Coevolution of Second-order-mutant

Mohamad Syafri Tuloli, Benhard Sitohang, Bayu Hendradjaya
Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, Indonesia

Article Info

Article history:

Received Dec 20, 2017
Revised Feb 1, 2018
Accepted Aug 18, 2018

Keyword:

Mutation analysis
Mutation testing
Software engineering
Software testing

ABSTRACT

One of the obstacles that hinder the usage of mutation testing is its impracticality, two main contributors of this are a large number of mutants and a large number of test cases involves in the process. Researcher usually tries to address this problem by optimizing the mutants and the test case separately. In this research, we try to tackle both of optimizing mutant and optimizing test-case simultaneously using a coevolution optimization method. The coevolution optimization method is chosen for the mutation testing problem because the method works by optimizing multiple collections (population) of a solution. This research found that coevolution is better suited for multi-problem optimization than other single population methods (i.e. Genetic Algorithm), we also propose new indicator to determine the optimal coevolution cycle. The experiment is done to the artificial case, laboratory, and also a real case.

*Copyright © 2018 Institute of Advanced Engineering and Science.
All rights reserved.*

Corresponding Author:

Mohamad Syafri Tuloli,
Sekolah Teknik Elektro dan Informatika,
Institut Teknologi Bandung,
Jl. Ganesha No.10, Lb. Siliwangi, Bandung, Kota Bandung, Jawa Barat 40132, Indonesia.
Email: Syafri_tuloli@students.itb.ac.id

1. INTRODUCTION

Testing phase usually becomes the most expensive phase of a software engineering process and can take up to 40-50% of development effort [1]. Most of the testing method is done by executing a test case to the program under test (PUT), to detect an error that still resides in the program. The testing result highly depends on the qualities of the test case, this makes important to improve test case qualities. Mutation Testing is one of the methods to evaluate and improve test cases. Mutation testing works by creating a variance of a program, this program variance (called mutant) is created by seeding an artificial fault to the original program. This fault is a simple fault that mimics faults usually done by programmers. The mutation testing has a caveat, that it involves generating a huge number of mutants and executing test case to the mutants.

To optimize the mutation testing process, there is already a lot of research using a meta-heuristic approach, mostly used a genetic algorithm method [2], and this meta-heuristics approach already gives a promising result [3]. Coevolution is one of the meta-heuristics that involves an evolution of multiple solution collections simultaneously, this trait is suitable for the mutation testing problem which consists of interaction between mutant problem and test case problem. Some research is already tapping on this potential [4], [5], but we think there is still a lot of exploration need to be done. In this research, we implemented a coevolution method to JMetal library, a java based meta-heuristics library created by Durillo, et.al [6]. We also improved our previous mutant generator [7], [8] by adding a test case generator functionalities. We combined these three functionalities to create a coevolution based mutation testing.

1.1. Mutation Testing (Problem)

Mutation testing used in many phases of software engineering, it has been used from a simple problem to a more complex one. Mutation testing works by generates a variation of a program called mutant, and execute a test case to the mutant. The mutation generates three categories of mutant. An easily-detected mutant is a mutant can be detected by an existing test case (test case before processed by mutation testing), we want to reduce this kind of mutant. A hard-to-detect-mutant, this mutant that cannot be detected by existing test case, but with further improvement of the test case, it can be detected, we want to increase this kind of mutant. Lastly, an equivalent mutant is a mutant that cannot be detected by any test case, this is because this kind of mutants are syntactically different from the original program, but are semantically same from the original program.

The problem with mutation testing is the process generates a huge number of mutants, and these mutants are to be executed to a huge number of test cases. This is why we need an optimization to reduce the mutants and the test cases involved in mutation testing without reducing its overall quality, test case qualities in detecting mutant, and mutant qualities in avoiding detection.

An equivalent mutant opens a separate branch of research in mutation testing because usually need a manual effort for detecting this mutant. In this research, we assume that equivalent mutant is a mutant that could not be detected by a complete list of test case from our benchmark test case.

Mutation testing mainly used to mutate source code or programming language feature [9], but mutation testing is also capable in testing software design [10], to mutate software model [11], [12], or even used as a tool for teaching software testing [13]. The research about mutation testing has been developed to expand its practice to a more complex application [14][15] or specific platform (i.e Android [16], [17] and web application [18]). The recent research is spread into many directions, from improving mutation operator [17], [19], improving mutant selection [20], improving [21] or creating [22] mutation testing tool, into implementing search-based optimization method (i.e. coevolution [4], [5]). In this paper, we use the most used search-based method [2] (i.e. Genetic Algorithm) that implemented in the work of Durillo et.al., [6] to create a coevolution optimization method.

Original Program	First Order Mutant	Second Order Mutant
…	…	…
while (hi < 50){	while (hi > 50){	while (hi > 50){
System.out.print(hi);	System.out.print(hi);	System.out.print(hi);
hi = lo + hi;	hi = lo + hi;	hi = lo * hi;
lo = hi – lo;	lo = hi – lo;	lo = hi – lo;
}	}	}
…	…	…

Figure 1. Example of First Order Mutant and Higher Order Mutant (Second Order Mutant), adapted from Nguyen and Madeyski [23]

1.2. Coevolution Optimization (proposed solution)

Coevolution method is one of the meta-heuristics optimization methods, that unlike a single population optimization method (i.e Genetic Algorithm), uses a multi-population in its process. These populations are evolved and evaluated by evaluation function named fitness function. In a single population method, the fitness function usually measures the quality of each individual (from the population) compared to other individuals in the same population. Fitness function in the coevolution method measures each individual compared to individuals in the other population. The evaluation formulation depends on the nature of the relation between the two/more population, it may be in competitive relation or cooperative relation.

