Random or evolutionary search for object-oriented test suite generation?

— Source link ↗

Sina Shamshiri, José Miguel Rojas, Luca Gazzola, Gordon Fraser ...+3 more authors

Institutions: University of Sheffield, University of Leicester, University of Milano-Bicocca, University of Passau ...+1 more institutions

Related papers:

- Whole Test Suite Generation

- A detailed investigation of the effectiveness of whole test suite generation

- EvoSuite: automatic test suite generation for object-oriented software

- Mutation-Driven Generation of Unit Tests and Oracles

- Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of Effectiveness and Challenges (T)

Share this paper: 📘 🐦 in ✉

Random or Evolutionary Search for Object-Oriented Test Suite Generation?

Sina Shamshiri[1]*, José Miguel Rojas[2], Luca Gazzola[3], Gordon Fraser[4],

Phil McMinn[1], Leonardo Mariani[3] and Andrea Arcuri[5]

[1] *Department of Computer Science, University of Sheffield, United Kingdom*
[2] *Department of Informatics, University of Leicester, United Kingdom*
[3] *Department of Informatics, Systems and Communication, University of Milano-Bicocca, Italy*
[4] *Chair of Software Engineering II, University of Passau, Germany*
[5] *Westerdals Oslo ACT, Norway*

SUMMARY

An important aim in software testing is constructing a test suite with high structural code coverage – that is, ensuring that most if not all of the code under test has been executed by the test cases comprising the test suite. Several search-based techniques have proved successful at automatically generating tests that achieve high coverage. However, despite the well-established arguments behind using evolutionary search algorithms (e.g., genetic algorithms) in preference to random search, it remains an open question whether the benefits can actually be observed in practice when generating unit test suites for object-oriented classes. In this paper, we report an empirical study on the effects of using evolutionary algorithms (including a genetic algorithm and chemical reaction optimization) to generate test suites, compared with generating test suites incrementally with random search. We apply the EVOSUITE unit test suite generator to 1,000 classes randomly selected from the SF110 corpus of open source projects. Surprisingly, the results show that the difference is much smaller than one might expect: While evolutionary search covers more branches of the type where standard fitness functions provide guidance, we observed that, in practice, the vast majority of branches do not provide any guidance to the search. These results suggest that, although evolutionary algorithms are more effective at covering complex branches, a random search may suffice to achieve high coverage of most object-oriented classes.
Copyright © 2017 John Wiley & Sons, Ltd.

1. INTRODUCTION

Automatically generating software test cases is an important task, with the objective of improving software quality. Many different algorithms and techniques for different types of software testing problems have been proposed. One particular application area in which search-based techniques have been successfully applied is generating unit tests for object-oriented programs, where test cases are sequences of object constructor and method calls.

---

*Correspondence to: Sina Shamshiri (sina.shamshiri@sheffield.ac.uk)

Automatically generated tests can be used to reveal crashes and undeclared exceptions (e.g., Csallner and Smaragdakis [1], Pacheco and Ernst [2]), to capture the current behavior for regression testing (e.g., Fraser and Zeller [3], Xie [4]), or they can simply be presented to the developer in order to support them in creating test suites [5]. There are various search-based tools available for languages such as Java and .NET, ranging from tools based on random search such as Randoop [2], JCrasher [1], JTExpert [6], T3 [7], or Yeti-Test [8], to tools based on evolutionary search such as EvoSuite [9], eToc [10], NightHawk [11], or Testful [12].

Search-based unit test generation tools dominantly use Genetic Algorithms (GAs) [13, 14], and are often thought to be superior to tools based on random search. However, it is neither clear whether this is actually the case in practice, nor whether it generalizes to other evolutionary search techniques. It could be that differences in performance across tools may be accounted for by the differences in the programming language that they target, or in the way they have been engineered, as opposed to any specific benefits of the particular search algorithm that they apply. In order to shed more light on these questions, we report on experiments to contrast the use of more than one different evolutionary algorithm with random search when applied to open source Java classes.

Evolutionary search algorithms generally mimic the metaphor of natural biological evolution, the social behavior of species, or other natural processes. Besides the many variants of GAs and the related families of evolution strategies [15], there are various algorithms based on, for example, how ants find the shortest route to a source of food [16], the foraging behavior of honey bees [17], the swarm behaviour of bird flocks or fish schools [18], as well as physical and chemical processes like chemical reactions [19]. However, in the context of evolutionary test generation of object-oriented unit tests, examples and applications of these other algorithms are rare or do not exist. Since it is infeasible to implement all different algorithms, we aimed to identify one suitable alternative algorithm to increase the external validity of our experimental results and to study if the findings generalize to other evolutionary search algorithms. In particular, the criteria for the selection of the alternative algorithm were that: (1) it is a population-based "global" search algorithm, like GAs; (2) it is suitable for optimization of the discrete search domains of unit test generation and can make use of the same representation as a GA (see Section 2); and (3) it is not just a minor twist to a GA, but a considerably different algorithm. Since many optimization algorithms focus on continuous domains and thus are not straight forward to apply to unit test generation, we identified Chemical Reaction Optimization (CRO) [19] as a suitable algorithm matching our criteria. CRO has been reported to be a promising technique in other domains (e.g. Lam et al. [20, 21] , Xu et al. [22, 23], and Yu et al. [24]), but has not previously been applied to test generation. We adapt CRO such that both algorithms, GA and CRO, optimize unit test suites for code coverage, while the random search algorithm optimizes code coverage by adding random tests to a test suite.

To allow for a fair comparison in these experiments, we use an implementation of the GA and random test generation in the EvoSuite tool, which is a state-of-the-art search-based test generation tool for branch-coverage of Java classes, as demonstrated by its successes at the annual unit testing tool competition [25] and its comparison to other tools in terms of fault-finding [26]. We present an implementation of CRO for test generation within EvoSuite, and further include EvoSuite's archive of solutions in our experiment, which allows the evolutionary search to focus on uncovered code, resembling more how random search optimizes for coverage. We run experiments on a stratified random sample of 1,000 classes from the SF110 corpus of open source projects [27] and evaluate the three techniques in terms of the achieved code coverage.

This paper extends and consolidates our previous experiment results [28] as follows. We extend the empirical evaluation to compare random search to Chemical Reaction Optimization (CRO) in addition to the GA, to validate that the results obtained by

Shamshiri et al. [28] generalize to other search algorithms beyond GAs. Note that this paper is the first one to investigate the application of the CRO search algorithm to the test suite generation problem. To increase our confidence in the data, we increase the number of repetitions from 50 to 100 times for all experiments. Finally, we compare random search with an extended version of the GA whereby an archive of past solutions (test cases) is used to assist with the generation of test suites with high coverage. We also expand the background on search-based test generation, as well as our analysis of the results.

To summarize, the main contributions of this paper are:

(1) an empirical study comparing the effectiveness of evolutionary and random search-based algorithms for generating branch coverage test suites for real-world Java classes;

(2) a classification of the types of branches that exist in Java bytecode, and the search landscape they create;

(3) a thorough investigation of how the nature of the branches that must be covered influences the effectiveness of random and evolutionary methods;

(4) the first use of CRO for test generation as another evolutionary search algorithm and a comparison of its effectiveness against the GA and random search;

(5) an empirical investigation of how an archive of past solutions can improve the capability of GA to focus on the uncovered code;

(6) a study of the effect that an extended search budget may have on the effectiveness of evolutionary and random search.

The results of our experiments suggest that the difference between the use of evolutionary algorithms and random search is smaller in practice than one might expect. While the two approaches have different performance profiles over time, the main reason for this finding is actually because of the types of branches that are prevalent in object-oriented programs. Fitness-guided search algorithms like GAs or CRO work well when trying to cover branches that result in a smooth gradient of fitness values, which the search can "follow" to the required test case. These branches are typically characterized by numerical comparisons. However, our study found that in practice such "gradient branches" are relatively few in number; allowing random search to generate test cases without much relative disadvantage, and with a similar level of effectiveness. These contributions have implications for future research and practice in unit test case generation, as discussed in Section 6.

2. SEARCH-BASED TEST GENERATION

In this paper, we study the application of evolutionary and random search to automatic test suite generation, as implemented in the EVOSUITE tool. EVOSUITE aims to generate unit test suites that cover as many branches (i.e., true/false outcomes of conditional statements) of a Java class as possible, while also executing all methods without any branches, which we refer to as "branchless" methods.

The selected methods and algorithms represent state-of-the-art solutions. In particular, the use of random search integrated in EVOSUITE could benefit from the specific capabilities of the tool (e.g., seeding – as discussed in Section 2.1.1) guaranteeing a fair comparison with the other methods, which would be difficult to achieve otherwise. For instance, the annual unit testing tool competition [25] revealed that other random testing techniques not benefiting of the EVOSUITE infrastructure are less effective than the evolutionary algorithms defined in EVOSUITE [27]. Furthermore, the GA defined in EVOSUITE is a state-of-the-art unit test generation algorithm. While a GA is probably the most commonly applied search algorithm in search-based testing [13], any differences between a random approach and the GA raise the question of whether this is a result of specific aspects of the GA, or evolutionary search in general. Therefore, we selected CRO

as a relevant alternative evolutionary algorithm, which provides the intrinsic ability to integrate GA-style and Simulated Annealing-like strategies [13].

We begin by introducing our implementation of random search to the problem of generating test cases (which we may interchangeably refer to as *tests* in this paper), and then introduce the two evolutionary algorithms evaluated in this paper for generating complete test suites – the GA and CRO algorithms.

2.1. Random Search for Tests

One strategy for finding branch-covering test cases is simply to generate sequences of statements to the class under test at random, coupled with randomly-generated inputs. The test generator is given a list of methods and constructors to consider (or derives this list with static analysis), and iteratively selects a random one. This is inserted into the existing sequence of statements such that parameter objects of the inserted call can either be satisfied with existing objects in that sequence (i.e., return values of previous statements), or recursively calls constructors or methods that generate the required objects. For primitive parameter types (e.g., numbers or strings) random values are generated. If a randomly-generated test case covers new branches that have not been executed before, it is added to a test suite for the class, else it can be discarded. One disadvantage of this approach is the size of the resulting test suite, which can be very large and therefore carry a high execution cost.

A further problem is finding inputs that need to be certain "magic" values required to execute certain branches, such as constant values, specific strings, etc., that are unlikely to be generated by chance. One way of circumventing this problem is to enhance the algorithm through *seeding*.

2.1.1. Seeding.
The process of *seeding* involves biasing the search process towards certain input values that are likely to improve the chances of enhancing coverage [29, 30, 31]. EVOSUITE obtains seeds both statically and dynamically (as documented by Rojas et al. [32]). The static approach takes place before test generation: EVOSUITE collects all literal primitive and string values that appear in the bytecode of the class of the test. Then, while tests are being generated, literals that are encountered at runtime may also be dynamically added to the pool of seeds. Some of these seeds are specially computed, according to a set of predefined rules. For instance, if the test case includes the statement "foo.startsWith(bar)", involving the strings foo and bar, the concatenation bar + foo will be added to the seed pool. During the search process, EVOSUITE will then choose to use a seed from the pool instead of generating a fresh value, according to some probability.

We study random search with and without seeding enabled in this paper. We refer to the enhanced version of random search incorporating seeding as *Random+*, and the basic implementation without seeding as *Pure Random*.

