KINETIK, Vol. 2, No. 4, November 2017, Pp. 273-282
ISSN : 2503-2259
E-ISSN : 2503-2267

Optimization of Genetic Algorithm Performance Using Naïve Bayes for Basis Path Generation

Achmad Arwan*[1], Denny Sagita Rusdianto[2]
[1, 2]Universitas Brawijaya
arwan@ub.ac.id[*1], denny.sagita@ub.ac.id[2]

Abstract

Basis path testing is a method used to identify code defects. The determination of independent paths on basis path testing can be generated by using Genetic Algorithm. However, this method has a weakness. In example, the number of iterations can affect the emersion of basis path. When the iteration is low, it results in the incomplete path occurences. Conversely, if iteration is plentiful resulting to path occurences, after a certain iteration, unfortunately, the result does not change. This study aims to perform the optimization of Genetic Algorithm performance for independent path determination by determining how many iteration levels match the characteristics of the code. The characteristics of the code used include Node, Edge, VG, NBD, and LOC. Moreover, Naïve Bayes is a method used to predict the exact number of iterations based on 17 selected code data into training data, and 16 data into test data. The result of system accuracy test is able to predict the exact iteration of 93.75% from 16 test data. Time-test results show that the new system was able to complete an independent search path being faster 15% than the old system.

Keywords: *Basis Path Testing, Genetic Algorithm, Naive Bayes*

1. Introduction

Software testing is a process of determining a program code being free from any existing errors. The testing can be executed using various methods. One of which is White-Box Testing. White-Box Testing is a testing method employing the source code as a basic knowledge in searching for code defects [1]. In order to perform White-Box Testing, the source code is converted into a graph form called CFG (Control Flow Graph) [2], a description of the source code structure of a program. CFG contains Node-Node representing the commands in a code /a pseudo code. On the other hand, DD-Graph (Decision-to-decision Graph) becomes a refinement of CFG, not establishing all codes into a graph despite only the beginning of the code to meet the branching conditions made by the graph [3]. Furthermore, DD-Graph was utilized as knowledge for the test scenario. The testing was executed by trying all the paths in the DD-Graph from the beginning to the end of the code by assigning values to the variables existing on the Node. This method was then called Basis Path Testing [4].

In Basis Path Testing, there are paths that must be skipped/tested at least once to ensure an error free code. Obtaining the paths can be completed manually or automatically. The previous research provides automatic basis path recommendations using Genetic Algorithms [5][6]. Genetic Algorithm is a method mimicing the pattern of chromosome evolution in the genes of living things. In the Genetic Algorithm, there are chromosome mutation operations, crossing the chromosomes on the genes. These chromosomes can be generated from codes, branching, iterations, assignments, etc. Ghiduk's research [5] was able to suggest an independent path on the Basis Path Testing after a certain iteration. One of the factors determining the success of finding basis path using genetic algorithms is population and iteration. For certain codes, independent paths occasionally show complete appearance. However, independent paths, in other cases, completely disappear. For example, certain codes showed 30 iterations, but in reaching the last iteration, the independent path combinations were completely unrecognized. Increasing the number of mutation iterations/cross over might become a possible solution in order to locate basis paths, a condition called less iteration. Furthermore, a given code, in another example, was identified to have 100 iterations for the path appearance. Nevertheless, until reaching the 70th iteration, it turned out all the basis path fully presented, a condition called by over iteration. The uncertainty of the number of iterations makes the performance of the Ghiduk

Arwan, A., & Rusdianto, D. (2017). Optimization of Genetic Algorithm Performance Using Naïve Bayes for Basis Path Generation. Kinetik : Game Technology, Information System, Computer Network, Computing, Electronics, and Control, 2(4). doi:http://dx.doi.org/10.22219/kinetik.v2i4.370
Paper submitted on July 31, 2017; Revision on August 17, 2017; Received August 18, 2017

research become less appealing because the user has to experiment with entering the number of iterations until a basis pathway appears. The complete appearance of the paths, sometimes, may be easily achieved, but re-attempting using higher iterations may sometimes be needed to achieve this complete path appearance.