Black-box optimizations usually suffer a no-free-lunch theorem that states: all optimization method are equal in average given a wide variety of cases. The coevolution optimization, however, gives a free-lunch [24]-[26], thus it may diminish the necessity to use a problem-specific information to be exploited in the optimization process. Another interesting point about coevolution optimization is in how it works by using a cooperation of multiple populations of solution. This makes it a good match to a problem with solution that consists of multiple separate but interdependent parts. Coevolution generally implemented as cooperative [27] or competitive, depends on the problem. A mutation testing problem is one of the problem that consists of multiple and contradicting goals, in one side it tries to have a mutant (program variation) that are hard to detect, but in another side, it tries to have a test case with high mutant detection ability. This makes mutation testing a potentially good match to use with a coevolution optimization method.

1.3. Related Works

If Assis, et al [4] offer alternatives of coevolution solution representation and mutation operator, here we offer a new way to implement coevolution by using Genetic Algorithm optimization library as a foundation and using a regex based mutant generator, we also proved it by using in a real case. Assis, et al implement an evaluation mechanism that only uses the best solution from each opponent population to be used in the fitness, here we use an evaluation that all solution contributes to the evaluation of the opponent population.

Adamopolous [5] implements coevolution mutation testing using a pre-generated mutant and test case pool, we think it may give a short-term benefit (i.e. faster search), but will sacrifice opportunity of getting a higher solution quality, and also are impractical to use. We instead implemented coevolution mutation testing using a mutant and test case generator, with this we hope both it can help research by widening the search space, and also later with further improvement, can be used in more practical context, because it does not require a developer to create mutant/testcase beforehand.

1.4. Contribution and Originality

This research prove the potential advantage of using coevolution on the inter-related competing problem instead of using single population optimization method, and also shows its caveat (overgrown sides). This research also contributes in proposing and implementing a coevolution optimization method into mutation testing problem, that different from other implementation: use population evalution instead of best individual [4], and use an mutant and test case generator instead of pre-generated mutant and testcase pool [5]. The population based fitness evaluation makes the evolution direction move toward an overall population improvement, and an automatic generated mutant and test case makes the resulted system to have a practical usage in industri and research.

The coevolution method is validated by executing it into an artificial case, and shows the coevolution is works as expected. The coevolution result is validated by comparing and executing it into an actual mutant and actual testcase (benchmark testcase), and shows that optimization can improve the mutant and testcase qualities, without having an overgrown sides. This research also the first to use a real world case on coevolution method in mutation testing.

2. EXPERIMENT DESIGN (METHOD)

The experiment consist of a couple experiment:

2.1. Implementing Coevolution

The coevolution method is implemented to the existing implementation of NSGAII (Non-dominated Sorting Genetic Algorithm) method in the JMetal library. In the JMetal library, it is already defined a Genetic Algorithm optimization method, we improve this implementation by modifying its evolution process (i.e. fitness evaluation and other processes) to be able to handle a multiple population processing.

The coevolution method consists of optimizing multiple collections (population) of solution concurrently. In the mutation testing, the potential solution can be divided into test case solution and mutant solution. This characteristic of mutation testing problem makes it ideal to use a coevolution optimization method, which consists of two solution population, a test case solution population and a mutant solution population as shown in Figure 2.

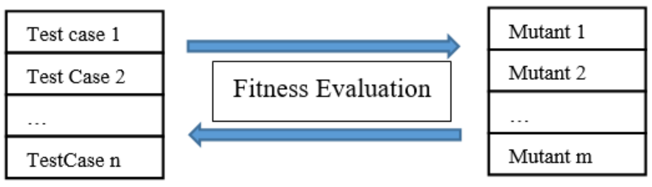


Figure 2. Coevolution optimization method in mutation testing

The coevolution method treat problem as a collection of solution candidate (called individual), this solution representation is in a form that can be processed in evolution. In this paper, the test case representation for first-order-mutant is a pair of test data value, so the representation may vary, depends on the case being used. As for mutant representation, is a pair of mutant operator code, sub-mutant code, and line number where the mutation operator executed. A mutation operators used in this paper is based on the definition of mutation operator by King and Offut [28] and at the time is already implemented six mutation operator [8], but for comparison with Polo works [29] that used here as a benchmark, then we only used four categories of mutation operator as shown in Table 4.

This form of FOM (First-Order-Mutant) representation make the mutation process have a possibility to generate an undefined sub-code mutant, this is because each of the mutation operators has a different number of sub-code size. For example in Table 1, it shows a variation of mutant that generated by using LCR (Relation Operator Replacement), from the original code we can have sub-code mutant 1, 2, to n. So the sub-code mutant range will differ from one mutation operator to another. This is the reason that we pick a different form of SOM (Second-Order-Mutant) representation. For second-order-mutant representation, we use a pair of an index from a pre-generated FOM (First-Order Mutant). This pre-generated FOM was a FOM that already cleaned from an undefined sub-code mutant, this makes the process to become faster since the process does not have to deal with undefined sub-code mutant FOM.

2.2. Experiment Using Artificial Cases

Before implementing the coevolution method to the mutation testing problem, firstly, the coevolution method must be tested separately, this is to isolate a method-implementation error from case related error. There is two couple of artificial problem used in this research, BeSame-BeDifferent and BeCloser-BeDistant.

a. BeSame-BeDifferent

BeSame-BeDifferent is an artificial case consist of two population with a contradicting goal, BeSame goal is to maximize the number of gen (part of solution/individual) that equal with the gen from the rival population, while BeDifferent goal is to maximize the number of gen that differs from the rival population. With an equal (a, b) function return 1 if a=b, and 0 if otherwise. Fitness value of an individual (in BeSame case) BS against PBD population, with n gen per individual, and m individual per population is:

$$Fitness(BS, PBD) = \sum_{i=1}^{n} \sum_{j=1}^{m} equal(BS_i, PBD_{ij}) \quad (1)$$

Fitness value of an individual (in BeDifferent case) BD against PBS population, with n gen per individual, and m individual per population is:

$$Fitness(BD, PBS) = \sum_{i=1}^{n} -1 \times \sum_{j=1}^{m} equal(BD_i, PBS_{ij} \quad (2)$$

First, we compared the artificial cases with using a coevolutionary method and genetic algorithm, this due to genetic algorithm still a most used in test case and mutant generation [2].

