A Fitness Function for Search-based Testing of Java Classes, which is based on the States Reached by the Object under Test

Ina Papadhopulli[1] and Elinda Mece[2]

[1] Polytechnic University of Tirana

Abstract

Genetic Algorithms are among the most efficient search-based techniques to automatically generate unit test cases today. The search is guided by a fitness function which evaluates how close an individual is to satisfy a given coverage goal. There exists several coverage criteria but the default criterion today is branch coverage. Nevertheless achieving high or full branch coverage does not imply that the generated test suite has good quality. In object oriented programs the state of the object affects its behavior. Thereupon, test cases that put the object under test, in new states are of interest in the testing context. In this article we propose a new fitness function which takes into consideration three factors for evaluation: the approach level, the branch distance and the new states reached by a test case. The coverage targets are still the branches, but during the search, the state of the object under test evolves with the scope to produce individuals that discover interesting features of the class and as a consequence can discover errors. We implemented this fitness function in the eToc tool. In our experiments the usage of the proposed fitness function towards the original fitness function results in a relative increase of 15.6

Index terms — structural testing, test case generation, search based software testing, fitness function, object state, coverage criteria, mutation score.

1 I. INTRODUCTION

ue to the fact that the influence of software in all areas has grown rapidly in the past 40 years, the software has become very complex and also its reliability is fundamental. All the software development phases have been adapted to produce these complex software systems, but especially the testing phase is of critical importance and testing thoroughly today's software systems is still a challenge. According to a study [1] conducted by the National Institute of Standard & Technology, approximately 80% of the development cost is spent on identifying and correcting defects. It is a well-known fact that it is a lot more expensive to correct defects that are detected during later system operation. Considering past experiences, inadequate and ineffective testing can result in social D problems and human/financial losses. In order to improve the testing infrastructure, several efforts have been made to automate this process.

In the unit testing level, there are three approaches towards automation: random testing, static analysis (Symbolic Execution [3]) and metaheuristic search. A considerable number of tools have been developed based on these approaches; eg. RANDOOP [4], EvoSuite [5], AgitarOne [6]. Nevertheless, the effectiveness of these tools is still not completely proved, because the results obtained from the experiments depend on the subjects under test. Usually, a coverage criteria is used to evaluate these tools, but achieving a high degree of code coverage does not imply that a test is actually effective at detecting faults [7]. According to [8], today there is no tool to find more than 40.6% of faults.

This article is focused on structural testing at the unit level of Java programs using Search-Based Software Testing (SBST) [9]. According to [10], SBST has been used to automate the testing process in several areas

1

44 including the coverage of specific program structures, as part of a structural, or white-box testing strategy.
45 Every unit (class) of the software must be tested before proceeding to the other stages of the development cycle.
46 SBST is a branch of Search Based Software Engineering (SBSE). SBSE is an engineering approach in which
47 optimal or near optimal solutions are sought in a search space of candidate solutions. The search is guided by a
48 fitness function that distinguishes between better and worse solutions. SBSE is an optimization approach and it
49 is suitable for software testing since test case generation is often seen as an optimization or search problem. Since
50 SBST techniques are heuristic by nature, they must be empirically investigated in terms of how costly and effective
51 they are at reaching their test objectives and whether they scale up to realistic development artifacts. However,
52 approaches to empirically study SBST techniques have shown wide variation in the literature. There exist several
53 search-based optimization methods used for test automation; e.g. genetic algorithms, hill climbing, ant colony
54 optimization and simulated annealing, etc, but Genetic algorithms (GAs) are among the most frequently applied
55 in test data generation.
56   GAs have several components which need to be defined in order for the GA to be implemented.
57   According to [10], the component that affects mostly the results obtained from the search is the fitness function.
58 The fitness function is a mathematical representation of the coverage goal the search should achieve. There are
59 different coverage goals each of them aims at covering certain parts of the unit under test. These different
60 coverage criteria verify the quality of a test suite. The gold criterion is strong mutation, but today this criterion
61 it is mainly used by the research community for evaluation of proposed techniques. The most used criterion is
62 branch coverage [11]. However achieving high branch coverage (even 100%), for some classes is not sufficient.
63   In object oriented programs the state of the object is a factor that affects the execution of a method. This is
64 why the state of the object of the Class Under Test (CUT), should evolve during the search in order to discover
65 hidden features of the class [12]. A test case that puts the object in one or several new states is of interest in
66 the testing context. The scope of this paper is to propose and evaluate a new fitness function, which rewards
67 the test cases according to branch coverage and also according to the new states the object has taken during the
68 execution of the test.
69   The rest of this paper is organized as follows: In the second section we explain in what unit testing of java
70 programs consists and in the third section we present an overview of GAs. The fourth section is focused on
71 branch coverage and the fifth section presents the proposed fitness function. The implementation of the proposed
72 fitness function is described in section six. The seventh section gives details of the experimental setup and in
73 the eighth section the results achieved are presented and discussed. We conclude finally with the conclusions we
74 have come preparing and accomplishing this study.

