JAMBURA
JOURNAL OF INFORMATICS
ISSN 2656-467X
http://ejurnal.ung.ac.id/index.php/jji

EXPLORING A BETTER SEARCH –BASED IMPLEMENTATION ON SECOND –ORDER MUTANT GENERATION

Mohamad Syafri Tuloli*, **Benhard Sitohang**, **Bayu Hendradjaya**

Electrical Engineering and Informatics
Institut Teknologi Bandung, Jl. Ganesha 10, Bandung 40132, West Java, Indonesia
** Corresponding author, email: syafri.tuloli@ung.ac.id*

ABSTRAK

Pengujian perangkat lunak adalah bagian dari proses pengembangan perangkat lunak, dengan tujuan utama untuk mengurangi/menghilangkan kesalahan pada perangkat lunak, hal ini umumnya dilakukan dengan menjalankan kasus-uji. Salah satu teknik untuk mengkur dan meningkatkan kualitas dari kasus uji adalah pengujian mutasi, tetapi walaupun sudah terbukti keefektifannya, teknik ini masih memiliki suatu kendala besar, yaitu tidak praktis untuk digunakan karena melibatkan pembangkitan dan eksekusi dari jumlah mutan yang besar. Pada penelitian ini, dilakukan eksplorasi penggunaan optimasi berbasis-pencarian pada pembangkitan mutan (variasi dari program), dengan tujuan untuk menghasilkan mutan yang tidak dapat dideteksi oleh kasus-uji, karena mutan jenis ini memiliki dapat kekurangan dari kasus-uji. Metode usulan dibandingkan dengan algoritma pembangkitan second-order mutant yang umum digunakan, dan juga dibandingkan dengan pendekatan berbasis-pencarian lainnya. Hasil menunjukkan bahwa metode usulan dapat membangkitkan lebih banyak mutan-tidak-terdeteksi (undetected-mutant) daripada dengan metode pembangkitan mutan yang umum. Metode usulan memiliki performansi yang lebih rendah daripada metode pembangkitan berbasis-pencarian benchmark, tetapi performansinya dapat ditingkatkan dengan melakukan perubahan pada representasi solusi, dan dengan adopsi parameter optimasi yang digunakan oleh metode pembanding.

Kata kunci: analisis mutasi, pengujian mutasi, pengujian perangkat lunak berbasis pencarian, rekayasa perangkat lunak berbasis pencarian

ABSTRACT

Software testing is a part of a software development process with a major concern is to reduce/eliminate fault in the software, and mainly done by executing a test case. One of the techniques for measuring and improving test case quality is mutation testing, but despite it is good effectiveness, this technique has a major problem that is impractical because it involves generation and execution of huge amount of mutant. This trend also happens in software testing, with the main focus on optimizing the test case generation. In this research, we explore the used of search-based optimization to the mutant (program variant) generation, with a goal to generate mutants that can escape test case detection, because these mutants have a probability to show test case deficiency. In this research, the proposed method is compared with a general second-order mutant generation algorithm and with other search-based mutant generation. The result shows that the proposed method can generate more undetected-mutant than a general second-order mutant generation. The proposed method performs worse than the benchmark search-based mutant generation, but this performance improved by altering it is solution representation and by the adoption of an optimization parameter.

Keywords: mutation analysis, mutation testing, search-based software testing, search-based software engineering

INTRODUCTION

One of software quality indicator is the numbers of fault in the software. The more fault can be detected (and fixed) the higher the quality of the software. To have a higher fault detection capability, it needs to have a good quality of test case. A test case is a set of data and method/function call sequence executed to software/program under test (**PUT**). In the test case is also included an expected output to be compared with an actual output from the PUT, if the actual output is different from the expected output, it can be concluded that there is a bug in the PUT. Because of the importance of the test case role to guard against software defect, the test case itself needs to be guarded against it is own defect (*custodiet ipsos custode*).