b. BeCloser-BeDistant

While the character of BeSame-Bedifferent problem to be discrete (either same or different), we design BeCloser-BeDistant problem to be more continuous, we hope it can show more information about the coevolution process. BeCloser goal is to get the mean value of its population to be closer to the mean value of the opposing population. BeDistant goal is to get a mean value of its population to be more distant from the mean value of the opposing population. Fitness value of an individual (in BeCloser case) BC with n number of gen, against pBD (BeDistant) population with m total gen in population:

$$Fitness(BC, pBD) = -1 \times abs\left(\frac{\sum_{i=1}^{n} BC_i}{n} - \frac{\sum_{j=1}^{m} pBD_j}{m}\right) \quad (3)$$

Fitness of an individual (in BeDistant case) BD, with n number of gen, against pBC (BeCsoler) population with m total gen in population:

$$Fitness(BD, pBC) = abs\left(\frac{\sum_{i=1}^{n} BD_i}{n} - \frac{\sum_{j=1}^{m} pBC_j}{m}\right) \quad (4)$$

The experiment result for artificial cases was measured by:

a. Max Fitness: fitness value of the best individual found throughout the evolution process.
b. All Generation Average Fitness (AGA Fitness): Fitness average of all individual throughout all generation.
c. All Generation Changed Only (AGCO Fitness): Fitness average of all individual only when the fitness fluctuate (increase/decrease) from the previous generation.

2.3. Measuring the Coevolution in Laboratory Cases

To make this experiment easier to compare with other research, the experiment was done to the most used benchmarking case [30], which is Bisect, BubbleSort, Find, FourBall Mid, and TriType as shown in Table 3. We execute a coevolution with parameters shown in Table 2, test case fitness function from Ghiduk [31], and mutant fitness function from Delgado [32].

The coevolution process consists of general genetic-algorithm based evolution, with the exception of the use of multiple populations that processed concurrently. These populations affect each other in the evaluation phase, the evaluation involves a measurement of each solution in population, against all of the solutions in the rival population. The test case population is measured by an ordinary mutation testing score, the fitness of an individual I against the mutant population S is:

$$Fitness(I,S) = \frac{Number\ of\ Mutant\ detected\ by\ I\ in\ S}{Total\ of\ Mutant\ in\ S} \tag{5}$$

The mutant is measured by fitness function below, this fitness function formulation is taken from [32], for a population of mutant size M, and a population of Test Case size T, the fitness of an individual I against to test case population S is, with m_{ij} is mutant j detected by test case i :

$$Fitness(I,S) = M\ x\ T - \sum_{j=1}^{T}\left(m_{Ij}\ \times \sum_{i=1}^{M} m_{ij}\right) \tag{6}$$

We assume that these fitness function will not reflect the real effectiveness because measurement of one side (test case or mutant) will depend on the current (generation) quality of its rival. The question is, does this will impact the quality of the resulted test case/mutant. To answer this, we measure a sample of generation 1, 50, and 100 of the solution to the actual mutant that generated from second-order mutant generation algorithm (for test case), and to the benchmark test case (for mutant).

Table 1. Example Mutant Variation from Using One Mutation Operator to One Line of Code

Original Program	Mutation Operator LCR #1	Mutation Operator LCR #2	…	Mutation Operator LCR #n

…
while (hi < 50)
{System.out.print(hi);
hi = lo + hi;
lo = hi – lo;}
…
	…
	while (hi > 50)
	{System.out.print(hi);
	hi = lo + hi;
	lo = hi – lo;}
	…
		…
		while (hi >= 50)
		{System.out.print(hi);
		hi = lo + hi;
		lo = hi – lo;}
		…
			….	…
				while (hi == 50)
				{System.out.print(hi);
				hi = lo + hi;
				lo = hi – lo;}
				…

Table 2. Coevolution Parameter

Parameter	Default Value
Algorithm	Coevolution
Population	2 (Test Case and Mutant)
Test Data Limit	-10 to 10
Objective	1
Constraint	0
Variable per Solution	Depend on case
Population Size	10
Max Evaluation	1000
Crossover Probability	0.9
Crossover Method	singlepointcrossover
Mutation Probability	1/total variable size
Mutation Method	BitFlipmutation
Selection Method	bynarytournament

3. RESULTS AND ANALYSIS

The result shows that:

a. The experiment using artificial case shows that there could be a possibility a growth imbalance in the coevolution, this can make one population to overgrowth opponent population and make the overgrowth population stagnate most of the time.
b. The artificial case experiment also shows that coevolution optimization method is better than a single population evolution (i.e. Genetic Algorithm) to apply on a problem that consists of interconnected

solution (i.e. BeSame-BeDifferent and BeDistant-BeCloser), this creates a hypothesis that coevolution will also work on mutation testing problem.

c. The AGCO and max fitness is a better indicator than the AGA fitness for determining the optimal generation cycle.
d. The coevolution method used in the mutation testing shows a rather balanced growth, and can improve the quality of mutant (undetected mutant) and test case (detected mutant).
e. The coevolution mutant and test case qualities shows to be related to redundancy level in the population, this can be used as further improvement in the coevolution method.

3.1. Coevolution Using Artificial Case Result

The two experiment with artificial cases shows the importance of having a good problem formulation when using a coevolution optimization method. The BeDifferent problem (in BeSame-BeDifferent) is overgrowth, this is caused by its problem formulation. BeDifferent goal is to maximize number of values that do not exist in its rival population, this makes its fitness formulation (1) becomes too easy to reach its peak because simply any random number has a high probability to get a different value than the rival population values, and thus easily reach its goal. This reflects in BeDifferent max fitness and AGA fitness that grow faster than its rival population as shown in Figure 1.