2   II.

3   UNIT TESTING FOR OBJECT ORIENTED SOFTWARE

77 Software testing at the unit level (Java classes) consists of three steps:
78   1) The design of test cases 2) The execution of these test cases
79   3) The determination of whether the output produced is correct or not. The second step is performed fully
80 automatically using frameworks like JUnit **??**2]. Automatically generating the test oracle is still a challenge
81 and there exists few research publications regarding this topic [13], therefore the third step is almost completely
82 performed manually by the testers. Regarding the first step, there exist a lot of research effort for the generation
83 of test cases automatically. Due to the complexity and the diversity of the programs under test this is still an
84 open research topic. Moreover test cases in object oriented unit testing are not just a sequence of input values like
85 in procedural languages. According to [14], a unit test of a Java class must accomplish the following four tasks:
86 The space of potential solutions is searched in order to find the best possible solution. This process is started
87 with a set of individuals (genotypes) which are generated randomly from the whole population space (phenotype
88 space). New solutions are created by using the crossover and mutation operators. The replacement mechanism
89 selects the individuals which will be removed so that the population size does not exceed a prescribed limit. The
90 basis of selection is the fitness function which assigns a quality measure to each individual. According to the
91 fitness function, the parent selection mechanism evaluates the best candidates to be parents in order to produce
92 better individuals in the next generation. It is the fitness function which affects the search towards satisfying
93 a given coverage criteria. Usually the fitness function provides guidance which leads to the satisfaction of the
94 coverage criterion. For each individual the fitness is computed according to the mathematical formula which
95 represents how close is a candidate to satisfy a coverage goal, e.g. covering a given branch in the unit under test.
96 GAs are stochastic search methods that could in principle run for ever. The termination criterion is usually a
97 search budget parameter which is defined at the beginning of the search and represents the maximum amount of
98 time available for that particular search.

4   IV. COVERAGE CRITERIA a) Types of Coverage Criteria

100 Automatic unit testing is guided by a structural coverage criterion. There exist many coverage criteria in
101 literature, each of them aims at covering different components of a CUT. Nevertheless, not all the criteria have
102 the same strength and can be fulfilled practically. Furthermore some criteria are subsumed by other criteria.
103 Below is a list of coverage criteria for structural testing of Java programs. [15]. This criterion is difficult to

apply and computationally expensive and it is practically only used for predicting suite quality by researchers. Another option to achieve high quality test cases with search based technique is to use a combination of multiple criteria. [16] performed an experiment to evaluate the effects of using multiple criteria and concluded that: ? Given enough time the combination of all criteria achieves higher mutation score than each criterion separately (except Weak Mutation). ? Using all the criteria increases the test suite size by more than 50% that the average test suite size of each constituent criterion used separately. ? The next best criterion (after Weak Mutation) to achieve high mutation scores is branch coverage.

The usage of multiple criteria increases the overall coverage and mutation score with the cost of a considerable increase in test suite length, so the usage of the combination in practice will be not feasible, because managing large test suites is difficult. A balance between mutation score and average test suite size is achieved with branch coverage criterion.

5 b) Branch Coverage

The most used criterion is branch coverage, but even though it is an established default criterion in the literature, it may produce weak test sets (mutation score less than 30% [17]). For example consider the Stack implementation in Figure **??**. public

6 Analyzing class Stack we notice the following errors :

? If method pop is called first and then is called method push, an uncaught exception is thrown (field size before calling push is -1). ? If method pop is called two times consequently an uncaught exception is thrown (field size before calling pop is -1). ? If method push is called four times consequently and then is called method pop an uncaught exception is thrown (field size before calling pop is 4).

? It is obvious that branch coverage is not sufficient for class Stack! Is there any possibility to improve the fitness function for branch coverage in order to obtain a test suite with higher quality? Both of the methods are covered by the test generated, but it is evident that the state of the object (the value of field size) before calling them affects the results of the tests. The same method called on different states of the object behaves differently. This is why, a possibility to improve the suite's ability to detect errors, is to evolve the state of the object during the search in order to put the object in new states that probably can discover interesting behaviors of the CUT. Since the search is guided by the fitness function, then this function should also consider the states reached by a test before evaluating it.

