, and for MIO 42% (see Figure 7).



(a) MOSA. (b) DynaMOSA.

(c) LIPS. (d) MIO.

Figure 6: Proportion of classes that have an average branch coverage (averaged out of 30 runs on all their classes) within each 10% branch coverage interval. X-labels show the upper limit (inclusive). For example, the group 30% represents all the classes with an average branch coverage greater than 20% and lower than or equal to 30%.



(a) MOSA. (b) DynaMOSA.

(c) MIO.

Figure 7: Proportion of classes that have an average overall coverage (averaged out of 30 runs on all their classes) within each 10% overall coverage interval. X-labels show the upper limit (inclusive). For example, the group 30% represents all the classes with an average overall coverage greater than 20% and lower than or equal to 30%.

The ranking of many-objective algorithms for single criteria (i.e., branch coverage) is in line with previous studies in which DynaMOSA outperformed its predecessor MOSA [11], and MOSA in turn was more effective than LIPS [28] at generating test cases for Java static methods with purely procedural behaviour. Note that although MOSA and DynaMOSA achieve the same branch coverage for single criteria on average, DynaMOSA is statistically significantly better on more comparisons (391 vs 370) and significantly worse on less comparison (31 vs 51) than MOSA. Thus, DynaMOSA is statistically better than MOSA. MIO achieves a branch coverage of 75% for single criteria, and 71% overall coverage for multiple criteria (see Table 6); therefore it is ranked third. This result is different to two studies conducted by Arcuri [30, 39], where MIO performed better than MOSA. Our conjecture is that the testing level influences this difference: Arcuri [30, 39] performed an empirical evaluation on the automatic generation of *system tests*, and we performed an empirical evaluation on the automatic generation of *unit tests*. Besides the larger number of coverage goals in system testing, a main difference is that system tests are usually computationally more expensive to execute than unit test, which would benefit algorithms with small populations, such as MIO. On the other hand, algorithms with large populations (e.g., Standard GA) would take longer for evaluating the fitness of its individuals, and therefore fewer solutions would be explored.

RQ3: DynaMOSA outperforms the other many-objective algorithms for individual and multiple criteria.

4.4. RQ4 – How does evolution of whole test suites compare to many-objective optimisation of test cases?

Table 6 compares each EA with the many-objective optimisation algorithms MOSA, DynaMOSA, LIPS, and MIO.

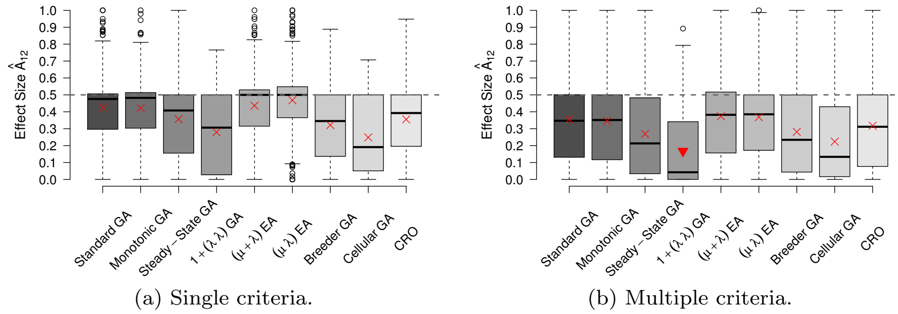
Our results confirm and enhance previous studies [9, 11] by evaluating eight different EAs (i.e., Standard GA, Steady-State GA, $1 + (\lambda, \lambda)$ GA, $(\mu + \lambda)$ EA, (μ, λ) EA, Breeder GA, Cellular GA, CRO) in addition to Monotonic GA, and show that MOSA and DynaMOSA perform better at optimising test cases than