2.2. Genetic Algorithm Search for Test Suites

While random search relies on encountering solutions by chance, guided searches aim to find solutions more directly by using a problem-specific "fitness function". A fitness function scores better potential solutions to the problem with better fitness values. A good fitness function will provide a gradient of fitness values so that the search can follow a "path" to increasingly better solutions that are increasingly fit for purpose. With a good fitness function, guided search-based approaches are capable of finding suitable solutions in extremely large or infinite search spaces (such as the space of possible test cases for a class as considered in this paper).

Genetic Algorithms (GAs) are one example of a directed search technique that uses simulated natural evolution as a search strategy. GAs evolve solutions to a problem based on their fitness. GAs evolve several candidate solutions at once in a "population". The

---
Algorithm 1 A genetic algorithm as used in search-based testing.

---

1    $seeds \leftarrow$ initialize $seeds$ with collected static literals from bytecode
2    $current\_population \leftarrow$ generate random population using $seeds$
3   **repeat**
4       $Z \leftarrow$ elite of $current\_population$
5   **while** $|Z| \neq |current\_population|$ **do**
6         $P_1, P_2 \leftarrow$ rank selection from $current\_population$
7   **if** crossover_probability **then**
8           $O_1, O_2 \leftarrow$ crossover P_1, P_2
9   **else**
10          $O_1, O_2 \leftarrow P_1, P_2$
11   **if** mutation_probability **then**
12   mutate O_1 and O_2 {The $seeds$ pool may be used}
13        $f_P = min(fitness(P_1), fitness(P_2))$
14        $f_O = min(fitness(O_1), fitness(O_2))$
15        $seeds \leftarrow$ update $seeds$ with collected dynamic seeds from fitness evaluations
16   **if** $f_O \leq f_P$ **then**
17          $Z \leftarrow Z \cup \{O_1, O_2\}$
18   **else**
19          $Z \leftarrow Z \cup \{P_1, P_2\}$
20       $current\_population \leftarrow Z$
21   **until** solution found or maximum resources spent

---

initial population of candidate solutions is generated randomly. Here, a *solution* is a test-suite, consisting of individual test cases that each contain a sequence of statements that invoke constructor calls and methods on the the class under test [33]. Each iteration of the algorithm seeks to adapt these solutions to ones with an increased fitness: "Crossover" works to splice two solutions to form new "offspring" while "mutation" randomly changes a component of a solution. The new solutions generated are taken forward to the next iteration depending on their fitness.

As shown in Algorithm 1, the GA first creates an initial population of solutions randomly (Line 2). Then, using rank selection, it selects two parents P_1 and P_2 (Line 6) and crosses them over (Line 7-10). With a certain probability, the GA applies the mutation operator on the resulting offspring O_1 and O_2 (Line 11-12), then it executes the solutions on the class under test, calculates their fitness values, and selects the chromosomes with the minimum (best) fitness values (Line 13 and 14), and then compares the fitness value of parents and their offspring to determine which one will be carried over to the next generation Z (Line 16-19). The GA repeats this process until a solution is found or the search budget is exhausted. During this process, to enhance the effectiveness of the generated solutions, the GA collects seeds statically and dynamically and uses them for the generation of new or mutated solutions (Line 1 and 15).

Crossover involves recombining test cases across two test suites while mutation works at two levels: at the test suite level and at test case level. At the test suite level, it adds fresh, randomly-generated test cases to an existing test suite, or selects individual tests for test case level mutation. Whenever a new test case is generated at random (when generating the initial population or during mutation of a test suite), this is done by starting with an empty test case and repeatedly applying mutation on it, with seeding enabled. Mutation of test cases either randomly adds new statements, removes existing ones, or modifies them and their parameters. Note that, in Algorithm 1, the `mutate` function (Line 12) encapsulates both mutation levels.

To guide the search towards achieving a high coverage test suite, the fitness value can be calculated based on the number of covered goals – where in the case of branch-coverage

with EVOSUITE, a *goal* is defined as either a branch or a branchless method. However, a fitness function based solely on the number of *covered* goals provides no guidance to goals that remain *uncovered*. As with previous work in search-based test generation [13], EVOSUITE incorporates branch distance metric [34], which indicates how "far" a branch is from being executed. For example, if a conditional "if (a == b)" is to be executed as true, the "raw" distance can be computed as "$|a - b|$". In this way, the closer the values of a and b are to one another, the lower the branch distance is, and the closer the search is to covering the goal. Note that the values or a and b in this example may not be directly controllable from the test cases, but may be internal variables set indirectly by the statements of a test as manipulated by the GA.

Since EVOSUITE aims to evolve test suites where each test case covers as many branches as possible, the fitness function involves adding the distance value $d(b, T)$ for each branch b within a test suite T, computed as follows [33]:

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

where $d_{min}(b,T)$ is the minimum raw distance value for the b for T, and ν is a function that normalizes a distance value between 0 and 1. Since the test suite must cover both the true and false outcomes of each individual branch, a distance value is not computed until the conditional is executed twice by the test suite. This is so that the initial execution of the predicate, with some specific true/false evaluation, is not lost in the process of pursuing the alternative outcome.

As longer test suites require more memory and execution time, controlling the length of the test suite can improve search performance [35]. Therefore, when deciding which test suites should proceed into the population for the next iteration of the search, EVOSUITE prefers shorter test suites to test suites with the same fitness but are composed of a higher number of statements.

Java programs are compiled to bytecode for execution on a Java Virtual Machine (JVM), and it is at the level of the bytecode at which EVOSUITE works – branch distances are computed by instrumenting and monitoring bytecode instructions. Different types of bytecode instruction can therefore give rise to different types of fitness landscapes that may or may not be useful in guiding the search, as we discuss in Section 3.

2.2.1. Archive of Tests. When generating solutions at a test suite level using a search algorithm, the computation of fitness values considers all coverage goals, even if they have already been covered by a test. This may have negative implications for the effectiveness of the search. For instance, the application of the mutation operator may lead a test suite to satisfy a particular coverage goal for the first time, but ceasing to satisfy two goals which were covered before the mutation. As a consequence, the search algorithm regards the mutation as detrimental and thus discards it, losing the improvement achieved by covering a new branch. A practical, effective solution to overcome this problem consists in using an archiving mechanism to store already covered goals and the tests that cover them, ensuring that the search keeps focused on as yet uncovered goals exclusively.

The use of an archive influences the search performance by changing the fitness evaluation, the mutation operation and the construction of the final solution. First, during fitness evaluation, each time a new goal has been covered, the GA adds it to the archive together with its covering test. Upon completion of the current iteration, the fitness function no longer takes these covered goals into account. Second, when the mutation operator adds a new test case to an existing test suite, the test added will be a mutated clone of an test stored in the archive instead of a purely random test, given a certain

probability. Finally, at the end of the search, the best individual is not directly regarded as the optimal test suite as customary. In contrast, and because this best individual may be missing goals that were covered by other individuals, the GA constructs the final result by incrementally extending the best individual with tests from the archive which enhance the resulting coverage. [36] provide more details and discuss the effectiveness of incorporating an archive of tests for search-based test generation.

2.3. Chemical Reaction Optimization

Chemical Reaction Optimization (CRO) is a metaheuristic algorithm that combines the advantages of population-based evolutionary algorithms, such as genetic algorithms, and simulated annealing [19]. CRO is inspired by real life chemical reactions, a process that transforms a set of molecules placed in a container to another set of molecules. In a reaction, the initial set of molecules is usually unstable but with high potential energy, while the set of molecules at the end of the reaction process is more stable but with less potential energy. Solving optimization problems with CRO requires mapping the possible solutions of a problem to molecules, the search operators to reactions, and the value of a solution to the potential energy of the molecules. Similar to evolutionary algorithms that manipulate individuals, the CRO optimization process manipulates molecules by iteratively applying reactions, transforming the initial set of molecules to a set of molecules with minimal potential energy. Similar to simulated annealing, CRO can accept reactions that increase rather then decrease the potential energy of the molecules. This is achieved by associating to molecules their kinetic energy, which represents the likelihood of a reaction that increases the potential energy.

Since CRO effectively combines global and local search operations by integrating GA-style and Simulated Annealing-style searches, it is a clear candidate for search-based unit test generation, which requires an effective local search strategy to cover branches, and also an effective global search strategy to effectively combine method calls.

In the following, we first present the basic CRO algorithm, and then describe how CRO has been instantiated to address test case generation.

2.3.1. CRO Algorithm. The CRO algorithm, shown in Algorithm 2, evolves an initial population of molecules executing reactions among molecules (called *collisions*). In particular, it uses four types of collision. The *on-wall ineffective collision* and the *inter-molecular ineffective collision* implement local transformations (i.e., they are local search operators) of one molecule and two molecules, respectively. The *decomposition* and *synthesis* collisions implement global transformations (i.e., they implement global search operations) of one molecule and two molecules, respectively. In the following, we describe how the CRO algorithm works, while we present the four collision types in the next subsection.

When running CRO, there are several parameters that must be defined. These parameters are specified as part of the signature of the CRO function shown in Algorithm 2 and are described below:

- f: is the fitness function, that is a function that associates to a molecule, which represents a possible solution, a potential energy, which represents the utility of that solution.
- $initSize$: is the initial number of molecules in the container
- $initKE$: is the initial value of the kinetic energy of each molecule
- $collRate$: is the probability that a collision is a collision between two molecules instead of a collision between a molecule and the container
- $KELossRate$: is the percentage of kinetic energy that a molecule loses after each collision
- $decThreshold$: is an integer value representing the number of ineffective collisions that can be tolerated before triggering a decomposition

---

Algorithm 2 A chemical reaction optimization algorithm adapted for search-based testing.

1 $seeds \leftarrow$ initialize $seeds$ with collected static literals from bytecode
2 $population$ = randomMolecules($initSize, initKE, seeds$)
3 **while** stopping criterion not met **do**
4      $r \leftarrow random[0, 1]$
5   **if** $r > collRate$ **then**
6   select a random molecule M
7   **if** M.hitsSinceLastMin() $> decThreshold$ **then**
8              $population$.decomposition($M, seeds, fitness$)
9   **else**
10             $population$.onWallIneffectiveCollision($M, KELossRate, seeds, fitness$)
11   **else**
12   select two random molecule M_1, M_2
13   **if** $M_1.KE < synThreshold$ AND $M_2.KE < synThreshold$ **then**
14         $population$.synthesis($M_1, M_2, seeds, fitness$)
15   **else**
16         $population$.interMolecularIneffectiveCollision($M_1, M_2, seeds, fitness$)
17     $seeds \leftarrow$ update $seeds$ with collected dynamic seeds from fitness evaluations
18   **if** $population.bestSolution.PE <= minimum.PE$ **then**
19         $minimum = population.bestSolution$
20   **else**
21   elitism($minimum, population$)
22 output $minimum$

- $synThreshold$: is the value of the kinetic energy under which two colliding molecules are automatically fused into one molecule

The algorithm starts with the generation of a random population of *initSize* molecules with *initKe* kinetic energy each (line 2 in Algorithm 2), and then enters the main loop (from line 3 to line 21). At each iteration, the main loop transforms the population of molecules while searching for the best solution.

The iteration starts by randomly selecting either a single or a multi-molecular collision (line 5). When a collision involving a single molecule is selected, CRO checks if the number of ineffective collisions that have not improved the potential energy of the selected molecule is greater than the parameter *decThreshold* (line 7). If the threshold has not been passed, local search is assumed to still have the potential to be useful and an *on-wall ineffective collision* is performed. Otherwise, CRO assumes it is not worth continuing with local searches based on that molecule and performs a global search based on *decomposition*.

