VOL. 10. NO. 3 FEBRUARY 2025
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480 /jitk.v10i3.6177

JITK (JURNAL ILMU PENGETAHUAN DAN TEKNOLOGI KOMPUTER)

DEVELOPMENT OF GRAPH GENERATION TOOLS FOR PYTHON FUNCTION CODE ANALYSIS

Bayu Samodra[1*]; Vebby Amelya Nora[2]; Fitra Arifiansyah[3]; Gusti Ayu Putri Saptawati[4]; Muhamad Koyimatu[5]

School of Electrical Engineering and Informatics[1, 2, 3, 4, 5]
Institut Teknologi Bandung, Bandung, Indonesia[1, 2, 3, 4, 5]
www.itb.ac.id[1, 2, 3, 4, 5]
23523307@std.stei.itb.ac.id[1*], 23523310@std.stei.itb.ac.id[2], fitra@staff.stei.itb.ac.id[3], putri@staff.stei.itb.ac.id[4], koyimatu@staff.stei.itb.ac.id[5]

(*) Corresponding Author
(Responsible for the Quality of Paper Content)

Abstract—*The increasing complexity of programs in software development requires understanding and analysis of code structure, especially in Python, which dominates machine learning and data science applications. Manual static analysis is often time-consuming and prone to errors. Meanwhile, static analysis tools for Python, like PyCG and Code2graph, are still limited to generating call graphs without including dependency and control flow analysis. This research addresses these shortcomings by proposing the development of a web-based tool that integrates the generation of function call graphs, function dependency graphs, and control flow graphs using Abstract Syntax Tree (AST), Graphviz, and Streamlit. With an iterative SDLC methodology, this tool was developed gradually to visualize Python function code as a heterogeneous graph. Evaluation of 11 Python function codes showed a success rate of 95.45% in analyzing and visualizing Python function codes with various levels of complexity. The limitations of Graphviz present an opportunity for future research to focus on improving scalability and Python code analysis.*

Keywords: *abstract syntax tree, control flow graph, graphviz, python, SDLC.*

Intisari—*Meningkatnya kompleksitas program dalam pengembangan perangkat lunak membutuhkan pemahaman dan analisis pada struktur kode, khususnya pada Python yang mendominasi aplikasi pembelajaran mesin, dan data sains. Analisis statis manual sering kali memakan waktu dan rentan terhadap kesalahan. Sedangkan, alat analisis statis untuk Python seperti PyCG dan Code2graph masih terbatas hanya pembangkitan graf pemanggilan fungsi tanpa disertai analisis dependensi dan aliran kontrol. Penelitian ini menjawab kekurangan tersebut dengan mengusulkan pengembangan alat berbasis website yang mengintegrasikan pembangkitan graf pemanggilan fungsi, graf keterkaitan fungsi dan graf aliran kontrol menggunakan Abstract Syntax Tree (AST), Graphviz dan Streamlit. Dengan metodologi SDLC iteratif, alat ini dikembangkan secara bertahap untuk memvisualisasikan kode fungsi Python sebagai graf heterogen. Evaluasi pada 11 kode fungsi Python menunjukkan tingkat keberhasilan 95,45% dalam menganalisis dan memvisualisasikan kode fungsi Python dengan berbagai tingkat kompleksitas. Keterbatasan pada Graphviz menjadi peluang untuk penelitian di masa mendatang dapat difokuskan pada peningkatan skalabilitas dan analisis kode Python.*

Kata Kunci: *abstract syntax tree, graf aliran kontrol, graphviz, python, SDLC.*

INTRODUCTION

The complexity of modern software development continues to grow in tandem with the increasing size of the system [1]. This increasing complexity highlights the importance of understanding and analyzing code structure, especially when dealing with many interrelated functions and control flows [2], [3]. Static analysis methods allow for analyzing program code without

Accredited Rank 2 (Sinta 2) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No.225/E/KPT/2022, December 07, 2022. Published by LPPM Universitas Nusa Mandiri

VOL. 10. NO. 3 FEBRUARY 2025
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v10i3.6177

execution, providing early detection of potential quality issues [4], [5]. However, manual static analysis of complex code is often time-consuming and error-prone. Static analysis tools that automatically perform analyzing code structure are needed, which can significantly improve software development efficiency by reducing debugging time. Existing static analysis tools such as SonarQube, Better Code Hub, and Coverity Scan still have low precision leading to false positives and incomplete understanding [4].

