ISSN: 2633-4828 Vol. 6 No. 3, September, 2024

International Journal of Applied Engineering & Technology

Porting And Enhancing Genetic Programming For Automatic Program Repair In Python

Natte Sai Bharath[1], Raja Pavan Karthik[2], Sista Sai Subrahmanya Mrinaal[3], Dr. Swaminathan J[4]
[1,2,3,4] Dept. Of Computer Science (Artificial Intelligence), Amrita Vishwa Vidyapeetham, Vengal, Tamil Nadu, India
Email: [1] amenu4aie20052@am.students.amrita.edu, [2] amenu4aie20060@am.students.amrita.edu, [3] amenu4aie20067@am.students.amrita.edu, [4] swaminathanj@am.amrita.edu

ABSTRACT

In the realm of software development, there is a growing need for efficient and automated solutions to address programming errors. This paper examines the utilization of genetic programming techniques in combination with established error fix approaches for automated programming repair, with a

targeted emphasis on Python syntax correction. Our primary objective is to seamlessly integrate error detection mechanisms with automatic repair processes, leveraging a unified genetic algorithm while tailoring settings and parameters to address various types of errors, thus presenting a novel and versatile approach to enhancing code quality. We have worked on different types of Python syntax errors that includes Indentation, Parenthesis, Braces, Arithmetic operators, Commas. By meticulously analyzing these errors and how they can be rectified, our research not only provides a comprehensive understanding of these methods in the field of Automatic Program Repair. In addition, we have developed a user-friendly interface where users can seamlessly select the type of error and input the erroneous code by pasting it into a designated window. Subsequently, the repaired version of the code is displayed in another window. We've compiled a diverse dataset of 400 Python code samples, manually crafted to include various syntax errors, Among them, 291 were successfully repaired, demonstrating a commendable patch fix rate. This study illuminates how the fusion of genetic programming, traditional error fixing methods, and user interface design holds promising potential for advancing the field of automatic programming repair, offering new ways for creating efficient and error-free Python code.

Index Terms – APR (Automatic Program Repair), GP (Genetic Programming), Python, Error

INTRODUCTION

Automated program repair (APR) has emerged as a promising approach to alleviate the burden of manual debugging and code maintenance. In software development, bugs are inevitable, and their timely resolution is crucial for ensuring the reliability and functionality of software systems. However, traditional debugging methods are often time-consuming and error-prone, requiring extensive human intervention to identify and rectify code faults. This inefficiency incurs significant costs in terms of both time and resources, hindering the overall productivity and competitiveness of software development endeavors. Amidst these challenges, the application of genetic programming [1] (GP) techniques to automate program repair has garnered substantial attention from researchers and practitioners alike. GP, a sub field of evolutionary computation, employs principles inspired by biological evolution to evolve solutions to complex problems.

The popularity of Python in various domains, ranging from web development to data science, underscores the importance of efficient bug-fixing mechanisms tailored specifically for this language. Furthermore, Python's dynamic and interpreted nature introduces unique challenges and opportunities for APR, necessitating innovative approaches to address its peculiarities effectively. The motivation behind focusing on Python program repair lies in the pressing need for automated tools capable of swiftly and accurately diagnosing and rectifying bugs in Python code bases. As the demand for Python-based software continues to surge across industries, the imperative to streamline the debugging process becomes increasingly pronounced.

Our research objective in the context of Python-specific automated program repair using genetic programming (GP) [2] is to develop efficient and accurate techniques for automatically detecting and repairing defects in Python code. Specifically, we aim to achieve the following objectives. We focus on precise defect localization within Python

programs. Our research aims to identify the exact locations (lines or blocks) where bugs occur. By leveraging GP, we intend to create tools that can pinpoint the root cause of Python-specific issues. Our objective is to generate high-quality patches that address the identified defects. These patches should be syntactically correct, maintain program semantics, and adhere to Python coding conventions. We explore how GP can evolve patches that not only fix the reported bugs but also enhance overall code quality. Python has unique features such as indentationbased scoping, dynamic typing, and duck typing. Our research considers these idiosyncrasies. We aim to create repair techniques that handle Python-specific constructs gracefully.