One of the methods to evaluate and improve test-case is mutation testing, it is a method that evaluates test case by the ability to detect a variant of a program (**mutant**). This method assumes that a good test case must be able to detect the artificial fault seeded in the PUT. Mutation testing is a promising method because research has shown that mutants have a strong relationship with an actual fault in PUT (Just, et al., 2014). But this method has it is own caveat, it is not practical since it needs to generate a huge amount of mutant, and this cost a great deal because the mutant must be executed to test case. The mutation testing process needs optimization.

The usage of Search-based optimization on the software engineering is starting to be explored by researchers (Harman, et al., 2012, 2009), and this trend includes in software testing domain (Mcminn, et al., 2004)(Orso & Rothermel, 2014). The advantage of search-based optimization because it is a black-box approach, it only needs a solution representation and an evaluation function to be implemented. In this research, we used search-based optimization to optimize the mutation testing process, explore different solution representation structure, and also compare the result with a general mutant generation algorithm and similar mutant search-based mutant generation approach.

MUTATION TESTING

The mutation testing is a test case evaluation and improving technique that stands on hypothesis DeMillo in (Yue Jia & Harman, 2011):

a. *Competent Programmer Hypotheses* that assumes programmers is a competent individual in developing software, and the fault in the program is only a simple fault that can be fixed with syntactical change.

b. *Coupling Effect*, that assumes that all complex fault is a combination of simple faults, then a test case that can identify these simple fault can identify the complex fault.

Based on these hypotheses, mutation testing generates an artificial fault henceforth called **mutant**, which created by altering one or more lines of code of the PUT (Figure 1). The execution of one mutation operator on one line of code can generate more than one mutant (Figure 1.a), this is because the effectiveness of each mutation operator is varying depending on the source code type on the PUT (Tuloli, et al., 2016).

Mutant can be divided into two categories: first-order mutant (**FOM**) that generated by executing a single mutation operator to it, and higher-order mutant (**HOM**) that generated by executing more than one mutation operator. Figure 1.b illustrates a higher-order mutant created by implemented mutation operator (ROR) twice, this mutant is called Second-Order mutant (**SOM**) because the mutation operator executed twice.

The HOM has more potential to generate an **undetected-mutant**, that is a mutant that cannot be detected by current test case (Y Jia & Harman, 2008). But the problem is the higher order of mutant is, the more the size of mutants increase.

Original Program	First Order Mutant (ROR operator)	First Order Mutant (ROR operator)

…
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;
		}
		…

a. Implementation of mutation operator to one line of code

Original Program	First Order Mutant	Second Order Mutant

…
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;
		}
		…

b. Second-order mutant

Figure 1. Example of first-order mutant and second-order mutant (Nguyen & Madeyski, 2014)

DESIGN

The mutant generation system is based on our previous mutant generation system based on regular expression (Tuloli, et al., 2016), the system is itself has been proven to be able to use in search-based First-Order Mutant generation (FOM) (Tuloli, et al., 2017). In this research, we explored the usage of this system on generating Second-Order Mutant (SOM).

Second-Order Mutant Algorithm

The second order mutant generation algorithm is used as shown in Figure 2. The algorithm main functionality is by implementing first and second mutant operator to line in a sequence. At first, the first-operator is checked to the code-line where the first-operator will be implemented, this is because not all mutation operator can be implemented to a code-line. For instance, an ABS operator (Table 1) cannot be implemented to a code-line where there is none existed arithmetic operator.

If the code-line can be mutated with the first mutation operator, then the second mutant operator is checked to the second code-line (can be same a code-line with the first). The Second-Order mutant is generated only when the second mutation operator can be implemented to the second code-line. The mutation operator itself is implemented using a regular expression (Tuloli et al., 2016).