7 V. THE PROPOSED FITNESS FUNCTION

Fitness functions are a fundamental part of any search algorithm. They provide the means to evaluate individuals, thus allowing a search to move towards better individuals in the hope of finding a solution [18]. The approach considered here is to minimize the fitness function during the search. The fitness function proposed in this paper rewards the individuals based on how close they are at covering a target (branch) and the states they put the object under test. This function is a mathematical equation depending on the:? Approach level ? Branch Distance ? New states achieved a) Approach Level

For each target, the approach level show how many of the branch's control dependent nodes were not executed by a particular input [20]. The fewer control dependent nodes executed, the "further away" an input is from executing the branch in control flow terms. The approach level is the most used factor in the fitness function for structural criteria, but the fitness landscape contain plateaus because the search is unaware of how close a test case was to traversing the desired edge of a critical branching node.

8 b) Branch Distance

The branch distance is computed using the condition of the decision statement at which the flow of control diverted away from the current "target" branch. For every operator the branch distance is calculated using the formulas introduced by Tracey [19].

The approach level is more important that the branch distance and as a consequence the branch distance should be normalized at the fitness function formula. This distance will be normalized at a value between 0.0 and 1.0. Value 0.0 means "true"; the desired branch has been reached. Values close to 1.0 means that the condition is far from being fulfilled. Intermediate values guide slightly the search towards the accomplishment of the condition (in order to remove plateaus in the fitness landscape). The formula for branch distance in our proposed fitness function is the formula introduced by Arcuri [21].

9 ????(?????????????????????) = ???? ???? + ??

BD is the branch distance before normalization and ?? is 1.

10 c) New States Achieved (NSA)

With the term state in this paper we refer to: Definition 1. State: The set of the values of all the fields in the CUT before calling a method + the method called.

For example, for the class Stack the two states:

? field size = 0 and filed st = !null, before calling method push ? field size = 0 and filed st = !null, before calling method pop are considered two different states and both of them are interesting in the testing context. The total number of states in the CUT is computed as a product of all the possible combinations Even though class Stack is very simple, and the branch coverage obtained is 100%, the mutation score is relatively low (29%). We added an assertion in the test (line 7) and used the JUnit framework to run it in Eclipse. The test passed. The tester may assume the class is correct with 100% branch coverage and a passing test.

Is branch coverage sufficient for this class? of the class fields (declared non final) after abstraction (explained in the next section), with the number of public methods.

The approach level is more important that the number of new states achieved and as a consequence this factor should be normalized at the fitness function formula. The normalization formula is:?????? = ????????????_?????????? -????????????_?????? ????????????_??????????

The greater the number of the new states achieved by a test case the smaller this factor in the overall fitness.???????????? = ??????????????_?????????? + ???? ????+1 + ???????????? _?????????? -????????????_?????? ???????????? _?????????? d) Abstract States

If we use the real values of the fields, the number of states will be infinite. Moreover, not all the states are of equal relevance during testing. For example, from the testing prospective, calling method pop() of the class Stack with field size = 1, is the same as calling method pop with filed size = 2. On the other hand calling method pop() with filed size = 0 in not the same, since this state reveals an interesting behavior of the object under test. Therefore, we use abstractions over the values of the fields rather than the concrete values themselves. We use a state abstraction function provided by Dallmeier at al. [34]. The abstraction is performed based on the three rules below:

? If the type of the field is concrete (int, double, long etc), the value will be translated in three abstract values: $x_i < 0$, $x_i = 0$ and $x_i > 0$.

? If the type of the field is an object, the value will be translated in two abstract values: x_i = null dhe x_i null ? If the type of the field is Boolean, there is no need to do translation, since there are only two values.

For example the combinations of the field values of class Stack, after abstraction are those listed in Table 1.

11 IMPLEMENTATION OF THE PROPOSED FITNESS FUNCTION

The proposed fitness function was implemented in the eToc [22] tool. eToc is a simple search based tool for unit testing of Java programs. Is uses GA and branch coverage criterion. This tool has been mentioned in many research works and has been used as the basis for the design of other tools. eToc is appropriate for the scope used in this work. In the high level architecture of this tool [22], the Branch Instrumentor module and the Test Case Generator module need to be differently implemented for the search to be guided by the proposed fitness function. The new implementation of these modules is described below.

