IAES International Journal of Artificial Intelligence (IJ-AI)
Vol. 14, No. 1, February 2025, pp. 252~259
ISSN: 2252-8938, DOI: 10.11591/ijai.v14.i1.pp252-259

Comparative analysis of genetic algorithms for automated test case generation to support software quality

Tiara Rahmania Hadiningrum, Siti Rochimah
Department of Informatics, Faculty of Intelligent Electrical and Informatics Technology, Institut Teknologi Sepuluh Nopember, Surabaya, Indonesia

Article Info

Article history:

Received Jun 19, 2024
Revised Jul 28, 2024
Accepted Aug 30, 2024

Keywords:

Automatic generation test cases
Genetic algorithms
Software quality assurance
Software testing
Test case generation

ABSTRACT

Software testing is crucial for enhancing software quality, but designing test cases is a labor-intensive, resource-intensive, and time-consuming process. Additionally, test case designers often introduce subjectivity when creating test cases manually. To address these challenges, this paper compares three different approaches for automatically generating program branch coverage test cases: the parallel data generation algorithm (PDGA), a standard genetic algorithm (SGA), and a random test generation method. By leveraging genetic algorithms and parallel data generation techniques, these automated approaches aim to reduce the manual effort, resources, and potential biases involved in test case design, while improving the efficiency and effectiveness of achieving comprehensive branch coverage during software testing. The experimental results, conducted using five datasets with programs written in PHP, demonstrate that PDGA outperforms both SGA and random methods across various tested programs, achieving higher maximum and average coverage. Specifically, PDGA achieved an average coverage of 100% in the "calculator" program, highlighting its superior stability and efficiency. While SGA also shows good performance, it is not as optimal as PDGA, and the random method shows the lowest performance among the three. These findings underscore the potential of genetic algorithms, particularly PDGA, to enhance the coverage and quality of software testing, thereby significantly improving system reliability.

Corresponding Author:

Siti Rochimah
Department of Informatics, Faculty of Intelligent Electrical and Informatics Technology
Institut Teknologi Sepuluh Nopember
Surabaya, Indonesia
Email: siti@its.ac.id

1. INTRODUCTION

Software testing is a critical stage in the software development cycle that aims to ensure the quality, reliability, and optimal performance of an application [1]–[3]. Through testing, various scenarios, and conditions are tested to identify bugs, logic errors, and potential failures in the software [4], [5]. Comprehensive testing methods involve functional testing, non-functional testing such as security and performance, as well as cross-platform testing if required [6], [7]. The results of testing provide a better understanding of the software's ability to address user needs and expectations and provide confidence that the application is ready for widespread use [8], [9]. Thus, software testing is an important foundation in ensuring product success and acceptance in the market [10].

Various algorithms have evolved to automate the test case generation process in software testing, with the aim of improving efficiency and effectiveness in detecting bugs and increasing test coverage [11], [12].

These algorithms include methods such as genetic algorithms, model-based testing, and mutation-based testing [13], [14]. Genetic algorithms utilize the concept of biological evolution to automatically create and adjust test cases based on certain criteria [15]. Genetic algorithms adopt the principles of biological evolution, such as natural selection and reproduction, to generate optimal solutions in the context of software testing [16].

The application of genetic algorithms in software automated testing is becoming increasingly important to improve the efficiency and effectiveness of the testing process [13], [17]. Genetic algorithms enable automatic creation, adjustment, and optimization of test cases, based on set criteria [14], [18]. Through integration with machine learning techniques, such as deep learning, genetic algorithms can improve their ability to generate more effective test cases [6]. The evolutionary process in genetic algorithms, which includes the formation of an initial population of test cases, genetic mutation, and natural selection, enables dynamic adaptation to change in the software under test [19]. Genetic algorithms make significant contributions in detecting bugs, increasing test coverage, and accelerating the overall software development cycle [20], [21].

Various studies have been conducted in the field of automatic test case generation, but there is still a lack of understanding of the relative effectiveness of the various algorithms used [22], [23]. This research provides a comprehensive approach that compares automatic test case generation algorithms, namely parallel data generation algorithm (PDGA), standard genetic algorithm (SGA), and random test generation method (Random). Through this research, it can determine the relative effectiveness of these algorithms, to provide guidance in the selection of the most appropriate and efficient algorithm for software testing needs. It is hoped that the results of this research can provide deep insight into the advantages and disadvantages of each approach, so that it can help in determining which algorithm is best to use to improve the efficiency and effectiveness of software testing.