When a collision involving two molecules is selected, CRO checks if the kinetic energy of both molecules is below the threshold *synThreshold* (line 13). If at least one of the values is above the threshold, local search is assumed to still have the potential to be useful and an *inter-molecular ineffective collision* is performed. Otherwise, CRO assumes it is not worth continuing with local searches based on the two selected molecules and performs a global search based on *synthesis*.

It is worth noting that the fitness function evaluation takes place within the search operators (collisions), unlike the previously presented genetic algorithm, where the fitness evaluation takes place after the mutation and crossover operations. This is due to the fact that, to calculate the residual kinetic energy of the molecules, the algorithm needs the potential energy value (which corresponds to the fitness value in GA).

At the end of each iteration, CRO checks if the current population includes the best solution discovered so far (line 18). If it is not the case, the best solution is automatically injected in the current population by replacing one of the existing molecules invoking the *elitism* function. Note that the original CRO algorithm does not include elitism. However,

we found that CRO without elitism is often too slow in reaching good solutions compared to the genetic algorithm used in EVOSUITE, and elitism worked well to mitigate this issue.

CRO iterates this procedure until a stopping criterion is satisfied. There are several options for the stopping criterion including the definition of a maximum number of iterations that can be executed, a maximum amount of time that can be spent evolving the molecules, and a level of potential energy that must be reached.

2.3.2. Test Case Generation with CRO. When CRO is used to generate test cases, molecules represent test suites. In particular, a single molecule represents a whole test suite with an arbitrary number of test cases. The potential energy of a molecule is the branch coverage achieved by the corresponding test suite. The evaluation of the fitness function on a molecule implies running the test cases associated with that molecule and collecting the coverage information. Collisions among molecules are used to evolve the test suites. In the following, for each collision type we first describe how it works in general, and then how it is specifically designed for test case generation.

Collisions are elementary reactions that change the structure of the molecules and their energy. Both potential and kinetic energies are influenced by the reactions. Since energy must always be balanced, the container is also associated with a potential energy that can be increased or decreased by the reactions.

Reactions can be local, that is they only require the molecules directly involved in the transformation to be performed, or global, that is they require more information than just the molecules involved in the reaction to be performed.

Local Reactions The *on-wall ineffective collision* is a reaction that involves one molecule only. It represents the case of a molecule hitting a wall of the container without producing any dramatic effect. This collision is modelled with the neighborhood search operator N that generates a new molecule x' that replaces the molecule x ($x' = N(x)$). The potential energy of x' is determined by the fitness function, that is $PE_{x'} = f(x')$. The molecule also loses some kinetic energy in the process (a random quantity, limited by a threshold) that is added to the energy of the container.

For the purpose of test case generation, N is implemented as an operator that mutates each test case in the test suite with probability $\frac{2}{|T|}$, where $|T|$ is the size of the test suite. When a test is mutated, each of its statements is mutated with a given probability (in the experiments we used a probability equals to 0.2) using one of the following statement-level operators: insert a statement, delete a statement, modify a statement.

The *inter-molecular ineffective collision* is a reaction that involves two molecules. It represents the case of two molecules hitting each other with little effect. This collision is essentially modelled as two independent on-wall ineffective collisions. In fact, the same operator N defined for the on-wall ineffective collision is used to mutate the two molecules. The energy is handled in a slightly different way because molecules can exchange energy.

Global Reactions The *decomposition* is a reaction that involves one molecule only. It represents the case of a molecule hitting the wall of the container and breaking into two or more molecules. In our evaluation, we only consider the case of two molecules, that is the decomposition operator D applied to a molecule x always produces two molecules ($D(x) = (x_1, x_2)$).

The energy of the original molecule must be enough to create two new molecules. Since it is often not the case, a small portion of energy can be withdrawn from the container and added to the kinetic energy of the newly generated molecules, making the decomposition less likely to fail.

For the purpose of test case generation, the decomposition operator is defined as an operator that generates new test cases by half random changes. In particular, given a test

suite x with n test cases, the operator returns two test suites x_1 and x_2 both with n test cases. The test suite x_1 inherits all the test cases of x in odd positions, while the test cases in even positions are generated randomly. The test suite x_2 inherits all the test cases of x in even positions, while the test cases in odd positions are generated randomly.

The *synthesis* is a reaction that involves two molecules. It represents the case of two molecules that collide fusing into one molecule. The synthesis operator S applied to two molecules x_1 and x_2 produces one molecule x' ($S(x_1, x_2) = x'$). The kinetic energy of the new molecule is the sum of the original molecules' kinetic energy.

For the purpose of test case generation, the synthesis operator is defined as an operator that generates a new test suite by selecting test cases from the two input test suites. If n_1 and n_2 are the number of test cases in the test suites x_1 and x_2, the test suite x' consists of the first an_1 test cases from x_1 and the last $(1-a)n_2$ test cases from x_2, where a is a random number in the range (0,1).

Seeding   Similar to the GA presented in Section 2.2, CRO also implements static and dynamic seeding for the generation of new or mutated solutions (Line 1 and 17).

3. BRANCH TYPES IN JAVA BYTECODE

Given that the fitness function is one of the key differences between the evolutionary and random search, and that a major component of the fitness function is the calculation of distances for the branches in the class under test, we now classify the types of branches that occur in the bytecode of Java programs, and discuss the level of guidance they can potentially afford the evolutionary search in EVOSUITE.

This is important because it has been long known that not all branch predicates give "good" guidance, the archetypal example being that involving the boolean flag [37, 38]. Boolean conditions in branch predicates can only ever evaluate to true or false, offering one of only two distance values. Since one of these values corresponds to execution of the branch, no guidance is given to the search. Nevertheless, several branch predicates do indeed provide guidance, and result in a smooth "gradient" in the fitness landscape that a guided search can use to easily find test inputs.

3.1. "Integer-Integer" Branches

"Integer-Integer" branches involve the comparison of two integer values. The range of values possible for this comparison can potentially create a gradient for the search. Figure 1a shows an example of such a comparison, in which a method receives an integer parameter "`a`", and has a conditional statement on the parameter ("`a == 1`") (Figure 1a-i). The bytecode (Figure 1a-ii) shows this is compiled to a "`if_icmpne`" instruction, which compares the last two integers pushed to the stack, performing a jump to some other instruction in the bytecode if those two integers are not equal. Figure 1a-iii shows how the distance value decreases as the chosen input value gets closer to the value that would execute the uncovered branch.

Of course, "Integer-Integer" branches may not always result in a gradient: it depends on the underlying program. One example of this is where two boolean values are compared, since boolean values are represented as the integer values 0 and 1 in Java bytecode. Therefore, source code comparisons involving two boolean values are compiled to an integer comparison involving the usage of the `if_icmpne` instruction. However, and as already discussed, boolean conditions do not induce any useful landscape gradient.

Furthermore, EVOSUITE's special handling of `switch` statements falls into the "Integer-Integer" category. Java `switch` statements are compiled to either a `tableswitch` or `lookupswitch` bytecode instruction. These instructions pop the top of the stack to look up a "jump" target instruction in a map data structure, for which the keys are the values

```
void m(int a) {
  if (a == 1) {
    // uncovered branch
  }
}
```

(i) Source code

```
void m(int);
  0: iload_1
  1: iconst_1
  2: if_icmpne 7
     [uncovered branch]
  7: return
```
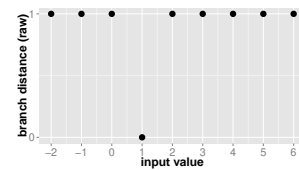
(ii) Bytecode
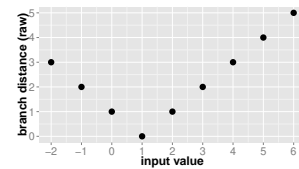
(iii) Raw branch distance

**(a) Int-Int Branch**

```
void m(int a) {
  boolean x = false;
  if (a == 1)
    x = true;
  if (x) {
    // uncovered branch
  }
}
```

(i) Source code

```
void m(int);
  0: iconst_0
  1: istore_2
  ...
  9: iload_2
 10: ifeq 15
     [uncovered branch]
 15: return
```

(ii) Bytecode

(iii) Raw branch distance

(b) Int-Zero Branch

```
void m(int a) {
  Object x = null;
  if (a == 1)
    x = this;
  if (this == x) {
    // uncovered branch
  }
}
```

(i) Source code

```
void m(int);
  0: aconst_null
  1: astore_2
  ...
  9: aload_0
 10: aload_2
 11: if_acmpne 16
     [uncovered branch]
 16: return
```

(ii) Bytecode
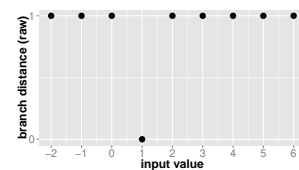
(iii) Raw branch distance

(c) Ref-Ref Branch

```
void m(int a) {
  Object x = null;
  if (a != 1)
    x = new Object();
  if (x == null) {
    // uncovered branch
  }
}
```

(i) Source code

```
void m(int);
  0: aconst_null
  1: astore_2
  ...
 15: aload_2
 16: ifnonnull 21
     [uncovered branch]
 21: return
```

(ii) Bytecode
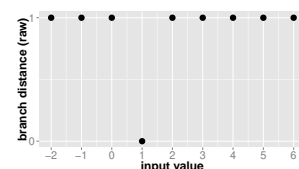
(iii) Raw branch distance

(d) Ref-Null Branch

Figure 1. Examples of different branch types (denoted "uncovered branch") and their effect on the respective fitness landscape for the GA through raw (unnormalized) branch distance values. We show both the original Java source and the compiled bytecode, as processed by EVOSUITE. Note that the target true/false evaluation of the branches is inverted by the Java compiler. Part (a) of the figure shows an example of a "gradient" branch, providing useful guidance to the search. Parts (b)–(d) of the figure show examples where no guidance is available: all possible inputs to the method except one lead to the same distance value, producing a flat fitness landscape.

```
void m(double a) {
   if (a == 1.0) {
      // uncovered branch
   }
}
```

```
void m(double);
   0: dload_1
   1: dconst_1
   2: dcmpl
   3: ifne 10
   [uncovered branch]
  10: return
```


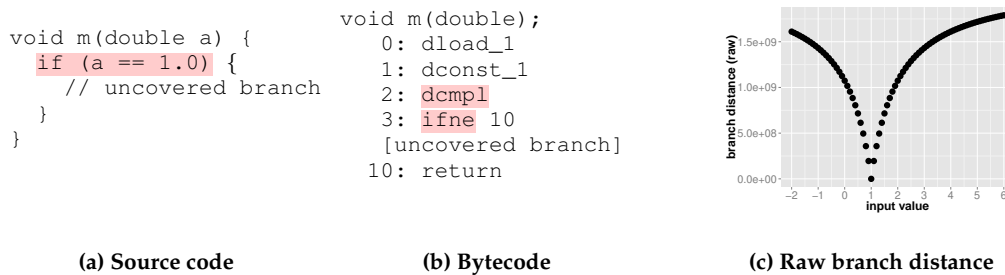| (a) Source code | (b) Bytecode | (c) Raw branch distance |

Figure 2. Branch distance plot for a branch involving a variable and constant of type `double`, showing the source code (a) and the bytecode (b). Although these branches fall into the "Int-Zero" category, EVOSUITE instruments the bytecode so that distances are recovered, resulting in a gradient landscape (c).

originally used in each `case` of the `switch`. For ease of fitness computation, EVOSUITE simply instruments the bytecode by adding an explicit `if_icmpeq` for each case before the original `tableswitch` or `lookupswitch` instruction, comparing the top of the stack to each `case` value.

