Jurnal Ilmu Komputer dan Informasi (Journal of Computer Science and Information)
17/1 (2024), 49-58. DOI: http://dx.doi.org/10.21609/jiki.v17i1.1198

Implementation Genetic Algorithm for Optimization of Kotlin Software Unit Test Case Generator

Mohammad Andiez Satria Permana, Muhammad Johan Alibasa, Sri Widowati

School of Computing, Telkom University, Bandung, Indonesia

Email: andiezpermana@student.telkomuniversity.ac.id, alibasa@telkomuniversity.ac.id sriwidowati@telkomuniversity.ac.id

Abstract

Unit testing has a significant role in software development and its impacts depend on the quality of test cases and test data used. To reduce time and effort, unit test generator systems can help automatically generate test cases and test data. However, there is currently no unit test generator for Kotlin programming language even though this language is popularly used for android application developments. In this study, we propose and develop a test generator system that utilizes genetic algorithm (GA) and ANTLR4 parser. GA is used to obtain the most optimal test cases and data for a given Kotlin code. ANTLR4 parser is used to optimize the mutation process in GA so that the mutation process is not totally random. Our model results showed that the average value of code coverage in generated unit tests against instruction coverage is 95.64%, with branch coverage of 76.19% and line coverage of 96.87%. In addition, only two out of eight generated classes produced duplicate test cases with a maximum of one duplication in each class. Therefore, it can be concluded that our optimization with GA on the unit test generator is able to produce unit tests with high code coverage and low duplication.

Keywords: *unit test generator, test case, optimization, Kotlin, genetic algorithm (GA)*

1. Introduction

Android application development has been popular since the last few decades. The Kotlin language is in great demand compared to its predecessor language, Java. One of the reasons is that Kotlin is found to be more effective than Java to build an android application [1]. To develop a high quality application, unit testing is crucial to make sure the implementation is correct. This unit testing process requires effort and time costs since the developers need to write test cases and test data. This issue encourages researchers and industries to develop automatic unit test generator systems. However, at the time of this paper is being written, there is no software or tool that can be used as a unit test generator for the Kotlin programming language. In addition, generating effective and efficient test cases can be challenging, especially in complex software systems. The generated test cases may have duplications or errors [2].

To overcome this challenge, we have explored various approaches to fix the problem of the generation of test cases. One such approach is using optimization problems [3, 4]. Optimization problems are finding optimal results by maximizing or minimizing a function [5]. Optimization can use algorithms such as grid search, random search, and genetic algorithm (GA) [6]. GA is a powerful optimization technique that mimics natural selection, where a population of potential solutions evolves to find optimal or near-optimal solutions to a given problem. GA can be used in complex problems with many parameters [7]. Several issues of implementing GA in test case generations include test case that might be duplicates and mutation process that can take long time. Therefore, there should be improvements on the mutation process to minimize the previous issues.

In this research, we propose and develop an automatic test case generator system that uses GA in optimizing the results of generating test cases. In our system, we also utilize ANTLR4 parser to

modify the mutation issues so that it can minimize the duplication issue and reduce the time to find the optimal solutions. By using ANTLR4, the mutation process can be optimized by using the correct static types from the parameters and the return value.

The structure of this paper is as follows: the second section discussed related works surrounding the topic. The third section explained our unit test generation and optimization method with GA. The evaluation and discussion of the research results were explained in the fourth section. Finally, the fifth section discussed the conclusion and future works.

2. Related Works

The test case generator has several approaches. One of them is the random input program approach that will be created. Various input probabilities can be used, ranging from uniform or biased inputs, such as random integers from zero to one [8]. Another approach is using fuzz testing or fuzzing. Fuzzing consists of taking well-formed inputs and repeatedly modifying them, more or less randomly, to generate new inputs [9].

In the research conducted by Bayusandya Tresnayatna et al. [2], another approach used is to use a basis path to generate test cases and test data. The basis path method analyzes the program domain to obtain appropriate test data based on Cyclomatic Complexity (CC) which helps the tester to estimate the number of test cases needed. This research also requires Abstract Syntax Tree (AST) by parsing source code using an ANTLR4 parser.

