Introduction to Web Science: Assignment #4

Dr. Nelson

Alexander Nwala

Thursday, October 9, 2014

Contents

Problem 1

From your list of 1000 links, choose 100 and extract all of the links from those 100 pages to other pages. We're looking for user navigable links, that is in the form of:

```
<A href="foo">bar</a>
```

We're not looking for embedded images, scripts, <link>elements, etc. You'll probably want to use BeautifulSoup for this.
For each URI, create a text file of all of the outbound links from that page to other URIs (use any syntax that is easy for you). For example:

```
site:
    http://www.cs.odu.edu/~mln/
links:
    http://www.cs.odu.edu/
    http://www.odu.edu/
    http://www.cs.odu.edu/~mln/research/
    http://www.cs.odu.edu/~mln/pubs/
    http://ws-dl.blogspot.com/
    http://ws-dl.blogspot.com/2013/09/2013-09-09-ms-thesis-http-mailbox.html
    etc.
```

Upload these 100 files to github (they don't have to be in your report).

SOLUTION 1

The solution for this problem is outlined by the following steps:

1. **Download links:** Given inputs derived from a file (**URIs.txt**) containing 1000 unique URIs downloaded from twitter, all links from all the URIs were extracted using Beautiful soup (Listing 1.) and stored in individual files as well as a master file of format:

```
parentURI0
    childURI0
    childURI1
parentURI1
    childURI0
    childURI1
.
.
.
parentURI(n-1)
    childURI0
    childURI1

In this representaion, childURI(n) was extracted from the
parentURI(n)
```

Listing 1: Extract links

```
#extract all children URIs
...
deleteIfFileEmptyFlag = True
for link in soup.find_all('a'):

    potentialURI = link.get('href')
    try:
     if( timeoutValueInSeconds > 0 ):
          response = urllib2.urlopen(potentialURI, timeout=timeoutValueInSeconds)
     else:
          response = urllib2.urlopen(potentialURI)
    except:
     continue
...
```

2. **Select 100 parent/child URI files:** This was achieved through the following selection criteria:

    a. Expand shortened URIs (Listing 2)
    b. To limit the size of the graph, if a parent has over 50 child links, select 50 or 25
    c. Given a collection of child links, select those that offer the most variety (those from a different domain than already selected)

The final set of 100 parent/child URIs were stored in a master file (**100_parentChildLinks.txt**). Finally, from this file, 100 individual parent/child files were written into the folder **100Links**

Listing 2: Expand short URIs

```
#Expand url
def followTheRedirectCurl(url):
     if(len(url) > 0):
          try:
               r = requests.head(url, allow_redirects=True)
               return r.url
          except:
               return url
     else:
          return url
```

Problem 2

Using these 100 files, create a single GraphViz "dot" file of the resulting graph. Learn about dot at:

```
Examples:
    http://www.graphviz.org/content/unix
    http://www.graphviz.org/Gallery/directed/unix.gv.txt

Manual:
```

```
    http://www.graphviz.org/Documentation/dotguide.pdf

Reference:
    http://www.graphviz.org/content/dot-language
    http://www.graphviz.org/Documentation.php

Note: you'll have to put explicit labels on the graph, see:
   https://gephi.org/users/supported-graph-formats/graphviz-dot-format/
   (note: actually, I'll allow any of the formats listed here:
   https://gephi.org/users/supported-graph-formats/
   but ``dot'' is probably the simplest.)
```

SOLUTION 2

Given input from the master file of 100 URIs of parent/child links (**100_parentChildLinks.txt**), due to Listing 3, the input file was converted into a dot file.

```
    The file parentChildURIs.dot contains the complete graph
```

Listing 3: Convert to dot file

```
#Convert to dot file
...
outputFile.write('digraph ParentChildLinks {\n')
     listOfLabelsUncompressed = []
     if( len(lines) > 0 ):
          for uri in lines:
               #is this parent
               uri = uri.lower()
               if( uri[0] == 'h' ):
                    #print "parent:", uri
                    currentParent = uri.strip()
                    listOfLabelsUncompressed.append(uri)
               else:
                    #print "...child:", uri
                    stringToWrite = '"' + currentParent.strip() + '" -> "' + uri.strip
                        () + '"'
                    outputFile.write(stringToWrite + '\n')
                    listOfLabelsUncompressed.append(uri)

          labels = getCompressedURLName(listOfLabelsUncompressed)

          #label graphs
          if( len(labels) == len(listOfLabelsUncompressed) ):
               for i in range(0, len(listOfLabelsUncompressed)):
                    outputFile.write('"'+ listOfLabelsUncompressed[i].strip() + '"' +
                        ' [label="'+ labels[i].strip() +'"];' +'\n')

          outputFile.write('\n}')
          outputFile.close()
...
```