3.2. "Integer-Zero" Branches

"Integer-Zero" branches involve the comparison of an integer value with zero. One type of "Integer-Zero" branch occurs when boolean predicates are evaluated[†], for example as shown by Figure 1b. Here the branch involves the evaluation of the boolean value `x` (Figure 1b-i). The corresponding bytecode evaluates `x`, pushing the result (an integer, 0 or 1) to the stack. The `ifeq` bytecode instruction then pops this value, performing a jump if it is zero. Such a condition can only be either true or false, and as such can only have one of two distance values, which, as shown by Figure 1b-iii, are not useful to guiding the GA to covering the branch. The "right" input must therefore be discovered purely by chance.

A further type of "Integer-Zero" branch occurs as a result of comparisons involving values of `float`, `double` and `long` primitive Java types. Figure 2 shows an example of a comparison involving `double` values. The original source (Figure 2a) performs the comparison in the branch predicate. This is decomposed into a sequence of bytecode instructions shown by Figure 2b. The comparison is performed by the `dcmpl` in relation to the top two double values pushed to the stack. The `dcmpl` instruction pushes an integer to the stack: -1 if the first value is greater than the second, 1 if the first is less than the second, else 0 if they are equal. The `ifne` then performs a jump if the top of the stack is not 0.

Since the original numerical comparison in the source code is transformed to a boolean comparison in the bytecode, a significant amount of useful distance information is "lost" in the compilation process that would have been useful in guiding the search. EVOSUITE therefore instruments the bytecode so that distance information can be recovered. The branch distance plot for the example, shown by Figure 2, therefore restores a gradient that can be used to optimize input values towards execution of the uncovered branch.

3.3. "Reference-Reference" branches

"Reference-Reference" branches are where two object references are compared for equality. Since references are not ordinal types, no meaningful distance metric can be applied, and the situation is similar to boolean flags – either the references are the same or they are not. Figure 1c shows an example of this. The original source code conditional is "`if (this == x)`" (Figure 1c-i), which Java compiles to the bytecode instructions 9–11 in

---

[†]Note that boolean predicate evaluations in branches differ in bytecode from comparing two boolean values – the latter type of branch falls into the "Integer-Integer" category.

Table I. Classification of Java bytecode branch types according to the search landscape they produce. Int-Int and Int-Zero branches in certain scenarios produce gradient or plateau conditions.
*For these comparisons, as discussed in Section 3.2 and shown in Figure 2, a gradient can be recovered.

Gradient Branch	Plateau Branch
Int-Int: Comparing two `integer` values **Int-Zero**: Comparing an `integer` with zero	**Int-Int**: Direct comparison of two `boolean` values **Int-Zero**: Checking `boolean` condition values returned by method-calls or those stored in variables **Int-Zero**: `float`, `double` and `long` comparisons* **Ref-Ref**: Memory reference comparisons **Ref-Null**: Memory reference comparison against `null`

Figure 1c-ii. Instructions 9 and 10 push the references onto the stack. Instruction 11 is the branching point in the bytecode, with "`if_acmpne`" popping the top two stack references and performing a jump if they are not equal. The resulting plot of branch distances (Figure 1c-iii) shows the resulting plateau, providing no guidance to the required input that makes the references equal and executes the uncovered branch.

3.4. "Reference-Null" branches

"Reference-Null" branches are similar to "Reference-Reference" branches, except one side of the comparison is `null`. Again, no meaningful distance metric can be applied. Figure 1d shows an example. The source code compares x with `null`. In the bytecode, x is pushed onto the stack by instruction 15. Instruction 16 is the branching point, where the `ifnonnull` instruction performing the jump if the element popped off the top of the stack is not `null`.

3.5. Summary

We have summarized and classified the different types of branches that can occur in Java bytecode. As outlined in Table I, some of these instructions will potentially give rise to a gradient in the fitness landscape, while others will not. We now study the prevalence of these types of branches in real-world code, whether they potentially involve a gradient, and their potential impact on the relative performance of random search and fitness-guided GA and CRO search. Note that even gradient branches do not *guarantee* gradients. For example, when the two numbers that are compared are constants or only have few possible values that can be assigned to them, then the resulting search landscape would be more plateau-like.

4. EXPERIMENTAL SETUP

We designed an empirical study to test the relative effectiveness of test case generation using random, GA, and CRO search, with the aim of answering the following research questions:

RQ1: Is the use of an evolutionary algorithm, such as GA and CRO, more effective at generating unit tests than random search?

RQ2: How do the results of the comparison depend on the types of branches found in the code under test?

RQ3: How do the results of the comparison depend on the time allowed for the search?

RQ4: How do the results of the comparison depend on an archiving functionality for covered goals?

Table II. Statistics for the sample of 975 classes. For each class, the number of "'Goals"' represents the sum of *branches* and *branchless methods*.

	Min	Avg	Max	Sum	SD
Total Branches	0	26.93	1,020	26,258	79.5
Branchless Methods	0	7.18	155	6,998	11.5
Total Goals	1	34.11	1,030	33,256	84.3

4.1. Subjects

In order to compare and contrast the relative effectiveness and performance of random and evolutionary search, we selected a sample of classes from the SF110 corpus of open source projects [27]. The SF110 corpus is made up of 110 open source projects from the SourceForge open source repository (http://sourceforge.net), where 10 of the projects were the most popular by download at the time at which the corpus was constructed (June 2014) and the remaining 100 projects selected at random. Due to the large variation in the number of classes available in each project, we stratified our random sampling over the 110 projects, such that our sample involved at least one class from each of the 110 projects in the corpus, and comprised 1000 classes in total[‡]. However, 25 classes were removed from the sample for reasons such as not having any testable methods (e.g., they consisted purely of enumerated types, or did not have any public methods) or test suites could not be generated for some other reason that would allow us to sensibly compare the techniques (e.g., the class contained a bug or other issue that meant it could not be loaded independently without causing an exception).

The final number of classes in the study therefore totaled 975, comprising small classes with just a single coverage goal to larger classes with over 1,000 coverage goals, as shown by Table II. In this table, *Branchless Methods* indicates the number of methods without conditional statements and can be covered by simply calling the method concerned.

4.2. Collation of Branch Type Statistics

In order to answer RQ2, we collated a series of statistics on the types of branches in the bytecode of each class.

Firstly, we simply collected the numbers of branches that fall into each of the categories detailed in Section 3 (i.e., "Integer-Integer" etc.) by statically analyzing the bytecode of each class in turn.

Secondly, we attempted to classify each branch as either *potentially* having a gradient distance landscape ("Gradient Branches"), or, a plateau landscape ("Plateau Branches"). We programmed EVOSUITE so that during test suite generation it would monitor the distance value of the predicate leading to the branch. If in any of the executions of a search algorithm in the experiments, a value other than 0 or 1 is observed, we assume a wider range of distance values is available for fitness computation and label the branch as a "Gradient Branch". Otherwise the search is labelled as a "Plateau Branch". Clearly, this analysis is only indicative (but helps in understanding our results, as we will show in the answer to RQ2). This is because a range of values does not necessarily imply a gradient that will be useful for guiding the search. Nor does only finding the distances 0 and 1 for a branch mean that there are not further distance values that could be encountered. For instance, given a branch predicate $x > 5$, if for the whole duration of the search only the values $x \in \{4, 5, 6\}$ are used, then this will result in the distance values of 1, 0, and 1 respectively; and the branch will be incorrectly classified as a plateau branch. However, it is quite unlikely that the branch would only be attempted with these values over the course of several executions of a search algorithm.

[‡]The list of classes is available on: https://sinaa.github.io/random-vs-ga-test-generation/

4.3. Experimental Procedure

We applied EVOSUITE to conduct our experiments, with implementations of the genetic algorithm (*GA*), chemical reaction optimization algorithm (*CRO*), and the two random search algorithms (*Random+* and *Pure Random*) as described in Section 2.

We use all four algorithms in RQ1 only. *Pure Random* features only in RQ1 in order to analyze for possible effects with *Random+* due to its seeding mechanism. *CRO* features in RQ1 and RQ2 only as an additional type of evolutionary algorithm with which to compare *GA* against *Random+*.

For RQ1 we applied each technique with a search time of two minutes (which has been shown to be a suitable stopping condition in previous work [27]). To answer RQ2, we investigated the influence of the type of conditional predicates on the outcome of each technique. To do so, we used the statistics on branch types, collected as we described in the last section. To better understand the influence of the search budget over the outcome of the techniques for RQ3, we executed EVOSUITE using the *GA* and *Random+* configurations with an increased search time of ten minutes and measured the level of coverage at one minute intervals.

For RQ4, we compared the effectiveness of the *GA* and random search algorithms with the test archive. Although the use of an archive of tests is expected to enhance coverage in general, it may also have undesired effects in some cases. For example, setting a high probability of reusing archived tests instead of using new random ones may hinder diversity in the population and therefore make it harder for the search to escape local optima. To prevent this from happening, a conservative probability value of 0.2 is used in our experiments.

For all other GA parameters we used the default values resulting from earlier tuning experiments [39]. As CRO is new in the field of search based test generation, there is no generally recommended set of default parameter values. We therefore used the parameter values as suggested by Lam and Li [40] as starting point, and then we ran CRO on a sample of classes from the SF110 projects [27], modifying one parameter at a time, repeating each run 100 times and observing the resulting average branch coverage. The following optimal configuration of CRO parameters emerged from these experiments:

- *decThreshold*: 25. We chose a substantially lower value than Lam and Li's [40] suggestion of 500, in order to increase the chances of decomposition. Higher values would bias the search strongly towards the local search aspects.

- *synThreshold*: 5. This is similar to the default (10 [40]); the slight decrease leads to less synthesis (global search), increasing local search.

- *initKE*: 1000. This matches the default value in the literature [40]. The higher the initial KE value is, the longer single individuals will explore their local search space (local search) before trying to explore different regions (global search).

- *KELossRate*: 0.1. (Default: 0.2 [40]) The lower KELossRate value is, the longer single individuals will explore their local search space (local search) before trying to explore different regions (global search).

- *collRate*: 0.1. (Default: 0.2 [40]) Higher collRate values indicate that individuals will exchange information more often (we have more syntheses and inter-molecular collisions at the expense of decompositions and on-wall ineffective collisions).

We conducted all our experiments on the University of Sheffield's HPC Cluster (http://www.shef.ac.uk/wrgrid/iceberg). Each node has a Sandy-bridge Intel Xeon processor with 3GB real and 6GB virtual memory. We used EVOSUITE's default configuration and ran it under Oracle's JDK 7u55. Our experiments resulted in over 680,000 generated test suites, requiring over 5.5 years of serial execution time.
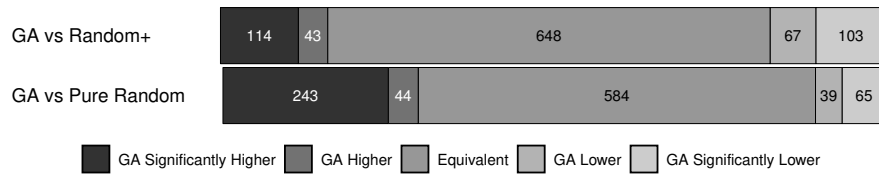
Figure 3. Comparing *GA* performance with *Random+* and *Pure Random* over the 975 SourceForge classes: For 78% of subjects, there was no significant difference between *GA* and *Random+*.
("GA Significantly Higher" is the number of classes for which *GA* obtained significantly higher coverage than *Random+*/*Pure Random* over the 100 runs of the experiment; "GA Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same (equal), etc.)