Abstract Syntax Tree (AST) is a tree designed to represent the abstract syntactic structure of source code. It is used widely in programming languages and software engineering tools [10]. One way to generate an AST is by using a parser. ANTLR is a parser used to read, process, execute or translate structured text. This parser is widely used to create AST [11]. The process in ANTLR involves a formal grammar file to generate the parser. The grammar file includes the lexical rules and syntax of the parser [12].

For the resulting test cases to have results that have few problems, such as duplication of test cases, several approaches can be used. One way is the optimization problem. Optimization problems have begun to be used in test case generators using grid search, random search, and genetic algorithms [6]. The genetic algorithm is one of the most used because it has the most efficient in time and space compared with the other algorithm [3, 13, 14].

The genetic algorithm is an optimization algorithm that uses a sequential approach to solve problems. It uses several processes to distinguish individuals at each solution iteration by swapping, adding, and selecting the parameters under test. Like evolution in biology, it simulates the process of natural selection. In this case, individuals with the best ability, called fitness value, will survive in the next iteration [6].

The process of population generation, swapping, and mutation in the genetic algorithm is done randomly just like in the process of evolution. An individual could be the reference for the randomization of this algorithm process. This individual will be compared with other randomized individuals. Compared to other search algorithms, such as grid search and random search algorithms [6], genetic algorithms do not require additional information about the problem to be solved. This particular feature addresses an unresolvable issue by other optimization algorithms due to factors such as discontinuity, degradation, and non-linearity [5, 15].

Pynguin is a unit test generator library that uses a genetic algorithm to optimize the results [3]. Pynguin is used to generate unit tests from the Python programming language. Pynguin uses search-based test creation to create tests that maximize code coverage [14]. This library is open-source so anyone can refer to Pynguin. Pynguin is based on previous research on the Java language, namely EvoSuite [3].

EvoSuite is a software tool created to generate test cases in the Java programming language. The generated tests have the achievement of high code coverage and include assertions that are appropriate to the subject being tested. Evosuite also uses a meta-heuristic search algorithm to generate test cases and test data. Evosuite aims to produce test suites with high code coverage with this method. [13, 16].

3. Methodology

The proposed method consists of several parts: parsing source code using ANTLR4 to collect AST, AST translation, initiating parent from AST to translate the methods that want to generate the test cases, and process optimization using GA. Each part contains several sub-parts, which will be discussed in this section.

Furthermore, the unit tests that have been created will be run on the Kotlin application manually. Each unit test that has been made will be recalculated for its code coverage and checked to determine whether the proposed method impacts the resulting code coverage unit test. This research was conducted to see whether optimization with GA on the unit test generator can produce test cases that have high

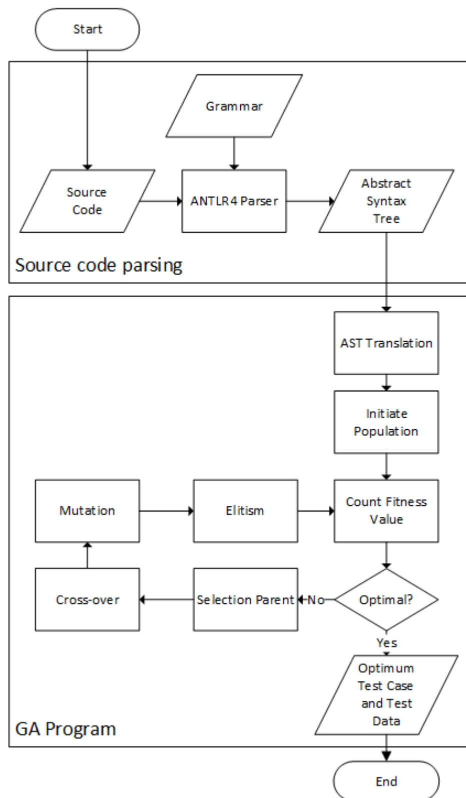

Figure 1. System diagram for unit test generator.

code coverage, especially in the Kotlin programming language.

3.1. Source Code

```
object Integer {
    fun isPrime(n: Int): Boolean {
        if (n <= 1) return false
        if (n == 2 || n == 3) return true
        if (n % 2 == 0 || n % 3 == 0) return false
        var i = 5
        while (i <= Math.sqrt(n.toDouble())) {
            if (n % i == 0 || n % (i + 2) == 0) return false
            i += 6
        }
        return true
    }
}
```

Figure 2. Source code subject of function determining an integer is a prime number.