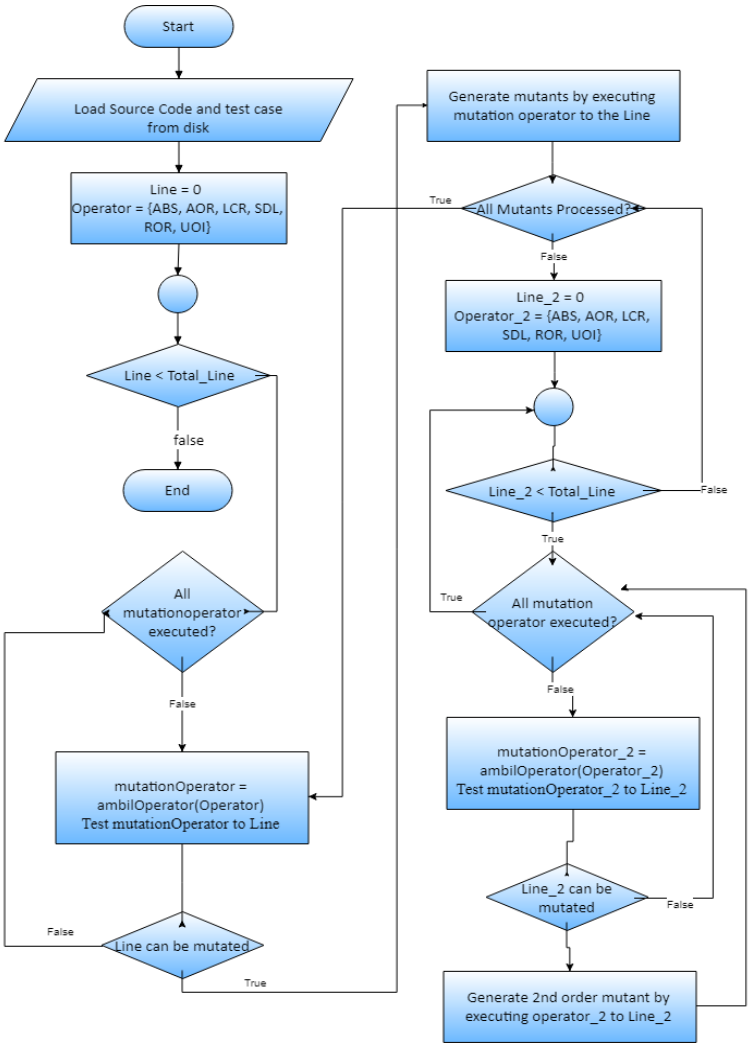

Figure 2. General algorithm to generate Second-Order Mutant

Table 1. Mutation operator used in the experiment

Operator	Operator Name	Description
ABS	ABSolute Value Insertion	Alter expression/sub-expression to add ABS operator (to get its absolute value) and NEGABS (to get absolute value and negate it)
AOR	Arithmetic Operator Replacement	Change the arithmetic operator (x, /, +, -, ^) to with another arithmetic operator.
LCR	Logical Connector Replacement	Change logical operator (equal, not equal, and, or) with another logical operator
ROR	Relational Operator Replacement	Change relation operator (<, >, <=, >=, =, !=) with another relation operator
UOI	Unary Operator Insertion	Insert unary operator (++, --, +, -, !, ~) into expression/sub-expression.
SDL	Statement Deletion	Delete one statement

Mutation Operator

The mutation used in the experiment is six operators which are selected because of its proven effectiveness. The six operators used are five operators from Offutt, et all (1996) (ABS, AOR, LCR, ROR, UOI) and one operator from Deng, et all (2013) (SDL). The mutation operator is

used to BubbleSort case, this case is selected because it is one of the most commonly used cases for mutation testing research (Yue Jia & Harman, 2011).

The Evolution Process

The search-based optimization method implemented is a genetic algorithm optimization method with the chromosome (solution representation) as shown in Figure 3. Each Chromosome is a set of second-order mutant, while each second-order mutant represents as a pair of First-Order Mutant. The first-order mutant represents as a pair of Mutation Operator (i.e. in Table 1), sub-code mutant, and line code (lines of the PUT where the mutation occurred). Because each implementation of one mutation operator on one line of code may generate more than one mutant (Figure 1.a), we need **sub-code mutant** to differentiate the mutants.