4.4. Threats to Validity

Threats to the *internal validity* of our study include its usage of only one test generation tool (EVOSUITE). While this was deliberate to facilitate a more controlled, fair comparison, it is plausible that specific implementation choices made in EVOSUITE may limit the extent to which our results generalize (an associated external threat). The size of the test suites, for example, may influence the comparison; whereas *Random+* has no constraint in the test suite size, both *GA* and *CRO* evolve test suites with limited size (100 test cases by default) which imposes boundaries in the search space.

The initial population of individuals and their evolution depend on the values of several parameters for both *GA* and *CRO*. Results might thus be affected by the specific parameter values that we used in the experiments. On the one hand, the *GA* has been used for long time in EVOSUITE and all the parameters are well optimized to address test case generation. On the other hand, however, CRO is controlled by a higher number of parameters and has been applied to test case generation for the first time in this paper. In an effort to mitigate this threat, we run preliminary tuning experiments on *CRO* in order to ensure that the optimal combination of parameters was used when comparing it with *GA* and *Random+*.

Another threat to internal validity stems from the branch-classification analysis described in Section 4.2, which can mis-categorize branches in certain cases. We acknowledge the results of this analysis may only be approximate, but while testing the experimental setup we validated that the analysis categorized all branches correctly. Furthermore, chance can affect the results of randomized search algorithms. To mitigate this threat, we repeated all experiments 100 times.

Threats to *external validity* affect the generalizability of our results. While we used a randomly selected sample of Java classes as subjects, our results may not generalize beyond the SourceForge project repository. Moreover, the algorithms and tools our study evaluates target the Java object-oriented programming language specifically; further studies will be needed to verify if our findings generalize to other programming languages and paradigms. Similarly, further work should look into whether our results hold when looking at other test suite quality measurement (e.g., size, length or fault detection ability) besides branch coverage.

Finally, we only used two different variants of evolutionary algorithms, GAs and CRO, and CRO has not previously been applied to unit test generation. While it is possible that other evolutionary algorithms would perform better at the specific problem of unit test generation applied to the classes used in our experiments [41], the use of two different algorithms is sufficient to observe differences and similarities in results compared to random approaches.

Figure 4. Comparing *Random+* performance with *Pure Random* over the 975 SourceForge classes. ("Random+ Significantly Higher" is the number of classes for which *Random+* obtained significantly higher coverage than *Pure Random* over the 100 runs of the experiment; "Random+ Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same (equal), etc.)

(a) Box Plot   **(b) Violin Plot**
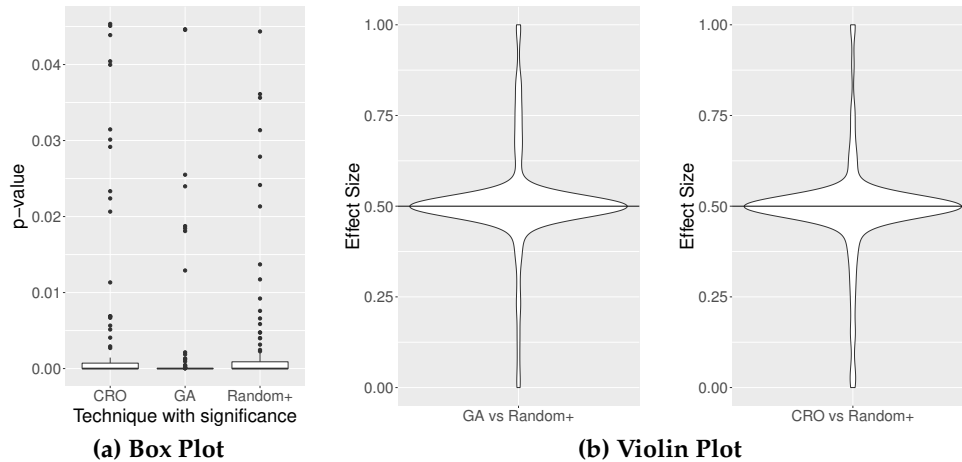
Figure 5. Comparing *GA* and *CRO* performance with *Random+*. (a) Box plot of *p*-values for classes where a significantly higher level of coverage was achieved with either the *GA*, *CRO* or *Random+*. (b) Violin plot of the effect sizes obtained using Vargha-Delaney's $\hat{A}_{12}$ statistic, here computing the proportion of the 100 repetitions for which the *GA* or *CRO* score a higher level of coverage than *Random+* for each class; thereby reflecting their relative effectiveness. The violin plots indicate a similar effect size between the evolutionary techniques (*GA* and *CRO*) and *Random+*.

5. RESULTS

RQ1: Coverage Effectiveness. On average over the 100 repetitions of the experiments, the *GA* attains 69.10% branch coverage, *CRO* 68.87%, *Random+* 68.76%, while *Pure Random* obtains 65.22%, across all classes. Notice the similarity of the level of coverage achieved by all techniques, specifically between the evolutionary techniques (*GA* and *CRO*), and the random techniques (*Random+* and *Pure Random*).

We observe further similarities in the coverage achieved by the *GA* and *CRO* against *Random+* with Figure 5b, which shows effect sizes computed with Vargha-Delaney's $\hat{A}_{12}$ statistic [42]. Here, the effect size estimates the probability that a run of *GA* achieves higher coverage than *Random+*. A value of $\hat{A}_{12} = 0.5$ indicates that both search strategies perform equally, $\hat{A}_{12} = 1$ indicates that all runs of the *GA* will achieve higher coverage than *Random+*, and vice versa for $\hat{A}_{12} = 0$. The overall average effect size for *GA* and *CRO* respectively amount to 0.51 and 0.49, which indicate that the *GA* is only very marginally more effective than *Random+*, and *CRO* is only very marginally less effective.

To compare the performance of the technique at class level, Figure 3 summarizes the number of classes for which the *GA* achieved a significantly higher or lower level of coverage than *Pure Random* and *Random+* over the 100 repetitions of the experiments. (We computed significance using the Mann-Whitney U test at a level of $\alpha = 0.05$.)

As it can be seen, for a majority of classes (78%), no significant difference exists between the evolutionary (*GA*) and random (*Random+*) techniques. Moreover, while there are 114 (11.7%) classes for which the *GA* achieves significantly higher coverage than *Random+*,

| CRO vs GA | 33 | 49 | 629 | 116 | 148 |
| CRO vs Random+ | 95 | 41 | 630 | 69 | 140 |

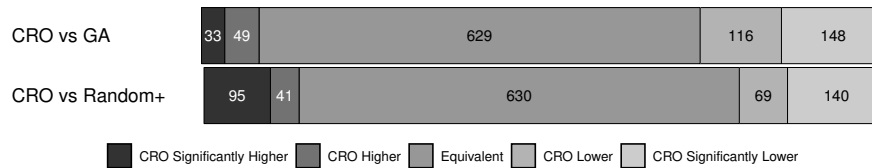■ CRO Significantly Higher   ■ CRO Higher   ■ Equivalent   □ CRO Lower   □ CRO Significantly Lower

Figure 6. Comparing the coverage achieved by CRO against *Random+* and *GA* over the 975 SourceForge classes. While for 148 classes *GA* achieved significantly higher coverage than *CRO*, for the majority of classes, the two evolutionary techniques had a similar performance. However, compared to *Random+* the outcome of *CRO* was similar to that of *GA*: for 76% of subjects *CRO* and *Random+* were as performant.
("CRO Significantly Higher" is the number of classes for which CRO obtained significantly higher coverage than *GA/Random+* over the 100 runs of the experiment; "CRO Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same (equal), etc.)

there are 103 (10.6%) classes on which *Random+* attains significantly higher coverage than *GA*. This indicates that no technique clearly achieves a better outcome than the other.

Besides subjects for which one technique achieves higher coverage than the other, for 648 classes the *GA* and *Random+* achieve identical (i.e., equal) coverage, and likewise for 630 of classes when comparing *CRO* and *Random+*. To a large extent, this can likely be attributed to the simplicity of these classes: *GA* achieves 100% coverage on 390 classes, *CRO* on 376, and *Random+* on 402 classes. Classes with lower but identical coverage are likely classes where the possible coverage is maximized, but less than 100% because of problems that EVOSUITE cannot overcome regardless of search algorithm (e.g., due to environmental factors such as classes depending on databases or web services that were not available during the experiments).

The common evolutionary nature of the *GA* and *CRO* is reflected on the results. In fact, for 94% of subjects for which *CRO* achieves significantly higher coverage than *Random+*, the *GA* also achieved higher coverage than *Random+*. Moreover, for 98% of subjects where *CRO* achieved significantly higher coverage than *Random+*, *GA* performed no worse than *Random+*, which further confirms the similarity of the two evolutionary algorithms. Similar to Figure 3, Figure 6 shows the same form of comparison for *CRO* against both *GA* and *Random+*. Although *CRO* had a higher number of subjects on which it performed significantly worse than both *GA* and *Random+* (148 and 140 classes respectively), it achieved a similar result to the *GA*. In particular, *CRO* achieved significantly better coverage than the *GA* and *Random+* for 33 and 95 classes, respectively. Figure 5a plots the *p*-values for the significant cases for the evolutionary techniques and *Random+* comparison showing that the majority of cases are highly significant (particularly in the case of the evolutionary algorithms) and thus unlikely to represent type-I errors.

The comparison between the *GA* and *CRO* against *Pure Random* shows larger differences, with 243 and 226 classes where *GA* and *CRO* respectively achieve significantly higher coverage. In particular, notice that in Figure 4 *Random+* achieves significantly higher coverage than *Pure Random* on 229 subjects. This indicates that optimizations such as constant and dynamic seeding, which are used in *Random+* (as explained in Section 2.1.1), are effective and help covering non-trivial classes.

> **RQ1.** Our experiments showed no significant difference between the evolutionary techniques (*GA* and *CRO*) and *Random+* in 78% and 76% of classes, respectively.