In order to facilitate visualization, the following label shortening scheme was applied:

1. Remove the scheme (http://, https://, etc)
2. Remove www.
3. Remove path
4. remove tld (.com, .edu, .org, etc)

Problem 3

Download and install Gephi: https://gephi.org/
Load the dot file created in #2 and use Gephi to:

```
- visualize the graph (you'll have to turn on labels)
- calculate HITS and PageRank
- avg degree
- network diameter
- connected components
```

Put the resulting graphs in your report.
You might need to choose the 100 sites with an eye toward creating a graph with at least one component that is nicely connected. You can probably do this by selecting some portion of your links (e.g., 25, 50) from the same site.

SOLUTION 3

In order to better visualize the graph, I chose the Fruchterman-Reingold algorithm[2] to render the graph. This method belongs to a class of force-directed algorithm which help reduce edge crossing.

Furthermore, in order to explore the connectedness of the graph I chose the YiFan Hu proportional algorithm[1] (also a force-directed algorithm) to visualize the graph. From Figure 3, we can see the graph has many isolated partitions (hub/spoke) and a couple of links exist between these.

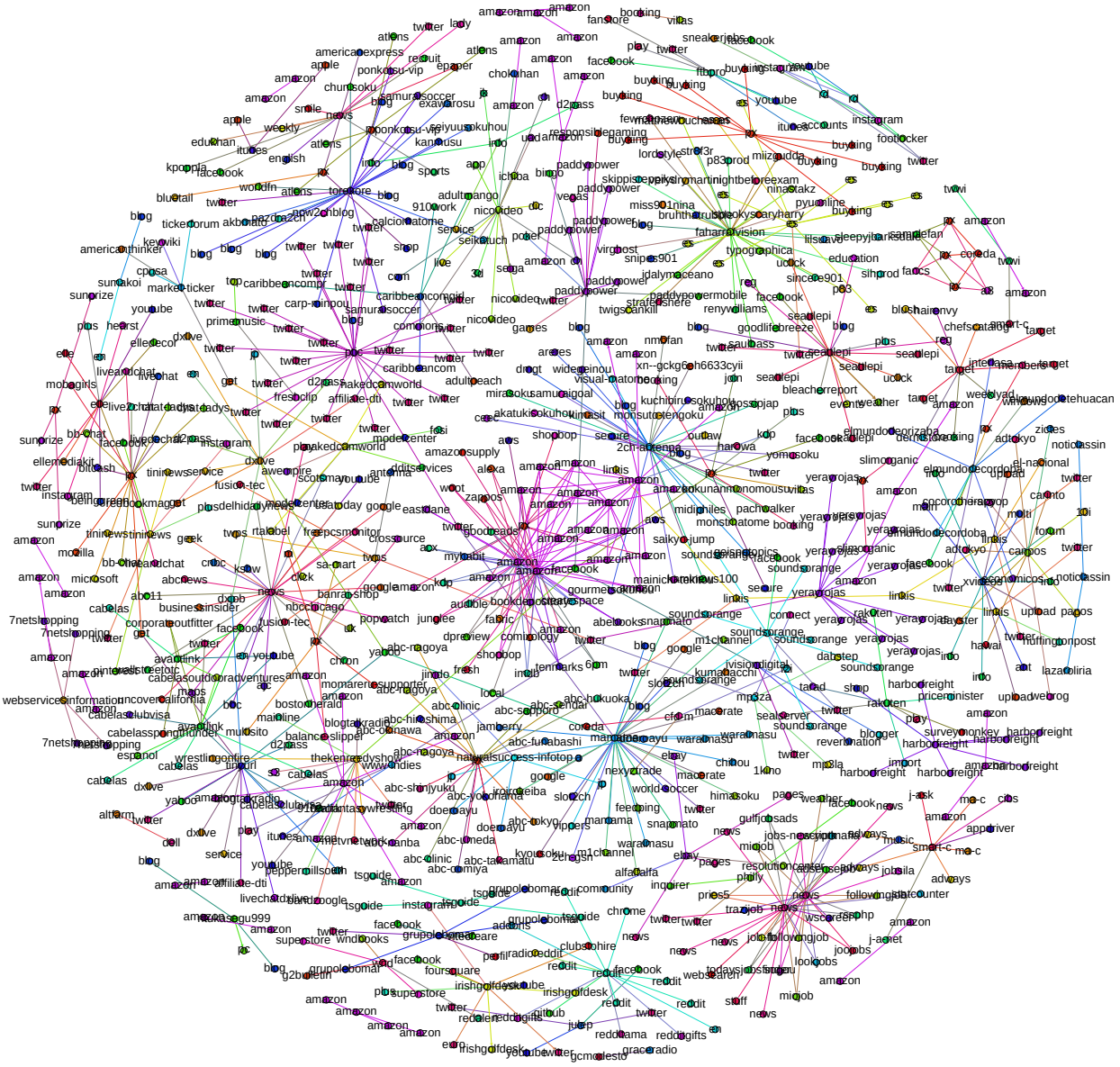
In addition, to add another dimension to the graph, the nodes are labelled and colored according to the respective labels.