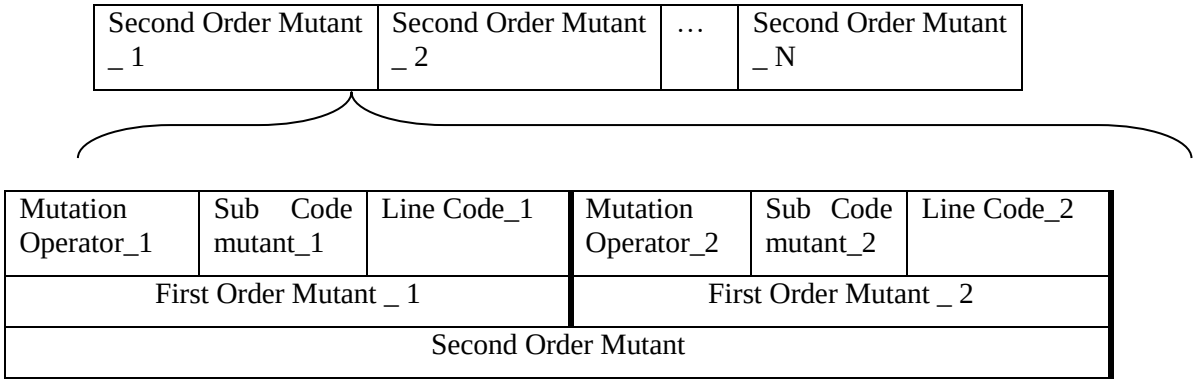

Figure 3. Solution structure algorithm 1

RESULT AND DISCUSSION

Comparison with other Second-Order mutant generation algorithm

The proposed search-based method is compared with second-order mutant generation algorithm: LastToFirst, DiffOp, and RandomMix from Polo et al (Polo, Piattini, & Garc, 2009). The proposed search-based approach is a genetic algorithm with the parameters shown in Table 2. We used the proposed method in three configuration Pop10, Pop100, and Pop200. **Pop10** is used with 10 solutions per population, **Pop100** is 100 solution per population, and **Pop200** is 200 solution per population (Figure 5).

The compared algorithm (LastToFirst, DiffOp, and RandomMix) generates second-order mutant (SOM) by combining a pre-generated first-order mutant (FOM) with a certain arrangement. **LastToFirst** combines FOM according to its index in the FOM-list (Figure 6), so FOM with index 1 combined with last-index FOM, FOM index 2 with FOM index last-1, and so on. **DiffOp** combines FOM but only FOM that generated using a different mutation operator (Table 1), for example, ROR with ABS, ROR with UOI, but never ROR with ROR even when the ROR is used in separates line of code. **RandomMix** combines a randomly selected FOM. All this second-order mutant generating algorithm minimizes total HOM by only use each FOM once for generating SOM.

The comparative evaluation is measured by using average mutant (undetected, detected, etc.), and the number of unique undetected/detected mutant exist in the population. The average undetected/detected mutant can only measures for each solution (chromosome) qualities, while the unique mutant can be used to measures search-diversity of all the mutants in the population (Figure 4). If a population have a high average undetected mutant with a low unique undetected

mutant, it means that the search process has reach convergence, and the search process cannot further improve the solution.

Table 2. Genetic Algorithm parameters

Parameter	Value	Description
PopSize	10, 100, and 200	Population size
MaxEval	100000	Stopping condition, a condition when the evolution stopped.
Variabel	60	Variable in each solution candidate (chromosome/ individual).
Crossover Probabilities	90%	The Probability of one solution candidates to be recombined with other solution candidate.
Mutation Probabilities	10%	Number of gen (variable) that mutated
Undetected	+2	Fitness value for every undetected mutant found in the solution candidates.
Detected	0	A Penalty for fitness value for every detected, non-exist-subcode, unmutated-line-of-code, and redundant mutant exist in the solution candidate
NASubKode	-2	
Unmutated	-2	
Redundant	-2	
SubCodeUpper	12	The Upper limit of the subcode mutant, this set in accord with the case being used.