Naïve Bayes is a method of probability-based classification that assumes features being independent, and classification results are unaffected by the dependency of each other's features [7]. Naïve Bayes can be used to classify documents, spam, health and many other fields of science.

Optimizing the use of Genetic Algorithms in searching for independent paths can be completed by predicting the exact number of iterations, so the users do not have to experiment with a different number of iterations to get the complete emersion of independent paths. Some metrics can be used to sharpen the prediction of the exact number of the iterations. The NBD metric is a metric calculating the complexity of the depth "if" structure [8]. LOC is a metric for finding code length. Node is a feature to find a number of statements in Java code, and Edge is the link between one Node with another [2]. V(G) is a complex metric measuring the number of branches in the source code [9]. The larger the V(G), the more complex the code is. Matrix V(G), LOC can also be utilized to determine the degree of convenience in code maintenance [10].

This research explores how to determine the corresponding number of iterations with proper code characteristics of LOC, NBD, Node, Edge, V (G) method with Naïve Bayes classification method. The research was able to improve the previous research [5] in basis path generation. Hence, over iteration or less iteration can be avoided. Therefore, performance improvement can be achieved when generating independent paths.

2. Research Method

This study aims to optimize the Genetic Algorithm by finding the right number of iterations to generate an independent path using the Naive Bayes classification technique. To achieve this result, the proper research method will be applied in this study. Figure 1 shows an overview of the research methods employed in this study.

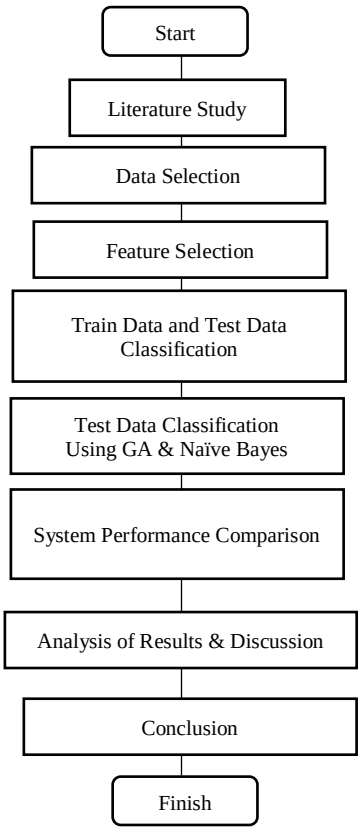

Figure 1. Research Methods

2.1 Literature Study

The literature study is imperative process in obtaining references and searching for relevant methods. These references are closely related to the topic of independent path generation, the Genetic Algorithm, and the selection of Java source code features.

2.2 Data Selection

Data selection is a process for selecting Java-based source code used as train data. The selection of these data was executed by considering the number of lines of code, the complexity of the code, the number of regions, the number of Nodes, and NBD metrics. The aspects ranged from small, medium, and large-scale to investigate which features were related to determine the optimal number of iterations. There were 7 Java files with the ± 40 total number of methods.

2.3 Feature Selection

Feature selection is the process of selecting the characteristics of the source code used for calssifying the training data. The selection was completed by a simple experiment finding features closely related to the corresponding number of iterations. The program code was inserted into the system. Afterwards, it was extracted into the features.

2.3.1 Node

Nodes are instructions in a single line of code. The instructions referred here are all instructions in the source code other than the decision instruction (if). In Figure 2, Nodes are illustrated with a circle having no sign (example no. 2, 5, 6, and11). Nodes were used as attributes to clarify how many Nodes were in one method. A method with the same VG does not necessarily have the same Node, so Node is considered a required attribute.

2.3.2 Edge

Edge is an arrow connecting Node with Node or Node with predicate Node (see Figure 2). Principally, all arrows from one instruction to the next instruction are the edge. A method with the same VG does not necessarily have the same edge, so edges are considered as attributes needed to be used.

