An npm Network Analysis

Project Report

Peter Huettl
Northern Arizona University
Flagstaff, Arizona
ph289@nau.edu

Garrison Smith
Northern Arizona University
Flagstaff, Arizona
gts35@nau.edu

ABSTRACT

This paper is an analysis of the development packages delivered through the npm package management system [1]. The purpose of this analysis is to quantify the interdependence of these packages to determine the fragility of the system as a whole. To accomplish this, we have crawled npm for package and dependency data that we then collected [2] and saved into a network to gather network characteristics. Using network characteristics such as *connectedness*, *clustering coefficient*, and *dependence*, we were able to draw conclusions regarding the network's structure. We found that the graphs would produce data in which nodes were dependent on other nodes which made it so the npm management systems are fragile.

1 INTRODUCTION

There are several questions we are interested in answering with the analysis of this data. Namely, how fragile is the modern web development ecosystem? How reliant are popular websites on external libraries? These are the concerns present in the modern web ecosystem, especially for web developers in large companies[1]. This is due to the fact that almost all websites rely on external libraries[2] that are downloaded by package managers such as npm.

As mentioned in the abstract, we plan to address these questions by analyzing the network characteristics of the graph we have built by crawling for npm data [2]. Due to the unbiased nature of our collection, this approach will allow us to draw conclusions that generalize to the entire npm package management system.

2 MOTIVATION

Large companies with a prominent web presence often rely on consistent traffic to their site. This may be due to the fact that the site produces sizable ad revenue, or that they provide goods or services through their website. In either case, it is of the utmost importance that their site is always available to be visited by consumers. Due to the scale of these sites, it is often the case that JavaScript libraries are utilized to simplify the development processes. Popular libraries such as Bootstrap [3], jQuery [4], and Promise [5].

Although these package managers provide an overall positive service to the web development community, there are several different cases of package outages causing development issues. Firstly, there was the *left-pad* [6] incident. This was a situation in which a relatively innocuous package entitled *left-pad* was inadvertently removed from the npm system. This 11 line [7] package simply prefixed a given string with a given character to a specified length. Despite this, many other packages and websites had included and were reliant on this package. In turn, its removal caused the failure of these dependent sites.

[1] Explained in section 2 (Motivation)

[2] Note that libraries can chain dependencies

Unfortunately, this is not the only case, as there was a similar incident regarding the deletion of a user named *floatdrop* [8]. The user's package was automatically marked as spam, and consequently triggered the deletion of their account. Their account hosted several highly depended on packages, with one such example being the *require-from-string* package [9].

There have been other in-depth analyses of the npm dependency network. One such example [10] also highlighted metrics such as between-ness centrality but we plan to focus our efforts on a wider array of network characteristics to glean broader insight. This is why and how we plan to perform our network analysis.

3 DATA

Firstly, we needed to establish the fundamental network model of the network we were creating. It was clear that the entities of our network would represent packages hosted on npm. To create a network model that most directly addresses the original problems posed, we then established that the relationships in our network would denote one package depending on another. Because package dependencies are uni-directional, our network will be represented by a directed graph.

3.1 Crawling npm

Next, we began collecting the data, but, because npm deprecated the use of the /-/all registry endpoint [11] which had previously served as a list of all packages currently hosted on npm. This, although reasonable as they have recently surpassed 600,000 [1] packages, meant that we were going to have to crawl and collect our own subset of packages. As such, we wrote an `npm_crawler.py` [12] script that crawled using a modified Snowball Sampling method[3] [13].

Our sampling script would begin at a random package and iteratively crawl to dependencies and dependents, storing all associated package info and all found package names. This process would repeat until all connected packages were gathered, and then the search would restart from a random package. This ensured that we were also collecting the packages with no dependencies.