The result shows that our proposed approach is able to generate more mutant than the general second-order mutant algorithm (Figure 5). The LastToFirst can only generate 3 undetected-mutant, DiffOp cannot generate any undetected-mutant, and RandomMix only generate one undetected-mutant, this is very few compared with total possible of undetected mutant 151 undetected-mutant (Table 3).

Table 3. The result of generating all Second Order-Mutant for BubbleSort case

Measurement	Value
Total First-Order Mutant	89
Total Second-Order Mutant	3916
Undetected-SOM	151
Detected-SOM	3442
Sub-Code-Mutant not exist	10
Unmutated-Line	313
Duration (millisecond)	451130

The population-size parameter proven to significantly affect the resulting undetected-mutant, this effect cannot be detected by measuring average-undetected-mutant but must be measured by unique-undetected-mutant in the population (Figure 5). The average-undetected-mutant cannot detect the difference between pop100 and pop200 performances, this is because the improvement of the population size (from 100 to 200) increase the variety of the undetected-mutant without increasing undetected-mutant/solution ratio, this makes the average-undetected-mutant does not increase (Figure 5). The unique-undetected-mutant on the other hand measure variety of the undetected-mutant, that makes the unique-undetected-mutant are a better indicator in measuring the effect of the population-size parameter.

No	Chromosome/ Solution			
1	E	A	B	D
2	F	D	C	A

For example A, B, C is undetected mutant, then to evaluate the population is:

- Average Undetected-Mutant for the population is population_total_undetected_mutant / total_chromosome = (2+2) / 2 = 2
- Unique Undetected Mutant is 3, which is A, B, and C

Figure 4. Illustration of population evaluation using average-undetected-mutant and unique-undetected-mutant

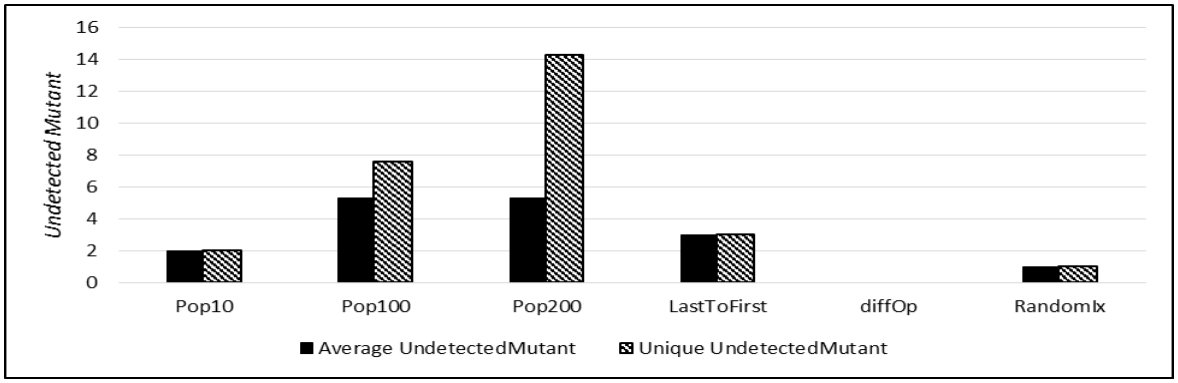

Figure 5. Comparison of average undetected-mutant and unique undetected mutant of proposed algorithm vs other second-order mutant generation algorithms

Comparison of Different Implementation of Proposed Method

The proposed Search-based method used in the previous experiment was using a solution structure that consists of pairs of mutation-position as illustrates in Figure 3. Later we found that this structure can cause:

1. undetected mutant,
2. detected mutant,
3. un-mutated line of code, unmutated by mutation first operator or second operator,
4. non-existing sub-code mutant, for mutation operator 1 or mutation operator 2.

The (1) and (2) condition is expected to happen and is useful to measure the method performances, but condition (3) and (4) can cause a decrease in evolution performance.