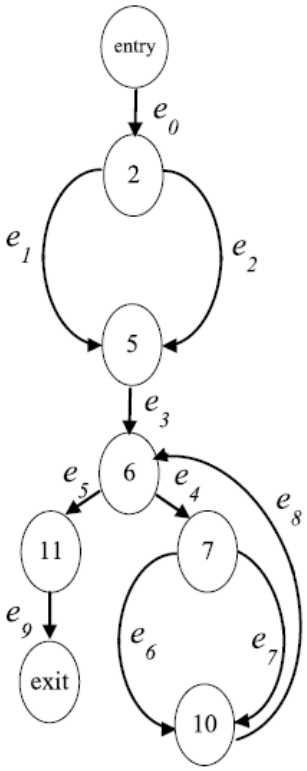

Figure 2. Node & Edge Code Representation

2.3.3 V(G)

V(G) is the complexity of a code. The value of V(G) is obtained from Node and Edge calculations. The results of V(G) vary from one to infinity. One means simple, and it grows into more complex code. the value of V (G) can be calculated by utilizing Equation 1.

$$V(G) = [E] - [V] + p \quad (1)$$

Annotation:
E (Edge) : Number of instructions.
V (Vertices) : Number of vertices.
P : Number of entry & exit points

2.3.4 LOC Metric

The LOC metric is a measurement of how many lines of code in a given method. The purpose of using LOC is to measure how many codes are needed to complete a method. There was higher complexity dispite of fewer lines of code used. However, there was also complexity in many code lines. This condition underlies why LOC was used as an attribute.

2.3.5 NBD (Nested Block Depth) Metric

NBD metrics are used to find the depth of the if condition in a code. The deeper if nested, the greater was the value of its NBD. NBD is different from VG because VG recognizes how many branches, but NBD only measures how deep is the nesting condition in a method.

2.4 Train Data and Test Data Classification

The classification of train data is the process of labeling the train data according to the optimal number of iterations of the train data. The classification process was executed by manual experiment. Java files, subsequently, were tested one by one into the implementation application of Ghiduk research. The steps of train data and test data classification can be seen in Figure 3.

The selection of test data was executed by using Java code data from http://freesourcecode.net/ and Java project from UB lecturer's. From both sources, it generated 4 projects with the number of method ± 250. Unfortunately, not all of these methods can be used in the experiment, some because the path was only one or the code cannot be well understood by the system created. This was because the Spoon Library cannot understand the existing code in Java. Therefore, there were 33 methods that can be employed. Each of these methods was independently searched for its path. There were 33 methods that matched the criteria, showing more than one basis path. These 33 methods were then divided into two (50% each) of train data as many as 17, and test data by 16. Tain data were used to create a classification model using Naïve Bayes. Test data were then used to test the system success rate in predicting iterations as the solution in the optimization method found by Ghiduk.

The selection was manually done (see Figure 3 and 4). Each of these data was stored in CSV format, in accordance with the format that can be processed by WEKA in classifying the data. The train data were stored in the datalatih.csv file while the data were stored into the datauji.csv file. The results of feature selection and classification of train data can be seen in Table 1. Meanwhile, the results of feature selection and test data classification can be seen in Table 2.

2.5 Test Data Classification using GA and Naive Bayes

This process is a process of classifying experiments utilizing test data. Test data would predict the number of iterations and measure the system recommendation to an independent path after iteration. The results of the system would be compared with the manual data, in recognizing whether or not the manual basis path had all been recommended by the system. The train data were then processed using a Naive Bayes classifier using WEKA library. The stages are illustrated in Figure 4.

Accuracy is a method to measure the classification quality of a system to the predicted results made by experts [11]. The expert results being the results of manual processes then become the basis for comparison with the classification results of the system. The level of accuracy in the result of classification was afterwards measured using Equation 2.

2.6 Performance Comparison of Old and New Systems

This process is a process of measuring the performance of the old system found by Ghiduk (GA without iteration classification) with the new system, the system in this research (GA with iteration classification using Naïve Bayes). The comparison was completed by measuring the time elapse in both mentioned systems. The performance comparison was measured by Equation 3.