We will concentrate on identifying and addressing common bug patterns specific to Python. These may include issues related to dynamic typing, list comprehensions, indentation errors, and module imports. Our research will explore how GP can effectively generate patches for these Python-specific issues. Our scope involves designing a suitable tree-based representation for Python code. This representation should capture Python's syntax and semantics accurately. We'll explore how to handle Python-specific constructs like decorators, context managers, and generator expressions within the GP framework. We'll investigate how GP can collaborate with existing Python static analysis tools (e.g., PyLint, Pyflakes, or mypy). Our objective is to enhance defect localization by combining GP-based repair with insights from static analysis results. Python programs often rely on external libraries and modules.

Genetic Programming (GP) is chosen for certain problem domains, including automatic program repair in Python, due to its unique characteristics and advantages. GP explores a vast solution space by representing candidate solutions as trees. This flexibility is beneficial when searching for correct and optimized program structures. In GP, programs are represented as abstract syntax trees (AST), which aligns well with the structure of programming languages. This representation allows for natural manipulation and transformation of code. GP is known for its ability to handle complex problems. Program repair often involves dealing with intricate code structures and dependencies, and GP's adaptability to complexity is an advantage in such scenarios. GP is versatile and can be adapted to various problem domains. It has been successfully applied to different areas of computer science, making it a candidate for a wide range of applications, including program repair. GP can incrementally evolve solutions and learn from the evolving population over generations. This can be advantageous in scenarios where the correction of one part of the code affects the correction of another, leading to a more coherent and integrated repair process. While GP has its strengths, it's important to note that the choice of the algorithm depends on the specific characteristics of the problem at hand, and different approaches may be more suitable for different scenarios.Genetic programming, a subset of evolutionary algorithms, presents a novel approach to addressing the challenge of automated syntax error correction in Python code. At its core, genetic programming harnesses principles inspired by biological evolution [3] to iteratively refine and improve code structures. This iterative process, known as the Evolution Process, unfolds through several key steps, each mimicking a facet of natural selection and genetic variation.

The first step in the Evolution Process is Selection. Here, a population of candidate solutions, each representing a potential code structure, is subjected to a selection process akin to natural selection. Individuals with higher fitness values, indicating their efficacy in correcting code errors, are favoured for advancement to the next generation. This step ensures that the subsequent generations are populated by code structures with greater potential for effective error correction.

Following selection comes Crossover [4] , which simulates genetic recombination. Pairs of code structures are selected, and segments of their Abstract Syntax Trees (ASTs) are exchanged. This process introduces genetic diversity into the population by combining advantageous features from different solutions. Crossover serves as a mechanism for exploring new code combinations that may lead to improved error correction strategies. Crossover and mutation are fundamental mechanisms driving genetic programming's ability to explore and exploit the solution space effectively. Crossover facilitates the exchange of genetic material between code structures, enabling the propagation of beneficial traits throughout the population. This process encourages the convergence towards

optimal solutions by combining the strengths of different individuals. On the other hand, mutation introduces stochasticity and diversity into the population by randomly altering individual code structures. This stochastic exploration allows the algorithm to search for novel solutions beyond the scope of the current population. Mutation serves as a mechanism for maintaining genetic diversity and preventing premature convergence, thereby enhancing the algorithm's ability to adapt to changing problem landscapes. In addition to crossover, Mutation [5] plays a crucial role in introducing variation within the population. Small, random changes are made to individual code structures, allowing for the exploration of novel solutions that may offer better error-correction capabilities. Mutation serves as a mechanism for adaptive exploration, enabling the algorithm to escape local optima and discover more effective code corrections. After the application of crossover and mutation, the fitness of each code structure is evaluated. Individuals with higher fitness scores, indicative of their proficiency in correcting code snippets, are retained for the next generation. This iterative process of selection, crossover, mutation, and fitness evaluation continues until satisfactory code corrections are achieved, or a termination criterion is met.

LITERATURE SURVEY

Numerous studies have delved into the application of genetic programming and automated techniques to transform Python programs into more efficient versions through automatic program repair. Despite these challenges, the field of automatic program repair continues to evolve, offering promising solutions for software development.