To address this problem, we designed an alternative approach, by adopting the second-order mutant approach (LastToFirst, etc) by using a different solution structure as shown in Figure 6. This structure uses an already generated FOM, this limit the search space only to the already generated FOM, and reduced the probability of condition (3) and (4) to emerge. The improvement is proven by the improvement of undetected-mutant and detected-mutant of the algorithm_2 (**Pop10v.2**, **Pop100v.2**, **Pop200v.2**) than our previous algorithm (Pop10, Pop100, Pop200) that shown in Figure 7 and Figure 8. In this experiment also shows once again the effective usage of unique-undetected-mutant indicator in the measurement, the average-undetected-mutant only show some improvement of the algorithm_2 (Figure 7), while the unique-undetected-mutant shows a significant improvement of the algorithm_2 both compared to algorithm_1, also in different population size (Figure 8).

Indices FOM	Mutation Operator	Subcode mutant	Line Numbers
1	UOI	3	14
2	SDL	5	10
…			

a. Pre-generated FOM list

Index FOM_1	Index FOM_2	….

b. Solution Structure

Figure 6. Solution structure for algorithm 2

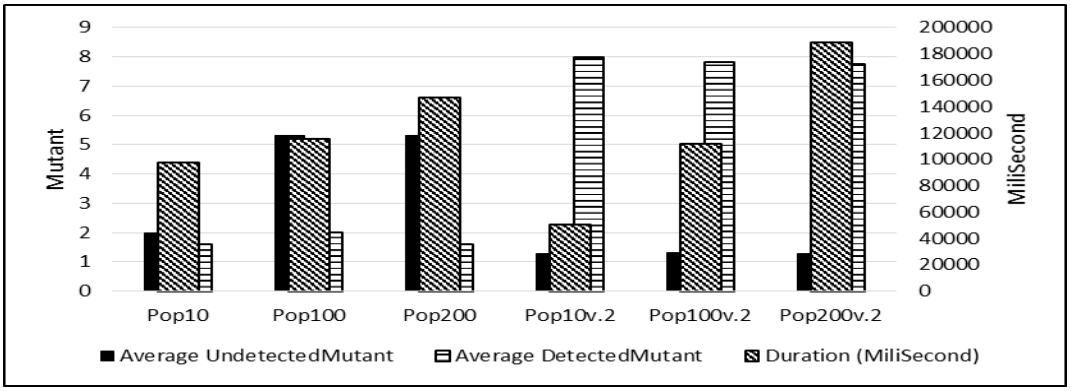

Figure 7. Comparison of average undetected-mutant, average detected mutant, and duration between algorithm 1 and algorithm 2

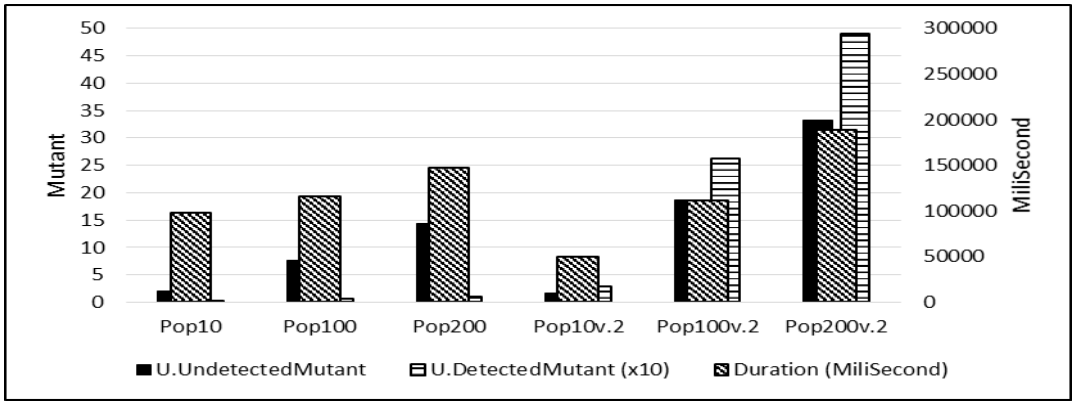

Figure 8. Comparison of unique undetected-mutant, unique detected mutant, and duration between algorithm 1 and algorithm 2

Comparison with Other Evolutionary Mutation Testing

To get a fair comparison, we took a Delgado et al approach (Delgado-Perez, et al., 2017), to compare with our proposed method, from this will be referred to as Delgado method. Delgado defines a strong mutant consist as:

1. potentially equivalent mutant: mutant that undetected from the current/existing test case,
2. difficult-to-kill: mutant that can be detected by only one test case, and this test case does not detect other mutants.