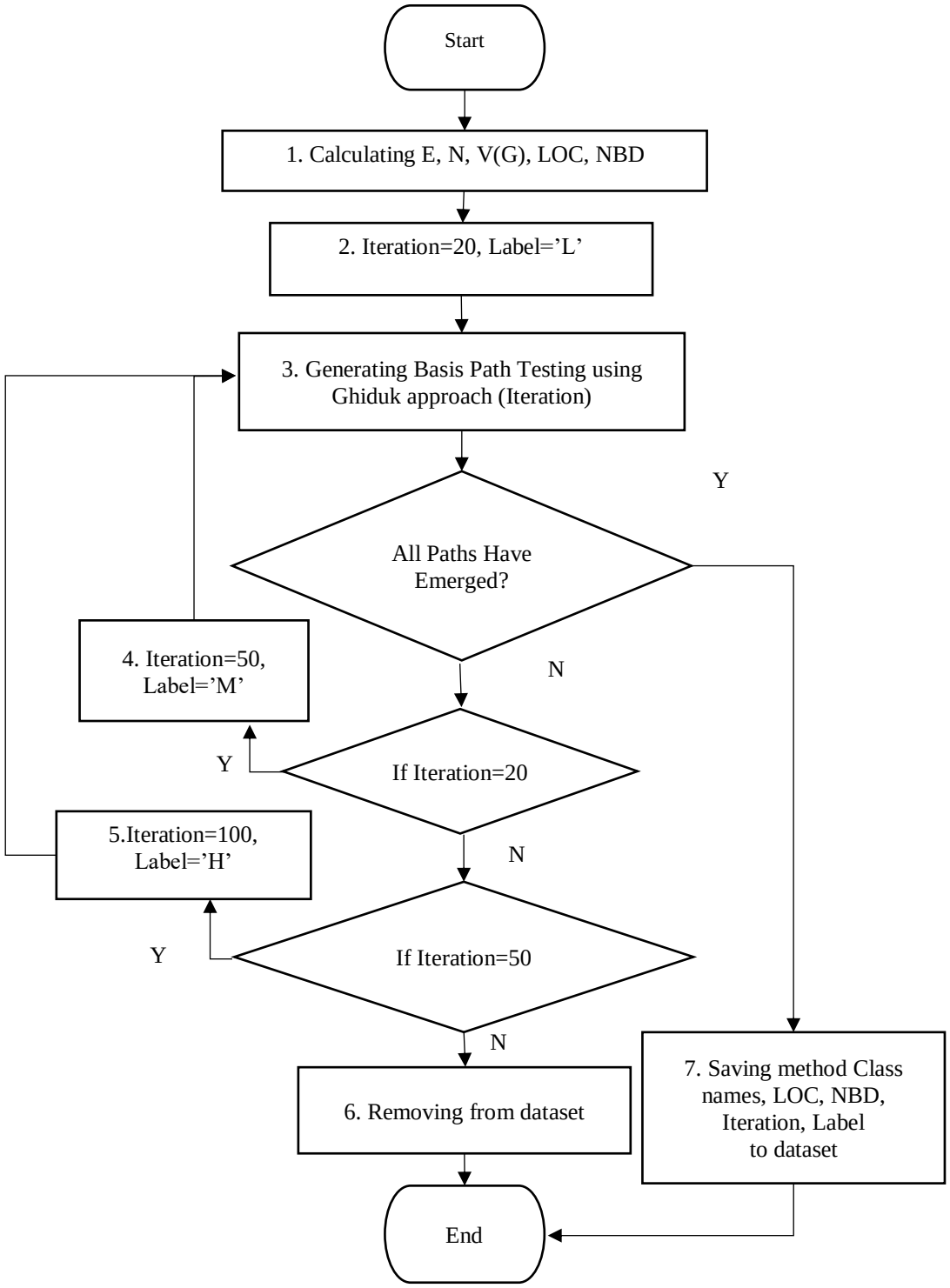

Figure 3. Manual Data Classification

Table 1. Train Data

No	Name	E	N	V(G)	LOC	NBD	Label
1	Edge.printByNode	6	6	2	8	2	L
2	Edge.getEdgeByNode	6	6	2	8	2	L
3	Edge.getSibling	9	7	4	14	3	L
4	Edge.addsucessor	7	6	3	8	2	L
5	Pong.PaintIntro	5	5	2	19	2	L
6	operasiGenetik.rankChromosome	11	9	4	12	4	M
7	DDG.SameEdge	10	8	4	7	3	L
8	DDG.closingBlockEdge	8	7	3	18	3	L
9	DDG.AllEdges	5	5	2	9	2	L
10	DDG.buildExitEdges	11	9	4	107	4	L
11	DDG.printNodeList	6	6	2	10	2	L
12	DDG.setEdgeSucessorWhile	7	6	3	93	3	L
13	DDG.setNodeList	7	6	3	7	3	L
14	PorterStemmer.Step3	84	56	30	41	2	H
15	PorterStemmer.Step4	31	22	11	19	2	H
16	PorterStemmer.m	31	22	11	31	2	H
17	TrafficSimulation.init	6	6	2	25	2	L

Table 2. Test Data

No	Name	E	N	V(G)	LOC	NBD	Label
1	dauber.mouseclicked	16	12	6	21	4	H
2	dauber.paint	10	8	4	5	3	L
3	edge.checkedgesucesor	8	7	3	8	2	L
4	operasigenetik.cekkeberadaan	8	7	3	3	3	L
5	operasigenetik.copychromosome	6	6	2	3	2	L
6	operasigenetik.selectbest	8	7	3	13	3	L
7	ddg.printnodelist	6	6	2	10	2	L
8	pong.run	53	38	17	83	5	H
9	pong.initgame1	9	8	3	16	2	L
10	pong.paint	13	10	5	19	2	L
11	pong.paintgame2	10	8	4	13	2	L
12	chromosom.printgene	6	6	2	8	2	L
13	chromosom.setzero	5	5	2	5	2	L
14	chromosom.isequal	8	7	3	10	3	L
15	chromosom.generaterandom	6	6	2	11	3	L
16	porterstemmer.step5	97	62	47	35	2	H

2.7 Analysis of Results and Discussion

This process is a process to explain the outcome of the system, both in terms of the accuracy of recommendations and in terms of performance of the old system and the new system. This process also depicted whether the new system will overcome the challenges of the old system optimization or vice versa. There were no significant results on the comparison results from the old and new system performance.

2.8 Conclusion

This process a concluding statement from the results obtained from this study. Moreover, it includes the improvement recommendations from the results obtained.

