Code2graph: Automatic Generation of Static Call Graphs for Python Source Code

Gharib Gharibi, Rashmi Tripathi, and Yugyung Lee
School of Computing and Engineering, University of Missouri-Kansas City, Kansas City, MO, USA
{ggk89}@mail.umkc.edu, LeeYu@umkc.edu

ABSTRACT

A static call graph is an imperative prerequisite used in most interprocedural analyses and software comprehension tools. However, there is a lack of software tools that can automatically analyze the Python source-code and construct its static call graph. In this paper, we introduce a prototype Python tool, named code2graph, which *automates* the tasks of (1) analyzing the Python source-code and extracting its structure, (2) constructing static call graphs from the source code, and (3) generating a similarity matrix of all possible execution paths in the system. Our goal is twofold: First, assist the developers in understanding the overall structure of the system. Second, provide a stepping stone for further research that can utilize the tool in software searching and similarity detection applications. For example, clustering the execution paths into a logical workflow of the system would be applied to automate specific software tasks. Code2graph has been successfully used to generate static call graphs and similarity matrices of the paths for three popular open-source Deep Learning projects (TensorFlow, Keras, PyTorch). A tool demo is available at https://youtu.be/ecctePpcAKU.

CCS CONCEPTS

• Software and its engineering → Automated static analysis

KEYWORDS

Static code analysis, call graph construction, Python Static code analysis, call graph construction, Python

ACM Reference format:

Gharib Gharibi, Rashmi Tripathi, and Yugyung Lee. 2018. Code2graph: Automatic Generation of Static Call Graphs for Python Source Code. In *Proceedings of the 2018 33rd ACM/IEEE International Conference on Automated Software Engineering (ASE '18), September 3–7, 2018, Montpellier, France. ACM, New York, NY, USA,* https://doi.org/10.1145/3238147.3240484

ASE '18, September 3–7, 2018, Montpellier, France

ACM ISBN 978-1-4503-5937-5/18/09…$15.00
https://doi.org/10.1145/3238147.3240484

1 INTRODUCTION

A call graph is a visual representation of the relationships (i.e., calls) among the system's functions [1]. It is typically visualized as a directed graph, where each node represents a function and each edge represents a function call. The function initiating the call is the caller function, and the invoked function is the callee. A call graph can be dynamic, constructed during run time, or static, constructed during compile time. A dynamic call graph represents all target functions of a single execution path (i.e., run) of the system [2]. A static call graph represents every possible execution path of the system [3]. The call graph can support a comprehensive understanding of the system and thus, leads to better software-related activities, such as maintenance and evolution. In addition, the call graph is a prerequisite to build a similarity matrix of all execution paths of the system. An execution path represents all function calls between an entry point (i.e., function) to an exit point in the system. For example, the similarity matrix can be used as an input to a Deep Learning model that generates similar, but more sophisticated applications. This was the main motive in developing the code2grap tool.

While there are several existing tools that can automatically construct static call graphs for several OOP languages [3-5], the methods and tools that address call graph construction for a Python source-code are still limited due to the dynamic nature of Python. The existing Python-based tools that attempt to automatically construct static call graphs either require the user's manual intervention, such as Inspect [6], or can analyze only one module at a time, such as ConstructCallGraph [7], or one directory at a time, such as Pyan [8]. In addition, these tools can neither construct all possible execution paths of the system nor their similarities. Note that from hereafter, a "call graph" is referring to the "static call graph," unless otherwise distinguished.

To this end, we introduce a tool, named code2graph, which can *automatically* (1) analyze Python code and extract its structure (i.e., the hierarchy of the packages, modules, classes, and functions), (2) construct and visualize the call graph, and (3) generate a list of all possible execution paths of the system and calculate their similarities. Code2graph uses static-analysis code-traversal techniques to extract the structure of the system using AST [9] to process trees of Python's abstract syntax grammar. Then, it constructs and visualizes the call graph and builds the similarity matrix of the system's paths, as explained in the following sections.