The source code used in this research is based on the Kotlin programming language. Subjects used for testing there are four classes. Based on Table 1, each class has one function with a different parameter data type and number of parameters. The source codes used in this research are function isPrime[1], checkTriangleValidity[2], and nthFibonacci [3]. In addition to the previous source codes, we added a function that we created to check the ability to cover all paths. Figure 2 shows the function isPrime used in this research. The source code is tested one by one in the GA program that has been created. The unit test results will be retested to analyze the code coverage used.

3.2. Source Code Parsing Using ANTLR4

The source code used in this research is based on the Kotlin programming language. The library used to get the AST from the source code is kotlinx.ast[4]. The AST will be generated by changing the source code through the ANTLR4 parser with the help of the grammar included in the library. The resulting AST will be in the form of a summary which can be seen in Figure 3.

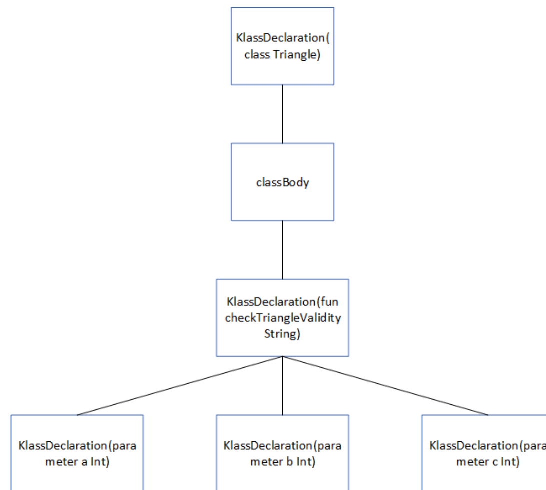

Figure 3. AST summary from kotlinx.ast4 library.

This library will generate an AST summary using KlassDeclaration objects and other markers. In Figure 4, each KlassDeclaration has parameters consisting of two, three, or more parts. The first part of the KlassDeclaration represents the type of the existing block, such as ”parameter” for parameters and

[1]https://www.geeksforgeeks.org/java-program-to-check-if-a-number-is-prime-or-not/ Function isPrime
[2]https://www.geeksforgeeks.org/check-whether-triangle-valid-not-sides-given/ Function checkTriangleValidity
[3]https://www.geeksforgeeks.org/java-program-for-program-for-fibonacci-numbers/ Function nthFibonacci
[4]https://github.com/kotlinx/ast Generic AST parsing library for kotlin multiplatform

Table 1. Source codes used as the subject.

Class	Function	Parameter		Return Type
		Number of Param	Data Type	
Integer	isPrime1	1	Integer	Boolean
TestVersion	testFunction	2	Integer, Integer	Integer
Triangle	checkTriangleValidity2	3	Integer, Integer, Integer	String
NthFibonacci	nthFibonacci3	1	Integer	Integer

”fun” for functions. Second is the name of the block, for example, the name of the parameter, function, or class. The third is the data type that represents the declaration. The class will be the implementation or extends class; the function is the return data type that will be returned and the parameters that define the data types in those parameters. The sequence will change when the type of block represented in the first part has modifiers such as private, open, or public. The entire series will shift to the right.

KlassDeclaration(fun checkTriangleValidity String)

Figure 4. KlassDeclaration object from kotlinx.ast4 library.

This research only requires parameters in the functions in the source code and the return data type. Both of these requirements can be obtained from the summary type AST. Requirements can be seen in Figure 4 with the KlassDeclaration object, which has parameter tags for parameters in each function. Note that the parameter for the function comes after the KlassDeclaration object, which has the name “fun.” The limit of the parameters of each function is known by checking whether after the last KlassDeclaration with type ”parameter” is a KlassDeclaration with type ”fun” or not at all. The function’s return data type can be seen from the last section in KlassDeclaration to the kind ”fun.”

3.3. AST Translation

The work in this section will convert the AST into an object class that will facilitate the process in the following area. There are three main classes, File, ClassBody, and Function, to hold the identifiers in the AST. The File object class has the name information of the Kotlin file that is read and the name of the package used in the Kotlin file. The package is used if the result wants to directly use the generated unit test result without changing the package manually. ClassBody is used to store the name of the class in the Kotlin file along with the functions in it. The Function class is a container for objects in the function, such as input parameters, return data types, and whether the parameters and return data types are nullable.