Research by Wambua and Wambugu [14] compares the effectiveness of Bat and genetic algorithms for test case prioritization in regression testing, finding the Bat algorithm superior in some metrics such as average percentage of fault detected (APFD), memory usage, and execution time. Rajagopal *et al.* [17] highlight the efficiency of an adaptive genetic algorithm in generating test cases, emphasizing the use of dynamic parameters and diverse datasets for effective test coverage. Chakraborty *et al.* [24] demonstrate the effectiveness of genetic algorithm in generating test cases using benchmark programs, improving test efficiency and performance. Alshammari *et al.* [25] propose a method using genetic algorithms to optimize test data generation in Python, reducing crossover and mutation operations to speed up testing. Katoch *et al.* [26] provide a comprehensive analysis of genetic algorithms, including their principles, improvements, and applications in complex problems such as combinatorial optimization and scheduling, demonstrating their effectiveness in finding optimal solutions.

Although research on automatic test case generation has been conducted extensively, understanding of the effectiveness of the various algorithms used is still limited. Identifying knowledge gaps related to the use of genetic algorithms in test case generation is important to provide a more in-depth view in selecting and developing the most suitable approach in software testing. The lack of direct comparison between traditional genetic algorithms and innovative approaches such as adaptive genetic algorithms and random test generation methods is a gap in previous research. Therefore, further research is needed to explore and compare the effectiveness of these three approaches in the context of software testing, to provide a more comprehensive understanding of the optimal genetic algorithm approach.

This research is structured with the following organization: in section 1 introduction, this section provides a general description of the background, motivation and objectives of the research conducted. Section 2 related work reviews previous research relevant to the topic discussed, providing context and theoretical basis for this research. In section 3 methodology, the methodology used in the research is explained in detail, including the experimental environment, genetic algorithm parameters, evaluation metrics, and the steps taken in the experiment. Section 4 experiment, results, and analysis presents the experimental results obtained, both in the form of pictures, tables and descriptive explanations. This section also analyzes and discusses these results, comparing the performance of the various genetic algorithms tested, and relating them to theory and previous research. Finally, in section 5 conclusion, this study summarizes the main findings, contributions, and potential for further research in the future based on the results obtained.

2. METHOD

The methodology of this work consists of three main phases, reflecting the preparation, implementation, and evaluation. The phases are data collection and preprocessing, algorithm implementation, and evaluation. Experimental design of this work is illustrated by Figure 1.

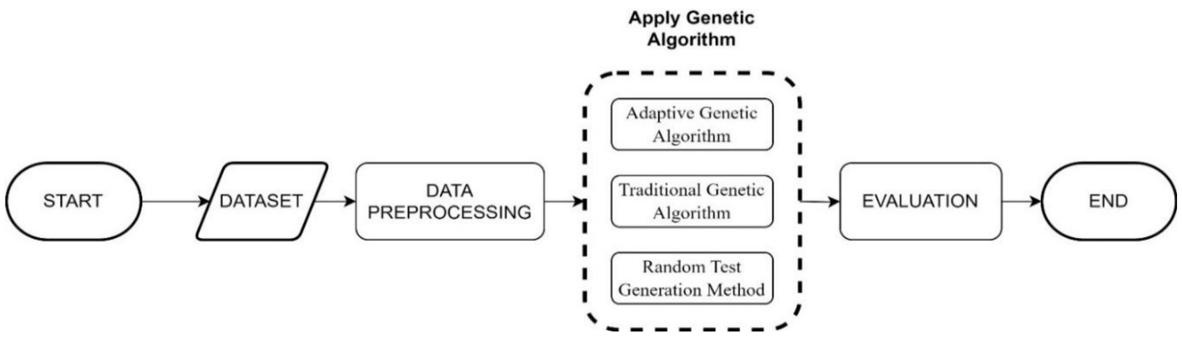
Apply Genetic Algorithm
START
DATASET
DATA PREPROCESSING
Adaptive Genetic Algorithm
Traditional Genetic Algorithm
Random Test Generation Method
EVALUATION
END

Figure 1. Methodology

2.1. Dataset

In this experiment, the primary objective is to evaluate the feasibility and effectiveness of the test case generation method proposed within this paper, with a particular emphasis on branch coverage. To accomplish this, a set of five distinct PHP programs serves as the focal point. These programs encompass diverse functionalities, including calculator, todo list management, weather forecasting, currency conversion, and body mass index (BMI) calculation. The structural intricacies, functionalities, and varying levels of complexity inherent in each program are meticulously outlined in Table 1. This comprehensive delineation enables a thorough assessment of the proposed genetic algorithm's efficacy in generating test cases that adequately cover branch conditions across a spectrum of PHP applications.

