Empirical Study of Python Call Graph

Yu Li
Nanjing University, China

***Abstract*—In recent years, the extensive application of the Python language has made its analysis work more and more valuable. Many static analysis algorithms need to rely on the construction of call graphs. In this paper, we did a comparative empirical analysis of several widely used Python static call graph tools both quantitatively and qualitatively. Experiments show that the existing Python static call graph tools have a large difference in the construction effectiveness, and there is still room for improvement.**

***Keywords*—Python, call graph, empirical study, quantitative, qualitative.**

I. Research Problem

The Python language has received more and more attention due to its widespread use, and in this case the analysis of Python programs is more valuable. Call graph construction is the basic work of program analysis, which can serve a lot of further analysis work. As a dynamic language, some features of Python bring many difficulties to the call graph construction.

There are a number of call graph studies for dynamic languages that give us a better understanding of the language's features[1, 2, 3, 4]. Some empirical work also provided inspiration for our experimental steps[5, 6, 7].

In this study, we applied three widely used Python static call graph tools to six open source projects. By comparing the constructed graphs quantitatively and qualitatively, we get some inspiration about Python call graph construction.

II. Methodology

A. Call Graph Extraction

We have selected three tools that are frequently mentioned in the developer community. Pyan[8] constructs call graph at syntactic level while Code2flow[9] constructs call graph at lexical level. Understand[10] is a commercial program analysis tool that provides call graph generation. For comparison, we also selected a dynamic call graph tool Pycallgraph[11] and approximated it as ground truth. The call graph construction of Pycallgraph depends on the execution of the test case. We conducted the experiment on projects of different sizes and application scenarios, including *scikit-learn, theano, networkx, numba, joblib* and *pandas*.

B. Study Process

For quantitative analysis, we first counted and compared the number of edges and nodes in the call graphs generated by the selected tools.

Then we used some graph metrics to compare the results of each tool at the method level.

In order to further observe the difference between the call graph contructed by different tools, we conducted a simple impact analysis and compare their performances.

At last, we did a manual review of the results by combining the source codes and the call graphs. We inspected some nodes that caused a huge difference in the number of calls, and indeed found some interesting phenomena.

III. Results

A. Comparison of results

In terms of quantity, the differences between tools are very large. We summarize the nodes and edges generated by all the tools in all the studied projects, and then calculate the intersection between them. Figure 1 shows the results. Each ratio represents the proportion of the marked part in the graph generated by all of the tools in all projects. The intersection between static tools and Pycallgraph is very small, which indicates that their performances are not as good as expected.

Additionally we have also made some attempts at the node level. We found out the nodes shared by all the tools in project *sklearn*. Then we calculated several centrality and connectivity metrics to further compare their effects. Figure 2 is the result of degree centrality[12] and average neighbor degree[13].

The tools have close degree centrality on the common nodes. The result of Pycallgraph is a bit larger than others, and Understand is closest to it. However, the gap between the results of average neighbor degree is obvious. Pycallgraph still has the highest results, which seems to indicate that the call graph it generates has good connectivity. The results of static tools are relatively poor, especially Code2flow. It is speculated that this is caused by missing calls.

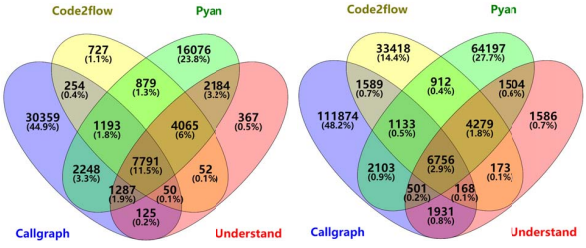


Fig. 1. Wayne diagram of the nodes(left) and the edges(right) found by all tools

B. Impact Analysis

We did a simplified experiment here to evaluate applicability of these tools. We made to find out 20 important nodes in the graph, using the pagerank value calculated before. We

978-1-7281-2508-4/19/$31.00 ©2019 IEEE
DOI 10.1109/ASE.2019.00160

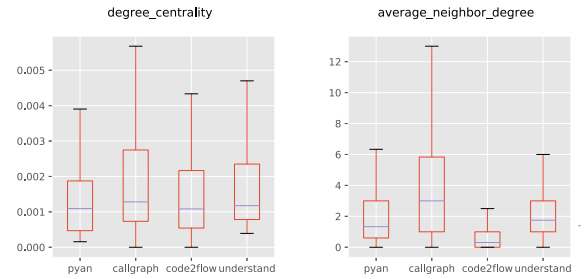
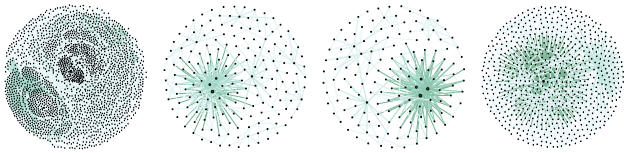


Fig. 2. Degree centrality and average neighbor degree of common nodes

traversed from these nodes in the reverse order of the call relationship. A node is impacted only if it can be reached. The results are shown in the way of network graph in figure 3.



(a) Pycallgraph (b) Pyan (c) Code2flow (d) Understand

Fig. 3. Network of impacted nodes

As we can see above, Pycallgraph finds out a large number of impacted nodes, forming a huge network. Pyan and Code2flow seem to get a similar network, which is much smaller than Pycallgraph. It is interesting that Understand find out more impacted nodes. In fact, Understand is also very good at some graph metrics. Its accuracy may be the advantage of a commercial software.