2 APPROACH OVERVIEW

Code2graph is a Python stand-alone tool that can automatically analyze Python source code, generate its call graph, and construct a similarity matrix of all possible execution paths of the system. Figure 1 illustrates an overview of the tool using an example from the TensorFlow case study, which is explained in Section 3. An overview of the approach is explained in the following stages.

2.1 Extracting the Code Structure

Our tool uses a static analysis code-traversal approach that crawls the codebase and builds a dictionary whose keys are the namespaces of each package, module, and class and the values are lists of functions and their calls (i.e., caller-callee relationships). In particular, the tool retrieves the hierarchical structure of the software's packages, modules, classes, and functions based on the user's specified root directory of the system (software). A package structure contains a list of Python modules and contains an additional "*__init__.py*" file to distinguish it from other packages that could contain random Python scripts. A module in Python is simply a file that contains Python code and it is treated as an object with an arbitrary number of attributes. The class structure contains information about the class, imported libraries, and the class attributes and functions. The output of this stage includes statistics on the program's structure (i.e., the number of packages, modules, classes, and functions) and a hierarchical view of the structure visualized using D3.js. An example of extracting the code structure with its visualization is illustrated in step 1 of Figure 1. The code structure is used to extract the caller-callee relationships as explained in the next subsection.

2.2 Extracting the Caller-Callee Relationships

Using the static analysis output from the previous stage, the tool automatically parses the function calls and forms a relationship between each caller function and its targets (i.e., callee functions). Calls to Python's built-in functions (e.g., *print()* and *input()*), libraries (e.g., *os* and *random*), and test cases (e.g., *assertEqulas() and assertNull()*) are ignored and not included in the graph due to the fact that they neither contribute to the overall structure nor the logical workflow of the project. However, code2graph is capable of parsing the library calls and visualizing them, but they are excluded to simplify the call graph visualization. A 2D matrix is used to represent the caller-callee relationships, in which the first column represents the caller functions and the second column represents the callee functions. The user can view the matrix in *csv* or *.xlsx* file formats and as shown in Figure 1. 2.

2.3 Constructing and Visualizing the Call Graph

A call graph is a rooted directed graph, *G=(V, E)*, in which each node *V* represents a function *v* and each edge, *E=(v, u)*, represents a function call from the function *v* (i.e., the caller) to function *u* (i.e., the callee). A function with no incoming calls is considered as an *entry point* to the program. A function that does not call any other function is considered as an *exit point* of the program. A function that is neither called nor calls any other function is called an *orphan function*. Orphan functions are visualized in the call graph, but are not included in any execution paths of the system. We use NetworkX [10] to create and construct the call graph from the caller-callee matrix generated in the previous stage. NetworkX is a Python graph data-structure that facilitates the creation and