The study by L. Gazzola, D. Micucci, et al [6] illustrates the algorithms and approaches used in automatic software repair and compares them on representative examples. It discusses the empirical evidence reported so far in the field of automatic software repair. The problem addressed in the paper is the need for efficient techniques to repair and maintain software applications. The general applicability of automatic program repair techniques in industrial settings still needs to be demonstrated. The paper does not address the scalability and efficiency challenges associated with automatic software repair techniques. Need for more empirical evidence to demonstrate the effectiveness of these techniques in real-world industrial scenarios.

Claire Le Goues, Michael Pradel, et al [7] outlined a problem of manually fixing programming mistakes and the burden it places on programmers and as well as technical challenges in scalability, patch quality, and integration into developer workflows in the context of automated program repair. They discusses the difficulty of automated program repair in complex software projects that contain legacy code. Performancerelated repair tools, such as MemoizeIt and Caramel, suggest code modifications to improve application-level caching and avoid repeated computations. It does not provide a detailed discussion of the limitations of the current state-of-the-art tools and techniques and explicitly mention any limitations related to the quality of repairs or the scope of problems addressed by repair. Developing learning-based approaches that can overcome the challenges of source code changes and obtaining high-quality human patches.

D.Li, W.E.Wong, M.Jian, et al [8] in their work delve into the realm of automated bug repair in software programs, aiming to diminish debugging expenses and enhance program quality. While existing techniques like GenProg have made strides, particularly for Java programs, Their paper introduces an enhanced automated patch generation tool tailored for Java programs, leveraging genetic programming and sequence-tosequence learning for patch identification. Despite its advancements, the proposed framework, ARJANMT, falls short in providing an exhaustive analysis of repair quality and integration into the development pipeline. Moreover, they highlight the need for performance-centric tools like MemoizeIt and Caramel to optimize code efficiency, along with a deeper exploration of the human-centric implications of automated patches on code maintenance and overall maintainability.

Y.Qi, X.Mao, Y.Lei et al [9] in their paper undertake an exploration into the efficacy of genetic programming (GP) compared to random search algorithms in the context of guiding the patch generation process for automated program repair. They aim to evaluate whether GP outperforms random search in the generation of patches for faulty programs. However, the authors noted the absence of a fitness evaluation mechanism in the random search algorithm, potentially limiting the selection of optimal patches for subsequent generations. Notably, the research

primarily focuses on legacy C programs lacking formal specifications, which may restrict the generalizability of the findings to other programming languages or program types. The authors advocate for broader experimentation across various program types and languages to gauge the transferability of the results and comprehend the influence of different program characteristics on the effectiveness of GP and random search algorithms.

Geunseok Yang, Youngjun Jeong, et al [10] introduces an innovative approach to automatic fault repair through genetic programming (GP), leveraging similar bug fix information.

Acknowledging the inherent complexity of software products, the authors emphasize the pressing need to alleviate the manual burden on developers and reduce associated costs in bug resolution. The proposed method intricately involves searching for comparable buggy source codes, identifying a related fixed code, and subsequently transforming it into abstract syntax trees for application in GP. Candidate program patches are generated through this process and subjected to validation via a fitness function based on provided test cases. This rigorous verification mechanism ensures the credibility and effectiveness of the proposed patches. Ultimately, the paper successfully contributes to the field by providing a systematic method for automatically generating program patches to rectify new instances of buggy code, offering a promising avenue for advancing automated fault repair in software development.

The paper by claire [11] addresses the critical issue of software quality, shedding light on the economic costs incurred by mature software projects shipping with known and unknown bugs. Enter GenProg, a groundbreaking approach introduced in the paper, leveraging genetic programming for the automatic patching of bugs in deployed and legacy software. GenProg is strategically guided by test cases and domain-specific operators, ensuring scalable, expressive, and high-quality automated repair. Empirical evidence supports the effectiveness of GenProg in rectifying various bug types across diverse programs, with repairs comparable to those executed by human developers. The paper's valuable contribution extends to the provision of new benchmark sets of real bugs and experimental frameworks, enriching the evaluation landscape for automated repair techniques.