The development of static analysis tools has grown significantly, especially in terms of representing code structure in graph form [6], [7], [8], [9]. Research by Rodriguez-Prieto et al. (2020) created ProgQuery, a static analysis tool with seven different types of graph representations [10]. Although ProgQuery offers a comprehensive range of graphs, three fundamental graph types, Call Graph, Dependency Graph, and Control Flow Graph, are effectively sufficient for many static code analysis tasks in most of their use cases to capture and provide insights into the code structure and behavior. Call Graph representation plays a particularly crucial role in understanding code structure [11]. One example is the Function Call Graph, which visualizes the relationship between functions and can identify potential dead code and redundancy in program logic [12], [13].

Then, the Dependency Graph can represent information about the relationship between functions, variables, and parameters [14]. Thus, it better supports code optimization and efficient modularization. Meanwhile, to identify critical code blocks or execution paths that can potentially cause runtime errors, the Control Flow Graph (CFG) can model the program execution flow, such as branches and loops [15]. This graph-based visualization provides an intuitive and comprehensive representation of code structures, enabling advanced analysis such as pattern identification, anomaly detection, and program optimization.

Although graph-based analysis techniques have proven effective, their implementation varies across programming language. Most static code analysis tools are designed to analyze Java code [16]. However, with the growing adoption of Python across various applications such as big data, machine learning, and data science, the need for Python static analysis tools is increasing. Python's flexibility and expressivity make it a preferred choice for developers, as evidenced by Python being at the top of the TIOBE Index 2024 with a ranking of 22.85% and recording a significant growth of 8.69% compared to the previous year [17].

Despite the increasing demand, static analysis tools for generating graphs from Python code are still limited. Existing tools such as Code2graph and PyCG attempt to address this issue by analyzing Python code, extracting its structure, and generating a function call graph [18], [19]. Code2graph focuses on transforming code into graph representations that facilitate structural analysis, whereas PyCG specializes in call graph generation for Python programs, enabling interprocedural analysis. These tools use Abstract Syntax Tree (AST) to extract code structure, allowing for deeper analysis compared to plain text parsing [20]. ASTVisitor traverses the code to extract syntactic and semantic information from the source code, facilitating the visualization of the program code [15].

While Code2graph and PyCG offer valuable functionalities, they are still limited in scope to visualizing program code in graph form. These tools do not provide integration with analysis outside the function call graph, such as control flow and dependencies between functions. This forces developers to rely on multiple separate tools, leading to inefficiencies in program analysis and potential inconsistencies in understanding code structure. Addressing these limitations, this research proposes a novel static analysis tool that integrates function call analysis, control flow, and variable dependency into a single platform. The proposed tool bridges the gap by incorporating control flow and dependencies analysis, offering a more holistic representation of code structure in Python.

This research introduces a novel approach by extracting structural information using AST, analyzing the Python code that contains the def() (function definition), and converting it into a single heterogeneous graph. The contributions of this research are:

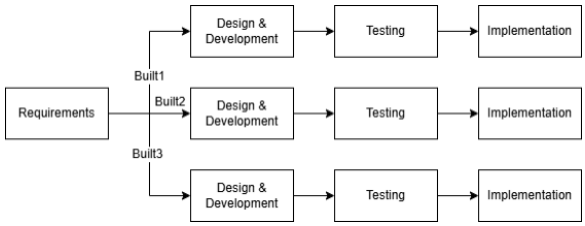
1. Develop an integrated static analysis tool that combines the Function Call Graph, Dependency Graph, and Control Flow Graph in a single heterogeneous graph.
2. A web-based platform that improves accessibility and usability for developers and to offer an efficient user experience.

MATERIALS AND METHODS

Research Methodology

The methodology for developing a graph generation tool for Python function code analysis in this research is based on the Software Development Life Cycle (SDLC) with an iterative and incremental approach. Figure 1 shows the SDLC iterative model

applied in this research. This model breaks down the development process into incremental cycles, ensuring continuous refinement to achieve a complete system. Each cycle involves design, development, testing, and implementation [21]. This iterative approach allows for progressive enhancement of the tool capabilities, from function call graph generation to the integration of dependency and control flow graph generation.



