Linear Programming and Network Flows Final Report

Steven Glasford

15 May 2020

All figures are attached to the bottom of this document and all of the python code for the parts of the project exist first within the their particular sections

1 Part 1

```python
import scipy.io as sio
from scipy.sparse import csc_matrix as csc
import networkx as nx
import numpy as np

mat_contents = sio.loadmat("Homo_sapiens.mat")
dense = mat_contents['network'].todense()
G = nx.from_numpy_matrix(dense)
numberOfNodes = np.shape(dense)[0]
numberOfEdges = np.sum(np.tril(dense))
averageDegree = (np.sum(dense) + np.trace(dense)) / np.shape(dense)[0]
numberOfSelfLoops = nx.number_of_selfloops(G)
# print(dense)

# G=nx.read_edgelist(dense)


print(
    "Part 1 of Final Project\n"+
    "Data from NetworkX\n" +
    nx.info(G) +

    "\n\nData from calculations from the adjacency list" +
    "\nIs the graph directed: " + str(nx.is_directed(G)) +
    "\nNumber of nodes: " + str(numberOfNodes) +
    "\nNumber of edges: " + str(numberOfEdges) +
    "\nNumber of self loops: " + str(numberOfSelfLoops) +
    "\nAverage degree: " + str(averageDegree) +
    "\n\nThe information from NetworkX is the same as the \n" +
    "calulations on the adjacency matrix, so we can have a high\n"
    +
    "level of confidence that our network is correct."
```

```
)
```

For the code for part 1 please refer to figure above. This produces the output in figure 1, which also produces the graph seen in figure 2.
To answer the question poised in this part, we are able to conduct testing on this code by making calculations on the matrix distinct from those calculations provided in NetworkX, which is what is apparent at the end (lines 16, 17, and 18) of figure 1.

2 Part 2

```
import part1

G = part1.G
nx = part1.nx
plt = part1.plt

print("\nPart 2 of the final project for CSC453")

print("Is G connected:", nx.is_connected(G))
print("The Degree Centrality of G:")
dc = nx.degree_centrality(G)
plt.figure()
plt.hist(list(dc.values()))
plt.show()

print("The Closeness Centrality of G:")
cc = nx.closeness_centrality(G)
plt.figure()
plt.hist(list(cc.values()))
plt.show()

print("The Betweenness Centrality of G:")
bc = nx.betweenness_centrality(G)
plt.figure()
plt.hist(list(bc.values()))
plt.show()

print("The Harmonic Centrality of G:")
hc = nx.harmonic_centrality(G)
plt.figure()
plt.hist(list(hc.values()))
plt.show()

print("The Eigenvector Centrality of G:")
ec = nx.eigenvector_centrality(G)
plt.figure()
plt.hist(list(ec.values()))
plt.show()

print("The Page Rank of G:")
pr = nx.pagerank(G)
plt.figure()
plt.hist(list(pr.values()))
plt.show()

```

```
print("The average clustering coefficient fo G:", nx.
    average_clustering(G))
cco = nx.clustering(G)
plt.figure()
plt.hist(list(cco.values()))
plt.show()
```

The code for this section can be found in figure above.
Part 2 required many different graphs to be outputted in particular degree centrality (figure 13), closeness centrality (figure 14), betweenness centrality (figure 12), harmonic centrality (figure 11), eigenvector centrality (figure 10), page rank (figure 9), and clustering (figure 8).
The output for this python code can be found in figure 7.

3 Part 3

```
import part2

G = part2.G
nx = part2.nx
plt = part2.plt

print("\nPart 3 of CSC 453 final project.")
print("Circular Layout of G:")
nx.draw_circular(G)
plt.show()

print("The Kamanda-Kawai force-directed layout of G:")
nx.draw_kamada_kawai(G)
plt.show()

print("The plannar layout of G")
nx.draw_planar(G)
plt.show()

print("Random layout of G")
nx.draw_random(G)
plt.show()

print("Spectral Layout of G")
nx.draw_spectral(G)
plt.show()

print("Spring Layout of G")
nx.draw_spring(G)
plt.show()

print("Shell Layout of G")
nx.draw_shell(G)
plt.show()

print("No layout looks well for our network since our network is
    extremely large.")

# Read the new file created by deepwalk
f = open("deeper", "r")
```

```
arr = []

# read the file in line by line
for line in f:
    col = []
    # save only the last 2 columns
    i = 0
    for j in line.split():
        col.append(j)
    arr.append(col)

# close the file after reading
f.close()

# # The next two lines are only for testing and not used in
    production
# for line in arr:
#     print(line[1], line[2])

# The following for statement saves the first column into an x
    variable
# used for the x variable in the graph
x = []
for line in arr:
    x.append(line[1])

# The following for statement saves the first column into an y
    variable
# used for the y variable in the graph
y = []
for line in arr:
    y.append(line[2])

# The following two lines make the scatter plot for the assignment
plt.scatter(x, y)
# remove the x and y axis names in the plot as they too bunched up
    to
# proved any real meaningful details
plt.xticks([])
plt.yticks([])
plt.show()

# plt.scatter(arr[:, 0], arr[:, 1])
```