12 a) The Intrumentor

The function of the instrumentor module is to transform the source code of the CUT in order to provide information about the executed branches, the branch distance and the states achieved during execution. The new statements added during instrumentation must not change the behavior of the CUT. In order to obtain information for the states reached by the object under test, for each of the attributes (except those declared final) of the CUT, a get method will be added. A static analysis can be used to provide information about the mutators and inspectors methods of a class [23][24], but in this case a static whole-program analysis is required, which is very expensive for this context used. Since it is not the purpose here to obtain a behavioral model of the CUT, the get methods are appropriate to be used as inspectors for obtaining the state of the object because these methods: ? Return the value of an attribute ? Do not take parameters ? Do not have any side effects in the execution of the program.

Based on the state definition given in section 5.C, the get methods should be called before the execution of each method of the CUT, so during instrumentation the statements calling the get methods are added before the existing statements of each method. The concrete values are translated in abstract values as described in section 5.D. Then the states reached by a test case are saved in a LinkedList and consequently during fitness evaluation the new states achieved by a test case can obtained. The instrumented version of the CUT is executed repeatedly with the scope to cover a specified target (branch of the CUT). The state lists resulting after each execution are compared with the state lists of the test cases that make up the population. The new states reached by an individual are used to compute part of its fitness.

This module is also responsible for the minimization of the generated test suite. Normally during minimization the tests that do not cover any target that is not covered by any other test are omitted from the test suite. Taking into consideration that a test case that reaches one or more new states is important in the testing context, before removing a test case because it does not cover any new target, it will be reconsidered regarding the states it puts the object under test in. The test cases which contain unreached states in their state lists, will be part of the final test suite. The proposed minimization has the advantage that it probably increases the number of tests in

the generated test suite and as a consequence it also increases the length of the test suite. On the other side the usage of the proposed fitness function is expected to increase the capability of the test suite to detect errors. An experimental evaluation of the new fitness function is presented in the In this work we aim to answer the following research questions:

? RQ1: How does the usage of the proposed fitness function affect the branch coverage? ? RQ2: How does the usage of the proposed fitness function affect the mutation score of the suite? ? RQ3: How does the usage of the proposed fitness function affect the number of suite's test cases and their size?a) System Characteristics

For the experiments we used a desktop computer running Linux 32 bit Operating System, 1 GB of main memory and a Intel Core 2 Duo CPU E7400 2.8GHz x 2 Processor.

13 b) Subject Selection

Selecting the classes under test is very important since this selection affects the results of the experiments. We chose 7 open source projects and selected randomly 23 classes from them. Also, the class Stack discussed throughout this paper was used as a subject for the experiments. To obtain comprehensive results, the evaluation must be done to real and not simple subjects. Also these subjects should not have any common characteristics which affect the obtained results. The characteristics of the 24 classes are listed in Table 2. The information about LOC (without comments and empty lines) and cyclomatic complexity is obtained using Metrics 1.3.6 [25], as a plugin in Eclipse. As can be noted from Table 2, the classes have very different characteristics and complexity.

Five of the projects were downloaded from SourceForge [26] which is today the greatest open source repository (more than 300,000 projects and two million of users). One project was downloaded from the Apache Software Foundation [27] which exists from 1999 and has more than 350 projects (including Apache HTTP Server). Class StringTokenizer was taken from the java.util package which is part of jdk 1.8.0. This package has been used by several studies for evaluation of automatic test case generation techniques. a) Parameters of GA Defining the parameters of GAs to obtain the optimal results is difficult and a lot of research effort is dedicated to this topic [28] [29]. Therefore we let the parameters of the GA to their default values [22]. The values of three of the most relevant parameters are listed in Table **??**. Regarding the search budget, it was determined depending on the experiment and will be shown next for each experiment. To overcome the randomness of the genetic algorithms each experiment was repeated 5 times.

The results of the experiments (average of all runs) are presented in Table **??**. The branch coverage was measured with EclEmma. For both functions the average branch coverage is greater when the search budget is 10 min. This result was expected since the individuals improve during the search and more time results in better solutions.

In order to do the best comparison of the approaches we focus on the case with search budget of 10 min in this section, since for the scope of the experiment, it is not appropriate to compare results affected by the limited search time.

The difference between the average branch coverage is inconsiderable (0.4%) when a search budget of 10 min is used. This difference may be due to the randomness of the results achieved by the search. Since the approach presented in this work does not change the targets to cover, the almost equal coverage was expected. For the class ExplorerFrame, there is an increase of 7% in the coverage achieved by the proposed approach. Even though the targets are identical, the proposed function rewards the individuals that reach more new states and therefore the test cases after minimization may be different and more complex. So, this increase probably is the effect of indirect coverage.