Source: (Yas et al., 2023) [21]

Figure 1. SDLC iterative model

The requirements stage focuses on an in-depth understanding of the graph generation tool's needs by identifying important features that the tool must possess. The identification involves selecting the platform and types of graphs to be generated. This tool was developed using a web-based platform for code visualization using the Streamlit library, which provides an intuitive and easy-to-use interface. Graph visualizations are presented using graph notations such as the DOT format [12]. DOT notation represents a text-based directed graph before being converted into a graphical visualization using the Graphviz library.

The main features of the graph generation tool from Python function code include visualization of function call graphs, dependency graphs, and control flow graphs. The final version of the graph generation tool is the integration of these three graphs into a heterogeneous graph [22]. Other features, such as a user-friendly interface, interactive visualization for exporting graphs in PNG format, and scalability in handling large code sizes, are also important features. The tool also needs to have informative features in the form of a usage guide, legend, table of code structure analysis results, and error handling when using the tool.

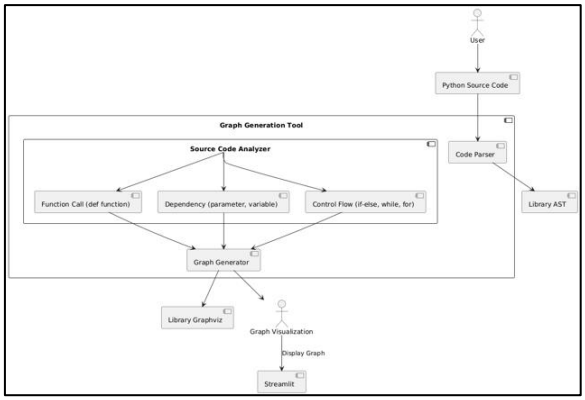
The Built 1 stage focuses on developing a function call graph generation tool. Graph visualization begins by analyzing Python function code through AST traversal to identify nodes representing function calls, then extracting information about the calling function (caller) and the called function (callee). Graph generation uses Graphviz, where nodes are functions and edges are function calls.

The Built 2 stage focuses on developing a dependency graph generation tool. The visualized graph is a further development of the Built 1 stage. It adds identification and visualization of parameters and variables used and defined in each function, as well as return statements. Variables used by more than one function are included as shared variables. Graph generation uses Graphviz, where nodes are functions, parameters, variables, and returns, while edges represent the relationships between parameters and variables.

The Built 3 stage focuses on developing a control flow graph generation tool. The fulfillment of all tool requirements is done at this stage. Each function is defined as a basic block. Nodes represent basic blocks, and each directed edge indicates the control flow between these blocks, with one entry block at the beginning and one exit block at the end of the instruction [23]. Analysis is performed by traversing the AST within each function block to identify control flow, such as conditional branches (if-else), and loops (while and for). It then visualizes function calls and the relationships of parameters and variables into a heterogeneous graph. Graph generation uses Graphviz, where nodes are functions, parameters, variables, returns, and control flow, while edges represent the relationships between these nodes.

System Architecture Design

The system architecture of the graph generation tool for Python code analysis uses a client-server architecture consisting of two main components: a front-end and a back-end. Figure 2 illustrates the system architecture of the graph generation tool. On the front-end side, the system uses the Streamlit library to provide an interactive user interface where users can input Python code to analyze and view graph visualizations.



Source: (Research Results, 2024)

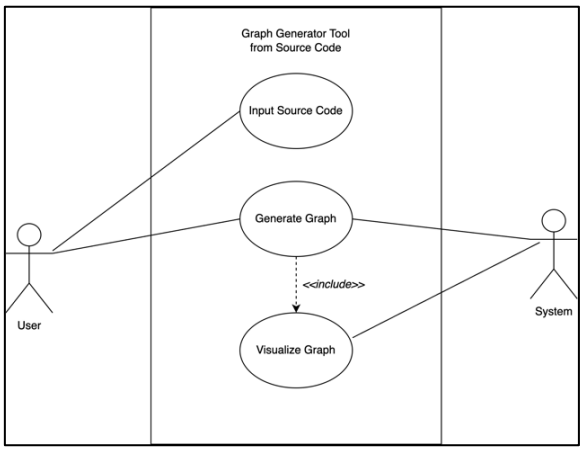
Figure 2. System Architecture of The Graph Generation Tool