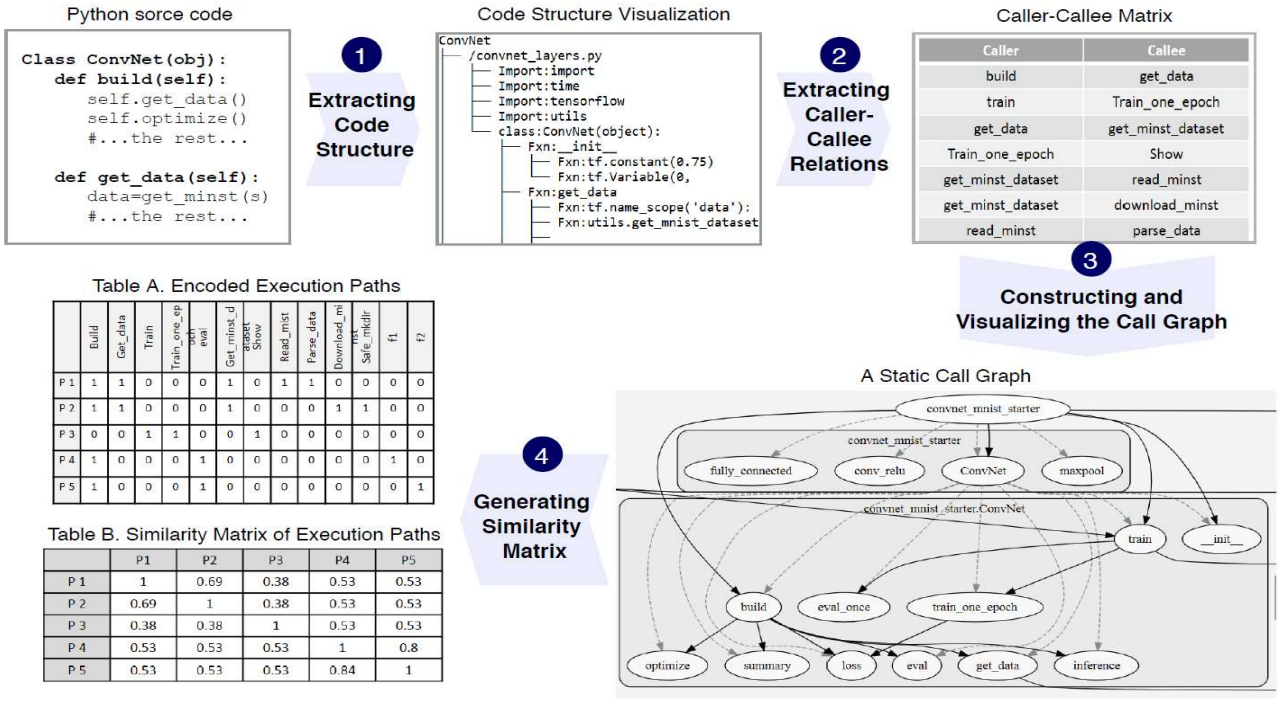

Figure 1. Approach overview with snippet examples from the TensorFlow case study.

manipulation of the graphs. Each node in the network represents a caller/callee function linked with a directed edge that shows the direction of the call. The object of NetworkX is translated automatically using code2graph into a graph description language written to a file with a *dot* file extension [11]. A *dot* file uses the DOT language to describe a graph, but does not provide facilities for rendering (i.e., visualizing) the graph. We use Graphviz [12], an open-source graph visualization software, to render the *dot* files. Graphviz can take a text description of any graph and render it into useful diagrams in several formats. Figure. 1. 3 shows an example of the automatically constructed call graph

2.4 Generating the Similarity Matrix of the Execution Paths

A similarity matrix is an imperative prerequisite for several innovative applications, such as clustering the similar execution paths into a logical workflow and generating machine learning models that focus on automating software-related activities. A similarity matrix illustrates the similarity score between each execution path and all other execution paths of the system. We explain the generation of the similarity matrix as follows.

2.4.1 Path Generation. The first step in this stage is to identify all possible execution paths of the program using the directed graph generated in the previous stage. A single execution path represents a possible run of the program from an entry point to an exit point. We used a modified version of the depth-first search, built in NetworkX, to generate a matrix of all possible execution paths. Each row in the matrix represents a single execution path as shown in Figure 2. This matrix is also available for the user to view in .csv or .xlsx file formats.

2.4.2 Encoding the Execution Paths. In order to simplify the matrix operations, we encoded the paths matrix generated in the previous step into a binary system inspired by the One Hot Encoding, also known as dummy variables [13]. The values of one and zero are used to represent the presence or absence of a particular function from each path respectively. The order of functions at this stage is not important since we have already generated the call graph and are more interested in generating the similarity matrix. A representation of the encoded paths is illustrated in Table A of Figure 1.

2.4.3 Calculating the Similarity Matrix. code2graph uses the Jaccard similarity coefficient [14], also known as the Jaccard Index, to identify the similarity between all system paths. Jaccard similarity compares two paths at a time and calculates a similarity coefficient between 0 and 1, or 0% and 100%. An index of 1, or 100%, means that the two paths are identical and an index of 0, or 0%, indicates that the two paths do not share any functions. The similarity matrix of size $n \times n$, where n is the total number of paths, is generated by calculating the Jaccard similarity coefficients between each path and all other paths of the program. The set of paths are treated as an ordered vector, $P_1, P_2,\ldots,P_n$, which is placed as the row and column headers of the matrix as shown in Table B of Figure 1 (the table included five paths only for explanatory purposes).

```
Class ConvNet(object):
    def build(self):
        self.get_data()
        self.eval()
        self.summary()
        #...function body

    def get_data(self):
        data = get_mnist_dataset(size)
        # ...function body

    def eval(self):
        self.huber_loss()
        #... function body

def get_mnist_dataset(size):
    read_minst()
    download_minst()
    #...function body
```

```
build -> get_data -> get_minst_dataset -> read_minst
build -> get_data -> get_minst_dataset -> doanlowd_minst
build -> eval -> huber_loss
build -> summary - > ...
```