3.2 The Collected Data

To improve the reproducibility of our conclusions, we have open sourced our crawler, data, and our Jupyter notebook [14]. We created and ran our Jupyter notebook in a Docker container that was set up with DataQuest's Data Science Environment[4] [15]. This environment includes:

- python3
- numpy
- Scipy

Our crawler has generated several different JSON data files that are also hosted in the data folder of our GitHub repo [2]. This data is in its raw form in that it includes any possible information we thought could ever be valuable to our metrics. So, for example, an item in one of our JSON files would look as follows[5]:


```
{
    "author": {
        "email": "sindresorhus@gmail.com",
        "name": "Sindre Sorhus",
        "url": "sindresorhus.com"
    },
    "dependencies": {
        "time-zone": "^1.0.0"
    },
    "description": "Pretty datetime",
    "devDependencies": {
        "ava": "*",
        "xo": "*"
    },
    "license": "MIT",
    "name": "date-time"
}
```


[3] Algorithm only initially inspired by Snowball Sampling

[4] Specifically the dataquestion/python3-starter image

[5] The example is the `date-time` package

Table 1: Network Characteristics (Sample 1)

Network Characteristic	200 Edges	1,000 Edges	2,000 Edges
# of Nodes	353	1402	2493
Betweenness Centrality	$2.1x10^{-7}$	$7.1x10^{-8}$	$7.7x10^{-8}$
Density	0.0016	0.0005	0.0003
Transitivity	0	0	0.0012
Avg. Path Length	0.3879	0.5298	0.7607
Avg. Node Connectivity[6]	0	0	0
Avg. Clustering Coefficient	0	0	0.0001
Avg. Neighbor Degree[7]	0.0255	0.0879	0.2574
Avg. Closeness Centrality	0.0016	0.0005	0.0004
Avg. Degree Centrality	0.0032	0.001	0.0006

Table 2: Network Characteristics (Sample 2)

Network Characteristic	5,000 Edges	7,000 Edges	10,000 Edges
# of Nodes	4617	5638	6595
Betweenness Centrality	$5.3x10^{-7}$	$4.5x10^{-6}$	$2.1x10^{-5}$
Density	0.0002	0.0002	0.0002
Transitivity	0.0053	0.0074	0.0120
Avg. Path Length	2.8633	6.392	7.7207
Avg. Node Connectivity	0	0	0
Avg. Clustering Coefficient	0.0018	0.0043	0.0114
Avg. Neighbor Degree	0.6898	1.1954	1.8215
Avg. Closeness Centrality	0.0004	0.0008	0.0028
Avg. Degree Centrality	0.0005	0.0004	0.0005

4 ANALYSIS

We conducted our analysis by using 2 main methodologies. Firstly, we calculated many network characteristics on many different network sizes to ensure the validity of our results. Secondly, we analyzed the visualizations of the networks, and their metrics compared to one another.

4.1 Network Characteristics

We decided on 10 key network characteristics to analyze to provide the most relevant data to answer the original question we posed. As such, we calculated the following metrics on graphs with various edge counts:

[6] Our networks are already disconnected due to the inclusion of standalone packages so our average node connectivity will likely always be 0 (unless our random sample selects a clique of packages)

[7] The neighbor degree increases as edge count increases

- Number of Nodes
- Betweenness Centrality
- Density
- Transitivity
- Average Path Length
- Average Node Connectivity
- Average Clustering Coefficient
- Average Neighbor Degree
- Average Closeness Centrality
- Average Degree Centrality

We aggregated this data into Table 1 and Table 2 and labeled these tables with respect to the graph edge count and the metric being analyzed. We chose to highlight these metrics for the edge counts 200, 1,000, 2,000, 5,000, 7,000, and 10,000. We chose these intervals specifically to ensure that the conclusions we were drawing from the data were founded on patterns rather than random chance.

On the low-end, we chose to consider 200 package dependencies. We chose this number deliberately to model the average number of packages included in a single large website package[8]. This small-scale simulation allows us to apply the metrics we calculate to an actual, real-world project.

Conversely, on the high-end, we considered a sub-network with 10000 dependency connections. The purpose of this sub-network was to create a manageable[9] dataset that more closely resembles the entire npm network. From this, we were able to collect data that can be more-or-less scaled up to represent the entire npm network. This sub-network also provided some of the most interesting network characteristics due to its large size. For example, this sub-network allowed us to see that the average path length was increasing as we kept more paths in our network because it filled out the network rather than shortening the average path.