Stephanie Forrest, ThanhVu Nguyen, et al [12] in their research combines genetic programming (GP) with program analysis methods to repair bugs in off-the-shelf legacy C programs. The paper introduces modifications to the GP technique, such as localizing genetic operations to the nodes along the execution path of the negative test case and representing high-level statements as single nodes in the program tree.The paper does not discuss the limitations or potential challenges of applying the genetic programming approach to different types of bugs or programming languages. The paper does not address the potential impact of the automated software repair process on the overall quality and maintainability of the repaired code. The paper suggests developing a generic set of repair templates to provide the genetic programming (GP) approach with a source of new code for mutation, beyond the statements present in the program being repaired.

Limitations include challenges in dealing with a wide range of Python program errors, occasional anomalies such as incorrect fixes or inefficient solutions, and the necessity for ongoing research to fine-tune algorithms and enhance the quality of repairs. Collectively, these studies contribute to the dynamic field of automated program repair techniques, each with its unique strengths, limitations, and a common objective of increasing efficiency and ease-of-use in applying genetic programming to repair Python programs.

METHODOLOGY

The methodology begins by defining a sample code snippet containing a specific type of syntax error. It then implements a genetic algorithm that operates on a population of such code snippets. Each individual in the population represents a potential solution to the syntax error. The algorithm iterates through generations, evaluating the fitness of each individual, selecting parents for crossover [13] , and applying mutation operators to introduce variations. The process continues until a solution with satisfactory fitness (a syntactically correct code snippet) is found, or a maximum number of generations is reached. We start by defining a population of candidate solutions, each representing a piece of Python code with a specific syntax error. Evaluate the fitness of each candidate solution

by checking if the code contains any syntax errors. This evaluation is typically performed using the ast.parse() function, which attempts to parse the code and raises a Syntax Error if any errors are encountered. Genetic operators applied, including mutation and crossover, to create new candidate solutions from the existing population. Mutation introduces random changes to individual solutions, while crossover combines features from two-parent solutions to produce offspring. Individuals are selected from the population to serve as parents for the next generation based on their fitness scores. This process often involves techniques like tournament selection or roulette wheel selection. The algorithm continues iterating through generations until a termination criterion is met, such as finding a solution with zero syntax errors or reaching a maximum number of generations.

GENETIC ALGORITHM FOR PYTHON

For more intricate indentation problems, the methodology incorporates the use of a Genetic Algorithm (GA) [14]. The process begins with the initialization of a population of code samples derived from the provided sample code. Each sample undergoes a fitness evaluation, wherein the checksyntax function is applied to determine its error status. Through genetic operations such as crossover, mutation, and selection, the algorithm explores different correction strategies. Crossover combines two samples to create offspring, mutation introduces random alterations, and selection ensures the propagation of fit samples while discarding unfit ones. This iterative process continues until a flawless code sample is obtained or a predefined termination criterion, such as a maximum number of generations, is met. The fix code function is responsible for fixing indentation errors and removing extra spaces in the provided code snippet. It works by dividing the code snippet into individual lines and then handles each line. For each line, leading and trailing white space is stripped. The function keeps track of the expected indentation level based on Python's syntax rules. If a line starts with certain keywords like if, elif, else, for, while, def, or class, it adjusts the indentation level by adding spaces based on the expected indentation. If a line starts with return, it reduces the expected indentation level. It also

