Journal of Information Technology and Computer Science
Volume 3, Number 1, 2018, pp. 104-112
Journal Homepage: www.jitecs.ub.ac.id

# Automation of Independent Path Searching using Depth First Search

Achmad Arwan[1], Denny Sagita Rusdianto[2]
Informatic Engineering, Faculty of Computer Science, Brawijaya University,
65145 - Malang, East Java, Indonesia
arwan@ub.ac.id, denny.sagita@ub.ac.id

Received 20 August 2018; accepted 13 September 2018

**Abstract** In a basis path testing, there are independent paths that must be passed/tested at least once to make sure there are no errors in the code and ensure all pseudocode have implemented on the code. Previously, the independent path was generated using the Genetic Algorithm, but the number of iterations influenced the likelihood of the emergence of the corresponding the independent path. Besides, the pseudocode was also unable to be used directly since it must be implemented first, this makes finding an independent path longer because it has to implement the code. This research aims to find out how to find the independent path directly from pseudocode using a graph and how well the Depth First Search algorithm in finding the independent path. It was chosen because it was able to find the paths from a point to a particular point in a graph. The result of the system accuracy test was able to find the correct independent path as much as 52 from 76 test data, where the result of accuracy is 68.4% on average.

## 1. Introduction

Builds a good quality of software is the dream of every software developers. There were many studies such as the measuring software quality[1], the development frameworks[2] and the software testing have invented to solving the problems. Software testing is an important part of the software development lifecycle. Software testing is a process to ensure there was no defect/bug within the code. If there were no defects in a software, it can not be said the software was good, maybe it could be the test cases were too weak(incomplete or miss test cases). Software testing took 50 percents of time in V-Model, 50% of construction time in both protoyping and incremental model on software development lifecycle[3]. It means taking a lot of time and money to do the test. Bad testing could make software bugs hide until it revealed and perform catastrophic when software used by customers. There are two major testing that can be done (White box & Black Box).

White Box testing is a tests method that uses code/pseudocode as a basic knowledge in searching for code defects[4]. In order to do White Box testing, the pseudocode must be converted into a graph form which called CFG(Control Flow Graphs)[5]. The CFG (see figures 1) contains N(nodes) as representations of commands and E(edge) as the flow/direction from single command to another command in a code/pseudocode. DD-Graph (decision-to-decision Graph) is a

simplification of CFG where not all codes are made into graphs, but only the beginning of the code to meet with the branching conditions that are used as a graph [5] (see figures 1).

Independent path is a path that contains a set of sequence of commands which the element on it is new and distinct from other paths that have revealed. In a White box testing, testing was done by trying all the independent paths in the CFG/DD-Graph from the beginning of the code to the end by assigning values to the variables that exist on the node. This method is called the basis path testing.

In the basis path testing, there are independent paths that must be passed/tested at least once to make sure there are no errors in the code. The independent paths were used as test cases to determine whether the code implementing all the pseudocode or not. The number of paths correlates with a complexity of the code. The bigger the number of paths means more complex the code. Revealing independent paths was able to do manually, but when the number of paths more than 10, it will complicating human to discover the correct of independent paths. Fault in finding independent paths may be resulting the bugs keep hide until sometime show up accidentally and perform catastrophic error.

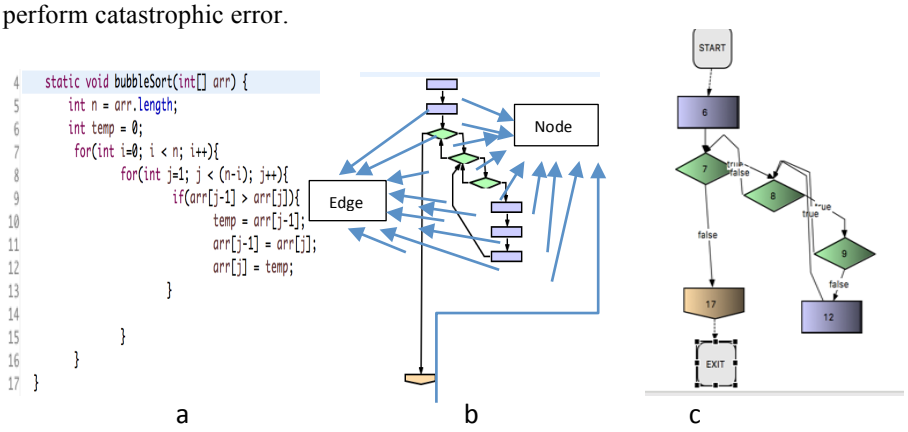


Fig. 1. (a) The BubbleSort Code, (b) BubbleSort CFG, and (c) BubbleSort DD-Graph.