Algorithm 1: Generate Parent

```
Input:
  parent_count: Number of parents
  functions: List of object Function
  class_name: Class name
  is_companion: Check companion object
Output: ClassParent generated and append
procedure generate_class_parent(self)
  for range from 0 to parent_count do
    class_parent := ClassParent();
    temp := [];
    foreach fun in self.functions do
      rand := random.randint(1, 50);
      parent := Parent();
      parent.branch := fun.branch;
      parent.tag := fun.name;
      for i in range(0, rand) do
        assertion :=
        generate_assertion;
        (see Algorithm 2)
        par-
        ent.assertions.append(assertion);
      end

      class_parent.parents.append(parent);
      code_coverage, parent.branch :=
      count_fitness;
      (see Algorithm 3)
      parent.code_coverage :=
      code_coverage;
      temp.append(code_coverage);
    end
    class_parent.code_coverage :=
    sum(temp) / len(temp);

    self.class_parents.append(class_parent);
  end
```

3.4. Initiating Parent

The previous object class from Section 3.3 will be used for parent creation. Creating a parent consists of object classes, such as ClassParent, Parent, and Assertion. The first is the ClassParent object which holds information about the code coverage for the entire file and its functions. The parent contains an assertion list containing the assertions used for the test case results and the code coverage value in the function. Assertion is a class that holds assertions, whether they assert equals or not equals. This class contains the function's name to call the intended function to get the original value of the function. Then there is a list of original values that will fill in the parameters in the function according to the data type. Another variable inside the Assertion class contains the expected value, the comparison value in the assertion with the original value generated in the function.

Algorithm 2: Generate Assertion

```
Input:
  class_name: Class name
  fun: An object Function
  i: Iteration
Output:
  assertion: A generated Assertion object
function generate_assertion(class_name:
string, function:Function, i:integer)
   assertion := Assertion();
   assertion.type := AssertType.EQUALS;
   assertion.name := fun.name + "_" + str(i);
   if fun.return_type == None then
   |   return None;
   end
   foreach param in fun.params do
   |   data_type := param.data_type;
   |   Generate real value based on
   |   data_type and nullable;
   |   assertion.real.append(real_value);
   end
   data_type := fun.return_type;
   Run Kotlin program using the
   function under test;
   Get expected value from Kotlin program;
   Translate value based on data_type;
   assertion.expected := expected_value;
   return assertion;
```

The population initiation process generates a ClassParent object based on a ClassBody object. A parameter in the GA class determines the population size [7], which will be explained in Section 3.5. Based on Algorithm 1, each iteration of the population generated will perform a loop to initialize the Parent class based on the Function class list. At each iteration of the looping function, a random number is selected to get the number of Assertions in each Parent object.

The initial assertion will rely on randomizing the original parameter value of the function based on the data type [17]. Based on Algorithm 2, the number of parameters, whether integer, float, or double, will be randomized from -100 to 100. Boolean parameters will be randomized between true and false. The string parameter will be randomized with a string length from 1 to 20, with letters, numbers, and characters. The values of these assertion parameters will be entered sequentially with the parameters in the class function, so there is no need to recheck the data type in the future.

The step to get the expected value requires a function call process with the values generated in the previous step. A Kotlin application is needed in this GA application to be able to run functions and also run tests to get code coverage. Added explanation to Algorithm 2, the action taken is to copy the Kotlin file, which is the subject, into the Kotlin application. The program will write a new Kotlin main file by taking the function result with the previous random parameter value and printing it on the command line. With the help of the subprocess module in Python, programs can run command shells to run Kotlin applications. The value will be taken from the command line and entered into the expected variable in the Assertion class.

The code coverage calculation process is carried out for each iteration of the looping function. JaCoCo[5] is one of the libraries used to get code coverage values in Java programs but can be used in the Kotlin language. JaCoCo5 will generate a report in CSV format and contain the number of instructions (byte codes), branches, and lines that are covered and not. The list of assertions made will be written to a file in the test section of Kotlin application to run the JaCoCo5 report. The program reads the CSV report file to get the coverage statement and includes it in the Parent object.

