Analyzing small world phenomenon and scale free network of python packages

Avadhoot Agasti
aagasti@indiana.edu

Abhishek Gupta
abhigupt@iu.edu

ABSTRACT

The *small world* phenomenon is a very important and interesting concept which can be practically used in optimizing imports for programming languages like python, java etc. This will also help us understand the network of the packages used by developers and classify them as scale free or random network. Further, this analysis can be extended to any other programming language and understand the package network. The scope of this project is to focus on python based packages and navigate to all dependent packages to build a network graph of these packages. Further, identify the important packages which could be like important nodes in scale-free network.

KEYWORDS

scale-free, random networks, small-world, packages

1 INTRODUCTION

When try to install a python library or rather when we use this python library first thing that we need to do is we have to *pip install* it. Most of the times, a python library is dependent on another library. Our idea is to build a network or graph of the python library dependency. In this network, every library will be the node of the network and the dependency will form the link. Such a graph can be queried to answer several questions like

- Which are the most core packages which are widely, directly or undirectly, used in large number of other packages?
- In which subject-area the new development is happening. We can pivot this solution around the small number of packages which are included in large number of packages. For example, if networkX is being used by lot of new packages then we can say that there is lot of development happening in network science
- What all packages will be impacted due to changes in a base package (e .g. if we find a severe bug in networkX, what are the other packages which can be potentially impacted due to the bug fix)

2 MOTIVATION

There is not much work done around this topic and after reading the concepts of network science it looks very interesting to find out how different module relate and how these relations can be interpreted. This paper *Power Laws in Software* [1] does analysis of power law distribution in a software application at class level and function level. It did analysis of java, perl, c/c++ etc applications and did establish a pattern that these applications do follow power law distribution.

Table 2: Review of other evidence.

Dataset	size	k_{in}	k_{out}
Java, C/C++ [Valverde and Solé 2003]	27-5,285	1.94-2.54	2.41-3.39
C/C++ [Myers 2003]	187-5,420	1.9-2.5	2.4-3.3
Java [Wheeldon and Counsell 2003]	NA	0.71-3.66	
Smalltalk [Marchesi et al . 2004]	1,797-3,022	2.07-2.39	2.3-2.73
Object graphs [Potanin et al . 2005]	15,064-1,259,668	2.5	3

Figure 1: Review of Languages

It depicts the size of datasets studied for this this analysis. However, we couldn't find a paper or research which analyzes these patterns on all modules present in a given language. Its very interesting problem to solve and find out how various python packages are dependent on each other and does it make sense to bundle some of the very common packages used along with base python distribution. It is interesting to understand the dependency structure but also to understand the community structure of these modules and what kind of network they form. Any other phenomenon exist when we analyze this graph. This analysis can be used to optimize the imports, remove cyclic dependencies between modules. Similar approach can be extended to other programming languages like java, ruby, perl etc.

3 REQUIREMENTS

The goal of this analysis is

- to create a network graph of these dependencies
- understand the dependencies and important nodes
- identify if its a scale-free network
- identify any other properties depicted by this graph
- update the graph and build the graph as and when needed

At the end of this exercise we should be able to run this analysis on available python [2] packages. Also, we need understand and compute the other properties of the graph like average degree, average clustering co-efficient, average path length etc Also verify if the graph is *scale free* or just a *random* graph.

4 TECHNICAL SOLUTION

4.1 Data Sources

Our approach is to identify the dependent packages by looking at the dependencies using *pip show* output. For example

```
pip show networkx

Name: networkx

Version: 1.11
```

```
Summary: Python package for creating and
manipulating graphs and networks

Home-page: http://networkx.github.io/

Author: NetworkX Developers

Author-email: networkx-discuss@googlegroups.com

License: BSD

Location: /Libs/anaconda/lib/python3.6/site-packages

Requires: decorator
```

Looking at the above output we can clearly see that *networkxx* depends on package *decorator* and further package *decorator* may be dependent on other package and so on. We continue to traverse we should be able to find all dependent packages till we reach code python libraries. Interestingly, there *128k* python packages available [2] which should give us a lot of data points for our analysis.

4.2 Solution

Python app builds the graph using the installed modules in the local virtual environment. It runs *pipdeptree*. The crawler module generates the dependency file with list of all recursive dependencies. Further, the parser recursively parses this dependency file and generates the directed graph as well as saves the graph in gml format.

Currently, this app relies on python modules installed in local environment only. Hence, we ended up installing bunch of python modules listed on pypi.org [3] to do this analysis. However, this work can be extended to crawl the python modules listed on pypi.org [3]. Installing all modules locally can be consume disk space, hence make sure you have enough disk space available if you plan to install lot of modules and run the analysis. Each package forms a node in the graph and each edge will represent the dependency with the next module. We should be able to plot this graph using *networkx* module and represent the most important nodes which become the hubs to the network. Also, compute the degree and clustering co-efficient for this graph. When you run the app, the app prints the number of vertices and edges along with average in and out degree of the the graph.

```
Type: DiGraph
Number of nodes: 242
Number of edges: 612
Average in degree:   2.5289
Average out degree:   2.5289
```

Further, we study this graph to identify if the graph follows the *power law* distribution or it is a *scale free* network or its just a *random* network. Also, see what kind of community structure these module form.