Algorithm	Branch Cov.	Overall Cov.	vs. MOSA $\hat{A}_{12}$	p	vs. DynaMOSA $\hat{A}_{12}$	p	vs. LIPS $\hat{A}_{12}$	p	vs. MIO $\hat{A}_{12}$	p
Search budget of 60 seconds – Single-criteria										
MOSA	0.82	—	—	—	0.47	0.31	0.78	0.05	0.64	0.15
DynaMOSA	0.82	—	0.53	0.31	—	—	0.78	0.05	0.65	0.12
LIPS	0.62	—	0.22	0.05	0.22	0.05	—	—	0.28	0.08
MIO	0.75	—	0.36	0.15	0.35	0.12	0.72	0.08	—	—
Standard GA	0.79	—	0.43	0.16	0.41	0.15	0.77	0.05	0.56	0.17
Monotonic GA	0.79	—	0.42	0.15	0.40	0.15	0.76	0.06	0.56	0.15
Steady-State GA	0.76	—	0.36	0.10	0.35	0.11	0.74	0.09	0.47	0.13
$1 + (\lambda, \lambda)$ GA	0.61	—	0.28	0.10	0.28	0.09	0.50	0.10	0.35	0.14
$(\mu + \lambda)$ EA	0.79	—	0.44	0.15	0.42	0.15	0.77	0.05	0.58	0.15
(μ, λ) EA	0.81	—	0.47	0.17	0.45	0.15	0.79	0.05	0.61	0.17
Breeder GA	0.72	—	0.32	0.10	0.31	0.10	0.68	0.08	0.43	0.14
Cellular GA	0.67	—	0.25	0.06	0.24	0.05	0.63	0.09	0.34	0.09
CRO	0.74	—	0.36	0.11	0.34	0.11	0.71	0.07	0.48	0.16
Search budget of 60 seconds – Multiple-criteria										
MOSA	0.79	0.81	—	—	0.37	0.17	—	—	0.72	0.10
DynaMOSA	0.84	0.86	0.63	0.17	—	—	—	—	0.78	0.09
LIPS	—	—	—	—	—	—	—	—	—	—
MIO	0.68	0.71	0.28	0.10	0.22	0.09	—	—	—	—
Standard GA	0.71	0.76	0.35	0.10	0.29	0.08	—	—	0.60	0.14
Monotonic GA	0.71	0.75	0.35	0.10	0.28	0.09	—	—	0.58	0.13
Steady-State GA	0.65	0.70	0.27	0.09	0.22	0.05	—	—	0.47	0.11
$1 + (\lambda, \lambda)$ GA	0.48	0.54	**0.17**	**0.05**	**0.14**	**0.04**	—	—	0.24	0.08
$(\mu + \lambda)$ EA	0.72	0.77	0.37	0.12	0.30	0.09	—	—	0.63	0.11
(μ, λ) EA	0.72	0.77	0.37	0.14	0.30	0.09	—	—	0.62	0.13
Breeder GA	0.66	0.71	0.28	0.10	0.23	0.07	—	—	0.48	0.13
Cellular GA	0.61	0.66	0.22	0.07	**0.18**	**0.04**	—	—	0.40	0.13
CRO	0.69	0.73	0.32	0.11	0.26	0.09	—	—	0.53	0.12

Table 6: Comparison of evolutionary algorithms on whole test suites optimisation and many-objective optimisation algorithms of test cases. Statistically significant effect sizes are shown in bold.

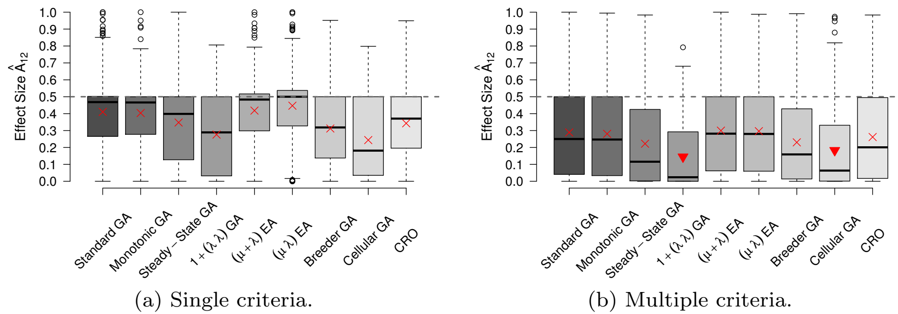
any EA at optimising test suites for single and multiple criteria (see Figures 8 and 9). Interestingly, and unlike any other algorithm, DynaMOSA achieves higher branch coverage on multiple criteria than on single criteria. This shows that DynaMOSA is suitable for optimising a large number of coverage goals (which is to be expected in a multiple criteria configuration) without negative effects on the final coverage.

We can only include LIPS in the single criterion scenario; here, all EAs performed better than LIPS (see Figure 10). When compared to MIO, only four EAs performed better than MIO for both single and multiple criteria: Standard GA, Monotonic GA, $(\mu + \lambda)$ EA, and (μ, λ) EA (see Figure 11).



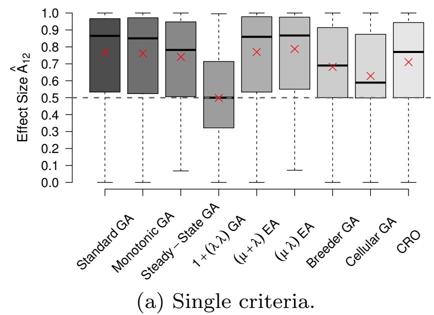
(a) Single criteria. (b) Multiple criteria.

Figure 8: Effect size $\hat{A}_{12}$ of EA X vs. MOSA. Please refer to Figure 4 for an explanation of each symbol.



(a) Single criteria. (b) Multiple criteria.

Figure 9: Effect size $\hat{A}_{12}$ of EA X vs. DynaMOSA. Please refer to Figure 4 for an explanation of each symbol.

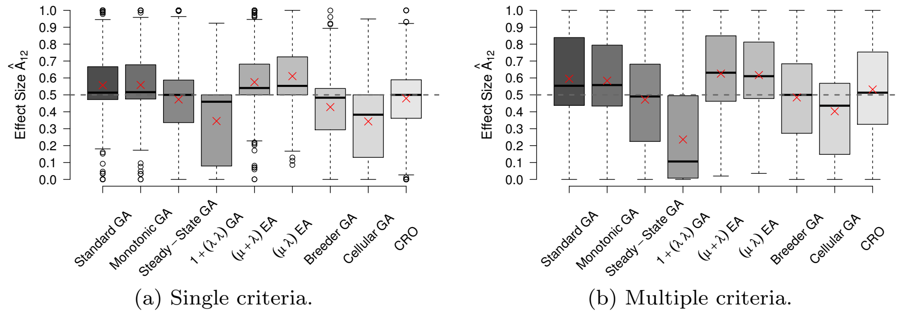


(a) Single criteria.