The resultant directed graph contains 781 nodes adn 801 edges. In Figures 1-3, common colors (node/edge) represent labels with common names. Note that due to the label shortening scheme, multiple nodes have the same labels, eventhough they have different full names.

From Figures 1 and 2, since the links were extracted from random site, the graph tends has many isolated partitions: this is reflected by the number of weakly connected components.

The following parameters were derived from the graph:

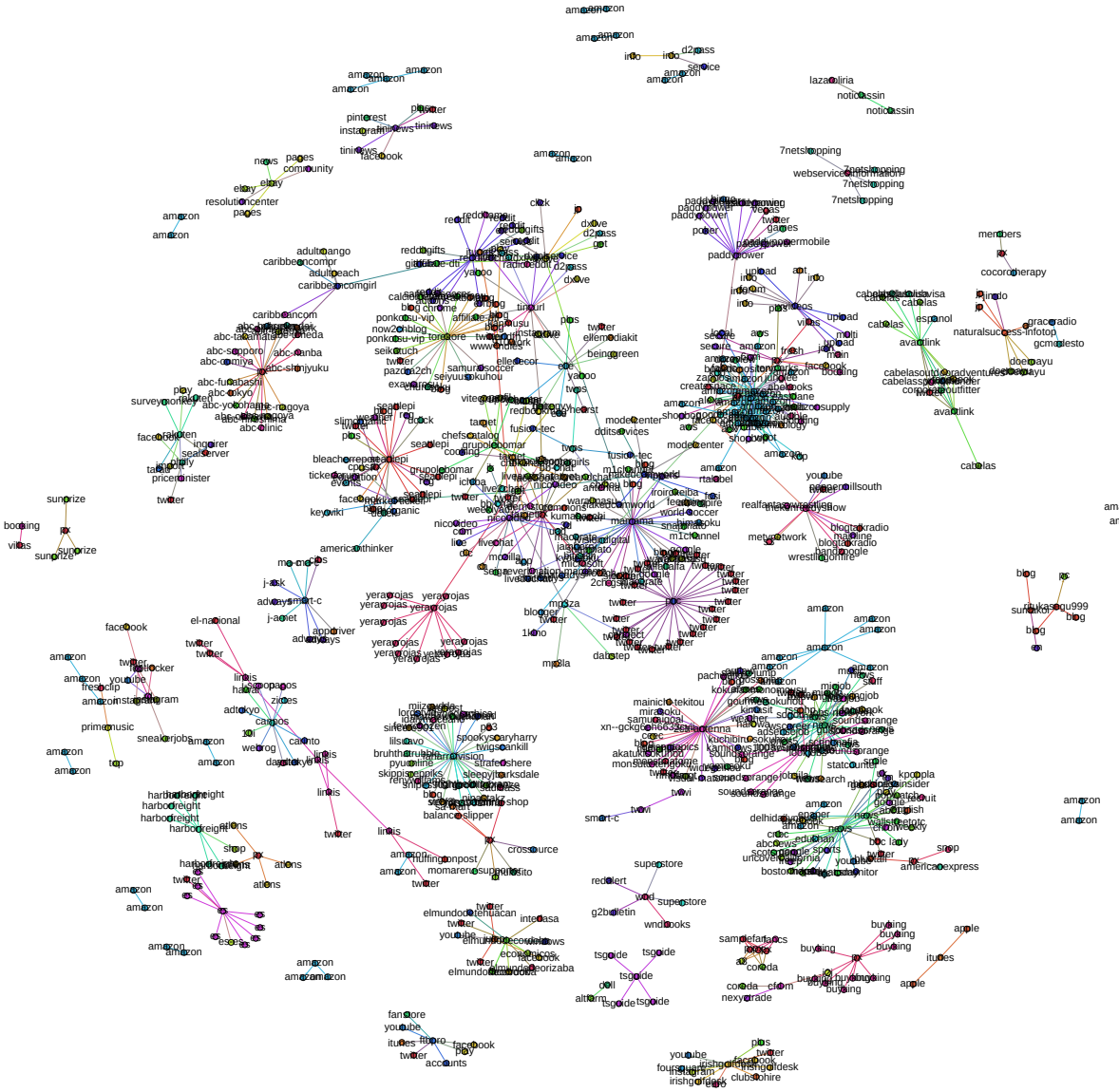
Figure 1: All nodes, Fruchterman Reingold visualization algorithm



Problem 3 continued on next page. . .