RQ2: Influence of Branch Types. Although the comparison between evolutionary techniques (*GA* and *CRO*) against *Random+* showed 648 and 630 classes with no difference in coverage respectively, there were also 217 and 235 classes with significant differences. RQ2 aims to shed light on these differences by studying the influence of different types of branches in a class on the effectiveness of the search algorithms.
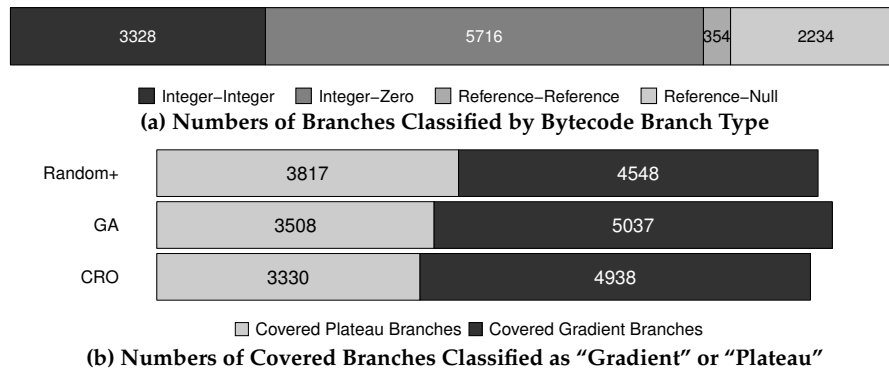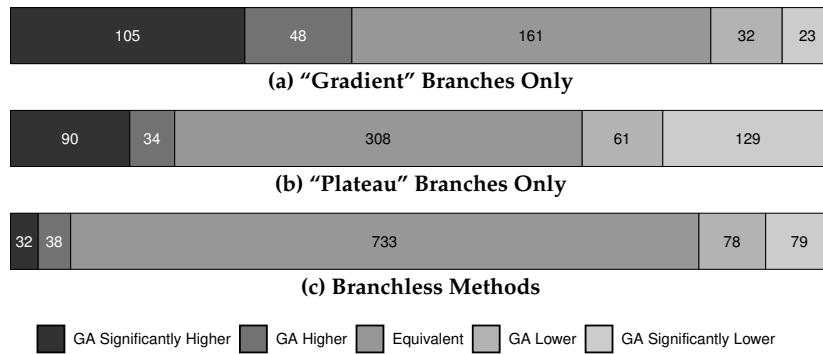
Figure 8. Comparing *GA* performance with *Random+* for different types of branch and with branchless methods: *GA* was more effective than *Random+* in covering gradient branches, while being less effective in covering plateau branches or methods without branches (branchless methods).

("GA Significantly Higher" is the number of classes for which *GA* obtained significantly higher coverage than *Random+* over the 100 runs of the experiment; "GA Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same (equal), etc.)
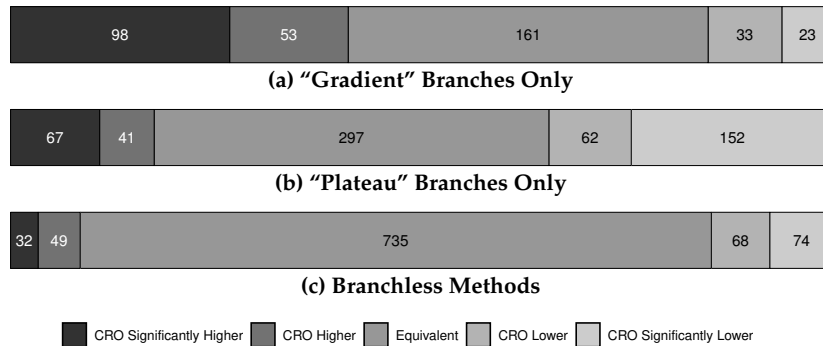

Figure 9. Comparing *CRO* performance with *Random+* for different types of branch and with branchless methods: Similar to *GA*, *CRO* was more effective than *Random+* in covering gradient branches, while being less effective in covering plateau branches or branchless methods.

("CRO Significantly Higher" is the number of classes for which CRO obtained significantly higher coverage than *Random+* over the 100 runs of the experiment; "CRO Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same (equal), etc.)

GA and *CRO* covered more branches overall, they covered fewer plateau branches. Since the branch distance for these branches only has two values there is no guidance that the *GA* could exploit – a plateau branch is either covered or it is not covered. A possible conjecture is a loss of diversity of the evolutionary search algorithms compared to the random search: While *Random+* continuously creates independent new objects and call sequences, *GA* and *CRO* spend more time exploring the neighborhood if existing individuals. In addition, the *GA* in EvoSuite prefers smaller test suites (when two test suites have the same fitness value, they are ranked by size) and thus further exacerbating the removal of random "noise", focusing the search operators on the exploitation of achieved coverage and mutating existing objects.

Effectiveness on Branchless Methods. Branchless methods represent a special case similar to plateau branches, and intuitively they are simple to cover – they just require test cases to call the method, without any need to search for specific parameter values. Figure 8c and Figure 9c compare *GA* and *CRO* against *Random+* with respect to the coverage of methods. Although *GA* achieves significantly higher coverage than *Random+* in 32 cases,
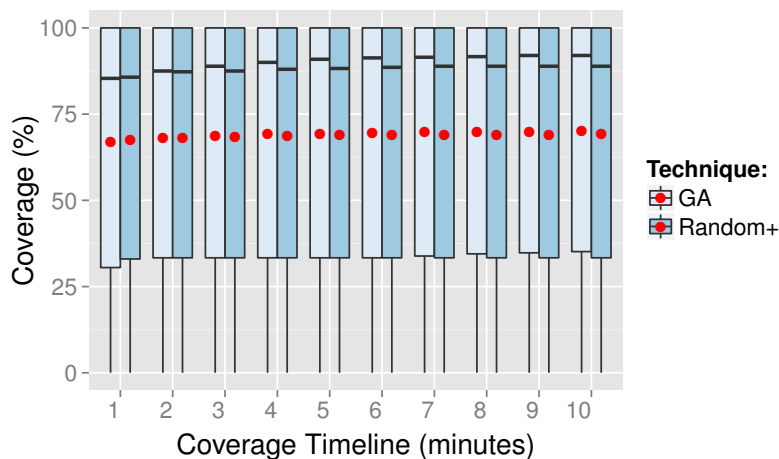
Figure 10. Branch coverage comparison between *GA* vs. *Random+* over 10 minutes with one minute intervals. Dots represent mean averages.

there are 79 classes where the *GA* results in lower coverage, which is similar in proportions to the plateau branches. Likewise, *CRO* achieved a similar outcome (32 and 74 classes respectively). It is maybe surprising that there can be difference in so simple coverage goals in the first place. Our conjecture is that this is because *Random+* has a higher probability of inserting new method calls: *CRO* and *GA* only mutate a test suite with a certain probability, and then each test in turn is only mutated with a certain probability, and finally insertion of new statements again does not always happen. In contrast, *Random+* generates tests by repeatedly adding new statements. Again it would only be a matter of time for evolutionary search to fully cover all branchless methods, although possibly more time than for *Random+*. Interestingly, classes on which the *GA* and *CRO* achieved more than 90% coverage have a median proportion of 100% branchless methods out of all coverage goals, providing further evidence that many classes in practice are trivial.

> **RQ2.** Our experiments show that the evolutionary techniques (*GA* and *CRO*) achieve higher coverage of gradient branches compared to *Random+*, but lower coverage of plateau branches, which constitute the majority of branches.

RQ3: Effects of the Time Allowed For the Search. The results so far have shown that *GA*, *CRO* and *Random+* perform similarly for the majority of classes after two minutes of search, with some differences in performance on plateau and gradient branches. This raises the question whether the results are influenced by the allocated search budget – given more time, do the results change?

To analyze the impact of the search budget, we repeated the experiments with *GA* and *Random+* using an increased search budget of 10 minutes, and measured the coverage values at one minute intervals. Figure 10 compares the average coverage per class for each interval: There is a slight increase of coverage for both the *GA* and *Random+* over time, and after 10 minutes the *GA* achieves an overall average of 69.81% branch coverage, while *Random+* achieves 68.95%.

Given more time, *GA* will catch up on branchless methods and plateau branches covered compared to *Random+*. Figure 11 compares the *GA* with *Random+* after 10 minutes, and shows that the *GA* has significantly lower coverage on only 83 classes after 10 minutes, compared to 103 after two minutes (Note that the number of classes with coverage data after 10 minutes is only 974, as there was 1 additional class for which EVOSUITE did not produce any data after 10 minutes). The *GA* will also continue to optimize gradient

131	39	664	57	83

◼ GA Significantly Higher ◼ GA Higher ▧ Equivalent ▨ GA Lower ☐ GA Significantly Lower
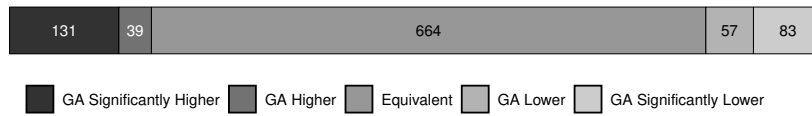
Figure 11. Branch coverage comparison between *GA* and *Random+* using a search budget of 10 minutes: While the effectiveness of *GA* increases compared to *Random+*, still for the majority of subjects there is no significant difference between the two techniques.
("GA Significantly Higher" is the number of classes for which *GA* obtained significantly higher coverage than *Random+* over the 100 runs of the experiment; "GA Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same (equal), etc.)

branches; however, the dynamic seeding used in EVOSUITE will also help *Random+* in many cases to cover gradient branches. Figure 11 shows that there are 131 classes where the *GA* has higher coverage after 10 minutes, compared to 114 after two minutes. For 760 classes the coverage is identical, which is likely because the maximum achievable level of coverage has been reached by both algorithms.

> **RQ3.** The coverage increase is higher for the *GA* than for *Random+* over time, suggesting that the disadvantage on plateau branches is overcome, although the coverage increase is small in absolute terms.

71	51	195	21	17

(a) "Gradient" Branches Only

164	59	310	28	55

(b) "Plateau" Branches Only

83	84	720	30	37

(c) Branchless Methods

◼ GA–Archive Significantly Higher ◼ GA–Archive Higher ▧ Equivalent ▨ GA–Archive Lower ☐ GA–Archive Significantly Lower
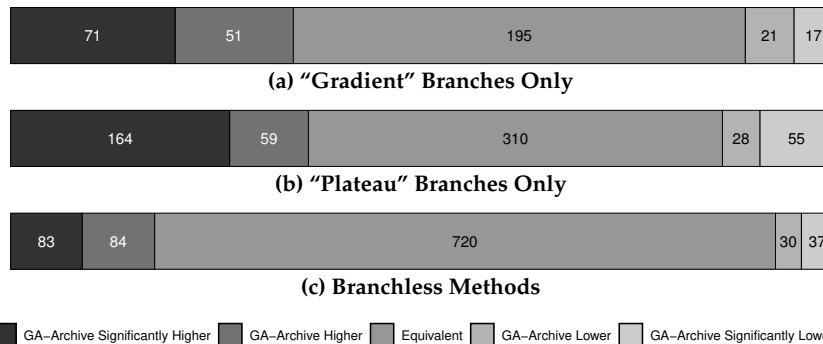
Figure 12. Comparing the effectiveness of *GA* with archive enabled against *GA* without archive for different types of branch and with branchless methods.
("GA–Archive Significantly Higher" is the number of classes for which *GA* with archive obtained significantly higher coverage than *GA* without archive over the 100 runs of the experiment; "GA–Archive Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same (equal), etc.)

RQ4: The Effects of an Archive of Solutions. By construction, the final solution of the *Random+* search includes every test that has at some point in the search covered a coverage goal for the first time. In contrast, due to the way in which the *GA* and *CRO* evaluate fitness and keep track of covered goals, it is possible that the final solution they produce does not contain all the goals covered during the search. Since the fitness function in *GA* aims to maximize coverage, the individual with the best fitness – hence highest coverage – will be preferred as a solution over another individual with worse fitness – and thus lower coverage – although the latter may still cover some goals which are not covered by the best individual. In this RQ we investigate whether keeping an archive of all covered goals during the search together with the tests covering them (i.e., the approach taken by *Random+*) can be beneficial to the effectiveness of evolutionary search for test generation. Specifically, we look at the test archive implemented for the *GA* in EVOSUITE [36].