Figure 10: Effect size $\hat{A}_{12}$ of EA X vs. LIPS. Please refer to Figure 4 for an explanation of each symbol.

701

RQ4: DynaMOSA outperforms any EA at optimising test suites for individual and multiple criteria.



(a) Single criteria.

(b) Multiple criteria.

Figure 11: Effect size $\hat{A}_{12}$ of EA X vs. MIO. Please refer to Figure 4 for an explanation of each symbol.

4.5. Discussion

Given the results of our study, we now discuss some of the implications and insights.

4.5.1. Does the choice of evolutionary algorithm matter?

In line with common wisdom on evolutionary algorithms, there is not a single EA that works best in all scenarios. Our experiments do, however, provide evidence that the choice of algorithm has a substantial impact in the coverage achieved in test generation. For instance, the range of branch coverage achieved by each EA for single criteria goes from 61% ($1 + (\lambda, \lambda)$ EA) up to 82% (DynaMOSA), and the overall coverage for multiple criteria from 54% up to 86% (see Figure 2). Thus, clearly the choice of evolutionary algorithm matters.

4.5.2. Does the representation of individuals in an evolutionary algorithm matter?

All EAs except MOSA, DynaMOSA, LIPS, and MIO represent the individuals of a population as test suites (i.e., a sets of test cases). On the other hand, algorithms such as MOSA, DynaMOSA, LIPS, and MIO represent individuals as test cases. An interesting question for future work therefore is to study the influence of the representation on the effectiveness of the search.

Algorithm	Branch Cov.	σ	CI	Overall Cov.	σ	CI	$\hat{A}_{12}$	#CUT
Search budget of 60 seconds – Single-criteria								
Standard GA	1.00	0.01	[1.00, 1.00]	—	—	—	0.94	1
Monotonic GA	0.93	0.08	[0.90, 0.96]	—	—	—	0.76	1
Steady-State GA	0.65	0.05	[0.63, 0.67]	—	—	—	0.84	1
(μ, λ) EA	0.67	0.23	[0.59, 0.75]	—	—	—	0.85	5
MOSA	0.85	0.08	[0.82, 0.88]	—	—	—	0.89	2
DynaMOSA	0.89	0.06	[0.88, 0.92]	—	—	—	0.93	21
MIO	0.69	0.12	[0.65, 0.74]	—	—	—	0.92	3

Table 7: Number of classes on which an EA X performed significantly better than all the other evaluated EAs. Note: for multiple criteria, no EA performed significantly better than all the other evaluated EAs for any class under test (CUT).

720 *4.5.3. Is there room for improvements?*

721 Table 7 reports the number of classes under test to which an EA X performed
722 significantly better than all the other evaluated EAs. For instance, for single
723 criteria DynaMOSA performed significantly better than all the other EAs for 21
724 classes, (μ, λ) EA for 5 classes, MIO for 3 classes, MOSA performed significantly
725 better for 2 classes, and Standard GA, Monotonic GA, and Steady-State GA
726 for only 1 class. Considering that there are classes on which other EAs (e.g.,
727 MIO) performed better than DynaMOSA, there might be potential to improve
728 DynaMOSA by incorporating some of MIO's features into DynaMOSA. For
729 example, rather than generating an offspring based on the population, in each
730 iteration DynaMOSA could (given a certain probability) sample individuals,
731 that still do not satisfy some coverage goals, from the archive as MIO does.
732 There may also be potential to develop entirely new search algorithms tailored
733 for test generation.

734 *4.5.4. Technical Limitations*

735 Overall, there is a large number of classes under test for which EAs were
736 able to achieve high coverage. For example, DynaMOSA covered half of all
737 classes under test with a branch coverage between 90% and 100%. However,
738 there are some classes for which all EAs and random approaches evaluated in
739 our empirical study failed to achieve any substantial coverage due to limitations