The code for this section can be found in figure above. There were several problems with the running of this code however, the only graph that was really able to run and display properly (as in display something that wasn't completely blackended by arcs to nodes as in figure 6), was the Spring layout drawing (figure 5).

- **Circular**: too big and not much information (figure 6)
- **Kamanda Kawai**: I could not run this graph in a reasonable amount of time, and system seemed to hang for hours.
- **Planar**: Our nodes are not planar so they cannot be represented in a plane.

- **Random**: Graph is too big, not much information is shown.
- **Spectral**: Took too long (several hours and produced nothing).
- **Spring**: The most interesting of the graphs, there is a level of separation in the graph that made it more interesting, however the user needs to zoom into the figure in order to actually see much information and it over all looks very chaotic. Figure 5.
- **Shell**: Looks very similar to the circular drawing in figure 6, which again doesn't show very much information.

In the end none of the graphs look very interesting or have too much information to extrapolate very much useful insight into the data, maybe Kamada Kawai is a good looking graph but it took too long to run on my machine and proved to be unsuccessful for me. The spring graph was a good looking graph, but its data is still bunched up to a point in which the data one can extrapolate from is limited.

The next part of part 3 had us us a program called deepwalk, after installation I ran the statement in figure 4. This produced a file that needed the first line removed, so instead of writing code to do that, I simply just edited the file and removed the first line in the file. Next, I added the code starting from line 43 until the end of the file in figure for the code in part 3. This resulted in a completely linear output as seen in figure 3.

4 End Remarks

Thank you for being a wonderful professor, you have been one of my favorite professor this semester. Stay healthy.

```
Part 1 of Final Project
Data from NetworkX
Name:
Type: Graph
Number of nodes: 3890
Number of edges: 38739
Average degree:  19.9172

Data from calculations from the adjacency list
Is the graph directed: False
Number of nodes: 3890
Number of edges: 38739.0
Number of self loops: 894
Average degree: 19.917223650385605

The information from NetworkX is the same as the
calulations on the adjacency matrix, so we can have a high
level of confidence that our network is correct.
```