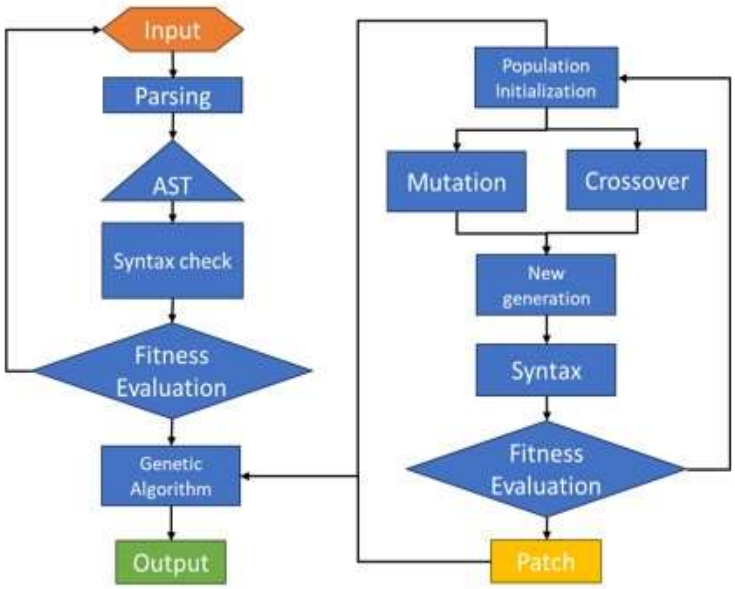

Fig. 1: Work flow of Algorithm

corrects print (to print (to follow the correct Python syntax. The corrected lines are then joined to form the fixed code snippet. The mutate(code) [15] function introduces a small random change into the given code snippet. It randomly selects a line in the code and either adds or removes indentation based on the presence of the print (sub string. This could potentially fix an indentation error, enhancing the correctness of the code.

A. Indentation Fixer

This section focuses on fixing incorrect indentation in Python code. It utilizes the genetic algorithm framework described above, with specific attention to adjusting indentation levels based on control flow statements like if, for, while, etc. it is adapted to address the specific task of correcting indentation errors. It identifies misaligned code blocks and applies mutations and crossovers to adjust their indentation levels until the syntax error is resolved.

B. Parenthesis Fixer

Here, the emphasis is on correcting syntax errors caused by missing closing parentheses. The genetic algorithm targets the insertion of the missing closing parenthesis at appropriate positions within the code. It is specifically designed to address the syntax error of missing closing parentheses. It randomly selects a position in the code and inserts a closing parenthesis to fix the error. The fitness function evaluates the correctness of the code by parsing it using ast.parse(). If the parsing succeeds without raising a Syntax Error, the code is considered correct, and it receives a fitness score indicating maximum fitness (max size). Otherwise, it receives a fitness score of 0.

C. Bracket Fixer

This part targets syntax errors arising from missing closing brackets. The genetic algorithm is adapted to insert the missing closing bracket in lists or tuples. The mutate() function is designed to insert a closing bracket (]) at a random position within the code, specifically where a comment (#) is found. This approach is tailored to address the unique syntax error of missing closing brackets in bracketed expressions.

D. Arithmetic Fixer

It addresses syntax errors due to missing arithmetic operators, such as +, -, *, or /, between operands. The genetic algorithm aims to insert the missing operator at suitable locations within the code. It utilizes a mutation operator that randomly inserts one of the arithmetic operators (+, -, *, /) at a random position within the code.

E. Colon Fixer

This section focuses on fixing incorrect keyword usage in Python code, such as missing colons (:) after for or if statements. The genetic algorithm is tailored to insert the missing colons at the appropriate positions. The mutation operator in this code section differs significantly from others. Instead of inserting or modifying characters directly, it replaces specific phrases in the code with corrected versions. For instance, it replaces for i in range(10) with for i in range(10): and if i >5 with if i >5:. This approach is tailored to address the specific nature of keyword-related syntax errors. Fitness is assessed based on the syntactic correctness of the code after applying mutations. The goal is to ensure that corrected versions of the code contain the necessary colons (:) after for and if statements, thus adhering to Python's syntax rules.

F. Grammar Fixer

Lastly, it tackles syntax errors caused by missing commas in lists, tuples, or function arguments. The genetic algorithm is adjusted to insert the missing commas to ensure syntactic correctness. The mutation operator in Code 6 is tailored to insert missing commas at appropriate positions within the code. It replaces instances of space-separated elements in lists or tuples with comma-separated elements. Additionally, it inserts commas between function arguments where they are missing. The fitness function evaluates the syntactic correctness of the code, similar to other sections. However, it specifically checks for the absence of syntax errors related to missing commas.

EXPERIMENTAL RESULTS

The genetic algorithm for Indent Fixer successfully corrected indentation errors in several code snippets, aligning statements properly according to Python syntax rules. These corrections included Proper indentation of loops (for and while), conditional statements (if), and function definitions, Fixing indentation inconsistencies within multi line structures like lists and dictionaries. On the other-hand Colon Fixer was able to successfully correct the syntax

errors in cases where missing colons were the primary issue, such as after for loops and if statements. Simple errors, like missing colons, were efficiently fixed, resulting in syntactically valid Python code. An arithmetic operator error in Python can critically impact the accuracy and reliability of computational results. We have a Arithmetic Fixer where it works by iteratively