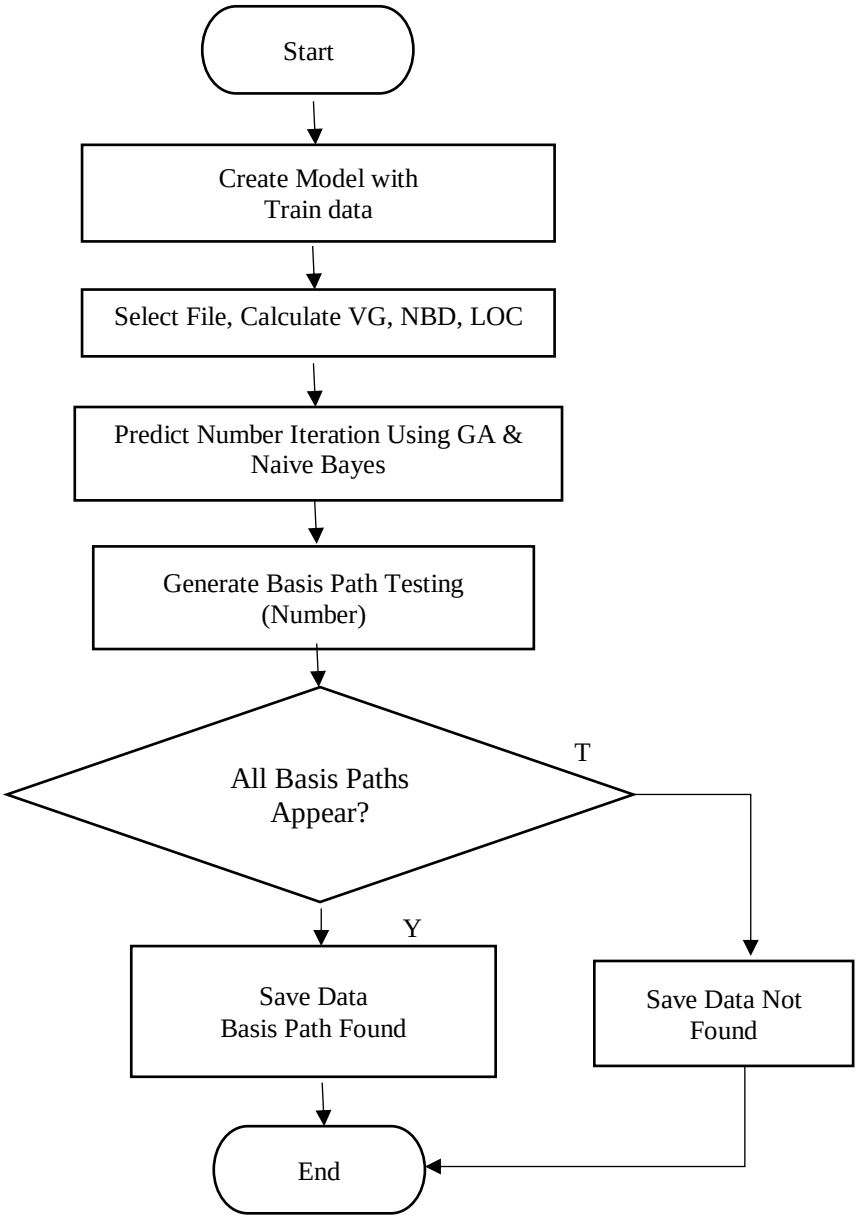

Figure 4. Data Test Classification Using GA & Naive Bayes

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{TP} + \text{FP} + \text{TN} + \text{FN}} \quad (2)$$

$$\text{Efficiency} = \frac{\text{Old System Time} - \text{New System Time}}{\text{Old System Time}} \quad (3)$$

3. Results and Discussion

3.1 System Architecture

The system was built by using: (1) Java programming language; (2) Spoon Library [12] to get VG and test case generation; (3) Weka [13] as a library classifier; and (4) source code

metrics. The input was a Java file that existed in the test data. The Figure 5 illustrates the architecture of the system established.

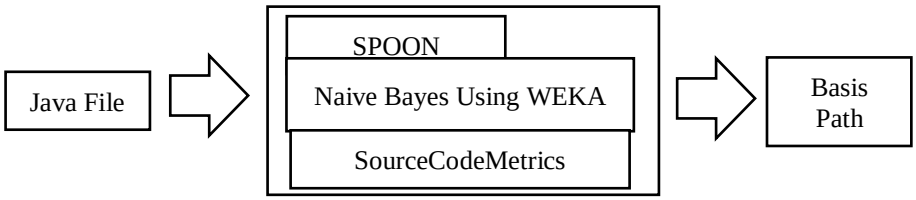

Figure 5. System Architecture

The Java file would first compute the number of Nodes, Edge, VG, NBD, LOC assisted by Spoon and SourceCodeMetric libraries. Afterwards, the results were processed with WEKA using the Naïve Bayes algorithm to predict the most appropriate iteration count based on the previous train data. Once predicted, the number of iterations of predicted results would then be used as parameters to generate independent paths. Consecutively, once a basis path was identified, the paths would be compared based on its number with the manual data related to the possibility in acquiring total appearance. If the complete appreance was identified, the data would be calculated as True. On the other hand, the data would be calculated as wrong data in the case of incomplete appearance.

The test was executed using 16 data (see Table 2) which were then inserted into the system (Figure 6). The system received the input in the form of Java file as well as the selected appropriate method in accordance with the provided 16 data. Each method of test data was tested one by one into the system to then recorded the number of appearing paths and its duration to complete the search for the independent path. The system predicted the number of iterations with the Naïve Bayes model based on the Node, Edge, VG, NBD, LOC parameters obtained from within the selected code (method). The system then answered the existing independent paths in the method. The number of paths was then compared with the manual results. Stroring the data in the correct data would be executed when finding equal number. Conversely, the data would be stored as wrong data after finding no appropriate number of paths.