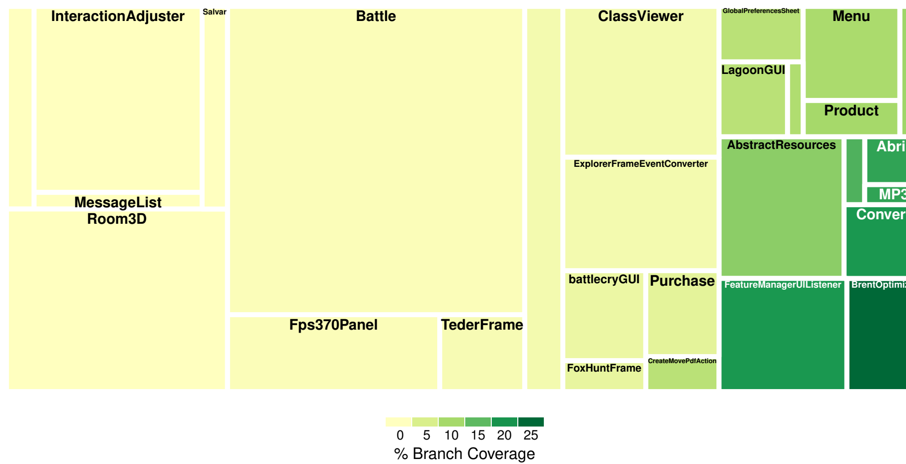


Figure 12: Classes on which all evaluated EAs and random approaches achieved less than 25% branch coverage. The area of each box is proportional to the number of branches in each class, and the colour represents the coverage achieved averaged over 30 repetitions.

of the test generation tool. Figure 12 shows the 28 classes on which all EAs and all random approaches failed to achieve more than 25% branch coverage. We looked closer at three problematic classes that stand out particularly:

1. `Battle` class from project `feudalismgame`, which represents the largest area in the figure. It consists of 786 branch goals, however only 1% of all goals have been covered. Despite the fact the class `Battle` is composed by eight public methods, all of them are invoked with Java reflection as described in the following snippet of code:

Listing 2: Piece of code from class `Battle` of project `feudalismgame`.

```
// Arguments for battle follows the following order: 1) Method name (attack
// target), e.g., vassal; 2) Attacker's Name
public void perform(Collection args) {
  try {
    Iterator argsIter = args.iterator();
    // The following will call a method dinamically according to the item
    // the player wants to buy
    Class aMethod = this.getClass().forName("feudalism.Battle");
    Class[] argType = {String.class};
```

```
        Method methodObj = aMethod.getMethod((String)argsIter.next(), new
            Class[]{Collection.class});
        methodObj.invoke(this, args);
        GameAutoActions.saveAll();
      } catch (Exception e) {
        e.printStackTrace();
    } }
```

Thus, in order to cover the methods of the class under test and therefore their branches, EvoSuite would have to generate a string parameter exactly as the name of one of the methods. When a string is required, EvoSuite either randomly generates one (with a certain probability) or uses static / dynamic seeds from the class under test [40]. Static seeds are all string constants in the bytecode of the class under test, and dynamic seeds are strings observed at runtime, for example, a call to the `equals` method of `String` class. It would be of interest to extend EvoSuite to also seed the name of methods or class fields for cases such as this particular one that uses Java reflection to invoke methods of the class under test.

2. `MP3` class from project `celwars2009`, which represents the smallest area in the figure (i.e., the smallest class represented in Figure 12). Although it only consists of 10 branch goals, EAs only managed to achieve a branch coverage of 18%. Although EvoSuite has been extended to support environment requirements such as interactions with the file system, console inputs, and many non-deterministic functions of the Java Virtual Machine (JVM) such as date and time [41], this particular class under test requires an MP3 file to successfully exercise the code under test, as described in the following snippet of code:

Listing 3: Piece of code from class `MP3` of project `celwars2009`.

```
public class MP3 extends Thread {
  AudioInputStream in = null;
```

```
AudioInputStream din = null;
String filename = "";

public MP3(String filename) {
  this.filename = filename; this.start();
}

public void run() {
  AudioInputStream din = null;
  try {
    File file = new File(filename);
    AudioInputStream in = AudioSystem.getAudioInputStream(file);
    AudioFormat baseFormat = in.getFormat();
    // +18 lines of code that are never executed because 'file' does not
    // point to a valid mp3 file
  } catch (Exception e) {
    e.printStackTrace();
  } finally {
    if (din != null) {
      try { din.close(); } catch(IOException e) { }
    }
} } }
```

Without guidance, EvoSuite is unlikely to produce data that represents valid MP3 files. To increase the adoption of EvoSuite, it would be of interest to extend it to generate not only music files, but also other types of files, e.g., image files that could be required to test a graphics editor software.

3. `MessageList` class from project `bpmail`. Despite the fact that it only consists of 24 branch goals, no EA or random approach was able to cover any goal at all. `MessageList` is an abstract class for which there is no concrete class, i.e., a non-abstract class that extends it, in the project. Therefore, no new objects of type `MessageList` could have been created. Although EvoSuite has been extended to mock certain type of classes,

823 e.g., interfaces [42], it will have to be further extended to handle cases such
824 as this one, i.e., an abstract class without a concrete class to instantiate.

825 These examples suggest that there are fundamental *technical* challenges
826 sometimes prohibiting high code coverage in practice; the choice of search al-
827 gorithm in such cases is minor. Consequently, it will be important to drive
828 research not only on algorithmic improvements, but to also accompany these
829 improvements with advances in the engineering of test generation tools.

830 **5. Related Work**

831 Although a common approach in search-based testing is to use genetic al-
832 gorithms, numerous other algorithms have been proposed in the domain of
833 nature-inspired algorithms, as no algorithm can be best on all domains [34].
834 Many researchers compared evolutionary algorithms to solve problems in do-
835 mains outside software engineering [43, 44, 45]. Within search-based software
836 engineering, comparative studies have been conducted in several domains such
837 as discovery of software architectures [46], pairwise testing of software product
838 lines [47], test case selection [48], or finding subtle higher order mutants [49].