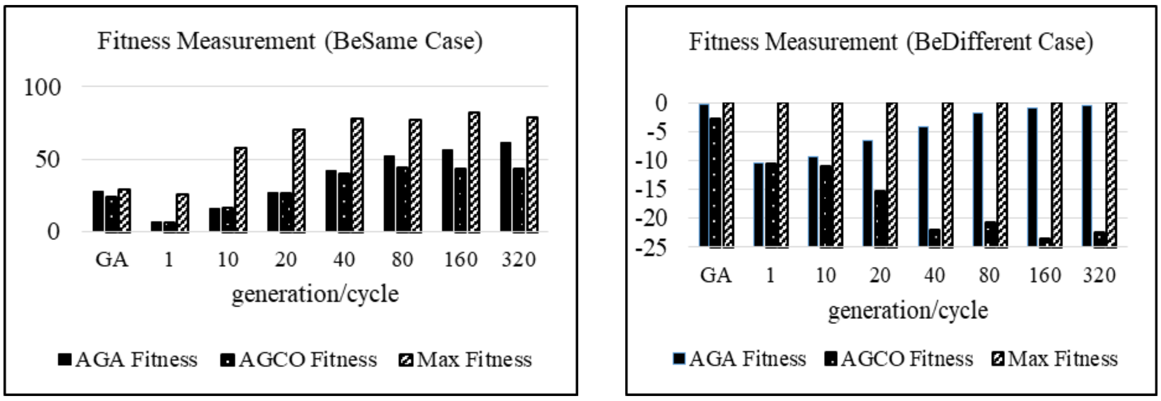


Figure 1. Fitness measurement in different cycle (besame-bedifferent case)

In the BeCloser-BeDistant case, overgrowth is reduced by using a fitness formulation that depends on the mean fitness of the population (instead of fitness accumulation). The effect is reflected in the steady improvement of fitness value for both of the population as shown in Figure 2.

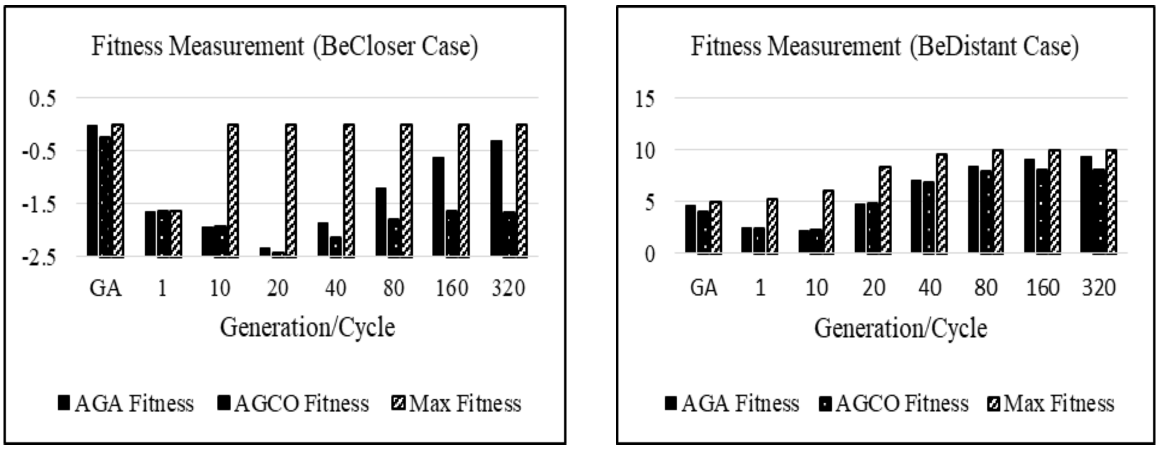
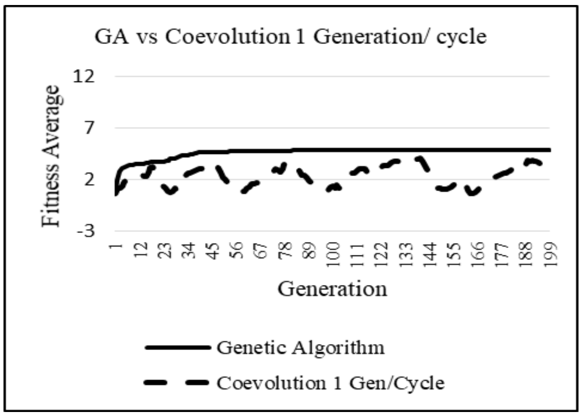


Figure 2. Fitness measurement in different cycle (becloser-bedistant case)



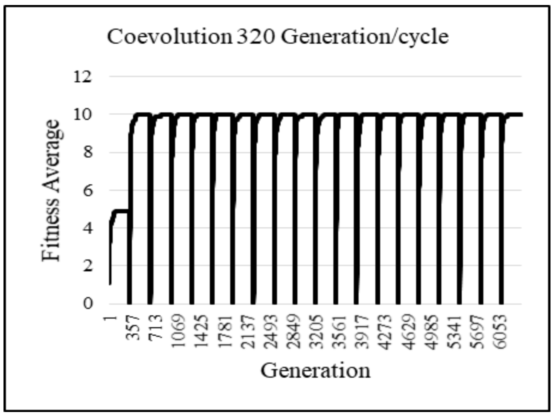


Figure 3. Fitness average comparison in bedistant problem using genetic algorithm and coevolution 1 and 320 generation/cycle

The Artificial case shows the effectiveness of coevolution method, as shown in Figire 3. In the first cycle of evolution, the population always have a lower growth rate in fitness level, this is as expected, because the first population is generated randomly. The second cycle growth is higher, this is because both of the population start to optimize away from the random starting point. The figure also shows the advantage of using coevolution, from single population optimization (i.e. genetic algorithm). The Figure 3.a shows that, for BeDistant case, the genetic algorithm can only reach close to 5 points on fitness average, while the coevolution in higher cycle (320 generation/cycle) can reach close to 10 points of fitness average Figure 3.b. This advantage is because, in the coevolution, evolution happens both in population (solution collection) and its rival population, this creates a dynamic search space that moves toward a better solution. This is not the case in the single population evolution (i.e. genetic algorithm) because the search space is static, the evolution is limited to the optimal value of the current search space.