Accredited Rank 2 (Sinta 2) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No.225/E/KPT/2022, December 07, 2022. Published by LPPM Universitas Nusa Mandiri

VOL. 10. NO. 3 FEBRUARY 2025
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v10i3.6177

On the back-end side, the system consists of three main components: Code Parser, Source Code Analyzer, and Graph Generator. The Code Parser uses the AST library to perform parsing and syntactic analysis of the input Python code. The Source Code Analyzer performs an in-depth analysis of program structure and control flow by traversing the code to generate information for visualization. The Graph Generator uses the Graphviz library to generate various graphs based on the analysis results.

Figure 3 illustrates a use case diagram that models user interactions with the graph generation tool system. This tool allows users to input Python function code for analysis. Users can click the "Create Graph" button to visualize the Python code. Subsequently, the system will display analysis results in the form of a function information table and graph generation.



Source : (Research Results, 2024)

Figure 3. Use Case Diagram of The Graph Generation Tool

RESULTS AND DISCUSSION

Result

Algorithm 1 represents the algorithm implemented in the graph generation tool for Python function code analysis. The tool performs parsing using Abstract Syntax Tree (AST) to read the input Python code, transform it into an AST tree structure, and traverse each node to analyze the program structure.

Algorithm 1 Graph Generation Tool

```
procedure AnalyzeCode(source_code)
    Parse source code into AST
    Initialize FunctionAnalyzer
    Visit and analyze AST
    procedure CodeAnalysis
        Visit AST nodes
        Process function definitions
        procedure BlockAnalysis
            Create control flow blocks
            Create entry block
            Process function body: if statements, while loops, for loops, function calls, and return statements
            Create exit block
        procedure VariableAnalysis
            Create edges between blocks
            Track variable usage
            Track variable definitions
    Run CodeAnalysis()
    Create visualization using Graphviz
    if visualization is successful then
        Display function info table
        Display heterogeneous graph
        Generate download link
    else
        Display error message
    return analysis results
```

Source: (Research Results, 2024)

Graph visualization is performed using Graphviz by creating basic blocks for each function in the source code and connecting the blocks according to the program's control flow. In each function block, there are at least entry and exit blocks. Additionally, it is possible to have function calls, parameters, used variables, and control structures (if-else, while, and for).

The tool has two outputs: a function information table and a heterogeneous graph with a download link to save the analysis results. Each node in the graph represents a code block with different color visualizations. Blue is used for functions and entry blocks, yellow for parameters and condition blocks, green for variables and normal blocks, and red for return and exit blocks.

The tool's development uses the iterative SDLC model, making the tool usable for visualizing graphs of Python function code from the first build stage. Thus, tool testing can be performed at each stage of the iterative model according to the established targets. The following is a Python function code tested on the tool.

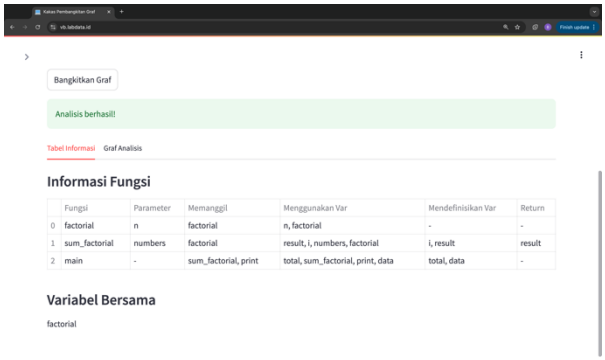
```
def factorial(n):
    if n <= 1:
        return 1
    else:
        return n * factorial(n - 1)

def sum_factorial(numbers):
    result = 0
    i = 0
    while i < len(numbers):
        result += factorial(numbers[i])
        i += 1
    return result

def main():
    data = [3, 4, 5]
    total = sum_factorial(data)
    print(total)
```

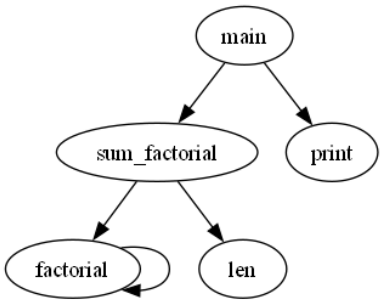
Source: (Research Results, 2024)

Figure 4 shows the function information table of the source code as the first output of the analysis results from this tool. The first function code, the definition of the factorial function, has a parameter n and is called the factorial function itself. The second function code, the definition of the sum_factorial function, has a parameter number, defines the variables result and i, and is called the factorial function. The third function is the main function that defines the data and total variable, called the sum_factorial function.