The system inputs were file names and method names. The system will then generate the basic path of the method. The system also recorded the duration in predicting the base path. The recording results were after that used to compare the possibility to have complete appearance of basic pathways. The recording results were also used to calculate the speed to find the code testing base path. Figure 6 shows the interface of the system.

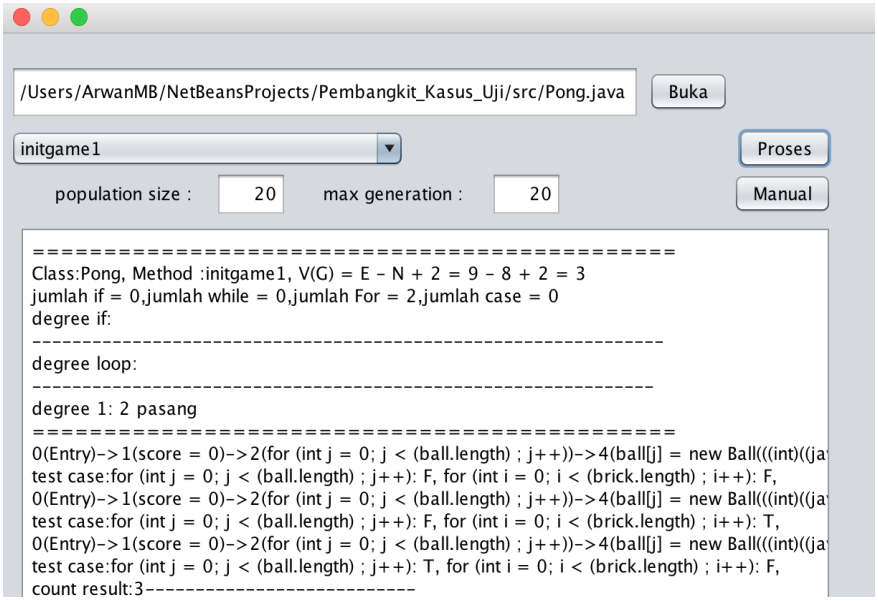

Figure 6. System Interface

3.2 Test Results Accuracy

The iterative prediction test using GA and Naïve Bayes showed the following results:

$$\text{Accuracy} = \frac{\text{TP} + \text{TN}}{\text{ALL}} = \frac{15}{16} = 0.937$$

The Naïve Bayes model can predict the exact number of iterations of 93.75% from the 16-test data. This result can be achieved because the test data and train data variation were considered low since most of them are labeled L(Low) meaning it only required low iteration to generate the basis path.

3.3 Comparison of Performance Test Results

The following test execution is a performance improvement test using time elapsed calculation compared to the old system. Testing was implemented by using the time record to calculate the duration required by the system in getting an independent path. Table 3 shows the comparative results between the old system and the new system.

Table 3. Test Data

No	Name	Label	Iterations			Old	New
			20	50	100		
1	Dauber.mouseClicked	H	1105	0	0	1105	1105
2	Dauber.paint	L	71			71	71
3	Edge.checkEdgeSucesor	L	9			9	9
4	operasiGenetik.Cekkeberadaan	L	60			60	60
5	operasiGenetik.copyChromosome	L	11			11	11
6	operasiGenetik.selectBest	L	36			36	36
7	DDG.printNodeList	L	8			8	8
8	Pong.run	H	169	1159	8084	9412	7667
9	Pong.initgame1	L	27			27	27
10	Pong.paint	L	614			614	614
11	Pong.PaintGame2	L	31			31	31
12	Chromosom.printGene	L	12			12	12
13	Chromosom.setZero	L	10			10	10
14	Chromosom.isEqual	L	6			6	6
15	Chromosom.generateRandom	L	10			10	10
16	PorterStemmer.Step5	H	0.4	0	0	0.4	0.4
					$\sum$	11422	9677.4

In the above experimental results, there were 16 data. 13 data have label L and 3 have label H. Label L is interpreted as requiring low iteration calculated by 20 to produce basis path. The time frame shows how many seconds required by the system to find a basis path. For codes labeled L, they managed to performe only one experiment having 20 iterations to get basis path. Thus, in iteration 50 and 100, there were no contents because 20 iterations were considered as adequate. Conversely, number 8 code is labeled H. It means that it needed to experiment 3 times in getting the independent path, so all columns have contents.