After investigating the network graphs generated, we found that a minor difference in a node can lead to huge global differences. Method *sklearn.metrics.pairwise.euclidean_distances* is one of the nodes determined by pagerank, and both Pycallgraph and Understand found more than ten nodes impacted by this node. However the other tools only found two impacted nodes, leading to dozens missing nodes.

C. Analysis of influencing factors

We first divide the nodes in call graphs into two categories: explicit nodes which represent the functions explicitly defined with the keywords ”def”, and implicit nodes representing other functions. Table I shows the number of implicit nodes in the *pandas* project and the edges associated with them. **Surprisingly, Pycallgraph and Pyan generate a large number of implicit nodes, and these nodes are associated with more call edges, which are likely to result in the huge differences.**

In our manual review process, we found some representative nodes. The most important problem is the missing nodes caused by dynamic binding. As shown in the following

TABLE I
NUMBER OF IMPLICIT NODES AND EDGES

	Pycallgraph	Pyan	Code2flow	Understand
Total nodes	8837	10162	4089	4059
Implicit nodes	5832	5378	147	470
The proportion of Implicit nodes	65%	53%	4%	12%
Total edges	38179	32722	15548	8695
Edges associated with implicit nodes	28430	19870	649	923
The proportion of edges associated with implicit nodes	74%	61%	4%	11%

code, the class *BaseEstimator* in project *sklearn* defines the *get_params* method, and all its subclasses inherit the method. Then the method *get_params* is called in the function *clone* through the dynamic parameter *estimator*.

```
class BaseEstimator:
  def get_params(self, deep=True):
    ...

def clone(estimator, safe=True):
  new_object_params = estimator.get_params(deep=False)
  ...
```

Depending on the runtime information, the parameter *estimator* can bind to several subclass objects of *BaseEstimator*. However, those static tools all generate a single call to *sklearn.base.BaseEstimator.get_params*. Such nodes are not uncommonly seen in the call graphs for the six studied projects, leading to lots of differences.

In general, Pyan uses an overly aggressive strategy in dealing with binding behavior. However, its practice has produced a lot of confusing nodes like mistaking variables as functions. In most cases, Code2flow and Understand ignore the importance of these nodes, so the number of calls generated by them is much less than those of Pyan and Pycallgraph.

D. Inspiration

Object-oriented features and binding behavior are the main obstacles affecting the call graph construction. It is quite complicated to solve these problems without the support of runtime information. Static tools have different ways to deal with these features. One way is just to neglect the callable property of parameters or uncertain objects. Another way is to take radical practice, making some speculations of the object type. We may need to weigh the pros and cons of these two practices based on the application scenarios. The designer of the algorithm may have to give the option to the developer.

IV. CONCLUSIONS

Through empirical analysis, we found some interesting phenomena in the existing call graph tools. In the future, we plan to further investigate the performances of existing call graph construction algorithms in Python language. We also attempt to combine and improve existing algorithms and tools to develop algorithms more suitable for the Python language.

REFERENCES

[1] Gábor Antal, Péter Hegedus, Zoltán Tóth, Rudolf Ferenc, and Tibor Gyimóthy. Static javascript call graphs: A comparative study. In *2018 IEEE 18th International Working Conference on Source Code Analysis and Manipulation (SCAM)*, pages 177–186. IEEE, 2018.

[2] Gregor Richards, Sylvain Lebresne, Brian Burg, and Jan Vitek. An analysis of the dynamic behavior of javascript programs. In *ACM Sigplan Notices*, volume 45, pages 1–12. ACM, 2010.

[3] J Dijkstra. *Evaluation of Static JavaScript Call Graph Algorithms*. PhD thesis, Software Analysis and Transformation, 2014.

[4] Gharib Gharibi, Rashmi Tripathi, and Yugyung Lee. Code2graph: automatic generation of static call graphs for python source code. In *Proceedings of the 33rd ACM/IEEE International Conference on Automated Software Engineering*, pages 880–883. ACM, 2018.

[5] Gail C Murphy, David Notkin, William G Griswold, and Erica S Lan. An empirical study of static call graph extractors. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 7(2):158–191, 1998.

[6] Michael Reif, Florian Kübler, Michael Eichberg, and Mira Mezini. Systematic evaluation of the unsoundness of call graph construction algorithms for java. In *Companion Proceedings for the ISSTA/ECOOP 2018 Workshops*, pages 107–112. ACM, 2018.

[7] Ond Lhoták et al. Comparing call graphs. In *Proceedings of the 7th ACM SIGPLAN-SIGSOFT workshop on Program analysis for software tools and engineering*, pages 37–42. ACM, 2007.

[8] The pyan tool. https://github.com/davidfraser/pyan.

[9] The code2flow tool. https://github.com/scottrogowski/code2flow.

[10] The understand tool. https://scitools.com/.

[11] The pycallgraph tool. https://github.com/gak/pycallgraph.

[12] R. Diestel. *Graph Theory*. Electronic library of mathematics. Springer, 2006.

[13] Alain Barrat, Marc Barthelemy, Romualdo Pastor-Satorras, and Alessandro Vespignani. The architecture of complex weighted networks. *Proceedings of the national academy of sciences*, 101(11):3747–3752, 2004.