4.2 Graph and Metric Visualizations

The next category of analysis that we utilized was creating visualizations. There were two main categories of visualizations that we employed. Those being:

- Graph visualizations
- Metric visualizations

The graph visualizations were an attempt to translate our dependency networks into graphs that could be visually parsed. This was the more challenging of the two visualization methods due to the sheer size of the networks. Even smaller sub-networks resulted in a complicated mesh of nodes.

That being said, we were able to simplify the graphs to a consumable scale in Figures 1 and 2 on the following page. The red ring of circles in these figures are the nodes or the packages of our sub-network. As you can see, this ring is more densely packed in Figure 2 due to the increased sub-network size. Figure 1 was the 500 edge sub-network while Figure 2 was the 1000 edge sub-network.

Inside this ring of nodes, there are the directed edges denoting a package being dependent on the other nodes that the arrows leaving to point to. Our visualization denotes arrowheads as thicker rectangles to assist in differentiating them from the rest of the edge.

We have also included several charts that help visualize the change in our metrics as we increase the sample size. For every relevant network characteristic gathered, we have created charts to demonstrate this change. Every chart is labeled with the metric that was gathered, and the axes are labeled by the size of the sub-network and the value of the characteristic. These charts also help accentuate the true values of the metrics and how these metrics could be extrapolated to draw conclusions on the entire npm system.