839 In the context of test data generation, Harman and McMinn [50] empirically
840 compared GA, Random testing and Hill Climbing for structural test data gen-
841 eration. While their results indicate that sophisticated evolutionary algorithms
842 can often be outperformed by simpler search techniques, there are more complex
843 scenarios (e.g., test data generation for Matlab Simulink models [51]), for which
844 evolutionary algorithms are better suited. Ghani et al. [51] compared Simulated
845 Annealing (SA) and GA for the test data generation for Matlab Simulink mod-
846 els, and their results show that GA performed slightly better than SA. Sahin and
847 Akay [52] evaluated Particle Swarm Optimisation (PSO), Differential Evolution
848 (DE), Artificial Bee Colony, Firefly Algorithm and Random search algorithms
849 on software test data generation benchmark problems, and concluded that some
850 algorithms performs better than others depending on the characteristics of the
851 problem. Varshney and Mehrotra [53] proposed a DE-based approach to gener-
852 ate 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. [5] 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 cover more branches when standard fitness functions provide guidance, but most branches of the analyzed projects provided no such guidance. Similarly, Sahin and Akay [52] showed that Random search is effective on simple problems.

Recently, Scalabrino et al. [27] compared LIPS (Linearly Independent Path-Based Search) and MOSA (Many-Objective Sorting Algorithm) [9] with respect to generating test data for C programs. They used 35 simple C functions extracted from different open-source C libraries on their evaluation. Results show that there are no major differences between LIPS and MOSA when it comes to branch coverage. However, authors found that LIPS outperforms MOSA with respect to running time, but MOSA produces shorter test suites. Motivated by the several threats to the validity of such empirical evaluation (e.g., most subjects are trivial and can be fully covered in a few seconds), Panichella et al. [28] replicated this empirical study by comparing LIPS and MOSA in different settings: LIPS were implemented within EvoSuite [2] and 33 functions from the original benchmark were implemented as Java static methods. Additionally, 37 static methods were randomly selected from open source libraries, which means the evaluation was performed over 70 subjects. Results show that the new LIPS implementation is superior than the original implementation given the flexibilities provided by EvoSuite. They noticed that the new LIPS implementation

884 reached higher branch coverage using less time budget. Despite these improve-
885 ments, results show that MOSA is more effective and efficient than LIPS when
886 new and more complex subjects are considered.

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

892 **6. Conclusions**

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

896 • The choice of algorithm can have a substantial influence on the perfor-
897 mance of whole test suite optimisation, hence tuning is important. While
898 EvoSuite provides tuned default values, these values may not be optimal
899 for different flavours of evolutionary algorithms.

900 • Although previous studies showed little benefit of using a GA over random
901 testing, our study shows that on complex classes and with a test archive,
902 evolutionary algorithms are on average superior to random testing and
903 random search.

904 • The Dynamic Many Objective Sorting Algorithm (DynaMOSA) is su-
905 perior to whole test suite optimisation and other many objective search
906 algorithms.

907 It would be of interest to extend our experiments to further search algo-
908 rithms. In particular, the use of other non-functional attributes such as read-
909 ability [54] suggests the exploration of multi-objective algorithms. Considering
910 the variation of results with respect to different configurations and classes under
911 test, it would also be of interest to use these insights to develop hyper-heuristics
912 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, the European Research Council (ERC) under the European Union's Horizon 2020 research and innovation programme (grant agreement No 694277) and the Research Council of Norway (grant agreement No 274385).

References

[1] G. Fraser, A. Arcuri, Whole Test Suite Generation, IEEE Transactions on Software Engineering 39 (2) (2013) 276–291.

[2] G. Fraser, A. Arcuri, EvoSuite: Automatic Test Suite Generation for Object-Oriented Software, in: Proceedings of the 19th ACM SIGSOFT symposium and the 13th European conference on Foundations of software engineering, ESEC/FSE '11, ACM, New York, NY, USA, 2011, pp. 416–419. doi:10.1145/2025113.2025179.
URL http://doi.acm.org/10.1145/2025113.2025179

[3] G. Fraser, A. Arcuri, A Large-Scale Evaluation of Automated Unit Test Generation Using EvoSuite, ACM Transactions on Software Engineering and Methodology (TOSEM) 24 (2) (2014) 8:1–8:42. doi:10.1145/2685612.

[4] G. Fraser, A. Arcuri, Evolutionary Generation of Whole Test Suites, in: Proceedings of the 2011 11th International Conference on Quality Software, QSIC '11, IEEE Computer Society, Washington, DC, USA, 2011, pp. 31–40. doi:10.1109/QSIC.2011.19.
URL http://dx.doi.org/10.1109/QSIC.2011.19

[5] S. Shamshiri, J. M. Rojas, G. Fraser, P. McMinn, Random or genetic algorithm search for object-oriented test suite generation?, in: Proceedings of the Conference on Genetic and Evolutionary Computation, ACM, 2015, pp. 1367–1374.