Many studies have been done to solve testing problems. The Artificial Bee Colony algorithm was used to determine the independent path and to solve local the optima problem[6]. The data that used only one(triangle problem) and there is no measurement of the success criteria. A Spanning Tree combined with Particle Swarm Optimization(PSO) was employed to generate the test data automatically[7]. The data also very simple(complexity number only three). The fault propagation path coverage was used to generate test case[8]. This research does not explain the data, but only the algorithm that they used only.

Despite to test by manual, many tools have invented to perform the automatic test. Tool based on search-based also have invented to generate automatic test data[7]. The tool was intended to simplify tester to test. It employed GA, PSO, Simulated Annealing to reveal the independent path. Several tools for test the Java-based code also have invented, they were JCrasher[8], eToc [9], Randoop[10], CarFast[11], TestFul[12], T3[13], Evosuite[14] and many more. The Evosuite was the most complete tools for testing on the Java-based code. It able to target a specific class and able to generate the test suite. There were also many tools for testing C-based code have invented such as CUTE[15], DART[16], KLEE[17], Pex[18]. The Pex was also great tools to perform white box test generation on ".Net". Most of all tools were intended to undertake the code coverage test, which means based on the code, not the

pseudocode.

The previous research was titled "Automatic generation of basis test paths using variable length genetic algorithm"[21]. This research using the genetic algorithm to generate basis test paths. The study yielded an accuracy rate of 80%. One of the factors determining the success of finding the independent path using genetic algorithms was population and iteration.

Further research was done by optimizing the use of genetic algorithms in searching for the independent path using Naïve Bayes. It was done by predicting the exact number of iterations based on code features (Node, Edge, NBD, LOC, V(G)), so users do not have to experiment with the various number of iterations to get the appearance of all independent paths [22]. The result of the new system was faster by 15%. Further research also was done by employing the J48 classifier to determine the independent path by recommending the number of iterations [23]. The result of this system was able to find the independent paths for about 84,5% and the performance was increased by 35% from what Ghiduk's did.

All of the above researches[21–23] was using codes (Java/C++) to be parsed as CFG. So when the test will conduct, all libraries needed should be loaded and configured properly, otherwise, the system can not find the independent paths. The problem raised when the data was only the pseudocode. The pseudocode must be implemented first to generate the independent path later. This research try to solve this problem by enabling automation of independent path searching directly from the pseudocode.

The independent paths ideally should be generated from the pseudocode. This step was intended to ascertain whether all pseudocodes have been applied to the code. If a given test case did not produce the result as expected, it means there was a missing/fault in implementation(code). For that reason, the pseudocode should be used as the source of CFG/DD-Graph. Since CFG/DD-Graph were using graph representation, then a theory of the graph was proper to be used as a solver for independent paths generation problems. To traverse from one node to another specific node can be done using some algorithms such as Dijkstra, Deep First Search (DFS)[24]. These algorithms widely used in network problems such as routing and traveling salesman problems. DFS has been used to determine test cases within system testing graph which was a combination between activity diagram and sequence diagram[25].

This research intended to generate the independent paths from pseudocode. The pseudocode converted into a graph and then searched independent path using DFS algorithm. This study also aims to measure how well the accuracy level obtained by finding an independent path on CFG/DD-Graph using the DFS algorithm. The system's input was the graph from pseudocode, and then it recommends the independent path which can be used further as test cases.

## 2. Research Method

To achieve the goal, the researcher contrived the research design which illustrated in figure 2. The processes were dataset determination, revealing independent path using DFS, checking independent path, save the data which correct/not correct, calculate accuracy. All process would be explained below.