[8] Keeping in mind that packages include other packages

[9] npm's entire network is 600,000 nodes

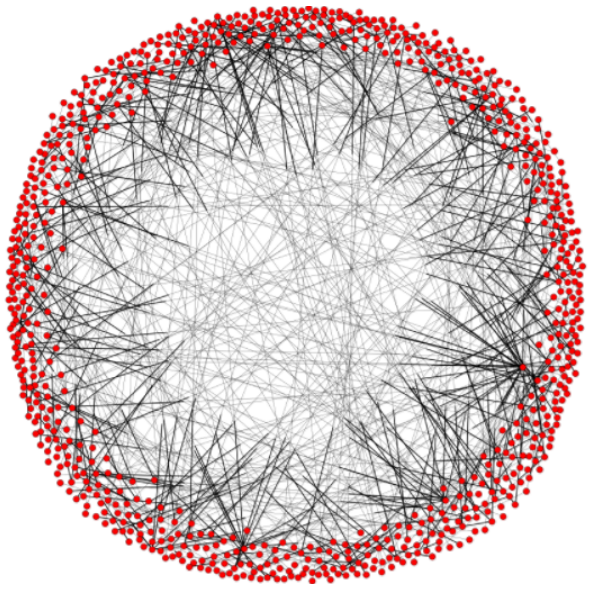

Figure 1: 500 Edge Sub-network

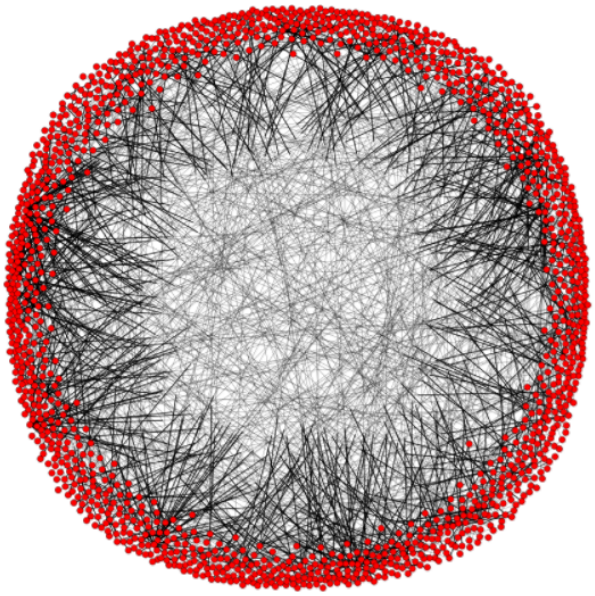

Figure 2: 1,000 Edge Sub-network

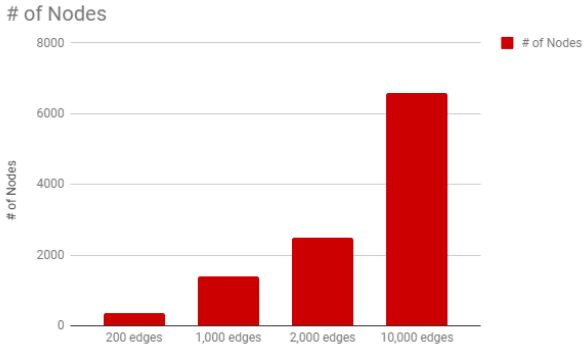

Figure 3: Number of Nodes

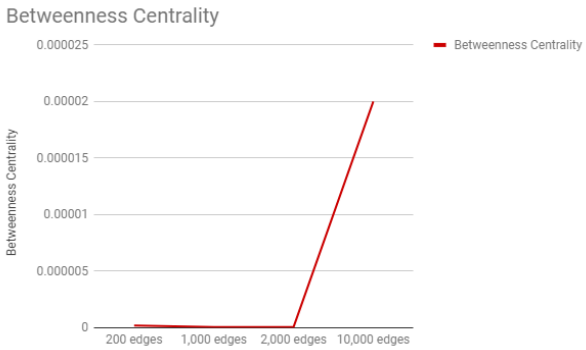

Figure 4: Betweenness Centrality

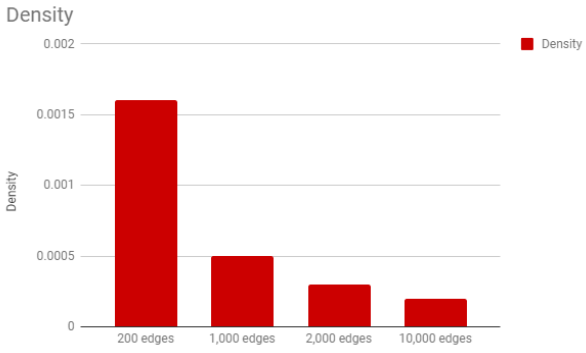

Figure 5: Density

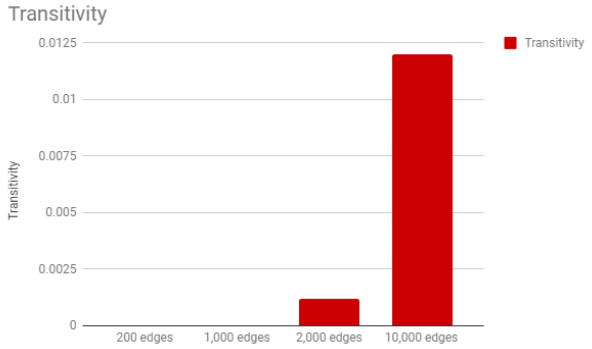

Figure 6: Transitivity

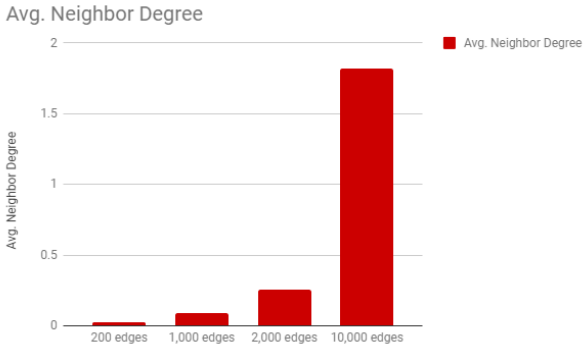

Figure 9: Average Neighbor Degree

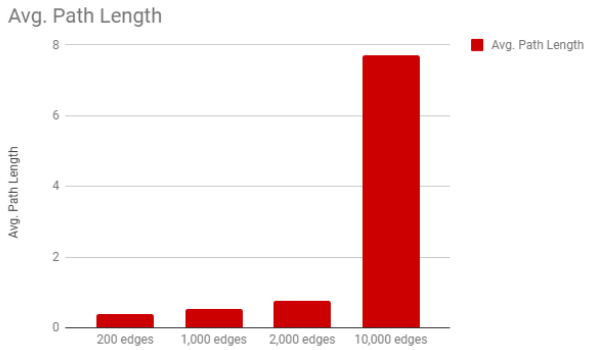

Figure 7: Average Path Length

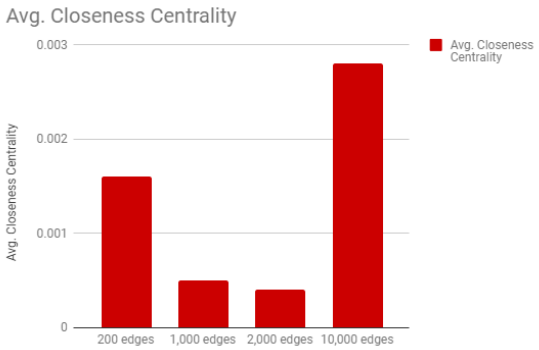

Figure 10: Average Closeness Centrality

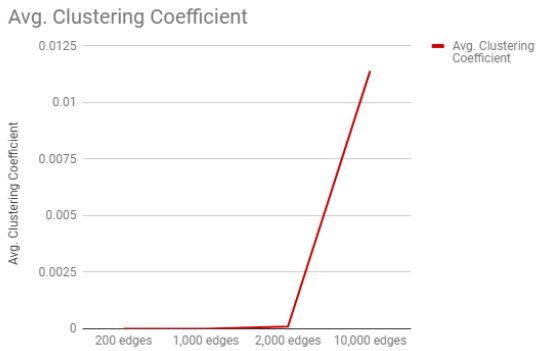

Figure 8: Clustering Coefficient

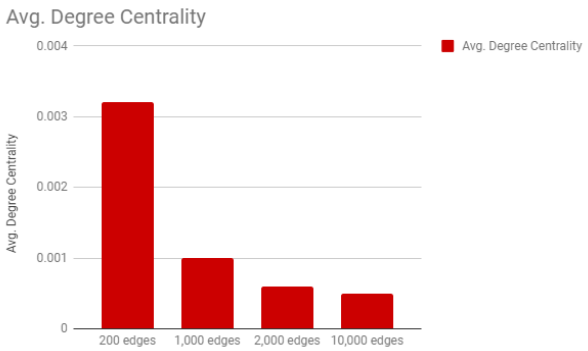

Figure 11: Average Degree Centrality

4.3 Outlier Data Points

To analyze specific packages, we calculated some metrics on individual nodes. In our analysis of the individual nodes, we found several interesting outlier data points. Firstly, we found that a package named *no-one-left-behind* [16] that depends on 1,000 total packages[10] which was the most dependent package in our sub-network.

The next interesting data point that we found was the package that was most depended on. In our sub-network, we found that a package named *lodash* [17] that was depended upon by 1357 packages, which was more than any other.

Finally, the last interesting metric we found was that when we ran the PageRank algorithm on our sub-network of 10,000, the highest value node was also *lodash* with a PageRank score of 0.0128.

Interestingly, upon further research, we found that these outlier packages we identified line up with the actual package data of npm [18]. In reality, *lodash* is the most depended upon package in the npm system, and according to other metrics ran, *no-package-left-behind* is the package with the most dependencies. In actuality, *lodash* is also the package with the highest PageRank score, but the score others have calculated is 0.0159.