941 [6] J. M. Rojas, J. Campos, M. Vivanti, G. Fraser, A. Arcuri, Combining Mul-
942 tiple Coverage Criteria in Search-Based Unit Test Generation, in: M. Bar-
943 ros, Y. Labiche (Eds.), Search-Based Software Engineering: 7th Interna-
944 tional Symposium, SSBSE 2015, Bergamo, Italy, September 5-7, 2015,
945 Proceedings, Springer International Publishing, Cham, 2015, pp. 93–108.
946 doi:10.1007/978-3-319-22183-0_7.
947 URL http://dx.doi.org/10.1007/978-3-319-22183-0_7

948 [7] G. Gay, The Fitness Function for the Job: Search-Based Generation of Test
949 Suites That Detect Real Faults, in: 2017 IEEE International Conference on
950 Software Testing, Verification and Validation (ICST), 2017, pp. 345–355.
951 doi:10.1109/ICST.2017.38.

952 [8] J. M. Rojas, M. Vivanti, A. Arcuri, G. Fraser, A Detailed Investigation
953 of the Effectiveness of Whole Test Suite Generation, Empirical Software
954 Engineering.

955 [9] A. Panichella, F. M. Kifetew, P. Tonella, Reformulating branch coverage as
956 a many-objective optimization problem, in: Software Testing, Verification
957 and Validation (ICST), 2015 IEEE 8th International Conference on, IEEE,
958 2015, pp. 1–10.

959 [10] J. Campos, Y. Ge, G. Fraser, M. Eler, A. Arcuri, An Empirical Evalua-
960 tion of Evolutionary Algorithms for Test Suite Generation, in: T. Menzies,
961 J. Petke (Eds.), Proceedings of the 9th International Symposium Search-
962 Based Software Engineering (SSBSE), Springer International Publishing,
963 Cham, 2017, pp. 33–48. doi:10.1007/978-3-319-66299-2_3.
964 URL https://doi.org/10.1007/978-3-319-66299-2_3

965 [11] A. Panichella, F. Kifetew, P. Tonella, Automated Test Case Generation
966 as a Many-Objective Optimisation Problem with Dynamic Selection of the
967 Targets, IEEE Transactions on Software Engineering PP (99) (2017) 1–1.
968 doi:10.1109/TSE.2017.2663435.

969 [12] R. Storn, K. Price, Differential Evolution – A Simple and Efficient Heuristic
970 for global Optimization over Continuous Spaces, Journal of Global Opti-
971 mization 11 (4) (1997) 341–359. doi:10.1023/A:1008202821328.

972 [13] J. Knowles, D. Corne, The Pareto archived evolution strategy: a new
973 baseline algorithm for Pareto multiobjective optimisation, in: Proceed-
974 ings of the 1999 Congress on Evolutionary Computation-CEC99 (Cat. No.
975 99TH8406), Vol. 1, 1999, p. 105 Vol. 1. doi:10.1109/CEC.1999.781913.

976 [14] S. Salcedo-Sanz, J. Del Ser, I. Landa-Torres, S. Gil-López, J. Portilla-
977 Figueras, The Coral Reefs Optimization Algorithm: A Novel Metaheuristic
978 for Efficiently Solving Optimization Problems, The Scientific World Jour-
979 nal 2014. doi:10.1155/2014/739768.
980 URL http://dx.doi.org/10.1155/2014/739768

981 [15] G. Fraser, A. Arcuri, Handling Test Length Bloat, Software Testing, Verifi-
982 cation and Reliability (STVR) 23 (7) (2013) 553–582. doi:10.1002/stvr.
983 1495.
984 URL http://dx.doi.org/10.1002/stvr.1495

985 [16] P. McMinn, Search-based Software Test Data Generation: A Survey, Soft-
986 ware Testing, Verification and Reliability 14 (2) (2004) 105–156. doi:
987 10.1002/stvr.v14:2.
988 URL http://dx.doi.org/10.1002/stvr.v14:2

989 [17] J. Wegener, A. Baresel, H. Sthamer, Evolutionary Test Environment
990 for Automatic Structural Testing, Information and Software Tech-
991 nology 43 (14) (2001) 841–854. doi:http://dx.doi.org/10.1016/
992 S0950-5849(01)00190-2.
993 URL http://www.sciencedirect.com/science/article/pii/
994 S0950584901001902

995 [18] A. Arcuri, It Does Matter How You Normalise the Branch Distance in
996 Search Based Software Testing, in: Software Testing, Verification and Vali-

997 dation (ICST), 2010 Third International Conference on, 2010, pp. 205–214.
998 doi:10.1109/ICST.2010.17.

999 [19] D. C. Karnopp, Random search techniques for optimization problems, Au-
1000 tomatica 1 (2-3) (1963) 111–121.

1001 [20] H. Mühlenbein, D. Schlierkamp-Voosen, Predictive Models for the Breeder
1002 Genetic Algorithm I. Continuous Parameter Optimization, Evolutionary
1003 Computation 1 (1) (1993) 25–49. doi:10.1162/evco.1993.1.1.25.
1004 URL http://dx.doi.org/10.1162/evco.1993.1.1.25

1005 [21] E. Alba, B. Dorronsoro, Cellular Genetic Algorithms, Operations Re-
1006 search/Computer Science Interfaces Series, Springer US, 2009.