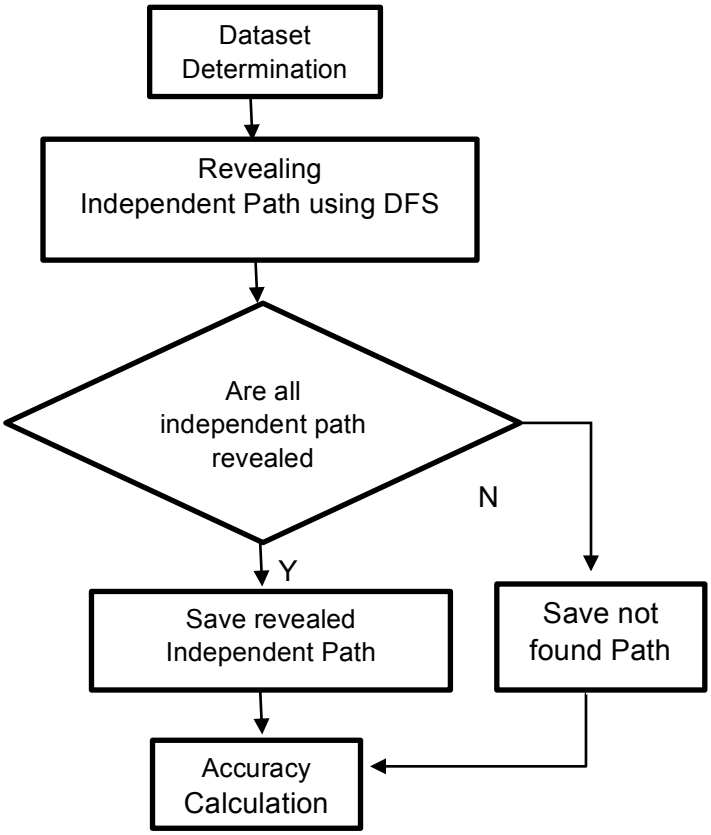


Fig. 2. Research Method.

### 2.1 Dataset Determination

The first process was dataset determination. This process intended to choose the data that would be used. The data must contain pseudocode/code or CFG and independent path. The researcher used the basis path samples from the internet(using the keyword "independent path") which were course slide and the journal[26][27]. The data can be accessed through website https://tinyurl.com/y9gz4ewe. The main reason why the data must comply with the requirements was to simplify when calculating the accuracy. The researcher only has to check whether the independent path on the graph was correct or not. After the search, there was 14 dataset(each dataset was single pseudocode or code or CFG) which contain 76 of the independent path. The sample of the dataset illustrated in figure 3.

### 2.2 Revealing Independent Path Using DFS

The second process revealed independent path using DFS. This process was done by using the implementation of DFS in the Java-based application. The input was a graph. The DFS application revealed the independent path. The output was the independent paths. The output would be used as comparison materials on the next process. DFS algorithm as illustrated in figure 4. DFS was able to search the path from the most depth into the shallowest.

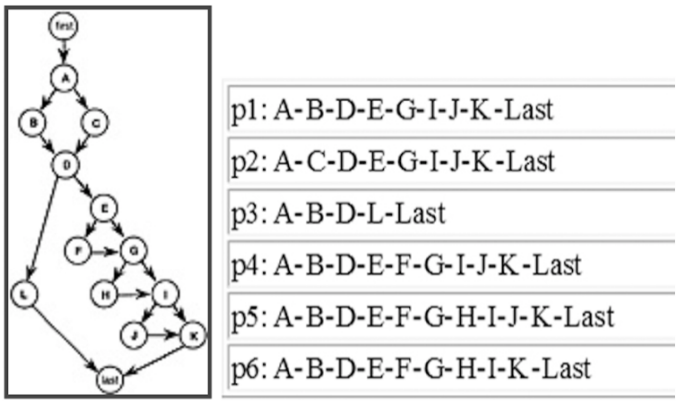


Fig. 3. Sample of Dataset.