Table 1. Dataset experiment

Program name	Number of input parameters	Number of branches	Number of lines of code	Program information
Calculator	2	10	89	Calculate arithmetic functions
Todo list	3	28	82	Record and organize tasks
Weather	2	18	167	Weather conversion calculations
Currency conversion	3	13	112	Currency conversion calculations
BMI calculator	2	25	75	Ideal body weight calculation

The selection of PHP programs, spanning different domains and functionalities, facilitates a robust evaluation of the test case generation method's versatility and adaptability. By encompassing a diverse array of programs, ranging from basic computational utilities to more complex functional implementations, the experiment aims to provide nuanced insights into the algorithm's performance across varied scenarios. Through this rigorous evaluation process, researchers and practitioners can gain a deeper understanding of the method's strengths, limitations, and potential areas for refinement, ultimately contributing to the advancement of software testing methodologies in PHP development environments.

2.2. Data preprocessing

Before applying genetic algorithms for automatic test case generation, an important step that needs to be taken is to pre-process the data. In this research, the data in question is the source code of the program to be tested. Data pre-processing aims to prepare the source code so that it can be accepted and processed properly by the genetic algorithm. The data pre-processing steps performed include code cleaning, comment removal, and tokenization. Code cleaning is carried out to remove characters or lines of code that are irrelevant or can interfere with the test case generation process. Comment removal is necessary because comments do not affect program execution, so they can be ignored in the test case generation process. Tokenization is the process of breaking down source code into smaller lexical units (tokens), such as keywords, variables, and operators.

After the tokenization process, the resulting tokens can be represented in the form of chromosomes for use in genetic algorithms. This chromosome representation can vary, such as a binary representation, an integer representation, or any other suitable representation. Choosing the right chromosome representation is very important because it will affect the effectiveness of genetic operators such as crossover and mutation in producing good test cases. In addition, in the data pre-processing process, dependency analysis between tokens or structural analysis of the source code can be carried out to obtain additional information that can be utilized in the test case generation process. This information can help direct the search for genetic algorithms in a more efficient direction and produce test cases that are better at uncovering errors or deficiencies in the program under test.

2.3. Genetic algorithm

The methodology adopted in the research entitled revolves around harnessing the power of genetic algorithms to streamline the process of generating test cases, thereby enhancing software quality. Genetic algorithms, recognized as a global optimization method with strong capabilities in global search compared to other intelligent optimization algorithms, are leveraged to automatically produce a diverse set of test cases that thoroughly exercise software functionality [27]. However, in traditional genetic algorithm optimization processes, parameters remain fixed. This rigidity becomes a hindrance when dealing with groups that continually adapt to external factors. Fixed parameters fail to cater to the dynamic requirements of individuals across various processes, consequently impacting the algorithm's performance and efficiency [28]. Therefore, the methodology adapts by exploring avenues to dynamically adjust parameters to better suit the evolving needs of the software under test. This flexible approach ensures that test cases are tailored to effectively uncover defects and vulnerabilities, ultimately contributing to the overall enhancement of software quality.

Traditional genetic algorithms typically perform the crossover operation before mutation. While this approach may yield favorable results early on, as the optimization progresses, the fitness values of individuals in the population tend to become quite similar [29]. When fitness values are closely clustered, the crossover operation can introduce significant changes in the offspring, making subsequent mutation operations more likely to disrupt highly fit individuals, thereby slowing down convergence. To mitigate this issue, an improved genetic algorithm adjusts the sequence of crossover and mutation operations based on the distribution of individual fitness values within the population. This adaptation aims to maintain population diversity by performing more effective genetic operations in later generations, facilitating better exploration of the search space and potentially leading to improved solutions.

Figure 2 illustrates the genetic operation of the genetic algorithm employed in the research study. This operation encompasses the selection of individuals (test cases), application of genetic operators (mutation and crossover), and the generation of a new population for subsequent generations. By analyzing the distribution of fitness values within the population, the algorithm adjusts the sequence of genetic operations to balance exploration and exploitation, aiming to enhance the effectiveness of test case generation for branch coverage.