Another finding is that the coevolution need to find an optimal cycle, because a shorter cycle wouldn't give the population to improve as shown in Figure 3.a, while a longer cycle will hit a stagnate phase (no improvement in evolution) longer, this stagnate phase is shown as a flat peak in the graph (e.g. flat peak in Figure 3.b). To find an optimal cycle, experiment was done with the AGA and AGCO indicator. The Figure 1 and Figure 2 shows that AGA Fitness is always increasing in accord with generation/cycle, this happens because a longer cycle creates a longer stagnate phase, and makes AGA could not be used as an indicator. The stagnation phase (flat peak in fitness average) start to appear in the 80 generation/cycle and above, the longer the cycle, the more this flat peak occur, for example in 320 generation/cycle the flat peak shows even more obvious as shown in Figure 1.b. An alternative indicator AGCO fitness is better from AGA fitness, this because AGCO does not affect by cycle duration, but only affected by the existence of fluctuation in the average fitness. This makes it does not susceptible to the effect of stagnating phase, and become a good candidate for optimum cycle indicator.

3.2. Coevolution Using Laboratory and Real World Cases

The coevolution evaluation fitness evaluation is only a relative quality of the solution (test case and mutant), to have a more accurate evaluation it needs to use an actual mutant (mutant resulted from second-order mutant generation) and the actual test case (benchmark test cases). The evaluation needs to represent broad space of test case and mutant probabilities but with a small possible size. For an actual mutant representation, we choose three second-order mutant generation algorithm that adapted from [29]: LastToFirst, DiffOp, and RandomMix. These second order mutant are frequently used in mutation testing research [33].

In this research these three algorithms compared to all possible second-order mutant (AllCombination) as a benchmark, to see if these three algorithms were adequate as a second-order mutant representation in our chosen cases. As shown in Figure 4 and Figure 5, for all of the laboratory case, one or more of the three second-order mutant generation algorithm (LastToFirst, DiffOp, and RandomMix) have a higher ratio than using AllCombination algorithm, and this applies both to Detected Mutant as shown in Figure 4 and also in Undetected Mutant category as shown in Figure 5. Furthermore, these three algorithms reach a higher detected and undetected mutant ratio than the benchmark while maintaining a much lower size of total mutant, down to 0.2%-0.6% compared to the total mutant from AllCombination.

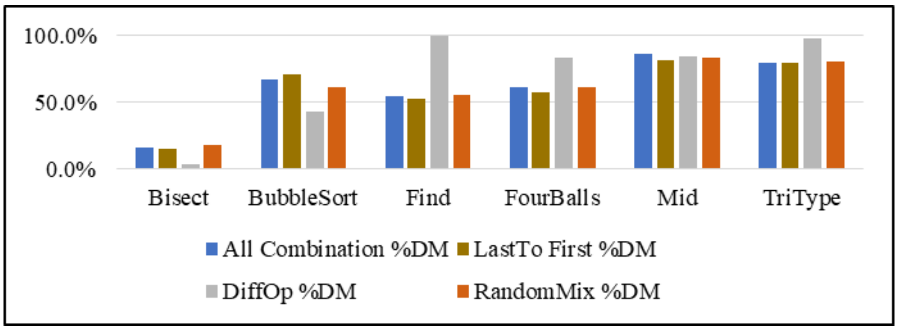


Figure 4. Detected mutant ratio, from mutant generated using second order mutant generation algorithm

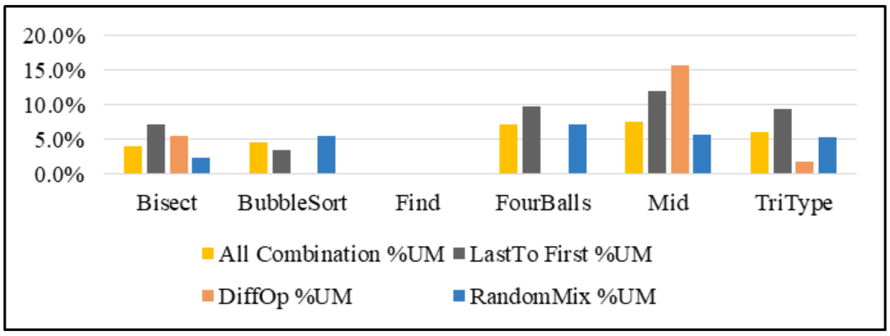


Figure 5. UnDetected-mutant ratio from mutant generated using second order mutant generation algorithm

The result of using coevolution to the mutation testing case is shown in Table 5. In this research we only take three samples from coevolution generation, the first generation (TC 1 and MT 1), the last generation (TC 100 and MT 100), this is to measure the improvement of the solution. Other samples are in the middle of the coevolution (TC 50 and MT 50), this is to measure if the improvement happens gradually or instantly. For TestCase solution as shown in Table 5, comparison is done by comparing the TC in the sample generation (TC 1, TC 50, and TC 100), to the sample Mutant generation (MT 1, MT 50, and MT 100) that shown in column (1), and also compared to the actual mutant generation that shown in column (2), (3), and (4). Comparison to the sample mutant is to evaluate the coevolution process, while the comparison to the actual mutant is to evaluate the actual quality of the test case solution. For the Mutant solution as shown in Table 5. column 1), a comparison is also done to measure both the coevolution process and the actual quality. This is done by executing the mutant sample (MT-1, MT-50, MT-100) to the Test Case sample (TC-1, TC-50, TC-100) and also to the Benchmark Test Case (BM TC).

To simplify the analysis, we categorize the result as shown in Table 5 into four categories, with each category is consist of multiple comparisons. The categories are: increase (increase steadily), decrease (consistent decrease), fluctuate (increase followed by a decrease, vice versa), mixed (increase in one comparison, paired with a decrease in another). Our previous experiment with artificial cases points out that there is a possibility for a coevolution method, to have a dominating side, and this may impair the development of the opposite sides, thus impair the overall progress of coevolution optimization. The coevolution implementation used in this research is a competition relation, so the improvement in one side (test case or mutant) will degrade the other, so the expected result is a rather balance improvement in all of the case, with no dominating side.