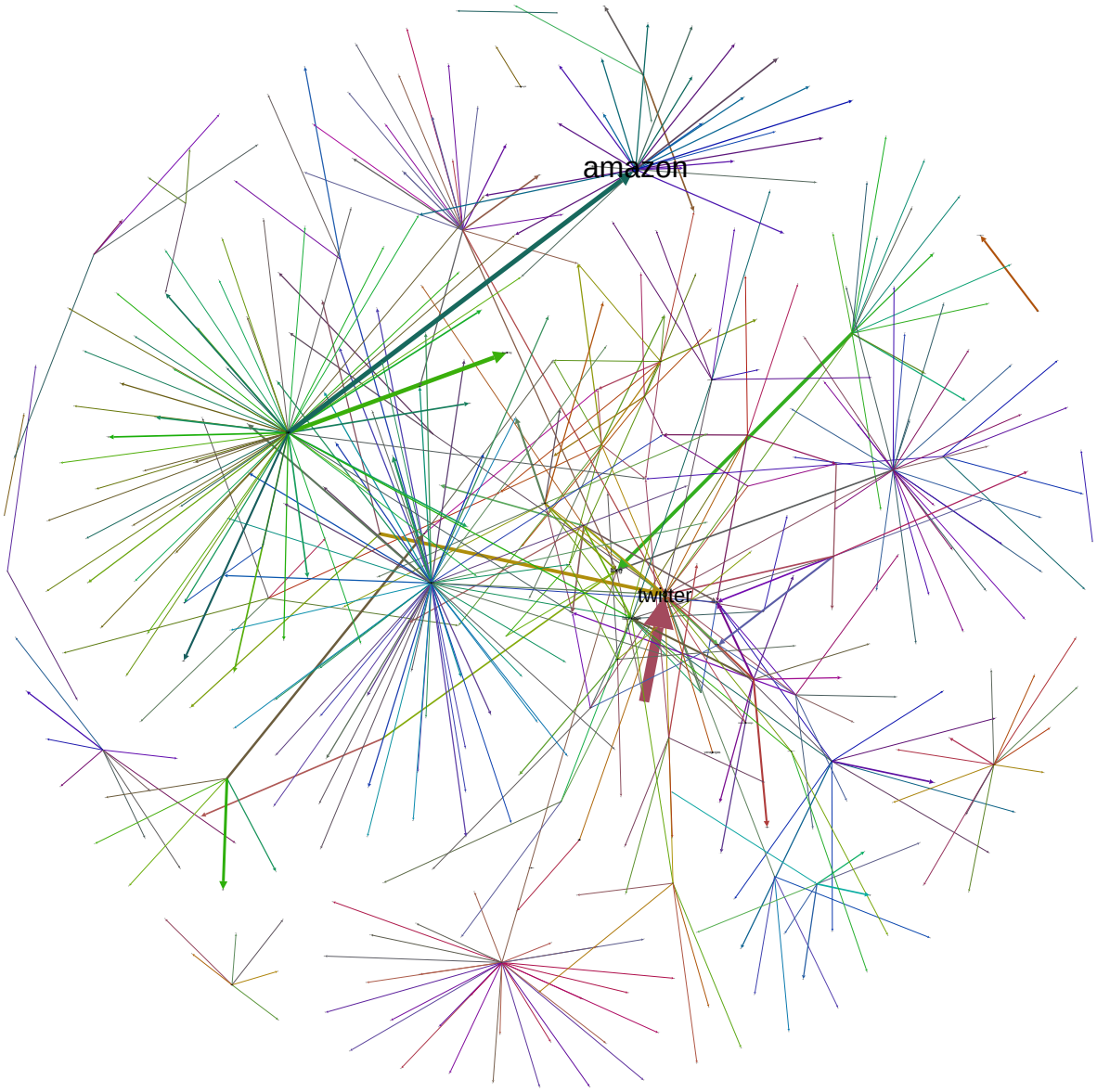
Figure 2: All nodes with boxed labels, Fruchterman Reingold visualization algorithm

Figure 3: All nodes, Yifan Hu Proportional visualization algorithm



Problem 3 continued on next page. . .