Let us first compare the overall effectiveness of the *GA* with archive against *GA* without archive (i.e., the treatment used in RQ1). Figure 12 shows that using the archive leads

| 113 | 46 | 151 | 24 | 15 |

(a) "Gradient" Branches Only

| 134 | 39 | 307 | 53 | 83 |

(b) "Plateau" Branches Only

| 45 | 63 | 747 | 48 | 52 |

(c) Branchless Methods

■ GA–Archive Significantly Higher   ■ GA–Archive Higher   ▨ Equivalent   ▨ GA–Archive Lower   ▨ GA–Archive Significantly Lower
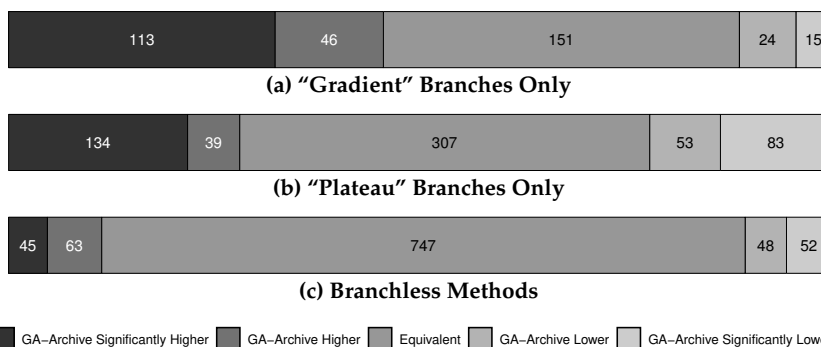
Figure 13. Comparing *GA* performance with *Random+* for different types of branch and with branchless methods, with Archive enabled.
("GA Significantly Higher" is the number of classes for which *GA* obtained significantly higher coverage than *Random+* over the 100 runs of the experiment; "GA Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same (equal), etc.)

GA vs Random+ | 163 | 53 | 653 | 26 | 75 |

■ GA–Archive Significantly Higher   ■ GA–Archive Higher   ▨ Equivalent   ▨ GA–Archive Lower   ▨ GA–Archive Significantly Lower
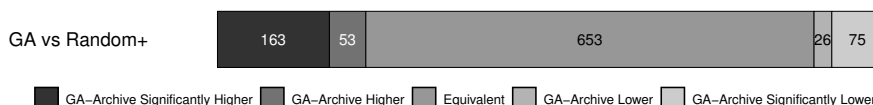
Figure 14. Comparing the coverage achieved by *GA* against *Random+* over the 970 SourceForge classes, with archive enabled: Compared to Figure 3, the archive of solutions is an effective way to increase the performance of *GA*, however, for a large number of subjects, the evolutionary and random techniques perform similarly.
("GA Significantly Higher" is the number of classes for which *GA* obtained significantly higher coverage than *Random+* over the 100 runs of the experiment; "GA Higher" – the number of class where a higher average coverage was obtained (but not significantly); "Equivalent", the number of classes where the average coverage level was the same (equal), etc.)

to significantly higher coverage of gradient and plateau branches, as well as branchless methods. Overall across all classes, *GA* with archive lead to significantly higher coverage in 179 cases and to significantly worse coverage in 67 cases. Whereas the improvements observed when using the archive are expected, the detrimental effects in some cases are worth discussing further. Although our experimental data does not shed light into what specific goals are being missed when using the archive, we conjecture that the negative effect is due to the way the archive influences the generation of new test chromosomes as explained in Section 2. Arguably, sampling from the set of archived tests may in some cases reduce the diversity among the evolved populations, thus resulting in limited exploration of the search space and a consequently lower coverage compared to the more randomized strategy applied by the *GA* without archive.

Having established that the effectiveness of the *GA* does improve with the use of the archive, let us now revisit the comparison of *GA* versus *Random+* presented in Figure 8. As Figure 13 and Figure 14 show, the *GA* with archive was significantly more effective in 163 cases and only significantly worse in 75 (note that we excluded five further classes for experiments with archive enabled as EVOSUITE failed to produce test cases due to execution failures). These results represent a considerable improvement with respect to the comparison results presented in Figure 8. In particular, the *GA* with archive covers a significantly higher number of gradient branches than *Random+* in 32% of subjects, significantly higher number of plateau branches in 22% of subjects, and significantly higher number of branchless methods in 5% of subjects. Furthermore, for the three kinds of branches, the number of subjects on which *GA* is significantly worse than *Random+* is consistently reduced (notably for plateau branches). Notice that the largest absolute improvement is observed for plateau branches: whereas the *GA* without archive has

significantly higher (resp. lower) coverage of plateau branches than *Random+* in 90 (resp. 129) classes (Figure 8), the *GA* with archive achieves significantly higher coverage of plateau branches than *Random+* in $134 - 90 = 44$ cases, and significantly lower coverage than *Random+* in $129 - 83 = 46$ (Figure 13).

Although it is reasonable to assume that the archive contributes to better coverage of plateau branches (for which RQ2 showed that the *GA* without archive did not perform as well as *Random+*) our data does not allow us to specifically link the archive to plateau branches, as improving coverage of any type of branch may open up the possibility to easily cover many dependent branches of other types. Overall, the positive results observed in terms of number of branches are validated in Figure 14 in terms of branch coverage effectiveness.

> **RQ4.** Our experiments show that archiving covered goals improves the performance of *GA* vs. *Random+*, leading to 49 additional subjects with significantly better coverage, and 28 fewer subjects with significantly worse coverage.

6. IMPACT OF THE FINDINGS

The results reported in this paper have important practical implications that may influence research and practice on search-based unit test case generation. In the following, we report and discuss these implications focusing on both the perspective of developers, who use these methods to test classes, and on the perspective of researchers, who work on the design of more effective and efficient test generation methods.

"Quick and dirty" coverage does not need evolutionary methods. If *developers* intend to use unit test case generators to achieve fast shallow coverage of their classes, methods based on random search are as effective as evolutionary methods, and might be even superior in quickly covering all the easy cases, such as executing all the branchless methods.

Residual coverage benefits from evolutionary methods. When covering gradient branches, evolutionary methods clearly outperform random methods. If part of the code in the target classes has been already covered with other methods, such as with some manually written or random test cases, achieving higher coverage by generating test cases that execute the uncovered areas of the code likely requires dealing with complex branch conditions and gradient branches. According to our findings, *developers* should exploit test case generation based on evolutionary search to address this case.

Results across evolutionary methods are consistent and complemental. The results obtained with GA and CRO are fairly consistent, and we can thus speculate that evolutionary methods have similar effectiveness on the same sets of classes. However, we also noticed differences that might be exploited in the future. For instance, GA has been generally more effective than CRO, but CRO worked on gradient branches more effectively than GA, suggesting that CRO might be stronger in the local search compared to GA. *Researchers* in evolutionary methods and testing might want to exploit these results to thoroughly investigate the complementarities between these methods and design highly effective test case generation methods that combine the strongest points of the individual techniques.

Switching across methods may lead to high effectiveness. The complementarities we observed among Random, GA and CRO can be exploited to design effective testing strategies. This might be of interest for the *developers*, who might simply start testing

Prepared using stvrauth.cls

classes using random methods, which are highly effective on the bootstrap phase, then switching to GA, which is more effective in executing the statements that are not trivial to reach, and finally switching to CRO, to deal with the gradient branches that have not been covered with GA. These complementarities are also interesting for *researchers* who might design methods that automatically switch from one approach to another when needed.

Testability transformation may be crucial to improve object-oriented testing. Our experiments indicate a dominance of plateau branches in object-oriented classes, which leads to a difficult search landscape. Similar findings have been reported by recent attempts to classify the search landscape [43]. One proposed way to improve problematic search landscapes in search-based testing is to apply testability transformation [44]. Initial results on transformation of Java bytecode [45] indicate feasibility, but *researchers* will need to investigate and develop advanced transformations to fully transform the search landscapes of object-oriented programs in order to enable search-based test generation tools to achieve higher code coverage.

The characteristics of the classes should influence the test generation strategy. The results reported in this paper show that each test generation method has several strong points that relate to the structural characteristics of the classes under test. For instance, the methods studied in this paper have shown a different effectiveness for gradient branches, plateau branches and branchless methods. This result might be very interesting for *researchers* in unit test generation. In fact, it motivates research on the definition of quick static analysis strategies that might be executed on a per-class basis to identify the optimal strategy for generating the tests for that class. What the set of characteristics that should be analyzed is, and how these characteristics should guide the identification of the strategy, is an open question, only partially answered by the results reported in this paper.

7. RELATED WORK

There have been several papers that have compared GAs with random search in the procedural domain (e.g., Harman and McMinn [46] and Wegener et al. [47]). This work has found guided search to always outperform random. In general, procedural code tends to consist of larger functions than the ones found in OO code, and each function tends to involve more parameters. While random search typically covers a large percentage of the branches involved, the GA covers significantly more.

Sharma et al. [48] showed on 13 examples that random testing of OO container classes achieves the same coverage as shape abstraction, a systematic technique specific for container classes. The results of our experiments suggest that in practice, many OO classes are, similarly to container classes, simple in nature and thus well suited for random testing.

Earlier experiments with EVOSUITE on the former SF100 corpus [27] showed that a large number of classes are either trivially covered, or uncoverable without providing the test generator with additional features (e.g., to handle environmental inputs such as web services or databases). This finding is in line with our results; however, a comparison with Randoop [2] in the same study suggested a large improvement of *GA* over random testing. The results of our experiments suggest that this improvement is largely due to the engineering of the tool rather than the search algorithm; for example, Randoop does not use seeding.

Eler et al. [49] analyzed the SF100 corpus from the point of view of test data generation using dynamic symbolic execution (DSE). They also reported the large number of reference comparisons and the challenges of handling those in a constraint solver. They further reported the relatively low number of branches involving integer comparisons, which result in constraints that DSE is typically strong at handling.

While Chemical Reaction Optimization, to the best of our knowledge, has not been applied to automatic test generation before, it has been shown that this algorithm can be applied to real problems in many disciplines, obtaining very competitive results. Lam and Li [19] applied CRO to a wide variety of optimization problems (quadratic assignment, resource-constrained project scheduling and channel assignment problem in wireless mesh networks), achieving superior results in many instances of those. CRO has also been used to tackle the population transition problem in peer-to-peer live streaming [21], the grid scheduling problem in grid computing [22], the stock portfolio selection problem [23], the 0–1 knapsack problem [50], and for artificial neural network training [24].

8. CONCLUSIONS AND FUTURE WORK

In this paper, we presented an empirical study comparing the effectiveness of evolutionary and random search-based algorithms for generating test suites aimed at maximising branch coverage. Experiments were carried out using a pool of 1,000 real-world Java classes. One might expect evolutionary algorithms such as a *GA* or *CRO* to vastly outperform random search for this task, but surprisingly we observed that all algorithms behaved similarly on the majority of classes, in particular when applying optimizations such as seeding of constant values, which applies to random search just as well as to evolutionary search in the domain of test generation. Although evolutionary search algorithms can exploit the guidance provided by certain types of branches, in practice there are many more branches that provide no such guidance. And, on some classes with many such branches, *GA* or *CRO* resulted in lower coverage than random search – even when a large search budget was used.