The result shows that, for the coevolution method evaluation, test case solution shows improvement in the quality of test case in Bisect and Find; stagnation in BubbleSort, FourBalls, HSLColor cases; and decline quality in Mid and TriType cases. For the mutant solution shows that increase in Mid, and Mixed result in TriType. This shows the coevolution was able to improve both of the solution (test case and mutant) quality on most of the cases and seems to have a rather balanced growth on both sides. For the comparison with actual mutant, the test case has an improvement in Bisect and Find; a fluctuation in BubbleSort and Tritype; stagnate in Fourball; decrease in Mid and HSLColor. While the comparison with actual/benchmark test case results in improvement in Mid and Tritype; decrease in Bisect, BubbleSort, Find, and FourBalls. All this suggest that the proposed coevolution method is able to balance the growth in both of the mutant and test case evolution.

The potential issue in the proposed method is redundancy, this happens because we instill no redundancy mechanism in the method, but since redundancy is a common issue in testing [34], we later investigate the result the result for redundancy. After further analysis of the test case and mutant solution, we got the redundancy level as shown in Table 6. For the test case redundancy (Table 6 column 4-12), on many of the case, there is a high rate of redundancy up to 100% (10 from 10 total population). In the mutant redundancy measurement, we categorize undetected mutant and uncompiled mutant as a redundant mutant because this kind of mutant has a high chance of being a redundant mutant since these mutant were undetected by all the test case. For the mutant redundancy (Table 6 column 1-2) the detected mutant redundancy is quite low (except for HSLColor case).

As shown in Table 5 and Table 6, it is proved that there is a relation between the redundancy level with the coevolution performance. In the Find case has the lowest test case redundancy level (Table 6, Find case column 4-12), and it also has a good performance in test case growth (Table 5, Find case column 2-4). The relation also shows in mutant solution, the high level of mutant redundancy in BubbleSort, Find and HSL Color (Table 6, column 1-3), correlated with the decrease in mutant growth (Table 5, column 1), while low redundancy level redundancy in Mid case, related with improved growth. This result shows a strong relationship between the redundancy and growth of test case and mutant.

There are many suggestion and further exploration to address the redundancy issues:

a. Increase evolution mutation (alteration of individual solution) probability: this will increase the variance in the solutions but could risk an information lost because of aggressive mutation have a higher probability to alter an already good quality solution;
b. integrate a redundancy penalty in fitness evaluation: this will reduce the chance of redundant mutant to move to the next generation, but it may reduce search-space;
c. add redundancy/similarity as another objective: It needs to implement more complex fitness function (to measure solution similarity), and can risk an increased processing time.

Table 3. First Order Mutant and Second Order Mutant of the Benchmark [29]

Name	LOC	Test Case/ Minimized	First-Order Mutant	Second Higher Order Mutant					
				LastToFirst		DiffOp		RandomMix	
				Mt (num/%)	Eq (num/%)	Mt (num/%)	Eq (num/%)	Mt (num/%)	Eq (num/%)
Bisect	31	25/2	63	32/58	5/15.63	44/69.8	5/11.36	32/50.8	2/6.25
BubCorrecto	54	256/1	82	41/50	0/0	44/53.7	0/0	40/48.8	1/2.5
Find	79	135/1	179	90/50.3	0/0	97/54.2	0/0	89/49.7	0/0
Fourball	47	96/5	212	107/50.4	5/4.67	128/60.4	6/4.68	106/50	7/6.6
Mid	59	125/5	181	91/50.3	8/8.79	110/60.8	4/3.63	91/50.3	7/7.69
Tritype	61	216/17	309	155/50.2	7/4.51	168/54.4	11/6.54	155/50.2	9/5.80

Table 4. Comparison of Mutation Operator Category

Tuloli	Polo [29]	Description
AOR	AORB	Arithmatic Operator Replacement
UOI	AORS	Replace ++ operator to --
	AOIU	Insertion of – (minus) operator
	AOIS	Insertion of ++ and -- operator
	LOI	Insertion of ~ (tilde) operator
LCR	COR	Logical Operator Replacement (&& and ||)
	COI	Insertion of ! operator
ROR	ROR	Relational Operator Replacement (<, <=, >, >=, ==, and !=)
Not Implemented	AODS	Deletion of ++ operator

Table 5. Resume of Test Case and Mutant Evaluation

Case	TestCase	Coevolution	Second Order Algorithm		
		MT 1, 50, 100	LastToFirst	DiffOp	RandomMix
		(1)	(2)	(3)	(4)
Bisect	TC 1,50,100	TC Improve, MT Stagnate	TC Improve	TC Stagnate	TC Improve
	BM TC	MT Decrease	-	-	
Bubble	TC 1,50,100	TC Stagnate, MT Decrease	TC Fluctuate	TC Stagnate	TC Fluctuate
Sort	BM TC	MT Decrease	-	-	-
Find	TC 1,50,100	TC Improve, MT Decrease	TC Improve	TC Improve	TC Improve
	BM TC	MT Decrease	-	-	-
FourBalls	TC 1,50,100	TC Stagnate, MT Decrease	TC Stagnate	TC Stagnate	TC Fluctuate
	BM TC	MT Decrease	-	-	-
Mid	TC 1,50,100	TC Decrease, MT Improve	TC Decrease	TC Decrease	TC Decrease
	BM TC	MT Improve	-	-	-
Tritype	TC 1,50,100	TC Fluctuate, MT Mixed	TC Fluctuate	TC Fluctuate	TC Fluctuate
	BM TC	MT Improve	-	-	-
HSLColor	TC 1,50,100	TC Stagnate, MT Decrease	TC Decrease	TC Decrease	TC Decrease
	BM TC	-	-	-	-

MT-x : Mutant population at generation x
TC-x : Test Case population at generation x
BM TC : BenchMark Test Case