The last is calculating the average code coverage to be included in the code coverage variable in the ClassParent object. Calculations in this section are used if there is more than one function in one class.

[5]https://www.eclemma.org/jacoco created by the EclEmma team

3.5. Genetic Algorithm Process

GA has several processes in it. Based on research conducted by Faisal Dharma et al. [18], the GA process goes through several steps described in Figure 1. This research makes the GA process in one GA class consisting of the population size parameter used as an iteration in the parent initiation process, list object class Function, class name, and is_companion with a Boolean data type. The is_companion variable is used when a class has a companion object or static modifier in Java. The goal is to write a function call that will be made unit test.

3.5.1. Fitness Function. The fitness function calculation uses the code coverage of the assertions made through JaCoCo5. According to research by B. I. Al-Ahmad et al. [19], JaCoCo5 has five types of coverage. The coverage includes instruction, branch, line, method, and cyclomatic complexity coverage. This research uses instruction coverage as fitness value because instruction coverage provides better precision than coarser coverage metrics [20]. The instruction coverage can be seen in Equation (1).

$$Cov_{ci}^{I} = \frac{CI_{ci}}{TNCI_{ri}} \tag{1}$$

where Cov_{ci}^{I}: Instruction coverage weight of the class, CI_{ci}: Covered Instructions of class ci, $TNCI_{ri}$: Total Number of all Classes' Instructions in each release ri [19].

Addition from Algorithm 3, calculating fitness value using JaCoCo5 can be done by writing a new file in the Kotlin application test section. Fitness calculation results can be obtained by running JaCoCo5 test report feature through the command line. The resulting test report will be in CSV format and translated using Pandas[6] libary. Fitness calculations are carried out in the population generation in Section 3.3, cross-over in Section 3.5.3, and mutation in Section 3.5.4.

3.5.2. Selection Parent. The selection process is done to select two parents that will be used in the cross-over process. This research uses the roulette wheel method based on research conducted by Faisal Dharma et al. [18]. Parent selection will be taken from a random number done N times. The selected parent must not use the same parent. The last parent selection process will be repeated if the same selected parent is selected.

[6]https://pandas.pydata.org/

Algorithm 3: Count Fitness Value

```
Input:
  parent: Object Parent
  fun_name: Function name
  functions: List of object Function
  is_companion: Check companion object
Output:
  code_coverage: Code coverage
  branch: Number of branch
function count_fitness(self, parent:Parent,
fun_name:str)
    Create file test in Kotlin program;
    Copy subject Kotlin file to Kotlin program;
    Run JaCoCo5 test report;
    data_frame =
    pd.read_csv("jacocoTestReport.csv");
    class_name := self.class_name;
    if self.is_companion then
    |   class_name += ".Companion";
    end
    df_result = df.loc[data_frame['CLASS'] ==
    class_name];
    instruction_covered =
    df_result['INSTRUCTION_COVERED'];
    instruction_missed =
    df_result['INSTRUCTION_MISSED'];
    branch_covered =
    df_result['BRANCH_COVERED'];
    branch_missed =
    df_result['BRANCH_MISSED'];

    branch = int(branch_covered + branch_missed);
    instruction = int(instruction_covered +
    instruction_missed);
    coverage = float(instruction_covered) /
    float(instruction);
    return coverage, branch;
```

3.5.3. Cross-over. This research performs the cross-over process by swapping the parameter values in the Assertion class on the two selected parents. There is a for loop to explore the list of class assertions in each parent. The number of iterations is determined by seeing which assertion list length is the least of the two parents. The parameter values selected will be randomized between one parent and the second parent. The parameter values in the Assertion class have been sorted according to the parameters in the function being tested. The parameter values will be saved for the next generation.

Each iteration of the assertion list is checked for duplication using the same method as calculating

fitness. The new assertion will be inserted into the parent and tested for code coverage. Suppose there is no increase in the code coverage value of the added assertion. In that case, the assertion is included in the duplication and not used [14]. Otherwise, the assertion will be included in the new generation. Iteration can end if the code coverage has reached 100%. The code coverage value obtained is stored in the object for the next generation.