Only in the case of class ArrayUtil there was a decrease of 1% in the coverage achieved, with budget 2 min, but more likely it is due to the randomness of the search. For the class NewsFactory the search failed to produce results for both approaches. We changed the parameters of the GA, but even for a population of 20, or 30 individuals, no results were generated. It is not the scope of this work to investigate the reasons why this happened.

? RQ2: How does the usage of the proposed fitness function affect the mutation score of the suite?

Since mutation score is the measure used in the strongest criterion (Mutation Coverage), here we have used it to measure the quality of the generated test suite. Computing the mutation score for a test suite requires determining, for every mutant, whether the test suite succeeds or fails when run on the mutant. In the worst case each test must be run on each mutant. For each of the classes the mutants were generated using as a plugin in Eclipse the tool MuClipse v1. **??** [30]. Mu Clipse generates mutants using the traditional operators and the operators in the class level [31]. The number of generated mutants for each class is given in generated, so that these cases can be used by MuClipse. Then, the generated tests were executed with JUnit against all the mutants and the presence of failures shows that the tests were able to kill the mutants.

The results of the mutation scores of each class for all the configurations are given in Table **??**.

The mutation scores achieved by both of the fitness functions are far from the optimal value (100%). Almost this range of mutation scores is also obtained from other studies [32]. The main reasons of these low scores are: ? the targets to cover are the branches and not the mutants ? the presence of equivalent mutants (behave the same as the original program) which cannot be killed.

280   Nevertheless, despite the relatively low mutation scores, our interest is focused on the difference between the
281 scores achieved by the original function against the proposed function.
282   For 6 classes (6/23 = 26%) there is an improvement in the mutation score achieved when using a search budget
283 of 10 min against a search budget of 2 min.
284   For the same reasons mentioned in the discussion of RQ1, to answer RQ2 we are focusing mainly at the
285 results achieved with a search budget of 10 min. The average mutation score reached by the original function is
286 35.9%, whereas the mutation score reached by the proposed function is 41.5%, thus a difference of 5.6%. The
287 improvement is 5.6/35.9 = 15.6%. For 15 classes out of 23 (15/23 = 65%), there is an improvement in the
288 mutation score achieved by the proposed function; for the remaining 8 classes (8/23 = 35%), the scores achieved
289 are identical. There is no class where using the proposed function results in a lower mutation score. Even though
290 we are aware that the results depend fact that CUT chosen have different characteristics), the results obtained
291 are very promising.
292   ? RQ3: How does the usage of the proposed fitness function affect the number of suite's test cases and their
293 size?
294   Automatically generated JUnit tests need to be manually checked in order to detect faults because automatic
295 oracle generation is not possible today. This is the reason why not only the achieved coverage of the generated
296 test suite is important, but the size of the test suite is of the same importance [33].
297   Here we refer to the size of a test suite as the number of statements after the minimization phase (without
298 assertions).
299   Only the results achieved with a search budget of 10 min, are shown in Table **??**, because in answering RQ3
300 we are interested in the number of tests generated and their size in the "worst case". The minimization phase
301 does not depend on the search budget, so the results with search budget of 10 min, subsume the scenario with a
302 search budget of 2 min. The LOC of the generated suite was obtained with the tool Metrics 1.3.6.
303   There is an increase of 314 -290 = 24 tests in the total number of test generated, or a relative increase of
304 24/290 = 8.2%. This increase is acceptable, although the number of tests in the test suite is not relevant in
305 respect to the size of the test suite, because having many short size tests is not a problem for the tester who is
306 detecting faults.
307   Regarding the size of the test suite, we can see from the results in Table **??**, that using the proposed fitness
308 function results in an average test suite size of 33.9 (781/23) statements. The relative increase is (33.9 -30.1) /
309 30.1 = 12.6%. For 8 of the classes (34%), there is no change in the average test suite size. Regarding classes
310 ExponentialFunction and GAAlgorithm (8.7% of the classes), there is a decrease in the average test suite size,
311 although there is no decrease either in branch coverage or mutation score. These results are explained with the
312 appearance of indirect coverage [36].
313   ArrayUtil is the class with the greatest test suite size because of the large number of branches (167). The
314 average increase in test suite size with the usage of the proposed function is the consequence of two reasons: ?
315 During the minimization phase the test cases that do not cover any target, but put the object under test in new
316 states, are added in the minimized test suite (as explained in Section 6)
317   ? Two different fitness functions probably will generate different test suites with different number of statements
318 (not necessarily a larger number).