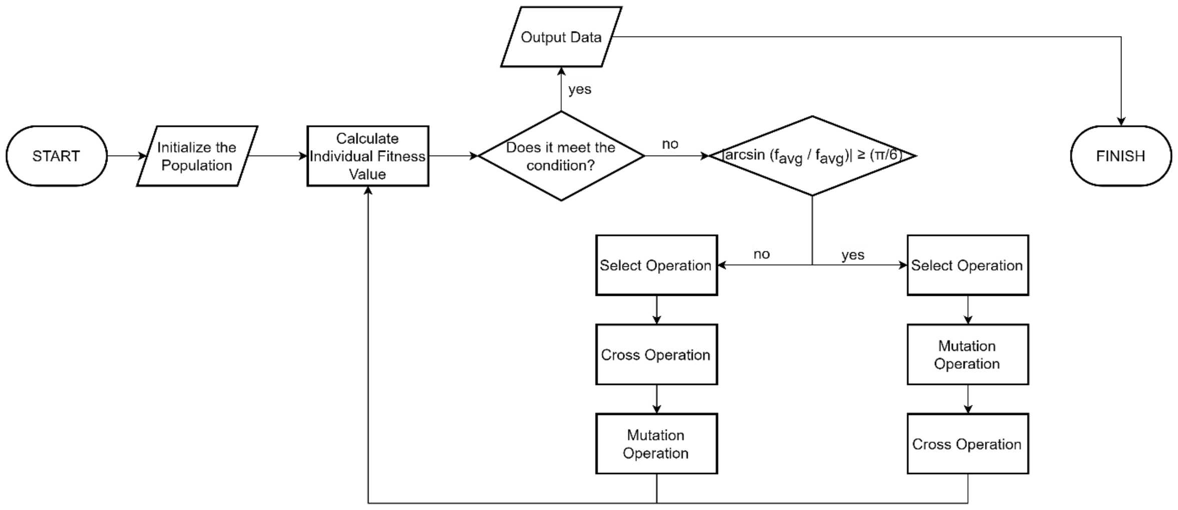

Figure 2. Genetic operation of genetic algorithm [30]

In the next step, the algorithm will decide whether the stop condition has been met or not. If not, the process will continue by choosing which genetic operation to apply, i.e. mutation or crossover. The mutation operation involves random modifications to the selected individuals, while crossover combines the traits of two individuals to produce a new individual. After the genetic operations are performed, a new population will be formed and re-evaluated iteratively until a stopping condition is reached, e.g. maximum number of generations or no significant improvement in fitness. Finally, the best population containing the test cases with the highest branch coverage will be generated as output.

2.4. Evaluation

In this study, we will compare three methods used to generate branch coverage test cases. Firstly, we have the adaptive genetic algorithm-based branch coverage adaptive genetic algorithm, designed to dynamically adapt to environmental conditions and optimize the test case generation process [31]. Secondly, we will evaluate the traditional genetic algorithm, which is a well-established approach but tends to have static parameters and lacks adaptability to environmental changes [17]. Lastly, we will consider the random test generation method, often used as a benchmark to measure the relative performance of more advanced methods [32].

The experiment's evaluation criteria include the average branch coverage rate, providing an overview of how well each method can reach all code branches; the maximum branch coverage rate, indicating each method's ability to find the most challenging code branches to reach; and the average convergence rate, measuring the speed at which each method achieves an optimal solution or approaches an optimal solution in a series of iterations. In the experiment, the standard parameters for all four-branch coverage test case generation methods are established as follows: a population size of 50 and a maximum of 100 evolutionary iterations. To mitigate the impact of randomness on the experiment's outcome, each method will be executed 50 times for every program under scrutiny. The assessment metrics for the experiment include the average branch coverage rate, the maximum branch coverage rate, and the average convergence rate. The average branch coverage rate (Ac) is calculated as in the(5):

$$Ac = \frac{\sum_{1}^{n} t_{(1,2,\ldots,n)}}{n} \quad (1)$$

Average branch coverage rate (Ac) is an important metric that indicates the average branch coverage rate achieved by a program when executed using a particular algorithm. To calculate Ac, branch coverage rate data from each program execution is collected and averaged. The Ac value provides an overview of the effectiveness of the algorithm in achieving good branch coverage. The higher the Ac value, the better the algorithm’s ability to generate test cases that cover most of the branches in the program. However, Ac alone is not enough to assess the overall performance of the algorithm.