Figure 1: Part 1 output

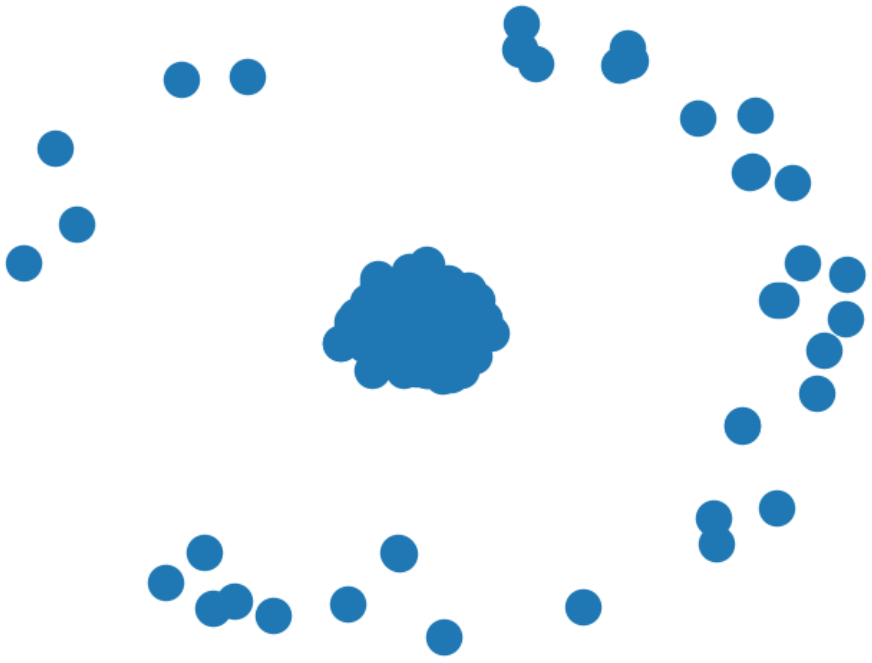

Figure 2: Network of Proteins for Homo Sapiens

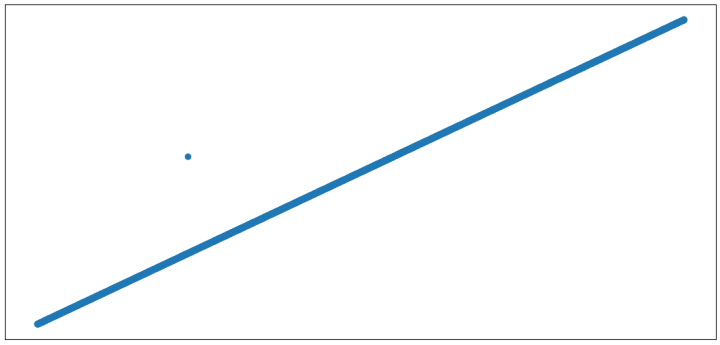

Figure 3: The graph produced by deep walk

```
$ deepwalk --format mat --input Homo_sapiens.mat --output deeper
--matfile-variable-name network --representation-size 2
```

Figure 4: Command to run deepwalk on our dataset

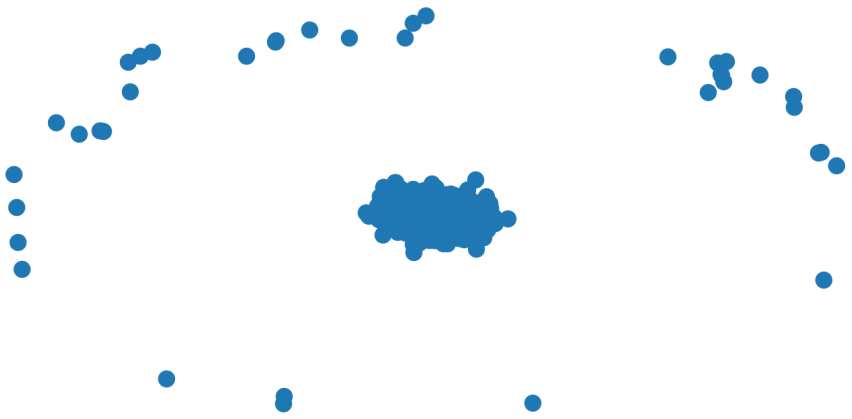

Figure 5: Spring drawing for part 3

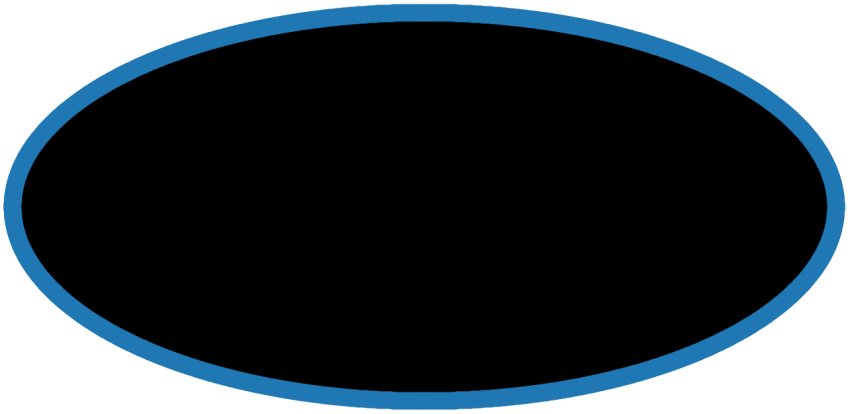

Figure 6: Circular layout for part 3

```
Part 1 of Final Project
Data from NetworkX
Name:
Type: Graph
Number of nodes: 3890
Number of edges: 38739
Average degree:  19.9172

Data from calculations from the adjacency list
Is the graph directed: False
Number of nodes: 3890
Number of edges: 38739.0
Number of self loops: 894
Average degree: 19.917223650385605

The information from NetworkX is the same as the
calulations on the adjacency matrix, so we can have a high
level of confidence that our network is correct.

Part 2 of the final project for CSC453
Is G connected: False
The average clustering coefficient fo G: 0.14644876266464119
```