Figure 2. An example of generating paths from the code.

3 CASE STUDIES

In this section, we demonstrate the usage of our tool using three popular open-source projects written in Python, including TensorFlow [15], Keras [16], and PyTorch [17]. The three case studies are platforms that facilitate Deep Learning computations in Python. Following this, we demonstrate the results of the first case study, TensorFlow. We present a complete list of the results with higher resolution and readable images for all case studies at the tools' website [18].

The code structure analysis showed that the current version of TensorFlow includes more than 1.6 million SLOC written in Python, C++, and other languages. We cloned the TensorFlow repository from GitHub and provided its local directory path to code2graph. The structure analysis using code2graph showed that TensorFlow codebase consists of 116 packages, 841 modules, 2,108 classes, 15,758 functions, and 1,187 import statements. A snippet of the TensorFlow structure view is shown in Figure 3, which shows the root directory, subdirectories, modules, and the functions inside each module.

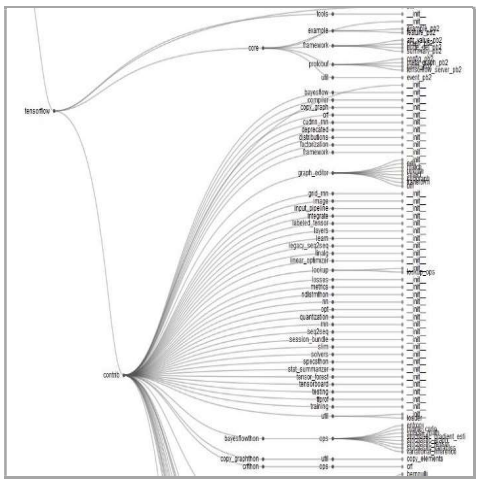

Figure 3. A snippet of TensorFlow's structure visualization.

The total number of extracted paths is 73,805 unique paths, out of them only 3,304 paths contained five or more nodes (i.e., functions). The maximum length of a path included 14 nodes. The paths' data are used in the following steps to generate the call graph and later the similarity matrix. Table 1 summarizes the results of the static analyses of the three case studies. The user can visualize this data in *excel* or *csv* file formats.

TABLE 1. Summary of the Case Studies' Analyses Results

Entity	TensorFlow	Keras	PyTorch
No. of packages	116	145	373
No. of classes	2,108	195	659
No. of functions	15,758	1,933	4,409
Total No. of paths	73,805	4,324	~5500
Maximum path length	14	10	9
No. of paths with more than five nodes	3,304	237	449
Analysis time (seconds)	146	60	80

The static code analyses results are used to construct the call graphs of the three projects. A complete list of the projects' call graphs is available on the tool's website [18]. Once the call graph is generated, the tool can then construct the similarity matrix and visualize it. Figure 4 depicts the similarity matrix of TensorFlow execution paths represented using a heat-map. The x- and y-axes represent the path. We used 50 random paths for explanatory purposes. The scale of the heat-map is [0,1]. The value of 0 represents no similarity between the two paths and it is represented using the color blue. The value of 1 represents identical paths and it is represented using the color red. The similarity matrix is a solid stepping stone for further in search and machine learning applications, such as clustering the execution paths and detecting the similarity between two projects.