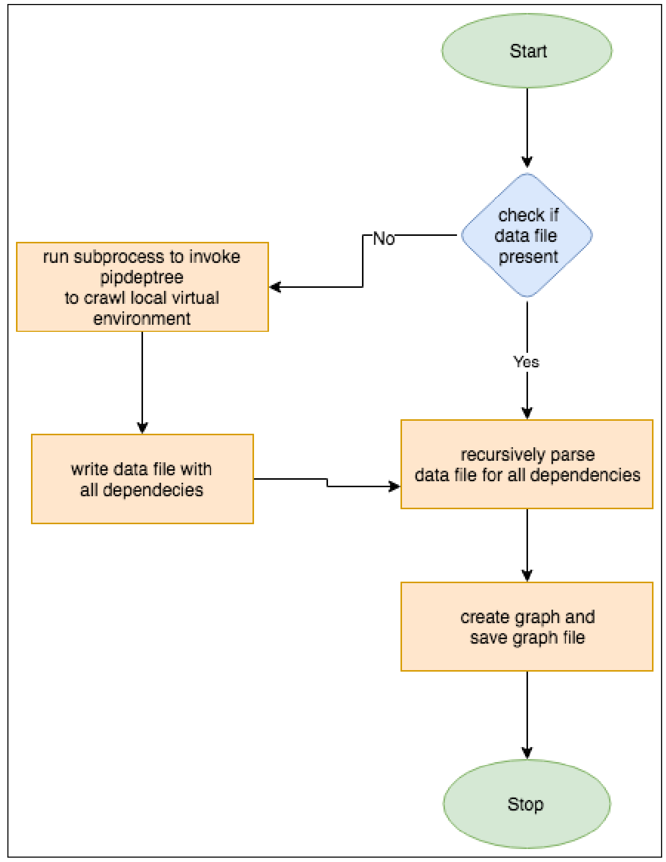

Figure 2: Flow chart

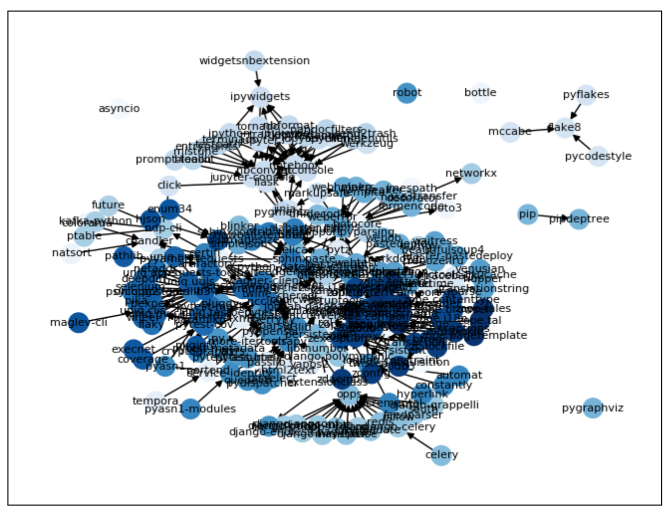

Figure 3: Generated Output Graph

4.3 Technology

We perform most of our coding in *Python* and *networkx* itself. We used *Gephi* for visualizing the network.

4.4 Challanges

During this exercise we ran into few technical challenges for example to perform this analysis we have to install each python package locally which could be really resource intensive specially in terms of disk space. Also, while doing the analysis and loading the graph for 128k packages could be memory and cpu intensive. To get around this problem we had to reduce the analysis to less number of python packages and keep adding mode packages as we make progress. Further enhancements can be made by running the crawler on pypi.org [3].

5 RELATED WORK

- The project Pipedtree - [4] is command line utility which allows user to see the installed packages in the form of dependency tree. However, this utility is focused more on solving dependency conflicts than the kind of analysis we propose to perform.
- Analysis of 30K Github projects [5]. The project was aimed to analyze the 30k different Java, Ruby and Javascript projects on Github to understand the top libraries being used. While their analysis was similar to what we plan to do, the approach was not network based.
- This paper *Power Laws in Software* [1] does analysis of power law distribution in a software application at class level and function level. It did analysis of java, perl, c/c++ etc applications and did establish a pattern that these applications do follow power law distribution

6 FURTHER ENHANCEMENT

Further enhancements can be done to crawler stage where instead of relying on modules installed locally it can crawl on module available on pypi.org. This will reduce an additional step of installing the modules locally and save the disk space as well as more flexible to crawl on web. Another enhancement can be made to make it more generic to do the analysis on any given language for example java maven dependencies or perl packages or ruby version manager and packages etc

7 ACKNOWLEDGEMENTS

The authors thank Prof. YY Ahn for his technical guidance. The authors would also like to thank TAs of Network Science class for their valued support.

8 REPO

All project and report document can be found at github project.

REFERENCES

[1] Panagiotis Louridas, Diomidis Spinellis, and Vasileios Vlachos. Power laws in software. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 18(1):2, 2008.
[2] Python.org. Python, packages. Web Page. Accessed: 2018-02-02.
[3] pypi.org. pypi, the python package index. Web Page. Accessed: 2018-02-02.
[4] Python.org. pipdeptree, command line utility to show dependency tree of packages. Web Page. Accessed: 2018-02-02.
[5] Tal Weiss. OverOps, blog by tal weiss. Web Page. Accessed: 2018-02-02.