Figure 7: Output for the code in Part 2

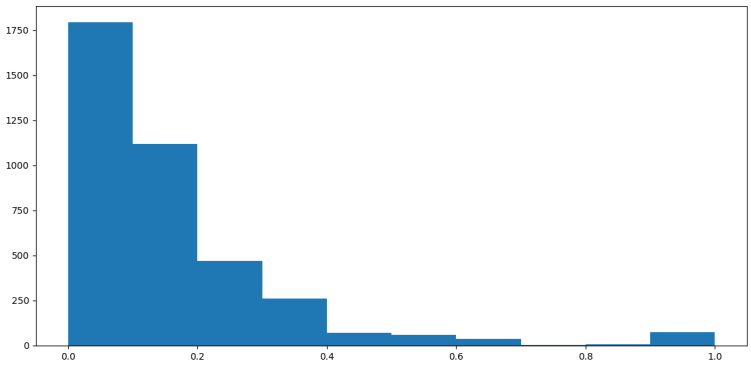

Figure 8: Clustering graph for Part 2

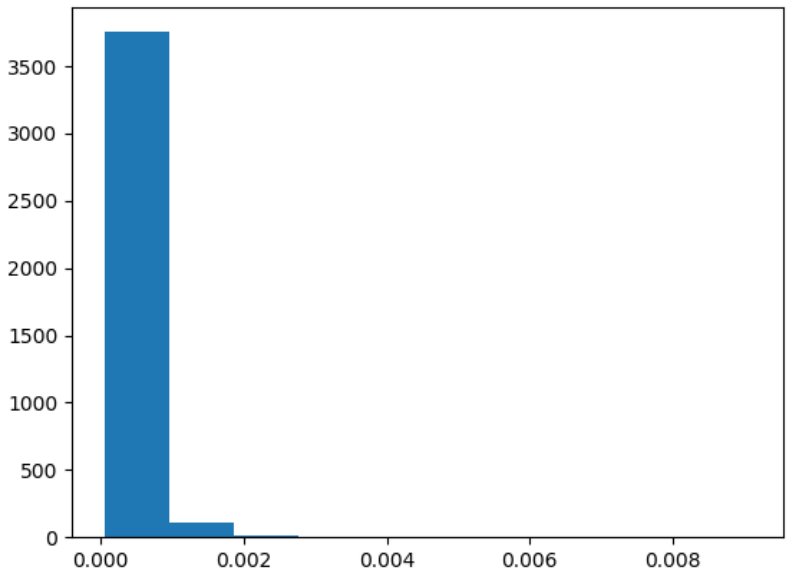

Figure 9: Page Rank graph for Part 2

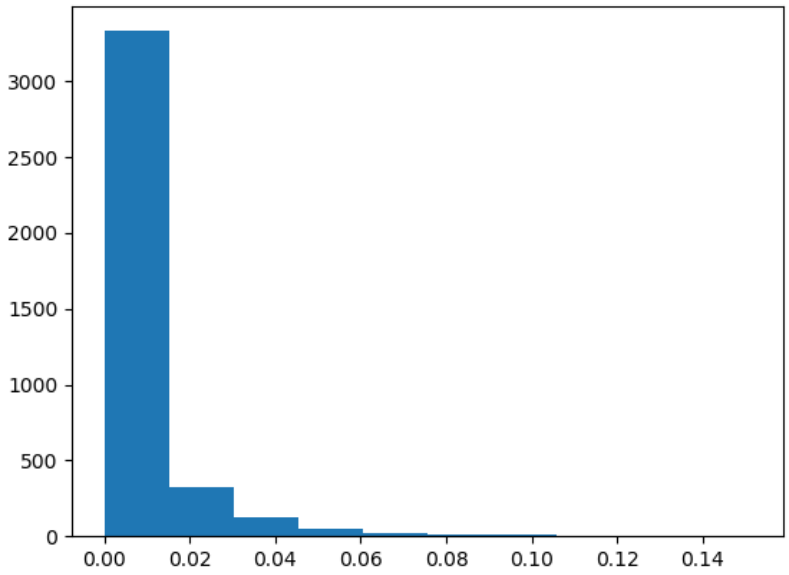