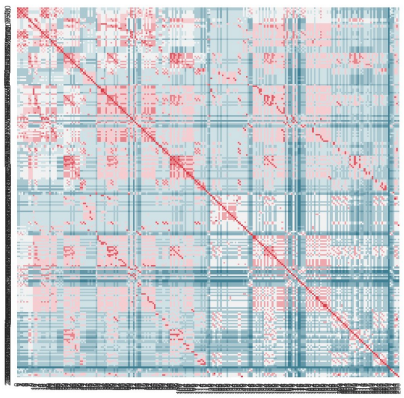

Figure 4. An example of the similarity matrix of TensorFlow's execution paths visualized using a heat-map.

4 PROPOSED EVALUATIONS

Code2graph was developed to serve in a bigger research project that requires the availability of the system's call graph and a similarity matrix of its execution paths. However, we have examined its potential in generating a high-level logical workflow of the system by clustering the similar execution paths. We are currently starting a user study to look at two hypotheses: First, code2graph assisting developers in understanding the overall structure of the system. Second, code2graph can automatically cluster execution paths based on their similarity in a faster and more accurate way than the existing tools.

5 CONCLUSIONS

This paper introduced a new tool prototype, code2graph, which can automatically analyze the Python source code, extract and visualize its structure, and build a similarity matrix of the system's execution paths. We plan to use the tool in an on-going work to automatically generate the logical workflows for new Deep Learning solutions based on the user's requirements. More information on the case studies are available on the tool's website [18]. A video demo is available at https://youtu.be/ecctePpcAKU.

REFERENCES

[1] Barbara Ryder. 1979. Constructing the Call Graph of a Program. *IEEE Transactions on Software Engineering*, SE-5, 3 (1979), 216–226.

[2] Michael Burch. 2016. The dynamic call graph matrix. In Proceedings of the 9th International Symposium on Visual Information Communication and Interaction. ACM. 1-8.

[3] Gail Murphy, et. al. 1996. An empirical study of static call graph extractors. In *Proceedings of the 18th international conference on Software engineering (ICSE '96)*. 90-99.

[4] Kareem Ali, et. al. 2015. Type-Based Call Graph Construction Algorithms for Scala. *ACM Transactions on Software Engineering and Methodology*, 25, 1, Article 9, 1-43.

[5] Demitry Petrashko, et. al. 2016. Call graphs for languages with parametric polymorphism. *ACM SIGPLAN Notices*, 51, 10, 394-409.

[6] Inspect, Python 3.6.5 documentation. 2018. Retrieved from: docs.python.org/3/library/inspect.html.

[7] Construct_Call_Graph. 2018. Retrieved from: blog.prashanthellina.com.

[8] Pyan – Constructing static call-graphs. 2018. Retrieved from: github.com/DavidFraser/pyan.

[9] Abstract Syntax Trees – Python 3.6.5 documentation. 2018. Retrieved from: docs.python.org/3/library/ast.html.

[10] NetworkX – Software for complex networks. 2018. Retrieved from: etworkx.github.io.

[11] Eleftherios Koutsofios and Stephanie North. 1991. *Drawing graphs with dot.* Technical Report 910904-59113-08TM, AT&T Bell Laboratories, Murray Hill, NJ.

[12] Ellson, John, et. al. Graphviz and dynagraph—static and dynamic graph drawing tools. 2004. In *Graph drawing software*, Springer, Berlin, Heidelberg, 127-148.

[13] Susan Garavaglia and Asha Sharma. A smart guide to dummy variables: Four applications and a macro. 1998. In *Proceedings of the Northeast SAS Users Group Conference*, 43.

[14] Niwattanakul Suphakit, et al. Using of Jaccard coefficient for keywords similarity. 2013. In *Proceedings of the International MultiConference of Engineers and Computer Scientists*, 1, 6.

[15] TensorFlow – AI Python platform. 2018. Retrieved from: github.com/tensorflow/tensorflow.

[16] Keras – Python AI Library. 2018. Retrieved from: github.com/keras-team/keras.

[17] PyTorch. Advanced AI platform. 2018. Retrieved from: github.com/pytorch.

[18] Website of Code2graph – Constructing static call graphs. 2018. Retrieved from: https://info.umkc.edu/UDIC_Research/index.php/code2graph/.