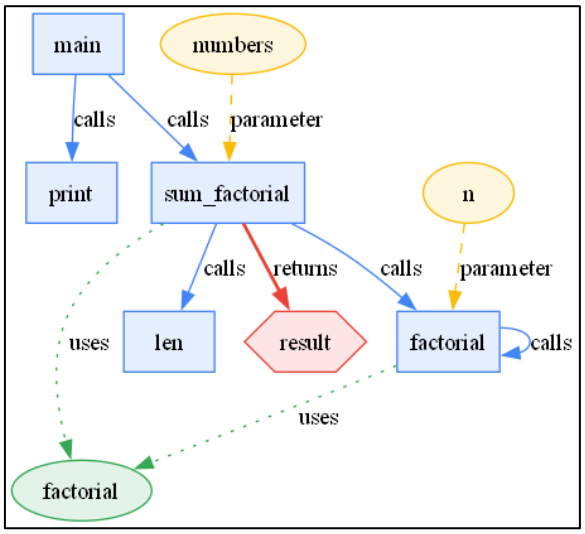
Source: (Research Results, 2024)
Figure 4. Function Information Table from Source Code

Figure 5 illustrates the Function Call Graph as a result of the static analysis process in the Build 1 stage. In the graph data structure, each node represents a function, and edges represent the calling relationships between functions. The main function calls sum_factorial function and print. Sum_factorial function calls factorial dan len. The factorial function also calls itself.



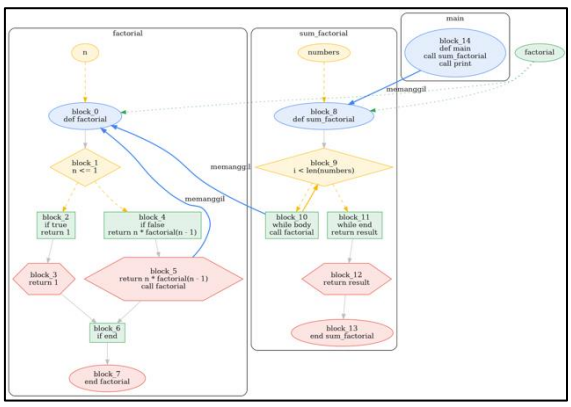
Source: (Research Results, 2024)
Figure 5. Function Call Graph

Figure 6 illustrates the Dependency Graph as a result of an analysis process that maps and visualizes the interconnections between functions in a Python program at the Build 2 stage. This process is done by analyzing the parameters and variables used. Each node represents a parameter and variable, and edges show their interconnections. Functions are represented using blue boxes, while parameters are defined using yellow ovals. Return values are represented using red hexagons. The sum_factorial function has a number parameter. The factorial function has a n parameter.



Source: (Research Results, 2024)
Figure 6. Dependency Graph

Figure 7 illustrates a Control Flow Graph (CFG) as a result of a complex analysis process that represents the control flow of a Python program at the Build 3 stage. Each node represents a block, and edges represent potential control flows between blocks, such as conditional branches (if-else), loops (while or for), function calls, and return statements. The entry block is represented using a blue oval, while the exit block uses a red oval. The condition block is represented using a yellow diamond, while the normal process after the condition is represented using a green box. The parameters and returns still use the exact representation as the Built 2 stage.



Source: (Research Results, 2024)
Figure 7. Control Flow Graph

Accredited Rank 2 (Sinta 2) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No.225/E/KPT/2022, December 07, 2022. Published by LPPM Universitas Nusa Mandiri

VOL. 10. NO. 3 FEBRUARY 2025
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v10i3.6177

Performance Evaluation

The graph generation tool successfully implemented several integrated core features and generated heterogeneous graph, including function call graph visualization that describes the relationships and interactions between functions in the program, function dependencies visualization showing parameter and variable dependencies, and control flow graph representing the program's control flow in detail. The tool can be accessed on a web-based platform at https://vb.labdata.id/. Figure 8 shows the user interface of the graph generation tool. This platform provides a guide to using the tool and examples of Python code so that users can perform analysis and visualization exploration.