Fig. 2: An error program before and after fix

mutating the code and introducing potential operators, the algorithm effectively resolved the issues, ensuring that the corrected code maintained syntactic correctness and logical coherence. The corrected versions demonstrate the ability of the algorithm to identify and rectify various types of missing operators, including addition, subtraction, multiplication, and division, thus enhancing the readability and functionality of the code. Where as the genetic algorithm approach for comma fixer is to fix missing commas in Python code shows limited success. It successfully fixed some simple cases such as missing commas in function arguments. However, it failed to address more complex scenarios such as dictionaries, sets, and class definitions. This failure highlights the challenges of using a genetic algorithm for this specific problem, including the need for a more sophisticated mutation operator and a deeper understanding of Python syntax and context.

This issue becomes more challenging to detect and fix in large code bases where the error may be buried deep within multiple files and functions. These are simple errors, such as missing closing parentheses, Using the Parenthesis Fixer the errors were addressed efficiently. However, more complex errors, such as those within loops or function definitions, proved challenging for the algorithm to resolve within the given constraints. Performance was influenced by parameters such as population size, mutation rate, and crossover strategy. Adjusting these parameters could potentially enhance the algorithm's ability to handle a wider range of syntax errors. Similarly the genetic algorithm for Bracket Fixer successfully fixed syntax errors in all provided scenarios. It efficiently identified the position where the closing bracket was missing and inserted it, thereby correcting the syntax error. The algorithm demonstrates its ability to handle various types of missing closing bracket errors in Python code, including lists and nested lists. These results highlight the effectiveness and versatility of the genetic algorithm approach in automatically correcting syntax errors in Python code, providing valuable insights for automated code repair techniques.

CONCLUSION

The findings underscore the importance of evolutionary techniques in code maintenance and debugging. By automating the process of identifying and rectifying syntax errors, developers can concentrate their efforts on more complex tasks. This approach increases productivity and reduces the likelihood of errors in software development projects. In closing, the success of these code snippets in fixing syntax errors underscores the promise of evolutionary approaches in automated code repair.

As we continue to refine and expand upon these techniques, we move closer to creating robust and efficient tools for code maintenance and debugging. This research is a significant step forward in the field of software engineering, bringing us closer to the goal of automated, reliable, and efficient code repair. In conclusion, while the genetic algorithm demonstrated promise in addressing certain types of syntax errors, particularly those related to simple indentation issues, its effectiveness was limited when confronted with more complex code structures and intertwined syntax errors.

The algorithm struggled to resolve issues such as missing quotes in multi line string definitions and improper method indentation within class definitions. These challenges underscored the need for enhancements in mutation

and crossover operators to tackle a broader range of indentation issues and the incorporation of additional heuristics or parsing techniques to identify and rectify complex syntax errors. Furthermore, the algorithm encountered difficulties in resolving instances of missing operators, particularly within loops and nested expressions, where the context of the error was critical for determining the correct fix. Its reliance on random mutations and a finite search space sometimes led to sub-optimal solutions or failure to converge within predefined parameters. Despite these limitations, several avenues for improvement and future research directions emerge. Enhanced search strategies, such as guided mutation or crossover operations based on domainspecific knowledge or heuristics, could steer the algorithm toward more promising solutions and accelerate convergence.

Additionally, the development of techniques for context-aware fixing, analyzing the surrounding code context and semantics, could improve the algorithm's ability to generate appropriate fixes for complex scenarios involving nested expressions and conditional statements. Integration with static analysis techniques or code parser could provide valuable insights into code structure and semantics, enabling more informed mutation decisions and error detection. Parallelization and optimization techniques could also expedite the search process and enhance the scalability of the genetic algorithm, making it more suitable for handling larger code bases and intricate code structures. However, it's essential to acknowledge that genetic algorithms may not always be the optimal choice for addressing syntax errors in code. Alternative approaches, such as static analysis, parsing, and rule-based correction, may yield better results in certain contexts. For instance, leveraging parser to analyze code structure and contextually determine where syntax errors occur could provide a more accurate and efficient solution than the simplistic mutation approach of genetic algorithms. Therefore, future research efforts should explore a combination of these approaches to develop robust and versatile automated code correction tools.