The correspondence of our data with the actual data of the entire npm system demonstrates that our network model is an accurate scaled down representation of the characteristics that can be found throughout the entire package ecosystem. The preservation of trends is important to ensure that accuracy and reproducibility of our analysis.

[10] Only 964 in our sub-network due to data restrictions

5 FINDINGS

In our graphs and from our data above we can conclude that there are some modules that have no dependencies and many dependents and others that have many dependencies and no dependents and as well there are many that are in between the two. However by looking at our metrics that we found we can evaluate the risks that come from software packages to download counts to update frequencies and how this can effect our data.

Looking through our data we can show that most critical modules have cascading impact on other modules. Since there is a cascading relationship between these dependencies then we see that there is a removal package which can affect another package, which affects other packages, and so on and so forth. To justify this we can always look at a smaller data set within our data and check nodes that are incredibly small and continue to nodes that are exponentially bigger. Doing this we can see that the clustering coefficient for some of the graphs start to get out of hand and we start to see many dependent nodes that rely on other nodes to get their information. This idea then shows us that the nodes get more and more dependent on each other and there we can conclude that the web software needs certain updated software from a different software which then as said above goes into a downward spiral until we get to a software system that does not even work anymore. Which means that out of date software causes many other software products to shut down because they were dependent on a certain type and now they cannot get updated from that type anymore.