The maximum branch coverage rate (MaxC) metric also needs to be considered. MaxC shows the highest level of branch coverage achieved by a program after multiple executions using the same algorithm. MaxC provides insight into the upper limits of an algorithm's ability to achieve branch coverage. By analyzing Ac and MaxC together, developers can gain a more complete understanding of the algorithm's performance in software testing. If Ac is quite high but MaxC is much higher, this indicates that the algorithm has the potential to improve its performance. On the other hand, if Ac and MaxC have relatively close values, then the algorithm has worked quite efficiently in generating test cases that cover most of the program branches.

3. RESULTS AND DISCUSSION

The experimental outcomes present the average branch coverage rate and the maximum branch coverage rate for the five tested programs. Each program was executed 50 times across the four algorithms: PDGA, SGA, Random, and control. Table 2 displays these results obtained from the 50 execution runs per program per algorithm. These outcomes offer insights into algorithm performance across multiple runs, crucial for refining testing methodologies and optimizing algorithmic efficiency in achieving comprehensive branch coverage.

Table 2. Branch coverage experiment results of the three algorithms

	PDGA		SGA		RANDOM	
Program Name	AC (%)	MaxC (%)	AC (%)	MaxC (%)	AC (%)	MaxC (%)
Calculator	100.00	100.00	99.50	99.80	67.39	89.20
Todo List	89.50	100.00	88.20	98.98	65.89	79.99
Weather	92.80	100.00	93.23	100.00	71.89	97.34
Currency Conversion	98.70	100.00	98.52	100.00	69.81	92.12
BMI Calculator	99.90	100.00	94.70	100.00	59.01	83.02

The results presented in Table 2 highlight the superior performance of the PDGA algorithm in achieving higher branch coverage rates across multiple programs. For the "calculator" program, PDGA attained a maximum branch coverage of 100%, whereas the Random algorithm only reached an average of 67.39%. In the "todo list" application, PDGA again achieved a maximum coverage of 100%, while the Random approach failed to surpass 80%, with an average coverage of merely 65.89%.

Similarly, for the "weather" program, both PDGA and SGA reached the maximum 100% branch coverage, outperforming the Random algorithm's maximum of 97.34% and average of 71.89%. The "currency conversion" program exhibited analogous results, with PDGA and SGA achieving the maximum 100% coverage, contrasted by the Random algorithm's maximum of 92.12% and average of 69.81%. For the "BMI calculator" program, both the PDGA and SGA algorithms demonstrated their superiority by achieving the maximum branch coverage of 100%. However, the SGA method exhibited a higher average branch coverage of 94.70% compared to the Random algorithm's average of only 59.01%. This disparity highlights the ability of SGA to consistently outperform the Random approach in terms of average branch coverage.

Notably, in the "calculator" program, the PDGA algorithm not only attained the maximum branch coverage of 100% but also maintained an impressive average branch coverage of 99.50%. This result underscores the stability and robustness of PDGA, particularly in scenarios where the number of branches increases, enabling it to deliver consistently high coverage rates, surpassing the performance of other algorithms. From Figure 3, the PDGA algorithm shows superiority compared to other genetic algorithms, namely SGA and RANDOM, in terms of the average code coverage rate. The PDGA algorithm can achieve a higher level of code coverage than other algorithms in various programs tested, such as calculator, todo list, weather, currency conversion calculator, and BMI calculator. These advantages can be attributed to efficient search strategies, appropriate solution representation schemes, as well as well-designed genetic operators for the automatic test case generation problem in the PDGA algorithm.

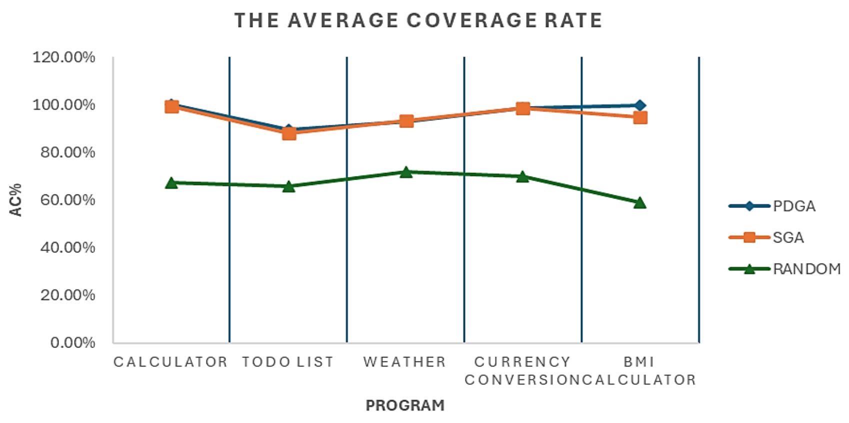

Figure 3. The average coverage rate of the program running in different algorithms