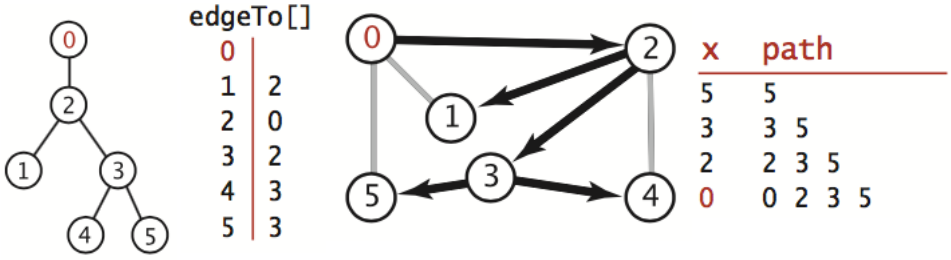


Fig. 4. DFS Graph Path [24].

### 2.3 Check and Save Independent Path

The third process was to check whether the independent path has revealed or not. This process was done by checking each path from as a result of the previous process. Each of these paths would be checked one by one to find out whether the paths was an independent path or not.

### 2.4 Accuracy Calculation

The final process was the calculation of accuracy. To measure the accuracy of the system, the researcher using the following equation.

$$Accuracy = \frac{System\ Independent\ Path}{Manual\ Independent\ Path}\ X\ 100\ \% \quad (1)$$

The system independent path was the number of independent paths which were come up from the system and matched with the independent path from manual. The manual independent path was the number of the independent paths which were taken from manual calculation from the dataset.

## 3. Results and Analysis

This research achieved the results which depicted in table 1. There was 14 dataset. Data complexity varies from an only single path(very simple) to eleven paths (quite complex). This data limitation was caused by the sample of basis path tests from the course or journal was too few.

**Table 1** Results of Accuracy

| Number | Data number | #Path using DFS | #Path using manual | Accuracy |
|---|---|---|---|---|
| 1 | 1 | 1 | 4 | 25% |
| 2 | 2 | 6 | 7 | 86% |
| 3 | 3 | 4 | 5 | 80% |
| 4 | 4 | 4 | 4 | 100% |
| 5 | 7 | 1 | 4 | 25% |
| 6 | 9 | 1 | 5 | 20% |
| 7 | 10 | 5 | 5 | 100% |
| 8 | 13 | 2 | 4 | 50% |
| 9 | 14 | 1 | 5 | 20% |
| 10 | 15 | 2 | 3 | 67% |
| 11 | 16 | 5 | 7 | 71% |
| 12 | 19 | 11 | 11 | 100% |
| 13 | 20 | 6 | 6 | 100% |
| 14 | 21 | 3 | 6 | 50% |
| | total | 52 | 76 | |
| | | | Avg | 68% |

Based on table one, it can be concluded DFS was able to reveal the independent path on average about 68%, 100% for the best, and 20% for the worst. This results might be the first discovery of the automation of independent path which based on pseudocode. There was no other research which using pseudocode have found previously. So it cannot be comparing using other methods since the previous researches using the code to generate independent path. Based on the results, there were good results and bad results. Both of it will be discussed in the following section.

### 3.1. Good Results Analysis

Good results mean accuracy was above 80, and it could be observed in data number 2,3,4,7,12,13. The graphs which good mostly have the branch that not directly connected into the end of nodes. The number of complexity was also not correlating with accuracy. Less number of the nested loop or iteration got the better result. Some of the good result graph shown in figure 5.

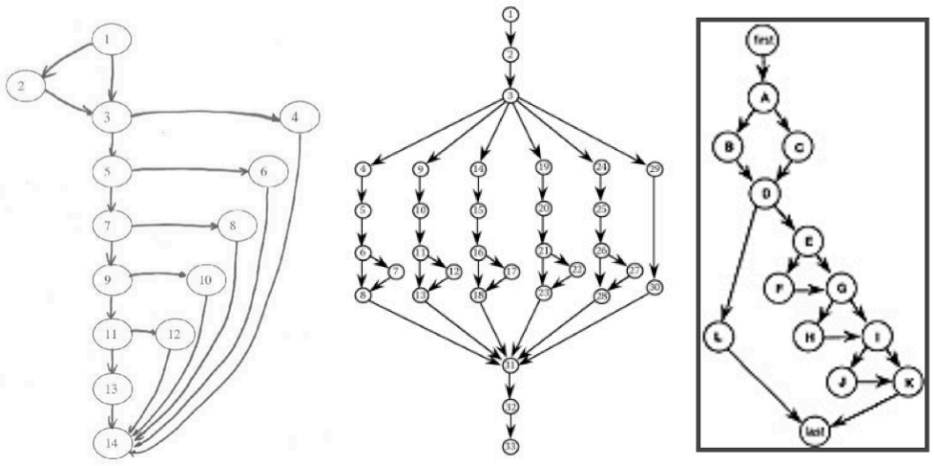

Fig. 5. Good Result Graphs Sample.