Another finding is that when there is a node connected to by 4 other nodes which make the initial node dependent on the other ones this can cause a vulnerability factor to play role in which the initial node becomes too dependent on other nodes and eventually will cause the chaos to happen between all nodes.

One aspect that we noticed was that the betweenness centrality would rise as we would add new nodes and the degree centrality would get small as new nodes were added. This caused some interesting ideas to arise while we were looking at the data. When a graph has high betweenness centrality and low degree centrality that means there are many clusters in the graph as the graph gets bigger. This also means that few and fewer nodes create bridges for other information to pull through these nodes. This can create a single point of failure more times than not which means that those failure points are the nodes that cause the chaos to happen when software packages get removed.

When we were calculating the graphs from our data that we had there was a something that happened while calculating transitivity and average path length. We did not notice it until we made out bar graphs but what happened is that in the lower levels of edges that we would calculate there was no data for transitivity and for average path length but when we hit 10000 edges the number skyrocketed and that seems intriguing. As we concluded earlier we noticed that there were many failure points that could happen in the graph and many nodes that were dependent on each other because of the degree centrality. However, transitivity states that if I can get to node x from node y and node y to node z then I can get from node x to node z. If this is the case then at 10000 nodes we had a very well average transitivity for the graph this would only mean that yes maybe there were many nodes together connected to each other they could have also been in a clique which means that there is a small bias that happens for these certain nodes.

Another idea is that because our connectedness was zero then there could have been small cliques that were not connected to the graph at all which was stated in the tables up above. This connectedness never became anything except zero so this can show that there is going to be at least one group that is possible in a transitivity state on the graph but is not connected to the rest of the graph this can also conclude that no matter what the graph is never connected.

Something that we noticed when calculating the density of our graphs was that when the edges were increased in the graph the nodes would start to cap at a certain point and the density would go down. This means that there are not many nodes that could potentially have connections to another node. In addition, this helped us better understand why certain nodes were not fully connected to the graph but would have made the transitivity high for the graph as well. This idea can also better understand what happens when we have nodes dependent on nodes which are dependent on more nodes. Furthermore better understand why there us failure nodes within modern web development applications. Finally, this can help conclude the idea of web development applications shutting down after a certain period of time.

The last finding that we found was that when we were finding metrics to build our data points we noticed that we could not get our diameter for our graph because there were too many separate nodes from each other on the graph and too many nodes that were not fully connected to the graph. This helps us better understand why we were getting the conclusions that we were getting in the data points and why the connectedness was always resulting in zero. In addition, we can state that the diameter was just impossible to find because of how the conclusions came to be.

As for all of our findings, the main idea was to show that there is going to be many nodes in our graph that are dependent on other nodes in the graph and those nodes are going to be dependent on other nodes as well. This can cause chaos within the graph because if one node wanted to get information about anything it would have to search through an entire tree and at the end of the tree there might not be a node that has the information it is looking for which can cause this node to fail and filter out all the other nodes that failed as well with it when they went to search for that data. These findings help us wrap up what we are trying to look for in the npm API and help us better understand what is happening in the modern web development ecosystem.

6 CONCLUSION

In conclusion, we were looking to answer the question: “how fragile is the modern web development ecosystem?” We were interested in this due to the implications such a question has on the modern web development ecosystem. We began by first crawling the npm web API to gather a meaningfully large dataset from which to draw conclusions. To accomplish this, we created a crawler tool that implements a modified Snowball Sampling method to iteratively build up a JSON file that stores information such as dependencies, package authors, and general package descriptions.

From this data, we were able to calculate meaningful network characteristics that allowed us to address the original question that we prompted. From our findings, we can conclude that the npm package management systems are fragile throughout each packages lifetime. This was shown through creating graphs in networkX that allowed us to output data on certain metrics. We were also able to take these metrics and produce them into tables that were later put into bar graphs and line graphs. These graphs from the tables and the data that we were able to pull from the metrics helped us better understand why these npm package management tools are fragile as time goes by. The main conclusion was that because nodes are dependent upon nodes which are dependent upon nodes this dependency can cause fragile nodes to take place within a graph which means that it can be very difficult to find a handful of important nodes within these graphs.