Our findings suggest several specific areas for future work in order to improve the effectiveness of evolutionary search algorithms for the task of unit test generation:

- To the best of our knowledge, this is the first time *CRO* has been applied to automatic software test generation, and the high degree of flexibility of the algorithm and the wide range of parameters has not been studied in detail for this particular problem. Although *CRO* on the whole performed comparable to the *GA*, there are some classes on which *CRO* performed significantly better than *GA*, and in these cases we observed that *CRO* covers more gradient branches. In particular, for these classes, out of the 54.67 total branches to be covered on average, *CRO* covered 11.53 gradient branches, while *GA* covered 11.15 gradient branches. This suggests that there are indeed potential benefits of the local search operators in *CRO*, and an in-depth study of how to exploit this potential remains as future work.

- Our experiments with EVOSUITE's *GA* used a basic implementation of the search algorithm. However, there are various attempts to extend this *GA* to a memetic algorithm, such as by applying dynamic symbolic execution as a type of local search [51], or using specifically designed local search operators [52]. While these local search operators would mainly benefit the search on gradient branches, the overall effects in comparison to a random search would need to be studied in detail.

- The high number of plateau branches suggests that testability transformation [44] could be used to convert some of these branches to gradient branches. While initial experiments on EVOSUITE [45] showed the potential of this approach, a significant engineering effort remains to be done before the effects can be studied at large scale.

- The analysis of RQ2 suggests that the search operators of the *GA* and *CRO* have an effect on the diversity: While random search constantly generates new tests, these evolutionary search algorithms spend more time exploring the neighborhood of existing tests through mutation, which may lead to less diversity, and negative effects

on covering plateau branches or branchless methods (cf. Figures 8b and 8c). Using an archive of solutions as an optimization, to focus the *GA* search on not yet covered goals, proved useful to enhance coverage effectiveness. This was specially the case for plateau branches, which seem to abound in open source Java projects.

- Since random search and evolutionary search are complementary in the types of branches they are good at covering, there is potential to combine the benefits of both in a hybrid approach: Random search could first be applied to more quickly cover the plateau branches, while evolutionary search could then be applied to target residual coverage. An interesting question in this context is to identify the point at which evolutionary search becomes more effective than random search.

- Future work could also explore the possibility of adapting the search to the specific fitness landscape of the problem at hand, and controlling search parameters such as the mutation rate. For instance, if a class appears to have mainly plateau branches, then the mutation rate could be increased.

From a practical standpoint, our empirical study shows that, if the objective is simply to quickly achieve a decent level of branch coverage on object-oriented classes, then using random search with seeding may be sufficient. However, considering that average branch coverage was at most 69.81%, there are plenty of possibilities for further improvements.

REFERENCES

1. Csallner C, Smaragdakis Y. JCrasher: an automatic robustness tester for Java. *Software: Practice and Experience* 2004; **34**(11).
2. Pacheco C, Ernst MD. Randoop: feedback-directed random testing for Java. *ACM SIGPLAN International Conference on Object-Oriented Programming, Systems, Languages, and Applications (OOPSLA)*, ACM, 2007; 815–816.
3. Fraser G, Zeller A. Mutation-driven generation of unit tests and oracles. *IEEE Transactions on Software Engineering (TSE)* 2012; **38**(2).
4. Xie T. Augmenting automatically generated unit-test suites with regression oracle checking. *European Conference on Object-Oriented Programming (ECOOP)*, Springer, 2006; 380–403.
5. Fraser G, Staats M, McMinn P, Arcuri A, Padberg F. Does automated unit test generation really help software testers? a controlled empirical study. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2015; **24**(4):23.
6. Sakti A, Pesant G, Gueheneuc YG. Instance generator and problem representation to improve object oriented code coverage. *IEEE Transactions on Software Engineering (TSE)* 2015; **41**(3).
7. Prasetya IWB. T3, a combinator-based random testing tool for Java: benchmarking. *FITTEST'13*. Springer, 2014.
8. Oriol M, Tassis S. Testing .NET code with YETI. *ICECCS*, IEEE, 2010.
9. Fraser G, Arcuri A. EvoSuite: automatic test suite generation for object-oriented software. *Proceedings of the Symposium on the Foundations of Software Engineering (FSE)*, ACM, 2011.
10. Tonella P. Evolutionary testing of classes. *ACM SIGSOFT Softw. Eng. Notes* 2004; **29**(4).
11. Andrews JH, Li FC, Menzies T. Nighthawk: A two-level genetic-random unit test data generator. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, ACM, 2007.
12. Baresi L, Lanzi PL, Miraz M. Testful: an evolutionary test approach for Java. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2010.
13. McMinn P. Search-based software test data generation: a survey. *Software Testing, Verification and Reliability (STVR)* 2004; **14**(2).
14. Ali S, Briand LC, Hemmati H, Panesar-Walawege RK. A systematic review of the application and empirical investigation of search-based test case generation. *IEEE Transactions on Software Engineering* 2010; **36**(6):742–762.
15. Bäck T, Hoffmeister F, Schwefel H. A survey of evolution strategies. *Proceedings of the International Conference on Genetic Algorithms (ICGA)*, Morgan Kaufmann, 1991; 2–9.
16. Dorigo M, Birattari M, Stutzle T. Ant colony optimization. *IEEE computational intelligence magazine* 2006; **1**(4):28–39.

17. Karaboga D, Basturk B. A powerful and efficient algorithm for numerical function optimization: artificial bee colony (abc) algorithm. *Journal of global optimization* 2007; **39**(3):459–471.
18. Kennedy J. Particle swarm optimization. *Encyclopedia of machine learning.* Springer, 2011; 760–766.
19. Lam A, Li VO. Chemical-reaction-inspired metaheuristic for optimization. *IEEE Transactions on Evolutionary Computation (EVC)* 2010; **14**(3).
20. Lam A, Victor L, Xu J. On the convergence of chemical reaction optimization for combinatorial optimization. *IEEE Transactions on Evolutionary Computation (EVC)*, IEEE, 2013.
21. Lam A, Xu J, Li VO. Chemical reaction optimization for population transition in peer-to-peer live streaming. *IEEE Congress on Evolutionary Computation (CEC)*, IEEE, 2010.
22. Xu J, Lam A, Li VO. Chemical reaction optimization for task scheduling in grid computing. *IEEE Transactions on Parallel and Distributed Systems* 2011; **22**(10).
23. Xu J, Lam AY, Li VO. Stock portfolio selection using chemical reaction optimization. *Proceedings of the International Conference on Operations Research and Financial Engineering (ICORFE)*, 2011; 458–463.
24. Yu JJ, Lam A, Li VO. Evolutionary artificial neural network based on chemical reaction optimization. *Evolutionary Computation (CEC), 2011 IEEE Congress on*, IEEE, 2011.
25. Rueda U, Just R, Galeotti JP, Vos TEJ. Unit testing tool competition: Round four. *Proceedings of the International Workshop on Search-Based Software Testing (SBST)*, ACM, 2016.
26. Shamshiri S, Just R, Rojas JM, Fraser G, McMinn P, Arcuri A. Do automatically generated unit tests find real faults? an empirical study of effectiveness and challenges. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE, 2015.
27. Fraser G, Arcuri A. A large-scale evaluation of automated unit test generation using EvoSuite. *ACM Transactions on Software Engineering and Methodology (TOSEM)* 2014; **24**(2).
28. Shamshiri S, Rojas JM, Fraser G, McMinn P. Random or genetic algorithm search for object-oriented test suite generation? *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, ACM, 2015.
29. Alshahwan N, Harman M. Automated web application testing using search based software engineering. *Proceedings of the International Conference on Automated Software Engineering (ASE)*, IEEE, 2011.
30. Fraser G, Arcuri A. The seed is strong: Seeding strategies in search-based software testing. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2012.
31. McMinn P, Shahbaz M, Stevenson M. Search-based test input generation for string data types using the results of web queries. *Proceedings of the International Conference on Software Testing, Verification and Validation (ICST)*, IEEE, 2012.
32. Rojas JM, Fraser G, Arcuri A. Seeding strategies in search-based unit test generation. *Software Testing, Verification and Reliability (STVR)* 2016; .
33. Fraser G, Arcuri A. Whole test suite generation. *IEEE Transactions on Software Engineering (TSE)* 2013; **39**(2).
34. Korel B. Automated software test data generation. *IEEE Transactions on Software Engineering (TSE)* 1990; **16**(8).
35. Fraser G, Arcuri A. Handling test length bloat. *Software Testing, Verification and Reliability (STVR)* 2013; **23**(7).
36. Rojas JM, Vivanti M, Arcuri A, Fraser G. A detailed investigation of the effectiveness of whole test suite generation. *Empirical Software Engineering (EMSE)* 2017; **22**(2):852–893.
37. Harman M, Hu L, Hierons R, Baresel A, Sthamer H. Improving evolutionary testing by flag removal. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, MK Pub., 2002.
38. Baresel A, Sthamer H. Evolutionary testing of flag conditions. *Proceedings of the Genetic and Evolutionary Computation Conference (GECCO)*, Springer, 2003.
39. Arcuri A, Fraser G. Parameter tuning or default values? an empirical investigation in search-based software engineering. *Empirical Software Engineering (EMSE)* 2013; **18**(3):594–623.
40. Lam A, Victor L, Xu J. Chemical reaction optimization: A tutorial. *Memetic Computing (2012)*, Springer, 2012.
41. Wolpert DH, Macready WG. No free lunch theorems for optimization. *IEEE Transactions on Evolutionary Computation (EVC)* 1997; **1**(1):67–82.
42. Vargha A, Delaney HD. A critique and improvement of the "CL" Common Language Effect Size Statistics of McGraw and Wong. *Educational and Behavioral Statistics* 2000; **25**(2).
43. Aleti A, Moser I, Grunske L. Analysing the fitness landscape of search-based software testing problems. *Proceedings of the International Conference on Automated Software Engineering (ASE)* 2016; .
44. Harman M, Hu L, Hierons R, Wegener J, Sthamer H, Baresel A, Roper M. Testability transformation. *IEEE Transactions on Software Engineering (TSE)* 2004; **30**(1).
45. Li Y, Fraser G. Bytecode testability transformation. *International Symposium on Search Based Software Engineering (SSBSE)*. Springer, 2011; 237–251.
46. Harman M, McMinn P. A theoretical and empirical study of search-based testing: Local, global, and hybrid search. *IEEE Transactions on Software Engineering (TSE)* 2010; **36**(2).
47. Wegener J, Baresel A, Sthamer H. Evolutionary test environment for automatic structural testing. *Information and Software Technology* 2001; **43**(14).
48. Sharma R, Gligoric M, Arcuri A, Fraser G, Marinov D. Testing container classes: Random or systematic? *Proceedings of the International Conference on Fundamental Approaches to Software Engineering (FASE)*. Springer, 2011.
49. Eler M, Endo A, Durelli V. Quantifying the characteristics of Java programs that may influence symbolic execution from a test data generation perspective. *Proceedings of the International Conference on Computer Software and Applications Conference (COMPSAC)*, IEEE, 2014.
50. Truong TK, Li K, Xu Y. Chemical reaction optimization with greedy strategy for the 0–1 knapsack problem. *Applied Soft Computing (ASOC)* 2013; **13**(4):1774–1780.

51. Galeotti JP, Fraser G, Arcuri A. Improving search-based test suite generation with dynamic symbolic execution. *Proceedings of the International Symposium on Software Reliability Engineering (ISSRE)*, IEEE, 2013.
52. Fraser G, Arcuri A, McMinn P. A memetic algorithm for whole test suite generation. *Journal of Systems and Software (JSS)* 2015; **103**:311–327.