3.5.4. Mutation. Mutation in the GA process is done by changing the parameter values in the parent [3]. The mutation value is determined based on the data type of the parameter value. In numeric data types, there is a random subtraction or addition of values. Randomized values also include negative and positive values. The Boolean data type will use random selection for Boolean operations such as "and" and "or". The string data type will truncate an existing string by a specific limit and replace it with another random string.

Like in the cross-over process, there is a process to get the expected value similarly. After the expected value is obtained, the last process is obtaining code coverage in the same way as in the cross-over process.

3.5.5. Elitism. The elitism process is required to select the population that deserves to be continued in the next generation as the new parent. With the help of sorting, elitism is done by sorting the number of assertions obtained from the cross-over and generation process. The smallest number will be at the top of the list. It is also sorted by code coverage or fitness value from largest to smallest. The number of parents taken is by the GA parameter number of parent classes. From this process, we will get a population with extensive code coverage and a small number of duplications.

At the end of the GA process, the parents are checked based on their fitness value. The best parent will be checked if its fitness value exceeds 85% to complete the iteration of the GA process. The final result of the GA process is a printed unit test file of the best parent generated in Kotlin file format.

4. Result and Discussion

This research chose a different number of parents and maximum generation in the GA process. Each class used pairs of parents and maximum generations: four parents and five maximum generations; eight parents and seven maximum generations. This difference was made to check if there was an effect on the results of the number of parents and the number of maximum generations. This research produces four Kotlin files that contain unit tests and test cases. Recalculation of code coverage with JaCoCo5 is also done by including branch coverage and line coverage, which can be seen in Equation (2) and (3).

$$Cov_{ci}^{B} = \frac{CB_{ci}}{TNCB_{ri}} \tag{2}$$

where Cov_{ci}^{B}: Branch coverage weight of the class, CB_{ci}: Covered Branches of class ci, $TNCB_{ri}$: Total Number of all Classes' Branches in each release ri [19].

$$Cov_{ci}^{L} = \frac{CL_{ci}}{TNCL_{ri}} \tag{3}$$

where Cov_{ci}^{L}: Line coverage weight of the class, CL_{ci}: Covered Lines of class ci, $TNCL_{ri}$: Total Number of all Classes' Lines in each release ri [19].

Table 2. Generated unit test's code coverages with four parents and five maximum generations.

Generated Unit Test	Instruction Cov.	Branch Cov.	Line Cov.
TestVersionTest	100%	100%	100%
TriangleTest	95.45%	66.67%	75%
NthFibonacciTest	93.94%	66.67%	100%
IntegerTest	86.36%	64.29%	100%

Table 3. Generated unit test's coverage counts with four parents and five maximum generations.

Generated Unit Test	Instruction		Branch		Test Case
	ctx	miss	ctx	miss.	
TestVersionTest	16	0	4	0	3
TriangleTest	21	1	4	2	2
NthFibonacciTest	31	2	4	2	1
IntegerTest	38	6	9	5	2

Table 4. Generated unit test's code coverages with eight parents and seven maximum generations.

Generated Unit Test	Instruction Cov.	Branch Cov.	Line Cov.
TestVersionTest	100%	100%	100%
TriangleTest	100%	66.67%	100%
NthFibonacciTest	93.94%	66.67%	100%
IntegerTest	95.45%	78.57%	100%

The experiment generates two unit tests on each class with differences in the number of parents and maximum generation. Table 2 and 4 show the code coverage results obtained from the unit test generator in experiments with different parameters. Instruction coverage generated by the first experiment on the TestVersion class from Table 1 gets the instruction,

Table 5. Generated unit test's coverage counts with eight parents and seven maximum generations.

Generated	Instruction		Branch		Test
Unit Test	ctx	miss	ctx	miss.	Case
TestVersionTest	16	0	4	0	3
TriangleTest	22	0	4	2	2
NthFibonacciTest	31	2	4	2	1
IntegerTest	42	2	11	3	4

branch, and line coverage of 100%. This code coverage also obtained the same results in the second experiment. The amount of coverage on the TestVersion class is all instructions and branches, as seen in Table 3 and 5. The number of test cases generated is the same in both experiments of 3 test cases.