14   IX. CONCLUSIONS

319 This paper concerns the fitness function used to guide the search during automatic unit test generation of Java
320 classes. The branch coverage criterion is easy to implement but can produce weak test sets. Test cases that
321 put the object under test in new states discover hidden behaviors and consequently are relevant in the testing
322 context. Targeting all the states during the search is impossible due to the fact that some of them are infeasible.
323 In this article we presented a new fitness function that takes into consideration the states reached during the
324 execution of a test case. The implementation of this fitness function is very simple since the targets to cover
325 remain the branches, but the state evolve during the search and the minimization phase the tests that reach one
326 or more new states are not removed even though these tests does not reach any uncovered branches. The usage
327 of the proposed fitness function does not decrease the branch coverage and results in a relative increase of 15.6%
328 in the achieved average mutation score with the cost of a relative increase of 12.6% in the average test suite size.
329 The results are promising but since the subjects under test are very different further evaluation of the proposed
330 approach needs to be performed.   [1] [2]

---

[1] RQ1: In our experiments, there is no difference in the average branch coverage achieved between the usage of the original fitness function and the proposed fitness function.

[2] RQ2: In our experiments, the usage of the proposed fitness function results in a relative increase of 15.6% in the average mutation score achieved against the original fitness function.RQ3: In our experiments, the usage of the proposed fitness function results in a relative increase of 8.2% in the average number of test cases and 12.6% in the average test suite size achieved against the original fitness function.

Figure 1: ?

Figure 2:

class Stack {
private int size = 0; private int st [] = new int [4]; void push (int x){ if (size < st.length) st[size++] = x; } int pop (){ return st[size–]; } The class Stack is very simple (8 LOC, 2 attributes, 2 methods). Suppose the test suite generated } is the test suite given in Figure 2. 1. @Test 2. public void test0() { 3. Stack s0 =new Stack();
4. s0.push(1);
5. s0.push(0);
6. int int0 = s.pop();
7. assertEquals(0, int0);
8. s.push(0);
9. s.push(0);

[Note: C © 2016 Global Journals Inc. (US) Figure1 : Example Stack implementation 10. s.push((-1916)); 11. s.push((-1916)); 12. } Figure 2 :]

Figure 3:

1

	size	st
	= 0	null
state1	> 0	null
state2	< 0	null
state3		
VI.		

Figure 4: Table 1 :

II

Project	Class	LOC	Branch	Mutants	next section. Non-	Public	Cyclomatic	URL
			es		final Fields	Methods	Complexity	y projektit
Comm	Staku Option	12	4 131	22 140	2 9	2 42	1.5 1.52	https://co
ons	TypeHandler	155	28 4	28 1 19	0 2	9 2	2.66 1	mmons.ap
CLI	AlreadySe-	124	21 36	161 31	2 2	8 19	1.875	ache.org
Math4	lectedExcepti	26 86	11	1769	1 0	9 36	1.526	https://sou
J jdk	OptionGroup	61 40	167	827	2 2	12 1 3.48	rceforge.n
Geneti	Rational Expo-	320	100	682	7 6	20 6	3.63	et/projects
c Al-	nentialFunction	245	24 78	434 6	3 4	8 4	1.091	/math4j
gorit	ArrayUtil Poly-	102	14 9	21 44	8 8	11 9	3.12 2	https://sou
h i	Function Complex	313	13 26	74 41	16 1	17	1.4 1.08	rceforge.n
Ob-	StringTokenizer	65	41 47	155 47	8 4	13 4	1.44	et/projects /
ject	GAAlgorithm	14 62	12 11	29 88	3	10 7	1.571	j https://sou
Ex-	Genome	158	45 12	59		4	2.235	rceforge.n /
plor	Population Ex-	114					2.2 1.27	https://sou
Newz	plorerFrame Ob-	177				4 2	rceforge.n
Grabb	jectViewManager	63 28						et/projects
er	DirectoryDialog	121						ber /newsgrab
	StringSorter	55						
	BatchJob							
	NewsFactory							
	SongInfo							
	OptionsPanel	363	75	214	15	4	9.8	
Jipa	Label	18	11	42	3	4	1.8	https://sou
	Variable	40	23	87	3	4	2.1	rceforge.n
Total		2762	943	5021	111	264		/

Figure 5: Table II :

lll

Parameter	Value
Population Size	10
Search Budget	600s
Maximal number of generations/target	10

b) Experiment

For each of the classes we run eToc with the following configurations:

1. Original Fitness (OF) function with search budget of 2 min

2. Proposed Fitness (PF) function with search budget of 2 min

3. Original Fitness (OF) function with search budget of 10 min

4. Proposed Fitness (PF) function with search budget of 2 min

Figure 6: Table lll :