1007 [22] B. Doerr, C. Doerr, F. Ebel, From black-box complexity to designing new
1008 genetic algorithms, Theoretical Computer Science 567 (2015) 87–104.

1009 [23] I. Rechenberg, Evolutionsstrategien, in: Simulationsmethoden in der Medi-
1010 zin und Biologie, Springer, 1978, pp. 83–114.

1011 [24] A. Y. S. Lam, V. O. K. Li, Chemical-Reaction-Inspired Metaheuristic for
1012 Optimization, IEEE Transactions on Evolutionary Computation 14 (3)
1013 (2010) 381–399. doi:10.1109/TEVC.2009.2033580.

1014 [25] S. Kirkpatrick, C. D. Gelatt, M. P. Vecchi, Optimization by Simulated
1015 Annealing, Science 220 (4598) (1983) 671–680. doi:10.1126/science.
1016 220.4598.671.
1017 URL http://www.sciencemag.org/content/220/4598/671.abstract

1018 [26] A. Y. S. Lam, V. O. K. Li, Chemical Reaction Optimization: a
1019 tutorial, Memetic Computing 4 (1) (2012) 3–17. doi:10.1007/
1020 s12293-012-0075-1.
1021 URL https://doi.org/10.1007/s12293-012-0075-1

1022 [27] S. Scalabrino, G. Grano, D. Di Nucci, R. Oliveto, A. De Lucia, Search-
1023 Based Testing of Procedural Programs: Iterative Single-Target or Multi-

1024 target Approach?, Springer International Publishing, Cham, 2016, pp. 64–
1025 79. doi:10.1007/978-3-319-47106-8_5.

1026 [28] A. Panichella, F. M. Kifetew, P. Tonella, LIPS vs MOSA: A Replicated Em-
1027 pirical Study on Automated Test Case Generation, Springer International
1028 Publishing, Cham, 2017, pp. 83–98. doi:10.1007/978-3-319-66299-2_6.

1029 [29] K. Deb, S. Agrawal, A. Pratap, T. Meyarivan, A fast elitist non-dominated
1030 sorting genetic algorithm for multi-objective optimization: NSGA-II,
1031 in: International Conference on Parallel Problem Solving From Nature,
1032 Springer, 2000, pp. 849–858.

1033 [30] A. Arcuri, Many Independent Objective (MIO) Algorithm for Test Suite
1034 Generation, Springer International Publishing, Cham, 2017, pp. 3–17. doi:
1035 10.1007/978-3-319-66299-2_1.

1036 [31] A. Arcuri, G. Fraser, Parameter tuning or default values? an empirical
1037 investigation in search-based software engineering, Empirical Software En-
1038 gineering 18 (3) (2013) 594–623.

1039 [32] S. Nakagawa, A farewell to Bonferroni: the problems of low statisti-
1040 cal power and publication bias, Behavioral Ecology 15 (6) (2004) 1044–
1041 1045. arXiv:/oup/backfile/content_public/journal/beheco/15/6/
1042 10.1093_beheco_arh107/2/arh107.pdf, doi:10.1093/beheco/arh107.
1043 URL http://dx.doi.org/10.1093/beheco/arh107

1044 [33] T. V. Perneger, What's wrong with Bonferroni adjustments, British Med-
1045 ical Journal 316 (7139) (1998) 1236–1238. doi:10.1136/bmj.316.7139.
1046 1236.
1047 URL https://doi.org/10.1136%2Fbmj.316.7139.1236

1048 [34] D. H. Wolpert, W. G. Macready, No free lunch theorems for optimization,
1049 IEEE transactions on evolutionary computation 1 (1) (1997) 67–82.

1050 [35] S. Shamshiri, J. M. Rojas, L. Gazzola, G. Fraser, P. McMinn, L. Mariani,
1051 A. Arcuri, Random or Evolutionary Search for Object-Oriented Test Suite
1052 Generation?, Software Testing, Verification and Reliability.

1053 [36] T. Jansen, K. A. De Jong, I. Wegener, On the choice of the offspring pop-
1054 ulation size in evolutionary algorithms, Evolutionary Computation 13 (4)
1055 (2005) 413–440.

1056 [37] Evolutionary Algorithms study – full data, http://www.evosuite.
1057 org/experimental-data/evolutionary-algorithm-study/, [Online; ac-
1058 cessed June-2008] (2018).