### 3.2. Bad Results Analysis

Bad results mean accuracy was under 50. It could be observed in data 1,5,6,9. The graphs which have the branch which connects directly into the end of nodes correlating with bad accuracy. The number of complexity not correlating with the accuracy. The number of the nested loop got a worse result. Some of the bad result graph shown in figure 6.

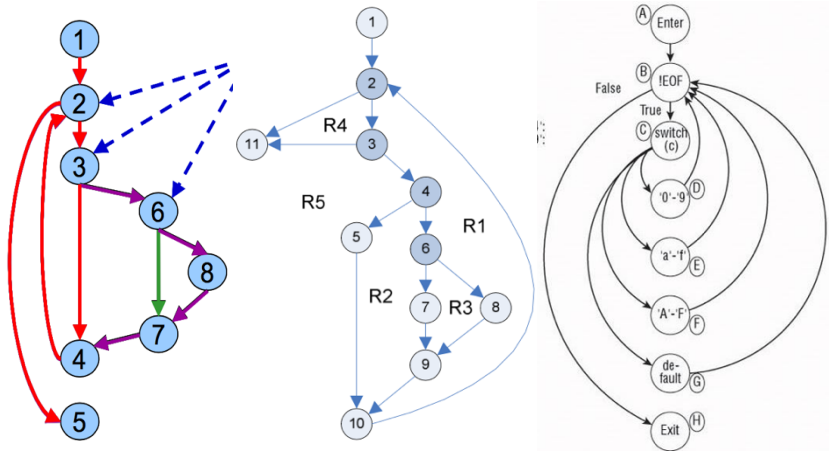


Fig. 6. Bad Result Graphs Sample.

## 4. Conclusion

Based on the experiments, it can be concluded that DFS was able to be used to generate the independent paths. The best result was 100% of the independent path have revealed. The worst was only 20% of the independent path have revealed. On average DFS was able to recommend 68.4 % of the independent path (56 paths of 76 paths). The number of a direct branch into the end of node corresponding with small

of accuracy. A number of the nested loop also have corresponded with small of accuracy. The number of complexity was not correlating with accuracy.

For future works, it can be extended using another algorithm, evaluating using a complex dataset (complexity more than 15), or develop the GUI tool to generated independent paths. It also can be extended by integrating with IDE or automation tools.