IV

A Fitness Function for Search-Based Testing of Java Classes, which is Based on the States Reached b

	Object under Test					
VII. ExponentialFunction 100 100 EXPERIMENTAL EVALUATION 100				100	60	55
ArrayUtil	100	99	100	100	9	9
PolyFunction	-	-	85	87	-	-
Complex	100 100		100	100	34	37
StringTokenizer	65	65	69	69	15	21
GAAlgorithm	93	93	93	93	33	33
Genome	44	44	55	55	0	4
Population	92	92	100	100	32	32
ExplorerFrame	8	15	8	15	0	3
ObjectViewManager 54 DirectoryDialog 6 NewsFactory -	54		54 6 -	54	17	24
	6			6	0	0
	-			-	-	-
SongInfo BatchJob	50 100 100 50			50 100	50	22	27
					100	62	69
StringSorter OptionPanel Label Variable Average Total	100 100 –100 100 100 100 60.5 69 –			100 37 100 100 74.8 75.2		

Year 2016

12

Volume XVI Issue II Version I

Global Average Of All Runs For Each Cut BC with PF (2 min) BC with OF (10 BC with PF (10 MS with O

Global Journal of Computer Science and Technology C ( )

Option	69	69	69	69	41	49
TypeHandler	75	75	75	75	46	46
AlreadySelectedException 100		100	100	100	100 100
OptionGroup	100	100	100	100	84	89
Rational	94	94	94	94	75	79
ExponentialFunction	100	100	100	100	60	55

Figure 7: Table IV :

2

Year 2016

13

( )

Figure 8: Table 2 .

332 [Tracey ()] *A Search-Based Automated Test-Data Generation Framework For Safety-Critical Software*, N Tracey
333   . 2000. University of York (PhD thesis)

334 [A simple and practical approach to unit testing: The JML and JUnit way The Economic Impacts of Inadequate Infrastructure fo
335   'A simple and practical approach to unit testing: The JML and JUnit way'. 01-12. *The Economic Impacts*
336   *of Inadequate Infrastructure for Software Testing*, Nov. 2001. NIST (National Institute of Standards and
337   Technology ; Department of Computer Science, Iowa State University (Technical Report)

338 [Fraser and Arcuri ()] 'Achieving Scalable Mutationbased Generation of Whole Test Suites'. G Fraser , A Arcuri
339   . *Empirical Software Engineering*, 2014.

340 [Lakhotia et al. ()] 'An empirical investigation into branch coverage for C programs using CUTE and AUSTIN'.
341   K Lakhotia , P Mcminnb , M Harman . *Journal of Systems and Software* 2010.

342 [Lakhotia et al. ()] 'AUSTIN: A Tool for Search Based Software Testing for the C Language and Its Evaluation
343   on Deployed Automotive Systems'. K Lakhotia , M Harman , H Gross . *International Symposium on SBSE*,
344   2010.

345 [Rojas et al. ()] 'Combining Multiple Coverage Criteria in Search-Based Unit Test Generation'. J Rojas , J
346   Campos1 , M Vivanti , G Fraser , A Arcuri . *Proceedings of the 26th IEEE/ACM International Conference*
347   *on Automated Software Engineering (ASE)*, (the 26th IEEE/ACM International Conference on Automated
348   Software Engineering (ASE)) 2011. p. .

349 [Papadhopulli and Meçe (2015)] 'Coverage Criteria for Search Based Automatic Unit Testing of Java Programs'.
350   E Papadhopulli , Meçe . *International Journal of Computer Science and Software Engineering* October 2015.
351   4 (10) .

352 [Ma and Ouffut (2014)] *Description of Class Mutation Mutation Operators for Java*, Y Ma , J Ouffut . August
353   2014.

354 [Shamshiri et al. ()] 'Do Automatically Generated Unit Tests Find Real Faults? An Empirical Study of
355   Effectiveness and Challenges'. S Shamshiri , R Just , J Rojas , G Fraser , P Mcminn , A Arcuri . *Proceedings*
356   *of the 30th IEEE/ACM International Conference on Automated Software Engineering (ASE)*, (the 30th
357   IEEE/ACM International Conference on Automated Software Engineering (ASE)) 2015.

358 [Fraser et al. ()] 'Does Automated Unit Test Generation Really Help Software Testers? A Controlled Empirical
359   Study'. G Fraser , P Mcminn , A Arcuri , M Staats . *ACM Transactions on Software Engineering and*
360   *Methodology* 2015.