1059 [38] A. Aleti, I. Moser, L. Grunske, Analysing the Fitness Landscape of Search-
1060 based Software Testing Problems, Automated Software Engineering 24 (3)
1061 (2017) 603–621. doi:10.1007/s10515-016-0197-7.
1062 URL https://doi.org/10.1007/s10515-016-0197-7

1063 [39] A. Arcuri, Test suite generation with the Many Independent
1064 Objective (MIO) algorithm, Information and Software Technol-
1065 ogydoi:https://doi.org/10.1016/j.infsof.2018.05.003.
1066 URL http://www.sciencedirect.com/science/article/pii/
1067 S0950584917304822

1068 [40] J. M. Rojas, G. Fraser, A. Arcuri, Seeding Strategies in Search-based Unit
1069 Test Generation, Software Testing, Verification & Reliability 26 (5) (2016)
1070 366–401. doi:10.1002/stvr.1601.
1071 URL https://doi.org/10.1002/stvr.1601

1072 [41] A. Arcuri, G. Fraser, J. P. Galeotti, Automated Unit Test Generation
1073 for Classes with Environment Dependencies, in: Proceedings of the 29th
1074 ACM/IEEE International Conference on Automated Software Engineer-
1075 ing, ASE '14, ACM, New York, NY, USA, 2014, pp. 79–90. doi:10.1145/
1076 2642937.2642986.
1077 URL http://doi.acm.org/10.1145/2642937.2642986

1078 [42] A. Arcuri, G. Fraser, R. Just, Private API Access and Functional Mocking
1079 in Automated Unit Test Generation, in: 2017 IEEE International Confer-
1080 ence on Software Testing, Verification and Validation (ICST), 2017, pp.
1081 126–137. doi:10.1109/ICST.2017.19.

1082 [43] A. Basak, J. Lohn, A comparison of evolutionary algorithms on a set of
1083 antenna design benchmarks, in: L. G. de la Fraga (Ed.), 2013 IEEE Con-
1084 ference on Evolutionary Computation, Vol. 1, Cancun, Mexico, 2013, pp.
1085 598–604.

1086 [44] M. Wolfram, A. K. Marten, D. Westermann, A comparative study of evolu-
1087 tionary algorithms for phase shifting transformer setting optimization, in:
1088 2016 IEEE International Energy Conference (ENERGYCON), 2016, pp.
1089 1–6. doi:10.1109/ENERGYCON.2016.7514056.

1090 [45] E. Zitzler, K. Deb, L. Thiele, Comparison of multiobjective evolutionary
1091 algorithms: Empirical results, Evolutionary computation 8 (2) (2000) 173–
1092 195.

1093 [46] A. Ramírez, J. R. Romero, S. Ventura, A comparative study of many-
1094 objective evolutionary algorithms for the discovery of software architec-
1095 tures, Empirical Softw. Engg. 21 (6) (2016) 2546–2600. doi:10.1007/
1096 s10664-015-9399-z.

1097 [47] R. E. Lopez-Herrejon, J. Ferrer, F. Chicano, A. Egyed, E. Alba, Com-
1098 parative analysis of classical multi-objective evolutionary algorithms and
1099 seeding strategies for pairwise testing of software product lines, in: Pro-
1100 ceedings of the IEEE Congress on Evolutionary Computation, CEC, 2014,
1101 pp. 387–396.

1102 [48] A. P. Agrawal, A. Kaur, A comprehensive comparison of ant colony and
1103 hybrid particle swarm optimization algorithms through test case selection,
1104 in: Data Engineering and Intelligent Computing, Springer Singapore, Sin-
1105 gapore, 2018, pp. 397–405.

1106 [49] E. Omar, S. Ghosh, D. Whitley, Comparing search techniques for finding
1107 subtle higher order mutants, in: Proceedings of the Conference on Genetic
1108 and Evolutionary Computation, GECCO '14, ACM, 2014, pp. 1271–1278.
1109 doi:10.1145/2576768.2598286.

1110 [50] M. Harman, P. McMinn, A Theoretical & Empirical Analysis of Evolu-
1111 tionary Testing and Hill Climbing for Structural Test Data Generation,
1112 in: Proceedings of the 2007 International Symposium on Software Testing
1113 and Analysis, ISSTA '07, ACM, New York, NY, USA, 2007, pp. 73–83.
1114 doi:10.1145/1273463.1273475.
1115 URL http://doi.acm.org/10.1145/1273463.1273475

1116 [51] K. Ghani, J. A. Clark, Y. Zhan, Comparing algorithms for search-based
1117 test data generation of matlab simulink models, in: 2009 IEEE Congress
1118 on Evolutionary Computation, 2009, pp. 2940–2947. doi:10.1109/CEC.
1119 2009.4983313.

1120 [52] O. Sahin, B. Akay, Comparisons of metaheuristic algorithms and fitness
1121 functions on software test data generation, Applied Soft Computing 49
1122 (2016) 1202 – 1214.

1123 [53] S. Varshney, M. Mehrotra, A differential evolution based approach to gen-
1124 erate test data for data-flow coverage, in: 2016 International Conference on
1125 Computing, Communication and Automation (ICCCA), 2016, pp. 796–801.
1126 doi:10.1109/CCAA.2016.7813848.

1127 [54] E. Daka, J. Campos, G. Fraser, J. Dorn, W. Weimer, Modeling Readability
1128 to Improve Unit Tests, in: Proceedings of the 2015 10th Joint Meeting on
1129 Foundations of Software Engineering, ESEC/FSE 2015, ACM, New York,
1130 NY, USA, 2015, pp. 107–118. doi:10.1145/2786805.2786838.
1131 URL http://doi.acm.org/10.1145/2786805.2786838