The Integer, Triangle, and NthFibonacci classes from Table 1 have different results in the two experiments. The unit test generated by the Integer class produces unit tests in Table 2 and 4 with different instruction, branch, and line coverage. The first experiment produces 86.36% instruction coverage, 64.29% branch coverage, and 100% line coverage. The second experiment, which can be seen in Table 4, experienced an increase in code coverage to 95.45% instruction coverage and 78.57% branch coverage. In Table 3 and 5, it can be seen that the Integer class has 38 out of 44 instructions covered in the first experiment and 42 out of 44 instructions from the second experiment. However, 9 out of 14 branches and 11 out of 14 are covered. The resulting test cases differ on different parameters, with the first experiment totaling 2 test cases and the second counting 4 test cases.

Judging from Table 3 and 5, the Triangle class produces a difference in the number of instructions covered. There are 21 out of 22 instructions in the first experiment. In the second experiment, all instructions are covered. However, the branches covered have the same number of 4 out of 6 branches covered in both experiments. This affects the results of instruction coverage which can be seen in Table 2. Instruction coverage increased from the first experiment, which resulted in 95.45% to 100% in the second experiment. The branch coverage results remained the same at 66.67% in both experiments. The number of test cases generated was 2 test cases in both experiments.

The last class, NthFibonacci, has no change in code coverage and the number of codes covered in both experiments based on Tables 2, 3, 4, and 5. The resulting code coverage is 93.94% in instruction coverage, 66.67% in branch coverage, and 100% in line coverage. The covered instructions amounted to 31 out of 33 instructions, and 4 out of 6 branches were covered. The number of test cases generated is 1 test case.

```
class IntegerTest() {
    @Test
    fun isPrimeTest() {
        assertEquals(false,Integer.isPrime(-237))
        assertEquals(true,Integer.isPrime(199))
    }
}
```

Figure 5. IntegerTest class from the first experiment.

```
class IntegerTest() {
    @Test
    fun isPrimeTest() {
        assertEquals(false,Integer.isPrime(20))
        assertEquals(true,Integer.isPrime(53))
        assertEquals(false,Integer.isPrime(125))
        assertEquals(false,Integer.isPrime(-52))
    }
}
```

Figure 6. IntegerTest class from the second experiment.

There is a difference in the number of test cases generated from experiments with the number of parents and maximum generation. The results seen in Figure 5 and 6 are in the IntegerTest class in which the first experiment produced 2 test cases and the second experiment produced 4 test cases. In Figure 6, it can be seen that there is duplication which causes the number of test cases generated to be excessive. The test case in Figure 2 is optimal, but this can happen because the mutation process causes changes in the optimal parameter values.

Looking at Table 3 and 5, the resulting branch coverage has a value significantly different from instruction coverage. Since this research uses instruction coverage, it can be concluded that high instruction coverage does not always give high branch coverage.

5. Conclusion and Future Works

The source code derived from the Kotlin programming language has successfully generated the unit tests of this research. This research uses the help of ANTLR4 as a parser to generate ASTs used in the GA process. The resulting unit test is the most optimum based on the fitness value of instruction coverage. The average value of code coverage in the unit test on instruction coverage is 95.64%, branch

coverage is 76.19%, and line coverage is 96.87%. Only two classes generate duplicate test cases, but there is only one duplication in each class. To get maximum results, determining the number of parents and maximum generation has affected the resulting unit test.

This research still uses instruction coverage as a fitness value, and there is still duplication in the results. Implementation of a combination of another type of code coverage can be used as a fitness value to get better results. Furthermore, the source code used is based on Kotlin language with limitations for the data type of parameters and return value. The data types that can be used are limited to integer, double, float, boolean, and string data types. In addition, this research can be translated into Kotlin language to run the expected value retrieval and code coverage faster. Optimization in code can also be done in the future by improving better duplication checks.

References

[1] L. Ardito, R. Coppola, G. Malnati, and M. Torchiano, "Effectiveness of kotlin vs. java in android app development tasks," *Information and Software Technology*, vol. 127, p. 106374, 2020.

[2] B. Tresnayatna, S. Widowati, and I. L. Hakim, "Pembangkit test case untuk pengujian perangkat lunak menggunakan metode basis path," *eProceedings of Engineering*, vol. 6, no. 1, 2019.

[3] S. Lukasczyk, F. Kroiß, and G. Fraser, "Automated unit test generation for python," in *Search-Based Software Engineering: 12th International Symposium, SSBSE 2020, Bari, Italy, October 7–8, 2020, Proceedings 12*. Springer, 2020, pp. 9–24.