Figure 4 shows the maximum code coverage level achieved by each algorithm in the programs tested. Once again, the PDGA algorithm outperforms the SGA and Random algorithms in achieving the maximum level of code coverage in most of the programs tested. In the "calculator" program, the PDGA algorithm was able to achieve a maximum code coverage rate of 100%, which is a remarkable achievement. These results show that the PDGA algorithm not only excels in average code coverage rate, but also has great potential in generating a set of test cases that cover all branches in the program under test.

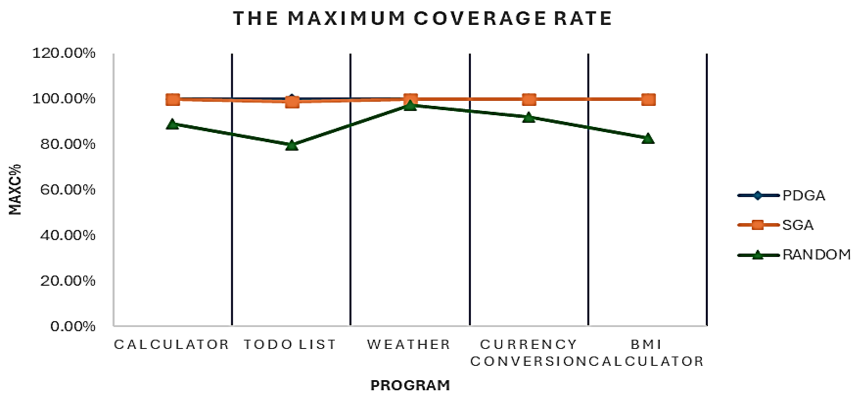

Figure 4. The maximum coverage rate of the program running in different algorithms

The success of the PDGA algorithm in generating high quality automated test cases makes a significant contribution to improving the quality and reliability of software testing. With its ability to achieve high code coverage, these algorithms can help uncover more errors or flaws in software, allowing for more effective fixes before the software is released to the market. Additionally, automated test cases generated by PDGA can be reused in subsequent software development cycles, saving the time and effort required to write test cases manually. Thus, the PDGA algorithm contributes significantly to improving the efficiency and quality of the overall software testing process.

4. CONCLUSION

In terms of automatically generating program branch coverage test cases, this paper proposes comparing 3 genetic algorithms, namely PDGA, SGA, and Random. The results confirm that the PDGA algorithm stands out with superior performance compared to SGA and Random methods in various tested programs. Analysis of the five programs presented shows that PDGA can achieve higher maximum and average coverage, illustrating strong stability and efficiency. For example, in the "calculator" program, the PDGA algorithm even managed to achieve an average coverage of 100%, showing its dominance and robustness in generating automatic test cases when compared with other algorithms. Nevertheless, the SGA algorithm also shows good performance, although not as optimal as PDGA, while the random algorithm shows the lowest performance among the three. Overall, these findings confirm that genetic algorithms, especially PDGA, have great potential to improve the coverage and quality of software testing, presenting exciting prospects for significantly improving system reliability.

REFERENCES

[1] Dr. Naveenkumar Jayakumar, Nilofar Mulla, “Role of machine learning and artificial intelligence techniques in software testing,” *Turkish Journal of Computer and Mathematics Education (TURCOMAT)*, vol. 12, no. 6, pp. 2913–2921, 2021, doi: 10.17762/turcomat.v12i6.5800.

[2] M. Krichen, “How artificial intelligence can revolutionize software testing techniques,” *Innovations in Bio-Inspired Computing and Applications (IBICA 2022)*, pp. 189–198, 2023, doi: 10.1007/978-3-031-27499-2_18.

[3] Z. Khaliq, S. U. Farooq, and D. A. Khan, “Artificial intelligence in software testing: impact, problems, challenges and prospect,” *arXiv-Computer Science*, pp. 1-13, 2022.