Source: (Research Results, 2024)

Figure 8. User Interface of Graph Generation Tool

The graph generation tool's performance was evaluated using several Python function codes. The scale of Python code, both in terms of lines of code (LOC) and number of def() (function definitions), served as performance evaluation parameters. Eleven Python function codes with varying scales were used to test the graph generation tool. Table 1 presents the results of the tool performance evaluation.

Table 1. Tool Performance Evaluation

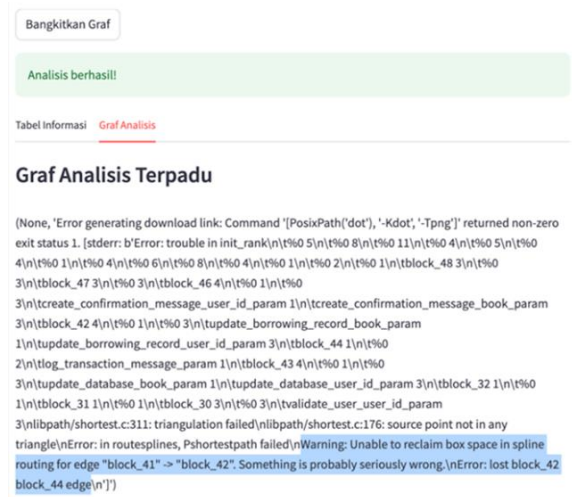
No	Program Code	Scale LOC	Function	Result Function	Graph	Success Percentage
1	P1	11	0	0	x	100%
2	P2	14	4	4	v	100%
3	P3	33	8	8	v	100%
4	P4	23	5	4	v	100%
5	P5	48	5	5	v	100%
6	P6	68	8	8	v	100%
7	P7	70	7	7	v	100%
8	P8	85	10	10	v	100%
9	P9	87	12	12	v	100%
10	P10	86	17	17	x	50%
11	P11	122	18	18	v	100%
Average Success Rate						95.45%

Source: (Research Results, 2024)

Based on Table 1, the success percentage of this tool is 100% if it can analyze Python function code accurately and display the results in the form of a function information table and graph visualization. Meanwhile, if it only displays one function information table or graph visualization accurately, the success percentage of this tool is 50%.

Two program codes, P1 and P10, were not visualized as graphs from the 11 Python source codes tested on the graph generation tool. P1 is a Python code without function definitions, so the tool's success rate is 100% because it correctly did not visualize it as a graph.

Program code P10 is a Python code containing 17 function definitions. In terms of complexity, program code P11 is more complex than program code P10. Figure 9 shows the error for program code P10. The tool could not generate a graph for program code P10, resulting in the error "unable to reclaim box space in spline routing for edge 'block_41' → 'block_42'". This issue arises due to a limitation in the Graphviz library, which is used for visualizing the graph structure. When the tool displays labels such as names, functions, parameters, variables, and related return values for each instruction block, there are space limitations for generating graph visualizations.

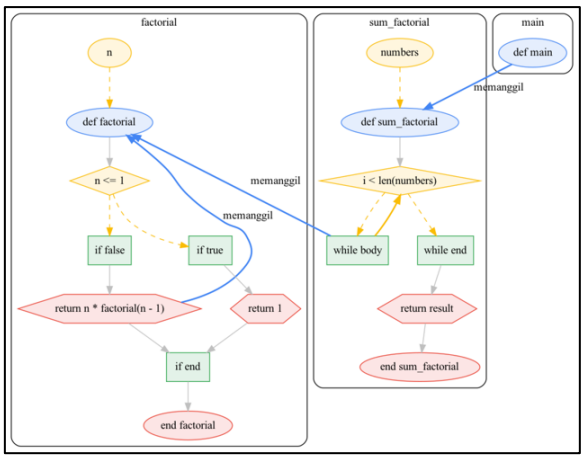


Source: (Research Results, 2024)

Figure 9. Error for Program Code P10

As shown in Figure 10, when all labels are displayed on each node, the tool can only generate the graph up to program code P7. To address this issue, optimization was applied to the node labeling scheme. Figure 10 illustrates the label modification on the Control Flow Graph. By simplifying the labels displayed on each node, the tool successfully generates the graphs for program codes P8, P9, and

P11 without space limitations for graph visualizations.