Figure 4: Partitioning based on node importance



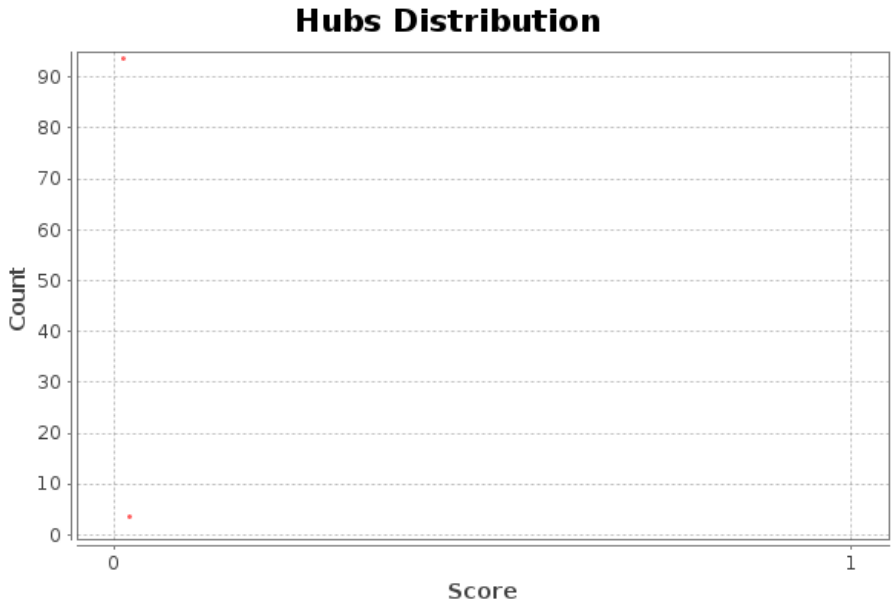
Problem 3 continued on next page. . .

HITS Metric Report

Parameters:

E = 1.0E-4

Results:



Algorithm:

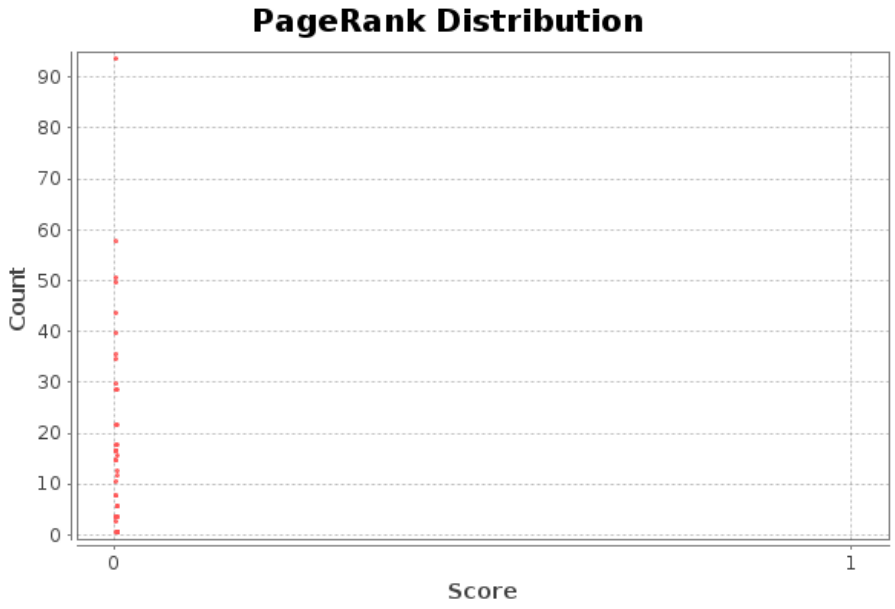
Jon M. Kleinberg, *Authoritative Sources in a Hyperlinked Environment*, in Journal of the ACM 46 (5): 604–632 (1999)

PageRank Report

Parameters:

Epsilon = 0.001
Probability = 0.85

Results:



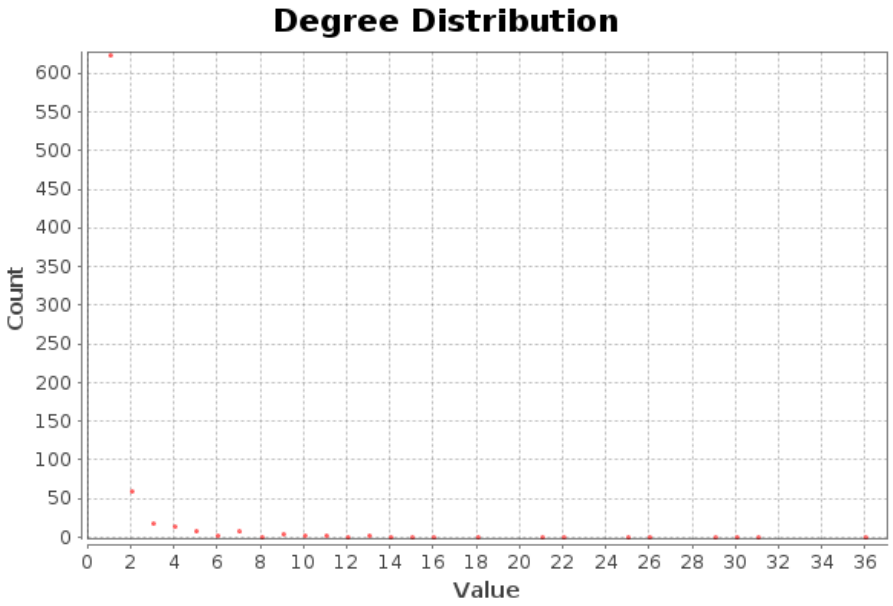
Algorithm:

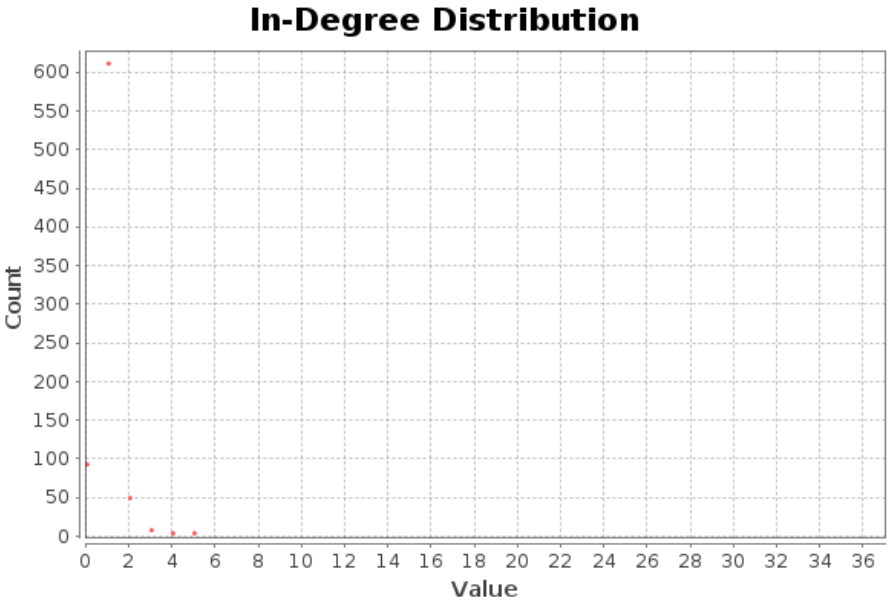
Sergey Brin, Lawrence Page, *The Anatomy of a Large-Scale Hypertextual Web Search Engine*, in Proceedings of the seventh International Conference on the World Wide Web (WWW1998):107-117

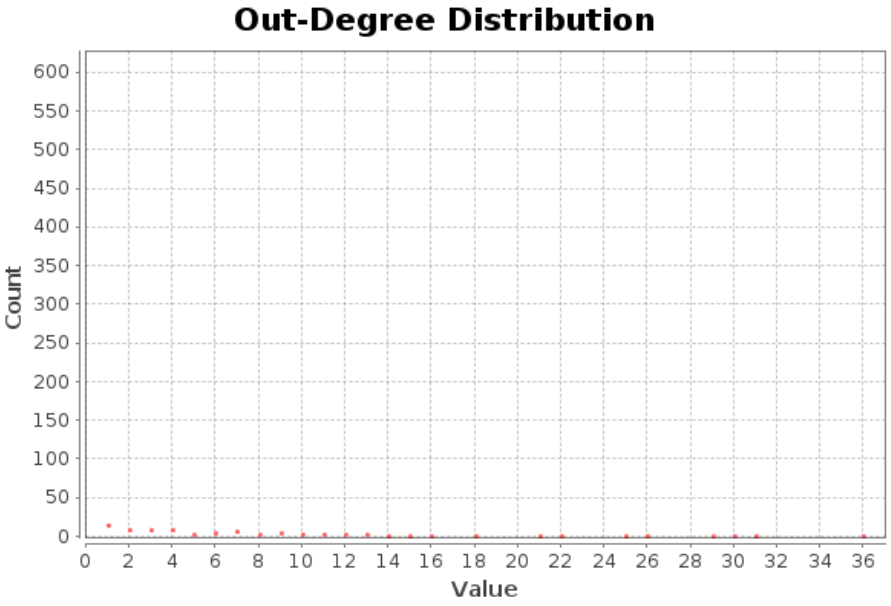
Degree Report

Results:

Average Degree: 1.026




In-Degree Distribution
Count
Value
0
50
100
150
200
250
300
350
400
450
500
550
600
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36


Out-Degree Distribution
Count
Value
0
50
100
150
200
250
300
350
400
450
500
550
600
0 2 4 6 8 10 12 14 16 18 20 22 24 26 28 30 32 34 36

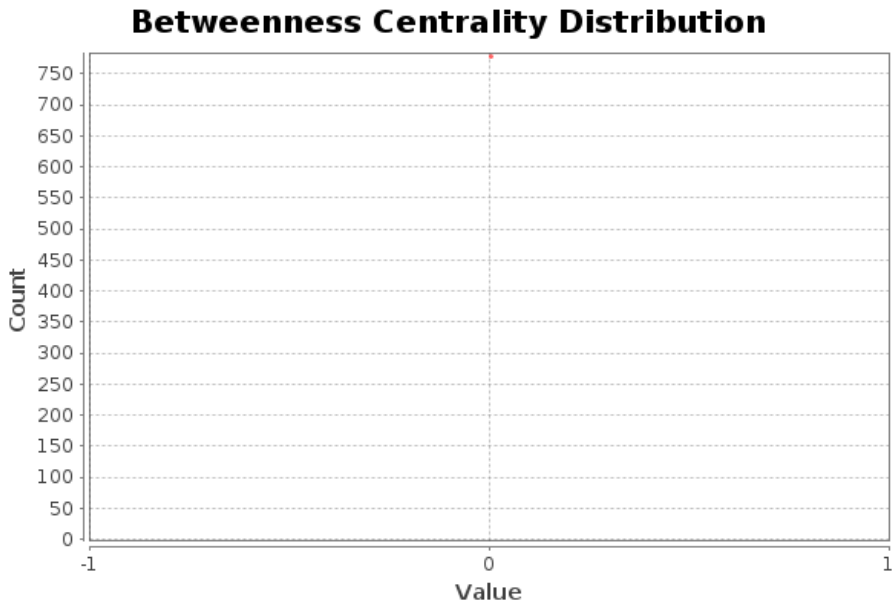
Graph Distance Report

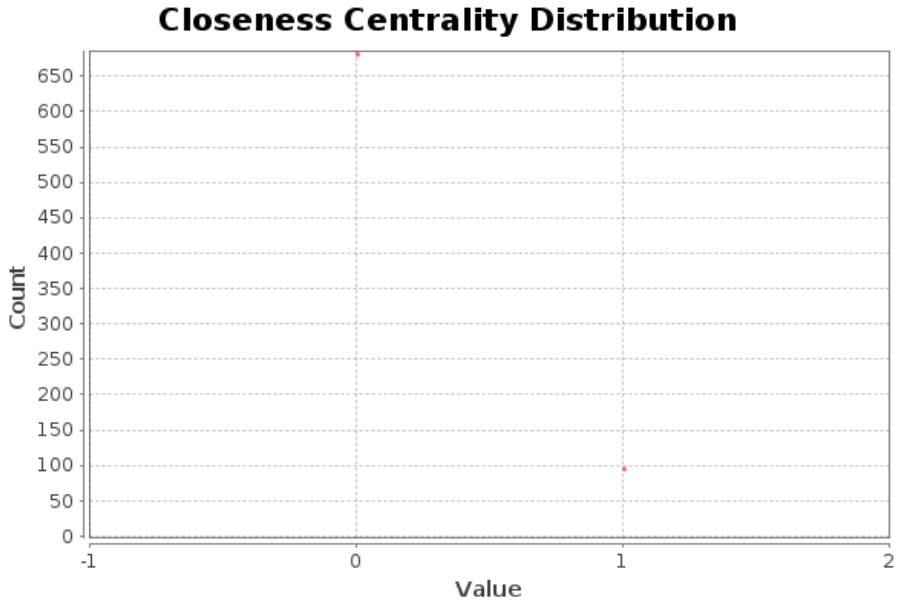
Parameters:

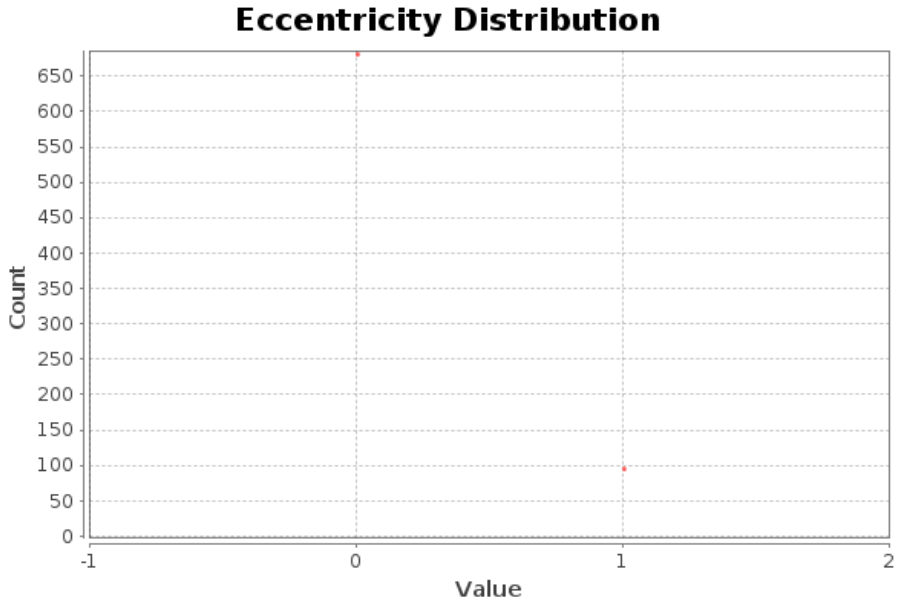
Network Interpretation: directed

Results:

Diameter: 1
Radius: 0
Average Path length: 1.0
Number of shortest paths: 797







Algorithm:

Ulrik Brandes, *A Faster Algorithm for Betweenness Centrality*, in Journal of Mathematical Sociology 25(2):163-177, (2001)

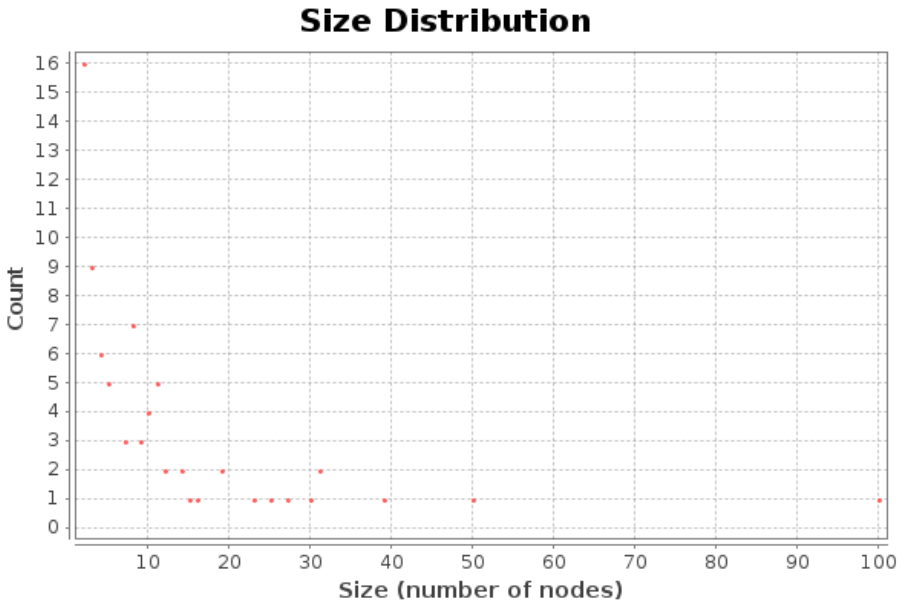
Connected Components Report

Parameters:

Network Interpretation: directed

Results:

Number of Weakly Connected Components: 75
Number of Stronlgy Connected Components: 781



Algorithm:

Robert Tarjan, *Depth-First Search and Linear Graph Algorithms*, in SIAM Journal on Computing 1 (2): 146–160 (1972)

Summary: The number of strongly connected components given by Gephi is incorrect since it is clear the graph is made up of multiple isolated partitions.

Even though the graph has many disconnected partitions due to the random nature of the data, we can see that the nodes with label "amazon" and "twitter" are very important since they are linked by other partitions regardless of the random nature of the data (Figure 4).

References

[1] Efficient and High Quality Force-Directed Graph Drawing. http://yifanhu.net/PUB/graph_draw_small.pdf. Accessed: 2014-10-09.

[2] Fruchterman-reingold. https://wiki.gephi.org/index.php/Fruchterman-Reingold. Accessed: 2014-10-08.