As before these ideas and conclusions that we found were based on all the facts that we were able to find in our graphs and the fact that npm had a nice API documentation to get all the nodes we need to justify our conclusion. As said before many large companies need to recognize that the web development packages that they may use could go out of date and if that were to happen then the that web development package could fail because if that package depends on other packages and those packages were to also fail because they depend on other packages then the idea of losing the web package and losing the traffic flow that goes through their website could devastate their company as a whole. The point of this project is to show the data that supports these claims and show that many packages that are used in web development are too reliant on each other and this can cause many companies to lose money.

To conclude the purpose of this project is to analyze the development packages delivered through the npm package management system. The general idea of this analysis is to quantify the interdependence of these packages to determine the fragility of the system as a whole. To accomplish this, we have crawled npm for package and dependency data that we then collected and saved into a network to gather network characteristics. The conclusion is that there are many packages that are dependent upon other packages and those packages are dependent on more packages and because of this scenario the network as a whole is fragile and can be easily broken.

REFERENCES

[1] Anon. npm. Retrieved April 28, 2018, from https://www.npmjs.com/

[2] Peter Huettl. 2018. petetetete/cs499-project. (March 2018). Retrieved April 28, 2018, from https://github.com/petetetete/cs499-project/tree/master/data

[3] Twitter. 18AD. bootstrap. (April 18AD). Retrieved April 28, 2018, from https://www.npmjs.com/package/bootstrap

[4] JS Foundation. 18AD. jquery. (January 18AD). Retrieved April 28, 2018, from https://www.npmjs.com/package/jquery

[5] Forbes Lindesay. 17AD. promise. (September 17AD). Retrieved April 28, 2018, from https://www.npmjs.com/package/promise

[6] Npmjs. 2016. kik, left-pad, and npm. (March 2016). Retrieved April 28, 2018, from https://blog.npmjs.org/post/141577284765/kik-left-pad-and-npm

[7] David Haney. NPM & left-pad: Have We Forgotten How To Program? Retrieved April 28, 2018, from http://www.haneycodes.net/npm-left-pad-have-we-forgotten-how-to-program/

[8] Npmjs. 2018. Incident report: npm, Inc. operations incident of January 6, 2018. (January 2018). Retrieved April 28, 2018, from https://blog.npmjs.org/post/169582189317/incident-report-npm-inc-operations-incident-of

[9] floatdrop. 18AD. require-from-string. (April 18AD). Retrieved April 28, 2018, from https://www.npmjs.com/package/require-from-string

[10] Burak Arikan. 2016. Analyzing the NPM dependency network – Graph Commons – Medium. (April 2016). Retrieved April 28, 2018, from https://medium.com/graph-commons/analyzing-the-npm-dependency-network-e2cf318c1d0d

[11] Npmjs. 2017. Deprecating the /-/all registry endpoint. (February 2017). Retrieved April 28, 2018, from https://blog.npmjs.org/post/157615772423/deprecating-the-all-registry-endpoint

[12] Peter Huettl. petetetete/cs499-project. Retrieved April 28, 2018, from https://github.com/petetetete/cs499-project/blob/master/src/npm_crawler.py

[13] Stephanie. Snowball Sampling: Definition, Advantages and Disdvantages. Retrieved April 28, 2018 from http://www.statisticshowto.com/snowball-sampling/

[14] Peter Huetl. petetetete/cs499-project. Retrieved April 28, 2018, from https://github.com/petetetete/cs499-project

[15] Vik Paruchuri. 2017. Docker: Data Science Environment with Jupyter. (December 2017). Retrieved April 28, 2018, from https://www.dataquest.io/blog/docker-data-science/

[16] zalestax. 18AD. no-one-left-behind. (January 18AD). Retrieved April 28, 2018, from https://www.npmjs.com/package/no-one-left-behind

[17] Anon. 18AD. lodash. (April 18AD). Retrieved April 28, 2018, from https://www.npmjs.com/package/lodash

[18] Anvaka. npm rank. Retrieved April 28, 2018, from https://gist.github.com/anvaka/8e8fa57c7ee1350e3491