A potentially equivalent mutant can be associated with our undetected-mutant definition, even though there is no guarantee that our undetected mutant is not an equivalent mutant. A difficult-to-kill mutant can be associated with our detected-mutant, but of course, it needs to be analyzed, because the mutant must be exclusively detected by only one test case, and the test-case cannot detect other mutants.

Delgado method and implemented and further combined with using indexed of pre-generated FOM, we named this **DelgadoVersi2**. At first experiment, DelgadoVersi2 was executed normally, but since the running time is much longer than our proposed method (**Pop200v.2**), it gives a rather unfair advantage to the DelgadoVersi2 method. To give a better comparison, we also experiment with a Delgado method but with a limited time that matches the Pop200v.2 duration, we called this **Delgado(Timed).**

Because of the earlier experiment shows Delgado method has better performance, and we suspect one of the factors is the Delgado choice of parameters, we adapt Delgado parameters to the proposed method and named this experiment **Pop200v.2 Param Delgado**. We also compared to all possible combination of FOM, we named the experiment **GenerateAll**.

As shown in Figure 9, the result shows that the DelgadoVersi2 method is able to generate a higher number of undetected mutant that proposed mutant (Pop200v.2) but it needs a longer duration. The duration even longer than the generation of all possible mutant (GenerateAll), but this is as expected because the DelgadoVersi2 method needs overhead processing for the evolution.

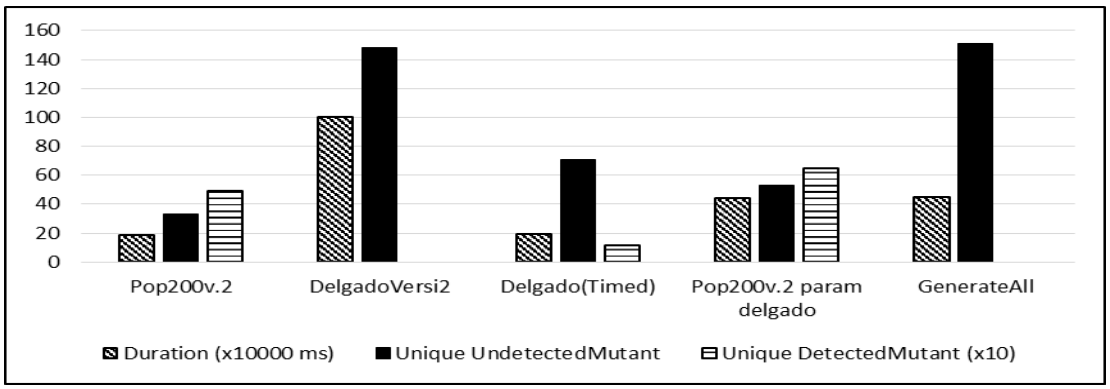

Figure 9. Comparison proposed algorithm with other search-based approaches

In the same duration, the Delgado method (DelgadoTimed) has a better performance than the proposed method (Pop200v.2). One of the determining factors is the parameter used by Delgado method, it is proven by the improvement of the proposed method after adapting the parameter used by Delgado (Pop200v.2 param Delgado). We analyze that the cause of this improvement is because of the parameter in the Delgado method is more emphasized on exploration than exploitation. This shows on its higher value of crossover probability and mutation probability (Table 2).