[4] P. Singhal, S. Kundu, H. Gupta, and H. Jain, “Application of artificial intelligence in software testing,” *The 2021 10th International Conference on System Modeling and Advancement in Research Trends, SMART 2021*, pp. 489–492, 2021, doi: 10.1109/SMART52563.2021.9676244.

[5] N. Jha and R. Popli, “Artificial intelligence for software testing-perspectives and practices,” *2021 4th International Conference on Computational Intelligence and Communication Technologies, CCICT 2021*, pp. 377–382, 2021, doi: 10.1109/CCICT53244.2021.00075.

[6] Y. Wang, “Software testing resource allocation algorithm based on improved evolutionary algorithm,” *The 8th International Conference on Communication and Electronics Systems, ICCES 2023*, pp. 674–678, 2023, doi: 10.1109/ICCES57224.2023.10192849.

[7] S. Martínez-Fernández *et al.*, “Software engineering for AI-based systems: a survey,” *ACM Transactions on Software Engineering and Methodology*, vol. 31, no. 2, 2022, doi: 10.1145/3487043.

[8] G. J. Myers, T. Badgett, and C. Sandler, “The art of software testing,” *IEEE Proceedings of the National Aerospace and Electronics Conference*, vol. 2, no. 112, pp. 757–760, 1991, doi: 10.1109/naecon.1991.165837.

[9] S. Kumar, “Reviewing software testing models and optimization techniques: an analysis of efficiency and advancement needs,” *Journal of Computers, Mechanical and Management*, vol. 2, no. 1, pp. 32–46, 2023, doi: 10.57159/gadl.jcmm.2.1.23041.

[10] J. Wang, Y. Huang, C. Chen, Z. Liu, S. Wang, and Q. Wang, “Software testing with large language models: survey, landscape, and vision,” *IEEE Transactions on Software Engineering*, vol. 50, no. 4, pp. 911–936, 2024, doi: 10.1109/TSE.2024.3368208.

[11] S. Alagarsamy, C. Tantithamthavorn, and A. Aleti, “A3Test: assertion-augmented automated test case generation,” *arXiv-Computer Science*, pp. 1-18, 2023.

[12] C. Lemieux, J. P. Inala, S. K. Lahiri, and S. Sen, “CodaMosa: escaping coverage plateaus in test generation with pre-trained large language models,” *International Conference on Software Engineering*, pp. 919–931, 2023, doi: 10.1109/ICSE48619.2023.00085.

[13] R. Sheikh, M. I. Babar, R. Butt, A. Abdelmaboud, and T. A. E. Eisa, “An optimized test case minimization technique using genetic algorithm for regression testing,” *Computers, Materials and Continua*, vol. 74, no. 3, pp. 6789–6806, 2023, doi: 10.32604/cmc.2023.028625.

[14] A. W. Wambua and G. M. Wambugu, “A comparative analysis of bat and genetic algorithms for test case prioritization in regression testing,” *International Journal of Intelligent Systems and Applications*, vol. 15, no. 1, pp. 13–21, 2023, doi: 10.5815/ijisa.2023.01.02.

[15] B. Alhijawi and A. Awajan, “Genetic algorithms: theory, genetic operators, solutions, and applications,” *Evolutionary Intelligence*, vol. 17, no. 3, pp. 1245–1256, 2024, doi: 10.1007/s12065-023-00822-6.

[16] R. R. Chandan *et al.*, “Genetic algorithm and machine learning,” *Advanced Bioinspiration Methods for Healthcare Standards, Policies, and Reform*, IGI Global, pp. 167–182, 2023, doi: 10.4018/978-1-6684-5656-9.ch009.

[17] M. Rajagopal, R. Sivasakthivel, K. Loganathan, and L. E. Sarris, “An automated path-focused test case generation with dynamic parameterization using adaptive genetic algorithm (AGA) for structural program testing,” *Information*, vol. 14, no. 3, 2023, doi: 10.3390/info14030166.

[18] W. Wang, S. Wu, Z. Li, and R. Zhao, “Parallel evolutionary test case generation for web applications,” *Information and Software Technology*, vol. 155, 2023, doi: 10.1016/j.infsof.2022.107113.