Table 6. Mutant Redundancy and Test Case Redundancy Measurement

Cases	Mutant Redundancy v.s. BenchMark TC			Test Case Redundancy								
				v.s. LastToFirst			v.s. DiffOp			v.s. RandomMix		
	MT 1 (1) DM/UM	MT 50 (2) DM/UM	MT 100 (3) DM/UM	TC 1 (4)	TC 50 (5)	TC 100 (6)	TC 1 (7)	TC 50 (8)	TC 100 (9)	TC 1 (10)	TC 50 (11)	TC 100 (12)
Bisect	20%/0%	0%/0%	0%/0%	0%	100%	80%	90%	70%	100%	100%	100%	100%
BubbleSort	50%/0%	60%/0%	40%/0%	0%	100%	100%	0%	0%	0%	0%	100%	100%
Find	20%/0%	60%/0%	0%/0%	0%	100%	80%	0%	100%	90%	0%	100%	80%
FourBalls	10%/0%	0%/0%	0%/0%	60%	60%	60%	80%	80%	80%	60%	60%	40%
Mid	0%/0%	0%/40%	0%/30%	0%	60%	100%	60%	100%	100%	0%	60%	90%
Tritype	0%/0%	10%/10%	0%/0%	60%	100%	100%	0%	100%	100%	60%	100%	100%
HSLColor	70%/0%	70%/0%	70%/0%	50%	90%	100%	50%	90%	100%	20%	90%	100%

MT-x : Mutant population at generation x
TC-x : Test Case population at generation x
DM/UM : Detected Mutant/ Undetected Mutant

4. THREAT TO VALIDITY

We understand that threat to the external validity of this findings is because we have not yet experimented to a wide variety of real cases. To address that we chose the most used and easily accessed laboratory cases, this is to make it easier for replication and comparison on further research. Another external validity threat is because the use of limited number mutation operator (four of six that supported in the framework), this is because of the need to use the same type of mutation operator with the benchmark case being used.

5. CONCLUSION

The coevolution method is able to show its advantage compared to other optimization methods (e.g. genetic algorithm), for a problem that comprises a competitive subproblem. This research point out the need for a better indicator for determining an optimal coevolution cycle, and suggest the use of an indicator that measures fitness fluctuation. Another thing to consider in coevolution is the possibility of one population to overgrown (dominate) its opponent, the design of coevolution must take this as consideration in designing aspect of coevolution (i.e. fitness definition, evolution parameter, etc).

This research also finds that proposed coevolution method implementation is able to be used in laboratory and real-world mutation testing cases. The proposed method is able to improve the test case and mutant solution quality, and show no sign of growth imbalance to one side of solution population. We also analyze a potential improvement in the method and suggest solutions to address redundancy issues. This implementation resulted in a more practical solution since the mutation system automatically generated test case and mutant, so we hope it makes easier to be used in larger scope after further improvement.

There is still so much thing to improve in the proposed system. The system needs to be tested with a more variety of real case to prove it is practical capability. For this we will search for the real cases that are used in other research, to make it possible to compare with other research findings. For the coevolution method, as suggested here, it will need to implement a redundancy reduction mechanism, but with a minimal processing cost. The coevolution method also needs to be explore, investigate the variety of configurations both to improve its effectiveness and also to reveal a connection between the case characteristic and coevolution method design decision.

REFERENCES

[1] L. Luo, "Software testing techniques," *Inst. Softw. Res. Int. Carnegie mellon Univ. Pittsburgh, PA*, vol. 15232, no. 1–19, p. 19, 2001.

[2] R. A. Silva, R. Senger, P. Sérgio, and L. De Souza, "A systematic review on search based mutation testing," *Inf. Softw. Technol.*, vol. 81, pp. 19–35, 2017.

[3] L. Thi, M. Hanh, N. T. Binh, and K. T. Tung, "Survey on Mutation-based Test Data Generation," *Int. J. Electr. Comput. Eng.*, vol. 5, no. 5, pp. 1164–1173, 2015.

[4] A. Assis, L. De Oliveira, C. G. Camilo-junior, and A. M. R. Vincenzi, "A Coevolutionary Algorithm to Automatic Test Case Selection and Mutant in Mutation Testing," *2013 Congr. Evol. Comput.*, 2013.

[5] K. Adamopoulos, M. Harman, and R. Hierons, "How to overcome the equivalent mutant problem and achieve tailored selective mutation using co-evolution," in *Proceedings of the Genetic and Evolutionary Computation Conference*, 2004, pp. 1338–1349.

[6] J. J. Durillo, A. J. Nebro, F. Luna, E. Alba, and C. De Teatinos, "jMetal: a Java Framework for Developing Multi-Objective Optimization Metaheuristics *Sci." Technol.*, vol. 1, no. March 2016, pp. 1–12, 2006.

[7] M. S. Tuloli, B. Sitohang, and B. Hendradjaya, "On the Implementation of Search-Based Approach to Mutation Testing," in *International Conference on Data and Software Engineering (ICoDSE)*, 2017.

[8] M. S. Tuloli, B. Sitohang, and B. Hendradjaya, "Regex Based Mutation Testing Operator Implementation," in *International Conference on Data and Software Engineering (ICoDSE)*, 2016, no. 1.

[9] R. Gopinath and E. Walkingshaw, "How Good Are Your Types? Using Mutation Analysis to Evaluate the Effectiveness of Type Annotations," *Proc. - 10th IEEE Int. Conf. Softw. Testing, Verif. Valid. Work. ICSTW 2017*, pp. 122–127, 2017.

[10] A. Sullivan, K. Wang, R. N. Zaeem, and S. Khurshid, "Automated Test Generation and Mutation Testing for Alloy," in *10th IEEE International Conference on Software Testing, Verification and Validation Automated*, 2017, no. 2, pp. 264–275.

[11] K. G. Larsen, F. Lorber, B. Nielsen, and U. M. Nyman, "Mutation-Based Test-Case Generation with Ecdar," in *10th IEEE International Conference on Software Testing, Verification and Validation Workshops Mutation-Based*, 2017, pp. 319–328.