CONCLUSION

The proposed test case generation method is proven to be able to generate a second-order mutant, both to use by a general second-order generation algorithm (DiffOp, LastToFirst, RandomMix) also can be used by a search-based optimization method. The proposed search-based method can generate more mutant that the general second-order mutant algorithm.

The proposed search-based method performed worse than Delgado search-based method, but this research discovers findings to help improve search-based performances. It shows a better solution representation of the mutants that reduced the existence of uncompiled mutant, this can improve the search by reducing the search-space into an only a valid mutant. Increasing Population size also can improve performances, because it improves solution population diversity. Another finding is about a better indicator to use in measuring performances, we suggest the use of unique-undetected-mutant because it has proven to be able to reflect the diversity of solution that cannot be detected using average-undetected-mutant.

In the future, we will use this finding to create a more efficient mutant generation method, while maintaining a good undetected-mutant ratio (mutation score). There are also many explorations can be made such as search-based method (e.g. Hill Climbing, A*, etc.) and or the parameter/configuration of the method (e.g. selection, crossover, mutation technique in Genetic Algorithm method), or other implementation related optimization.

REFERENSI

Delgado-Perez, P., Medina-Bulo, I., Segura, S., Garcia-Dominguez, A., & Dominguez-Jimenez, J. J. (2017). GiGAn : Evolutionary Mutation Testing for C ++ Object-Oriented Systems. *32nd ACM SIGAPP Symposium On Applied Computing*.

Deng, L., Offutt, J., & Li, N. (2013). Empirical evaluation of the statement deletion mutation operator. *Proceedings - IEEE 6th International Conference on Software Testing, Verification and Validation, ICST 2013*, 84–93. https://doi.org/10.1109/ICST.2013.20

Harman, M., Mansouri, S. A., & Zhang, Y. (2009). Search Based Software Engineering : A Comprehensive Analysis and Review of Trends Techniques and Applications, 1–78.

Harman, M., McMinn, P., Souza, J. De, & Yoo, S. (2012). Search-based software engineering: Techniques, taxonomy, tutorial. *Empirical Software Engineering and Verification*, 1–59. Retrieved from http://link.springer.com/chapter/10.1007/978-3-642-25231-0_1

Jia, Y., & Harman, M. (2008). Constructing Subtle Faults Using Higher Order Mutation Testing. Retrieved from http://discovery.ucl.ac.uk/1302155/

Jia, Y., & Harman, M. (2011). An analysis and survey of the development of mutation testing. *IEEE Transactions on Software Engineering*, *37*(5), 649–678. https://doi.org/10.1109/TSE.2010.62

Just, R., Jalali, D., Inozemtseva, L., & Ernst, M. (2014). Are mutants a valid substitute for real faults in software testing?, 654–665. https://doi.org/10.1145/2635868.2635929

Mcminn, P. (2004). Search-based Software Test Data Generation : A Survey. *Software Testing, Verification & Reliability*, *14*(2), 105–156.

Nguyen, Q. V., & Madeyski, L. (2014). Problems of Mutation Testing and Higher Order Mutation Testing. *Advanced Computational Methods for Knowledge Engineering*, 157–172. https://doi.org/10.1007/978-3-319-06569-4_12

Offutt, a. J., Lee, A., Rothermel, G., Untch, R. H., & Zapf, C. (1996). An experimental determination of sufficient mutant operators. *ACM Transactions on Software Engineering and Methodology*, *5*(2), 99–118. https://doi.org/10.1145/227607.227610

Orso, A., & Rothermel, G. (2014). Software testing: a research travelogue (2000–2014). *Proceedings of the on Future of Software Engineering - FOSE 2014*, 117–132. https://doi.org/10.1145/2593882.2593885

Polo, M., Piattini, M., & Garc, I. (2009). Decreasing the cost of mutation testing with second-order mutants. *Software: Testing, Verification, and Reliability*, (June 2008), 111–131.

Tuloli, M. S., Sitohang, B., & Hendradjaya, B. (2016). Regex Based Mutation Testing Operator Implementation. In *International Conference on Data and Software Engineering*.

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