[4] G. Fraser, "A tutorial on using and extending the evosuite search-based test generator," in *Search-Based Software Engineering: 10th International Symposium, SSBSE 2018, Montpellier, France, September 8-9, 2018, Proceedings 10*. Springer, 2018, pp. 106–130.

[5] J. Carr, "An introduction to genetic algorithms," *Senior Project*, vol. 1, no. 40, p. 7, 2014.

[6] P. Liashchynskyi and P. Liashchynskyi, "Grid search, random search, genetic algorithm: a big comparison for nas," *arXiv preprint arXiv:1912.06059*, 2019.

[7] A. Alhroob, W. Alzyadat, A. T. Imam, and G. M. Jaradat, "The genetic algorithm and binary search technique in the program path coverage for improving software testing using big data." *Intelligent Automation & Soft Computing*, vol. 26, no. 4, 2020.

[8] G. Candea and P. Godefroid, "Automated software test generation: some challenges, solutions, and recent advances," *Computing and Software Science: State of the Art and Perspectives*, pp. 505–531, 2019.

[9] M. Olsthoorn, D. Stallenberg, A. Van Deursen, and A. Panichella, "Syntest-solidity: automated test case generation and fuzzing for smart contracts," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 202–206.

[10] J. Zhang, X. Wang, H. Zhang, H. Sun, K. Wang, and X. Liu, "A novel neural source code representation based on abstract syntax tree," in *2019 IEEE/ACM 41st International Conference on Software Engineering (ICSE)*. IEEE, 2019, pp. 783–794.

[11] Z. Chang, Y. Sun, T.-Y. Wu, and M. Guizani, "Scratch analysis tool (sat): a modern scratch project analysis tool based on antlr to assess computational thinking skills," in *2018 14th International Wireless Communications & Mobile Computing Conference (IWCMC)*. IEEE, 2018, pp. 950–955.

[12] W. Zhu, N. Yoshida, T. Kamiya, E. Choi, and H. Takada, "Msccd: grammar pluggable clone detection based on antlr parser generation," in *Proceedings of the 30th IEEE/ACM International Conference on Program Comprehension*, 2022, pp. 460–470.

[13] S. Vogl, S. Schweikl, G. Fraser, A. Arcuri, J. Campos, and A. Panichella, "Evosuite at the sbst 2021 tool competition," in *2021 IEEE/ACM 14th International Workshop on Search-Based Software Testing (SBST)*. IEEE, 2021, pp. 28–29.

[14] S. Lukasczyk and G. Fraser, "Pynguin: Automated unit test generation for python," in *Proceedings of the ACM/IEEE 44th International Conference on Software Engineering: Companion Proceedings*, 2022, pp. 168–172.

[15] S. Mirjalili, J. Song Dong, A. S. Sadiq, and H. Faris, "Genetic algorithm: Theory, literature review, and application in image reconstruction," *Nature-Inspired Optimizers: Theories, Literature Reviews and Applications*, pp. 69–85, 2020.

[16] G. Fraser and A. Arcuri, "Evosuite: automatic test suite generation for object-oriented software," in *Proceedings of the 19th ACM SIGSOFT symposium and the 13th European con-*

ference on Foundations of software engineering, 2011, pp. 416–419.

[17] S. D. Immanuel and U. K. Chakraborty, "Genetic algorithm: an approach on optimization," in *2019 international conference on communication and electronics systems (ICCES)*. IEEE, 2019, pp. 701–708.

[18] F. Dharma, S. Shabrina, A. Noviana, M. Tahir, N. Hendrastuty, and W. Wahyono, "Prediction of indonesian inflation rate using regression model based on genetic algorithms," *Jurnal Online Informatika*, vol. 5, no. 1, pp. 45–52, 2020.

[19] I. Bilal, I. Al-Taharwa, S. Rami, I. M. Alkhawaldeh, and N. Ghatasheh, "Jacoco-coverage based statistical approach for ranking and selecting key classes in object-oriented software," *J. Eng. Sci. Technol*, vol. 16, pp. 3358–3386, 2021.

[20] A. Pilgun, "Instruction coverage for android app testing and tuning," Ph.D. dissertation, University of Luxembourg, Esch-sur-Alzette, Luxembourg, 2020.