## References


1. Pradana F, Priyambadha B, Rusdianto DS: Identifying Thresholds for Similarity-Based Class Cohesion ( SCC ) Metrics. J Inf Technol Comput Sci 1:72–81, (2016).
2. Lestari VA, Aknuranda I, Ramdani F.: Development Framework for the Evaluation of Usability in E-Government : A Case Study of E-Finance Government of Malang. J Inf Technol Comput Sci 2:76–89, (2017).
3. Pressman RS: Software Engineering A Practitioner's Approach 7th Ed - Roger S. Pressman, (2009).
4. Sommerville I: Software Engineering, 9th ed. Pearson, (2011).
5. Elodie V: White Box Coverage and Control Flow Graphs. 1–33, (2011).
6. Lam SSB, Raju MLHP, Uday KM, Swaraj C, Srivastav PR: Automated generation of independent paths and test suite optimization using artificial bee colony. Procedia Eng 30:191–200 . doi: 10.1016/j.proeng.2012.01.851, (2012).
7. Singla S, Kumar R, Kumar D: Natural Computing for Automatic Test Data Generation Approach Using Spanning Tree Concepts. Procedia Comput Sci 85:929–939 . doi: 10.1016/j.procs.2016.05.284, (2016).
8. Kun W, Yichen W: Software test case generation based on the fault propagation path coverage. Proc - Annu Reliab Maintainab Symp 2016–April: . doi: 10.1109/RAMS.2016.7448004, (2016)
9. Malhotra R, Poornima, Kumar N: Automatic test data generator: A tool based on search-based techniques. 2016 5th Int Conf Reliab Infocom Technol Optim ICRITO 2016 Trends Futur Dir 570–576 . doi: 10.1109/ICRITO.2016.7785020, (2016).
10. Csallner C, Smaragdakis Y: JCrasher: An automatic robustness tester for Java. Softw - Pract Exp 34:1025–1050 . doi: 10.1002/spe.602, (2004).
11. Tonella P, Ceccato M, Áòø Ò: Aspect Mining through the Formal Concept Analysis of Execution Traces, (2004).
12. Pacheco C, Ernst MD: Randoop: Feedback-Directed Random Testing for Java. Companion to 22nd ACM SIGPLAN Conf Object oriented Program Syst Appl companion - OOPSLA '07 5:815 . doi: 10.1145/1297846.1297902, (2007).
13. Park S, Hossain BMM, Hussain I, Csallner C, Grechanik M, Taneja K, Fu C, Xie Q: CarFast: Achieving Higher Statement Coverage Faster. Proc ACM SIGSOFT 20th Int Symp Found Softw Eng 1–11 . doi: 10.1145/2393596.2393636, (2012).
14. Baresi L, Lanzi PL, Miraz M: TestFul: An evolutionary test approach for Java. ICST 2010 - 3rd Int Conf Softw Testing, Verif Valid 185–194 . doi: 10.1109/ICST.2010.54, (2010).
15. Prasetya W: T3, a Combinator-based Random Testing Tool for Java: Benchmarking. 8432: . doi: 10.1007/978-3-319-07785-7, (2014).
16. Fraser G, Arcuri A: EvoSuite : Automatic Test Suite Generation for Object-Oriented Software. Proc 19th ACM SIGSOFT Symp 13th Eur Conf Found Softw Eng 416–419 . doi: 10.1145/2025113.2025179, (2011).
17. Sen K, Marinov D, Agha G, Sen K, Marinov D, Agha G: CUTE: A concolic unit testing engine for C. 10th Eur Softw Eng Conf 13th ACM SIGSOFT Int Symp Found Softw Eng 30:263 . doi: 10.1145/1081706.1081750, (2005).

18. Godefroid P, Klarlund N, Sen K: DART: directed automated random testing. Proc 2005 ACM SIGPLAN Conf Program Lang Des Implement 213–223 . doi: 10.1145/1065010.1065036, (2005).
19. Cadar C, Dunbar D, Engler DR: KLEE: Unassisted and Automatic Generation of High-Coverage Tests for Complex Systems Programs. Proc 8th USENIX Conf Oper Syst Des Implement 209–224 . doi: 10.1.1.142.9494, (2008).
20. Tillmann N, de Halleux J: Pex: White Box Test Generation for .NET. Proc TAP 134–153 . doi: 10.1007/978-3-540-79124-9_10, (2008).
21. Ghiduk AS (2014) Automatic generation of basis test paths using variable length genetic algorithm. Inf. Process. Lett. 114:304–316
22. Arwan A, Rusdianto D: Optimization of Genetic Algorithm Performance using Naïve Bayes for Basis Path Generation. Kinet Game Technol Inf Syst Comput Network, Comput Electron Control 2: . doi: 10.22219/kinetik.v2i4.370, (2017).
23. Arwan, A., & Rusdianto DS: Determining Basis Test Paths using Genetic Algorithm and J48. Int J Electr Comput Eng 8, (2018).
24. Sedgewick R, Wayne K: Algorithms, Addison-Wesley Professional. (2011).
25. Meiliana, Septian I, Alianto RS, Daniel, Gaol FL: Automated Test Case Generation from UML Activity Diagram and Sequence Diagram using Depth First Search Algorithm. Procedia Comput Sci 116:629–637 . doi: 10.1016/j.procs.2017.10.029, (2017).
26. Kurniawan TA: Pengujian Struktur Program Dengan Pengujian Jalur Dasar ( Basis Path Testing ) : Teori Dan Aplikasi. EECCIS 1:29–32, (2007)
27. Wang Q, Jiang S, Zhang Y: An Approach to Generate Basis Path for Programs with Exception- Handling Constructs. 51:330–335 . doi: 10.7763/IPCSIT.2012.V51.56, (2012).