[12] B. K. Aichernig, S. Marcovic, and R. Schumi, "Property-Based Testing with External Test-Case Generators," in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017.

[13] B. S. Clegg, M. Rojas, and G. Fraser, "Teaching Software Testing Concepts Using a Mutation Testing Game," in *IEEE/ACM 39th International Conference on Software Engineering: Software Engineering Education and Training Track (ICSE-SEET)*, 2017.

[14] I. Ahmed, R. Gopinath, A. Groce, and P. E. Mckenney, "Applying Mutation Analysis On Kernel Test Suites : An Experience Report," in *IEEE International Conference on Software Testing, Verification and Validation Workshops (ICSTW)*, 2017.

[15] A. Alberto, A. Cavalcanti, M. Gaudel, and A. Sim, "Formal mutation testing for Circus," *Inf. Softw. Technol.*, vol. 81, pp. 131–153, 2017.

[16] L. Deng, J. Offutt, P. Ammann, and N. Mirzaei, "Mutation Operators for Testing Android Apps," *Inf. Softw. Technol.*, vol. 81, pp. 154–168, 2017.

[17] S. Jabbarvand, Reyhaneh; Malek, "An Energy-Aware Mutation Testing Framework for Android," 2017.

[18] U. Praphamontripong, "Testing Web Applications with Mutation Analysis," George Mason University, 2017.

[19] F. Wu, J. Nanavati, M. Harman, Y. Jia, and J. Krinke, "Memory mutation testing," *Inf. Softw. Technol.*, vol. 81, pp. 97–111, 2017.

[20] C. Sun, F. Xue, H. Liu, and X. Zhang, "A path-aware approach to mutant reduction in mutation testing," *Inf. Softw. Technol.*, vol. 0, pp. 1–17, 2016.

[21] T. Laurent, A. Ventresque, M. Papadakis, C. Henard, and Y. Le Traon, "Assessing and Improving the Mutation Testing Practice of PIT," in *IEEE International Conference on Software Testing, Verification and Validation (ICST)*, 2017, pp. 430–435.

[22] A. Parsai, A. Murgia, and S. Demeyer, "LittleDarwin: a Feature-Rich and Extensible Mutation Testing Framework for Large and Complex Java Systems," in *International Conference on Fundamentals of Software Engineering*, 2017, pp. 148–163.

[23] Q. V. Nguyen and L. Madeyski, "Problems of Mutation Testing and Higher Order Mutation Testing," *Adv. Comput. Methods Knowl. Eng.*, pp. 157–172, 2014.

[24] T. C. Service and D. R. Tauritz, "Free lunches in pareto coevolution," *Proc. 11th Annu. Conf. Genet. Evol. Comput. GECCO 09*, pp. 1721–1727, 2009.

[25] D. H. Wolpert and W. G. Macready, "Coevolutionary free lunches," *IEEE Trans. Evol. Comput.*, vol. 9, no. 6, pp.

721–735, 2005.
[26] T. C. Service and D. R. Tauritz, “A No-Free-Lunch Framework for Coevolution,” *Proc. 10th Annu. Conf. Genet. Evol. Comput. - GECCO ’08*, pp. 371–378, 2008.
[27] M. A. Ismail, V. Mezhuyev, K. Moorthy, and S. Kasim, “Optimisation of Biochemical Systems Production using Hybrid of Newton method , Differential Evolution Algorithm and Cooperative Coevolution Algorithm,” *Int. J. Electr. Comput. Eng.*, vol. 8, no. 1, pp. 27–35, 2017.
[28] K. N. King and a J. Offutt, “A fortran language system for mutation-based software testing,” *Softw. Pract. Exp.*, vol. 21, no. 7, pp. 685–718, 1991.
[29] M. Polo, M. Piattini, and I. Garc, “Decreasing the cost of mutation testing with second-order mutants,” *Softw. Testing, Verif. Reliab.*, no. June 2008, pp. 111–131, 2009.
[30] Y. Jia and M. Harman, “An analysis and survey of the development of mutation testing,” *IEEE Trans. Softw. Eng.*, vol. 37, no. 5, pp. 649–678, 2011.
[31] A. S. Ghiduk, “Using Evolutionary Algorithms for Higher-Order Mutation Testing,” *Ijcsi*, vol. 11, no. 2, pp. 93–104, 2014.
[32] P. Delgado-Perez, I. Medina-Bulo, S. Segura, A. Garcia-Dominguez, and J. J. Dominguez-Jimenez, “GiGAn : Evolutionary Mutation Testing for C ++ Object-Oriented Systems,” *32nd ACM SIGAPP Symp. Appl. Comput.*, 2017.
[33] A. S. Ghiduk, “Reducing the Number of Higher-order Mutants with the Aid of Data Flow,” *e-Informatica Softw. Eng. J.*, vol. 10, no. 1, pp. 31–49, 2016.
[34] S. K. Mohapatra and S. Prasad, “Test Case Reduction Using Ant Colony Optimization for Object Oriented Program,” vol. 5, no. 6, pp. 1424–1432, 2015.

BIOGRAPHIES OF AUTHORS

Mohamad Syafri Tuloli, finish a Bachelor Degree in informatics Universitas Islam Indonesia, Yogyakarta in 2005. Finish a Magister in Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung, in 2007. Work as a lecturer in Universitas Negeri Gorontalo (UNG). Currently studying for a doctoral program in Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung.

Prof. Dr. Ing. Benhard Sitohang, is a Professor in Sekolah Teknik Elektro dan Informatika, Institut Teknologi Bandung. Leading researcher and also a promotor for a doctoral student in the in Software and Knowledge data Engineering area. Lead a research related to database, like scalability and performance of data management.

Dr. Bayu Hendradjaya, is a lecturer in Sekolah Teknik Elektro dan Informatika, on software engineering, programming, software quality, and software development security. Have an extensive practical experience in software development. Becomes a promotor for a doctoral student in the software engineering topics.