REFERENCES

[1] F. Y. Assiri and J. M. Bieman, "An Assessment of the Quality of Automated Program Operator Repair," 2014 IEEE Seventh International Conference on Software Testing, Verification and Validation, Cleveland, OH, USA, 2014, pp. 273-282, doi: 10.1109/ICST.2014.40.

[2] Wirsansky, Eyal. Hands-on genetic algorithms with Python: applying genetic algorithms to solve real-world deep learning and artificial intelligence problems. Packt Publishing Ltd, 2020.

[3] K. Harsha Saketh and Dr. Jeyakumar G., "Comparison of Dynamic Programming and Genetic Algorithm Approaches for Solving Subset Sum Problems", Computational Vision and Bio-Inspired Computing, vol. 1108. Springer International Publishing, Cham, 2020.

[4] Dhanya M. Dhanalakshmy, Pranav, P., and Dr. Jeyakumar G., "A Survey on Adaptation Strategies for Mutation and Crossover Rates of

[5] Differential Evolution Algorithm", International Journal on Advanced Science, Engineering and Information Technology (IJASEIT) (Scopus), vol. 6, no. 5, pp. 613–623, 2016.

[6] A. M. and Dr. Jeyakumar G., "Performance Enhancement of Differential Evolution Algorithm with Modified Mutation and Crossover Components", Proceedings of 4th IEEE International Conference on Computing Methodologies and Communication (ICCMC), Erode. pp. 13-20, 2020.

[7] L. Gazzola, D. Micucci and L. Mariani, "Automatic Software Repair: A Survey" in IEEE Transactions on Software Engineering, vol. 45, no. 01, pp. 34-67, 2019. doi: 10.1109/TSE.2017.2755013

[8] Claire Le Goues, Michael Pradel, and Abhik Roychoudhury. 2019.

[9] Automated program repair. Commun. ACM 62, 12 (December 2019), 56–65. https://doi.org/10.1145/3318162

[10] D. Li, W. E. Wong, M. Jian, Y. Geng and M. Chau, "Improving SearchBased Automatic Program Repair With Neural Machine Translation," in IEEE Access, vol. 10, pp. 51167-51175, 2022, doi: 10.1109/ACCESS.2022.3164780.

[11] Y. Qi, X. Mao, Y. Lei, Z. Dai and C. Wang, "Does Genetic Programming Work Well on Automated Program Repair?," 2013 International Conference on Computational and Information Sciences, Shiyang, China, 2013, pp. 1875-1878, doi: 10.1109/ICCIS.2013.490.

[12] Yang G, Jeong Y, Min K, Lee J-w, Lee B. "Applying Genetic Programming with Similar Bug Fix Information to Automatic Fault Repair". Symmetry. 2018; 10(4):92. https://doi.org/10.3390/sym10040092

[13] Goues, C.L. (2013) "Automatic program repair using genetic programming". Available at: http://www.cs.cmu.edu/ clegoues/docs/clairedissertation.pdf (Accessed: 15 April 2024).

[14] Stephanie Forrest, ThanhVu Nguyen, Westley Weimer, and Claire Le Goues. 2009. "A genetic programming approach to automated software repair". https://doi.org/10.1145/1569901.1570031

[15] S. Abhishek, Emmanuel, S. Coreya, Rajeshwar, G., and Dr. Jeyakumar G., "A Genetic Algorithm Based System with Different Crossover Operators for Solving the Course Allocation Problem of Universities", New Trends in Computational Vision and Bio-inspired Computing: Selected works presented at the ICCVBIC 2018, Coimbatore, India. Springer International Publishing, Cham, pp. 149 - 160, 2020.

[16] A. M. and Dr. Jeyakumar G., "Performance Enhancement of Differential Evolution Algorithm with Modified Mutation and Crossover Components", Proceedings of 4th IEEE International Conference on Computing Methodologies and Communication (ICCMC), Erode. pp. 13-20, 2020.

[17] D. Srivatsa, Teja, T. P. V. Kris, Prathyusha, I., and Dr. Jeyakumar G., "An Empirical Analysis of Genetic Algorithm with Different Mutation and Crossover Operators for Solving Sudoku", in Pattern Recognition and Machine Intelligence, Cham, 2019.