361 [Tsuji and Akinyele (2007)] 'Evaluation of AgitarOne'. T Tsuji , A Akinyele . *Analysis of Software Artifacts Final*
362   *Project Report*, April 24. 2007.

363 [Wegener et al. (2001)] 'Evolutionary Test Environment for Automatic Structural Testing'. J Wegener , A Baresel
364   , H Sthamer . *Information and Software Technology Special Issue on Software Engineering using Metaheuristic*
365   *Innovative Algorithms*, December 2001. 43 p. .

366 [Tonella] 'Evolutionary Testing of Classes'. P Tonella . *Proceedings of International Symposium on Software*
367   *Testing and Analysis (ISSTA)*, (International Symposium on Software Testing and Analysis (ISSTA)) p.
368   2004.

369 [Mirazz ()] *Evolutionary Testing of Stateful Systems: a Holistic Approach*, M Mirazz . 2010. University of Torino
370   (PhD thesis)

371 [Fraser and Arcuri] 'EvoSuite at the SBST 2015 Tool Competition'. G Fraser , A Arcuri . *Proceedings of*
372   *International Conference in Software Engineering (ICSE)*, (International Conference in Software Engineering
373   (ICSE)) p. 2015.

374 [Pacheco et al.] 'Feedback-directed Random Test Generation'. C Pacheco , S Lahiri , M Ernst . *Proceedings of*
375   *International Conference in Software Engineering (ICSE)*, (International Conference in Software Engineering
376   (ICSE)) p. 2007.

377 [Fraser and Arcuri ()] 'Handling test length bloat'. G Fraser , A Arcuri . *Proceedings of ICST*, (ICST) 2013.

378 [Arcuri ()] 'It Does Matter How You Normalise the Branch Distance in Search Based Software Testing'. Arcuri
379   . *Third International Conference on Software Testing, Verification and Validation*, 2010.

380 [Dallmeier et al. ()] 'Mining Object Behaviour with ADABU'. V Dallmeier , C Lindig , A Vasilowski . *Proceedings*
381   *of the International Workshop on Dynamic Systems Analysis*, (the International Workshop on Dynamic
382   Systems Analysis) 2006.

383 [Le et al. ()] 'MuCheck: An Extensible Tool for Mutation Testing of Haskell Programs'. D Le , M Alipour ,
384   R Gopinath , A Groce . *Proc. of the International Symposium on Software Testing and Analysis*, (of the
385   International Symposium on Software Testing and Analysis) 2014.

386 [Fraser and Zeller ()] 'Mutation-Driven Generation of Unit Tests and Oracles'. G Fraser , A Zeller . *IEEE*
387   *Transactions on Software Engineering* 2012.

388 [Eiben and Smit ()] 'Parameter tuning for configuring and analyzing evolutionary algorithms'. E Eiben , S K
389   Smit . *Journal: Swarm and Evolutionary Cmputation* 2011. p. .

390 [Rountev ()] 'Precise identification of side-effect-free methods in java'. Rountev . *20th IEEE International*
391   *Conference on Software Maintenance (ICSM '04)*, 2004. p. .

392 [Salcianu and Rinard (2005)] 'Purity and side effect analysis for java programs'. M Salcianu , Rinard . *Proceedings*
393   *of the 6th International Conference on Verification, Model Checking and Abstract Interpretation*, (the 6th
394   International Conference on Verification, Model Checking and Abstract Interpretation) January 2005. p. .

395 [Mcminn (2004)] 'Search-based Software Test Data Generation: A Survey'. P Mcminn . *Software Testing,*
396   *Verification and Reliability*, June 2004. p. .

397 [Gross et al. ()] 'Search-based system testing: high coverage, no false alarms'. F Gross , G Fraser , A Zeller .
398   *Proceedings of International Symposium on Software Testing and Analysis (ISSTA)*, (International Symposium
399   on Software Testing and Analysis (ISSTA)) 2012.

400 [Tonella ()] *Search-Based Test Case Generation*, P Tonella . 2013. TAROT Testing School Presenetation.

401 [Cadar and Sen ()] 'Symbolic Execution for Software Testing: Three Decades Later'. C Cadar , K Sen .
402   *Communications of ACM* 2013. p. .

403 [Aleti and Grunske ()] 'Test Data Generation with a Kalman Filter-Based Adaptive Genetic Algorithm'. L Aleti
404   , Grunske . *Journal of Systems and Software* 2014.

405 [Papadhopulli and Frasheri ()] 'Today's Challenges of Symbolic Execution and Search-Based for Automated
406   Structural Testing'. N Papadhopulli , Frasheri . *Proceedings of ICTIC*, (ICTIC) 2015.