Source: (Research Results, 2024)

Figure 10. Label Modification on the Control Flow Graph

CONCLUSION

This research successfully developed a tool using the SDLC method with an iterative model, achieving an average success rate of 95.45% in generating graphs from Python function code. The implementation using Abstract Syntax Tree (AST) effectively extracted structural and control flow information, where the generation of heterogeneous graphs for Python function code integrated three types of graph analyses: function call graphs, inter-function dependency graphs, and control flow graphs on a single platform. The development of this tool has contributed to the field of function code analysis, particularly for the Python ecosystem. The limitations of graph generation using the Graphviz library present an opportunity for future research.

Further development in the future may include enhancing the tool's capabilities to handle larger and more complex graph visualizations and expanding the analysis coverage to class and object levels in Python code. Additional testing with larger and more complex code and testing across various usage scenarios will also improve the effectiveness of this tool in real-world applications.

REFERENCE

[1] M. Alenezi and M. Zarour, "On the Relationship between Software Complexity and Security," *International Journal of Software Engineering & Applications*, vol. 11, no. 1, pp. 51–60, Jan. 2020, doi: 10.5121/ijsea.2020.11104.

[2] M. S. Khoirom, M. Sonia, B. Laikhuram, J. Laishram, and D. Singh, "Comparative Analysis of Python and Java for Beginners," *International Research Journal of Engineering and Technology*, 2020, Accessed: Dec. 03, 2024. [Online]. Available: https://www.irjet.net/archives/V7/i8/IRJET-V7I8755.pdf

[3] X. Zong, S. Zheng, H. Zou, H. Yu, and S. Gao, "GraphPyRec: A novel graph-based approach for fine-grained Python code recommendation," *Sci Comput Program*, vol. 238, Dec. 2024, doi: 10.1016/j.scico.2024.103166.

[4] V. Lenarduzzi, F. Pecorelli, N. Saarimäki, S. Lujan, and F. Palomba, "A Critical Comparison on Six Static Analysis Tools: Detection, Agreement, and Precision," *Journal of Systems and Software*, vol. 198, Apr. 2023, doi: 10.1016/j.jss.2022.111575.

[5] H. Bapeer Hassan, Q. Idrees Sarhan, Á. Beszédes, and H. B. Hassan, "Evaluating Python Static Code Analysis Tools Using FAIR Principles," Nov. 2024, doi: 10.1109/ACCESS.2024.0429000.

[6] G. Antal, P. Hegedus, Z. Herczeg, G. Loki, and R. Ferenc, "Is JavaScript Call Graph Extraction Solved Yet? A Comparative Study of Static and Dynamic Tools," *IEEE Access*, vol. 11, pp. 25266–25284, 2023, doi: 10.1109/ACCESS.2023.3255984.

[7] E. Fregnan, J. Fröhlich, D. Spadini, and A. Bacchelli, "Graph-based Visualization of Merge Requests for Code Review," *J Syst Softw*, vol. 195, p. 111506, 2023, doi: 10.1016/j.jss.2022.111506.

[8] Y. Jin *et al.*, "Graph-Centric Performance Analysis for Large-Scale Parallel Applications," *IEEE Transactions on Parallel and Distributed Systems*, vol. 35, no. 7, pp. 1221–1238, Jul. 2024, doi: 10.1109/TPDS.2024.3396849.

[9] K. Borowski, B. Balis, and T. Orzechowski, "Semantic Code Graph - An Information Model to Facilitate Software Comprehension," *IEEE Access*, vol. 12, pp. 27279–27310, 2024, doi: 10.1109/ACCESS.2024.3351845.

[10] O. Rodriguez-Prieto, A. Mycroft, and F. Ortin, "An Efficient and Scalable Platform for Java Source Code Analysis Using Overlaid Graph Representations," *IEEE Access*, vol. 8, pp. 72239–72260, 2020, doi: 10.1109/ACCESS.2020.2987631.