[19] M. A. Anuar, M. Z. Sahid, and N. Zainal, “Comparative analysis of test case prioritization using ant colony optimization algorithm and genetic algorithm,” *Journal of Soft Computing and Data Mining*, vol. 4, no. 2, pp. 52–58, 2023, doi:

10.30880/jscdm.2023.04.02.005.
[20] T. Abbas, “Optimizing software performance and bug detection: genetic algorithm-enhanced time convolution neural networks (GA-TCN),” *Easy Chair*, pp. 1-7, 2024.
[21] S. T. Cynthia, B. Roy, and D. Mondal, “Feature transformation for improved software bug detection models,” *ACM International Conference Proceeding Series*, 2022, doi: 10.1145/3511430.3511444.
[22] S. S. Sankar and S. S. V. Chandra, “An ant colony optimization algorithm based automated generation of software test cases,” *Advances in Swarm Intelligence (ICSI 2020)*, pp. 231–239, 2020, doi: 10.1007/978-3-030-53956-6_21.
[23] S. Ji, S. Zhu, P. Zhang, H. Dong, and J. Yu, “Test-case generation for data flow testing of smart contracts based on improved genetic algorithm,” *IEEE Transactions on Reliability*, vol. 72, no. 1, pp. 358–371, 2023, doi: 10.1109/TR.2022.3173025.
[24] S. Chakraborty, V. B. Gujar, T. Choudhury, and B. K. Dewangan, “Improving software performance by automatic test cases through genetic algorithm,” *International Journal of Computer Applications in Technology*, vol. 68, no. 3, pp. 228–234, 2022, doi: 10.1504/ijcat.2022.124946.
[25] M. Alshammari, M. A. Mezher, and K. Al-Utaibi, “Automatic test data generation using genetic algorithm for python programs,” *Proceedings of 2022 2nd International Conference on Computing and Information Technology, ICCIT 2022*, pp. 197–205, 2022, doi: 10.1109/ICCIT52419.2022.9711607.
[26] S. Katoch, S. S. Chauhan, and V. Kumar, “A review on genetic algorithm: past, present, and future,” *Multimedia Tools and Applications*, vol. 80, no. 5, pp. 8091–8126, 2021, doi: 10.1007/s11042-020-10139-6.
[27] D. Liu, “Mathematical modeling analysis of genetic algorithms under schema theorem,” *Journal of Computational Methods in Sciences and Engineering*, vol. 19, no. S1, pp. S131–S137, 2019, doi: 10.3233/JCM-191019.
[28] Y. Fang, X. Xiao, and J. Ge, “Cloud computing task scheduling algorithm based on improved genetic algorithm,” *Proceedings of 2019 IEEE 3rd Information Technology, Networking, Electronic and Automation Control Conference, ITNEC 2019*, pp. 852–856, 2019, doi: 10.1109/ITNEC.2019.8728996.
[29] A. Gerami Matin, R. Vatani Nezafat, and A. Golroo, “A comparative study on using meta-heuristic algorithms for road maintenance planning: Insights from field study in a developing country,” *Journal of Traffic and Transportation Engineering (English Edition)*, vol. 4, no. 5, pp. 477–486, 2017, doi: 10.1016/j.jtte.2017.06.004.
[30] X.-S. Yang, “Genetic algorithms,” *Nature-Inspired Optimization Algorithms*, Oxford, United Kingdom: Academic Press, pp. 91–100, 2021.
[31] X. Bao, Z. Xiong, N. Zhang, J. Qian, B. Wu, and W. Zhang, “Path-oriented test cases generation based adaptive genetic algorithm,” *PLoS ONE*, vol. 12, no. 11, 2017, doi: 10.1371/journal.pone.0187471.
[32] S. Lukasczyk, F. Kroiß, and G. Fraser, “An empirical study of automated unit test generation for Python,” *Empirical Software Engineering*, vol. 28, no. 2, 2023, doi: 10.1007/s10664-022-10248-w.

BIOGRAPHIES OF AUTHORS

Tiara Rahmania Hadiningrum successfully earned a bachelor's degree (S. Kom.) in information systems from Telkom University in 2023. She is currently studying for a master's degree at the Department of Information Engineering, Institut Teknologi Sepuluh Nopember. Her research interests include aspects of software quality, traceability, and testing. She can be contacted at email: 6025231079@student.its.ac.id.

Siti Rochimah successfully earned a doctoral degree (Ph.D.) in software engineering from Universiti Teknologi Malaysia in 2010. Currently, she serves as the head of the Software Engineering Laboratory at the Department of Informatics Engineering, Institut Teknologi Sepuluh Nopember. His work involves writing more than 80 articles related to software engineering. Her research interests include aspects of software quality, traceability, and testing. She can be contacted at email: siti@its.ac.id.