Overall, the old system took as long as 11.42 seconds to find all paths. The new system took as long as 9,678 seconds to find all paths due to a system being able to predict the exact number of iterations, saving 1.74 seconds. The time efficiency is as follows:

$$\text{Eficiencey} = \frac{\text{Old System} - \text{New System}}{\text{Old system}} = \frac{11.42 - 9.678}{11.42} = \frac{1.74}{11.42} = 0.15\%$$

4. Conclusions and Future Works

Performance optimization of the application of Genetic Algorithms to find independent paths in the source code was completed by using features in Edge, Node, VG, NBD, and LOC. The features were then extracted from the code to be used as train data to classify the exact number of iterations using the Naïve Bayes classification model. 17 data were used as train data. 16 data were used for test data. The measurement of classification accuracy was done to measure to what extent Naïve Bayes can predict the exact number of iterations. The result is a system capable of predicting the exact number of iterations of 93.75% from 16 test data. The measurement time optimization was used to measure time efficiency between the old system and the new system. The optimization result states the new system was 15% faster than the old system.

In the future, the researchers will try to implement Graph theory to generate basis path since having significant correlation with a graph used such as BFS, Dijkstra.

References

[1] I. Sommerville, "*Software Engineering*," 9th ed. Pearson, 2011.
[2] V. Elodie, "*White Box Coverage and Control Flow Graphs*," Pp. 1–33, 2011.
[3] A. Bertolino, R. Mirandola, and E. Peciola, "*A Case Study in Branch Testing Automation, Journal of Systems and Software*," Vol. 38, No. 1, Pp. 47–59, 1997.
[4] F. Zapata, A. Akundi, R. Pineda, and E. Smith, "*Basis Path Analysis for Testing Complex System of Systems*, Procedia Computer Science," Vol. 20, Pp. 256–261, 2013.
[5] A. Ghiduk, M. R. Girgis, and A. S. Ghiduk, "*Automatic Generation of Data Flow Test Paths Using a Genetic Algorithm*," February, 2014.
[6] W. Xibo and S. Na, "*Automatic Test Data Generation for Path Testing Using Genetic Algorithms*," 2011.
[7] I. Rash, "*An Empirical Study of the Naive {Bayes} Classifier*," January 2001, 2001.
[8] S. Herbold, J. Grabowski, and S. Waack, "*Calculation and Optimization of Thresholds for Sets of Software Metrics*," Empirical Software Engineering, Vol. 16, No. 6, Pp. 812–841, 2011.
[9] T. Ostrand, "*White-Box Testing*," Encyclopedia of Software Engineering, 2002.
[10] D. Kafura and G. R. Reddy, "*The Use of Software Complexity Metrics in Software Maintenance*," IEEE Transactions on Software Engineering, Vol. SE-13, No. 3, Pp. 335–343, 1987.
[11] A. Arwan, M. Sidiq, B. Priyambadha, H. Kristianto, and R. Sarno, "*Ontology and Semantic Matching for Diabetic Food Recommendations*," Proceedings - 2013 International Conference on Information Technology and Electrical Engineering: "*Intelligent and Green Technologies for Sustainable Development*", ICITEE 2013, Pp. 170–175, October, 2013.
[12] R. Pawlak et al., "*Spoon: A Library for Implementing Analyses and Transformations of Java Source Code,*" 2015.
[13] E. Frank, M. A. Hall, and I. H. Witten, "*The WEKA Workbench. Online Appendix for Data Mining: Practical Machine Learning Tools and Techniques*," Morgan Kaufmann, Fourth Ed., 2016.