[11] Z. Sagodi, E. Pengo, J. Jasz, I. Siket, and R. Ferenc, "Static Call Graph Combination to

Accredited Rank 2 (Sinta 2) based on the Decree of the Dirjen Penguatan RisBang Kemenristekdikti No.225/E/KPT/2022, December 07, 2022. Published by LPPM Universitas Nusa Mandiri

VOL. 10. NO. 3 FEBRUARY 2025
P-ISSN: 2685-8223 | E-ISSN: 2527-4864
DOI: 10.33480/jitk.v10i3.6177

Simulate Dynamic Call Graph Behavior," *IEEE Access*, vol. 10, pp. 131829–131840, 2022, doi: 10.1109/ACCESS.2022.3229182.

[12] R. Alanazi, G. Gharibi, and Y. Lee, "Facilitating Program Comprehension with Call Graph Multilevel Hierarchical Abstractions," *Journal of Systems and Software*, vol. 176, Jun. 2021, doi: 10.1016/j.jss.2021.110945.

[13] A. Bansal, Z. Eberhart, Z. Karas, Y. Huang, and C. Mcmillan, "Function Call Graph Context Encoding for Neural Source Code Summarization," *IEEE Transactions on Software Engineering*, vol. 49, no. 9, pp. 4268–4281, Sep. 2023, doi: 10.1109/TSE.2023.3279774.

[14] H. Liu, Y. Tao, W. Huang, and H. Lin, "Visual Exploration of Dependency Graph in Source Code Via Embedding-based Similarity," *J Vis (Tokyo)*, vol. 24, no. 3, pp. 565–581, Jun. 2021, doi: 10.1007/s12650-020-00727-x.

[15] H. V. Tran and P. N. Hung, "A Control Flow Graph Generation Method for Java Projects," *VNU Journal of Science: Computer Science and Communication Engineering*, vol. 40, no. 1, Jun. 2024, doi: 10.25073/2588-1086/vnucsce.668.

[16] P. Bedadala, M. D, and L. S. Nair, "Generation of Call Graph for Java Higher Order Functions," in *Proceedings of the Fifth International Conference on Communication and Electronics Systems (ICCES)*, IEEE, 2020. Accessed: Nov. 20, 2024. [Online]. Available: https://ieeexplore.ieee.org/document/9138056

[17] P. Jansen, "TIOBE Index for November 2024," https://www.tiobe.com/tiobe-index/.

[18] Z. Zhao, X. Wang, Z. Xu, Z. Tang, Y. Li, and P. Di, "Incremental Call Graph Construction in Industrial Practice," in *Proceedings - International Conference on Software Engineering*, IEEE Computer Society, 2023, pp. 471–482. doi: 10.1109/ICSE-SEIP58684.2023.00048.

[19] V. Salis, T. Sotiropoulos, P. Louridas, D. Spinellis, and D. Mitropoulos, "PyCG: Practical Call Graph Generation in Python," in *Proceedings - International Conference on Software Engineering*, IEEE Computer Society, May 2021, pp. 1646–1657. doi: 10.1109/ICSE43902.2021.00146.

[20] N. Mehrotra, A. Sharma, A. Jindal, and R. Purandare, "Improving Cross-Language Code Clone Detection via Code Representation Learning and Graph Neural Networks," *IEEE Transactions on Software Engineering*, vol. 49, no. 11, pp. 4846–4868, Nov. 2023, doi: 10.1109/TSE.2023.3311796.

[21] Q. Yas, B. Rahmatullah, A. Alazzawi, and Q. M. Yas, "A Comprehensive Review of Software Development Life Cycle methodologies: Pros, Cons, and Future Directions," *Iraqi Journal for Computer Science and Mathematics*, vol. Vol. 4 No. 4, pp. 173–0, 2023, doi: 10.52866/ijcsm.2023.04.04.0.

[22] Y. Huang, M. He, X. Wang, and J. Zhang, "HeVulD: A Static Vulnerability Detection Method using Heterogeneous Graph Code Representation," *IEEE Transactions on Information Forensics and Security*, 2024, doi: 10.1109/TIFS.2024.3457162.

[23] H. Il Lim, "An approach to comparing control flow graphs based on basic block matching," *Indian Journal of Computer Science and Engineering*, vol. 11, no. 3, pp. 289–296, May 2020, doi: 10.21817/indjcse/2020/v11i3/201103237.