Figure 10: Eigenvector Centrality for part 2

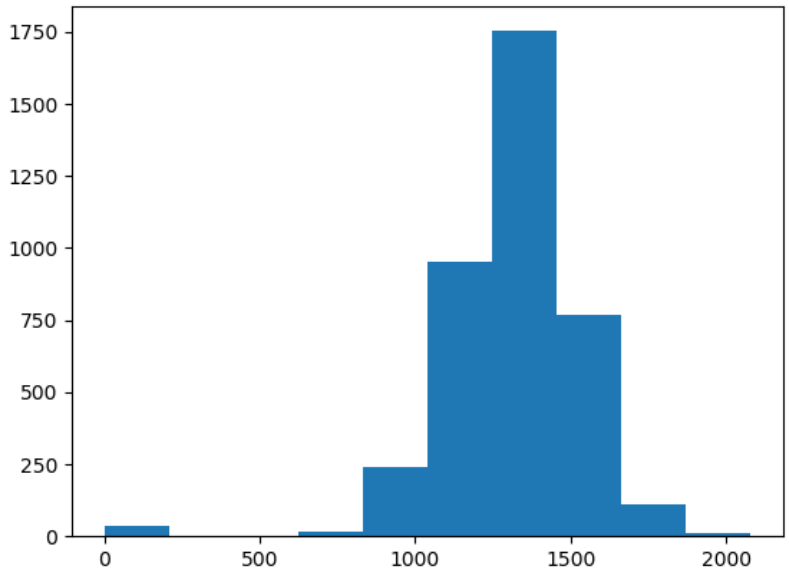

Figure 11: Harmonic Centrality for part 2

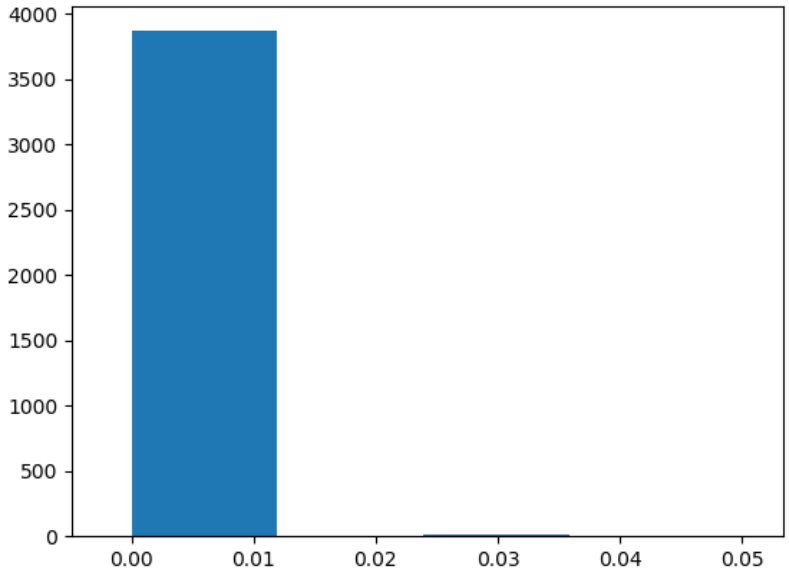

Figure 12: Betweenness Centrality

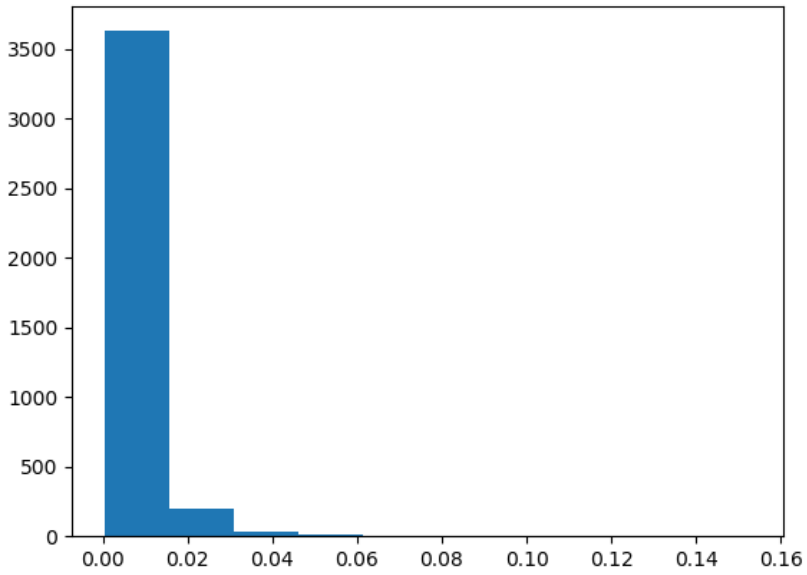

Figure 13: Degree Centrality output

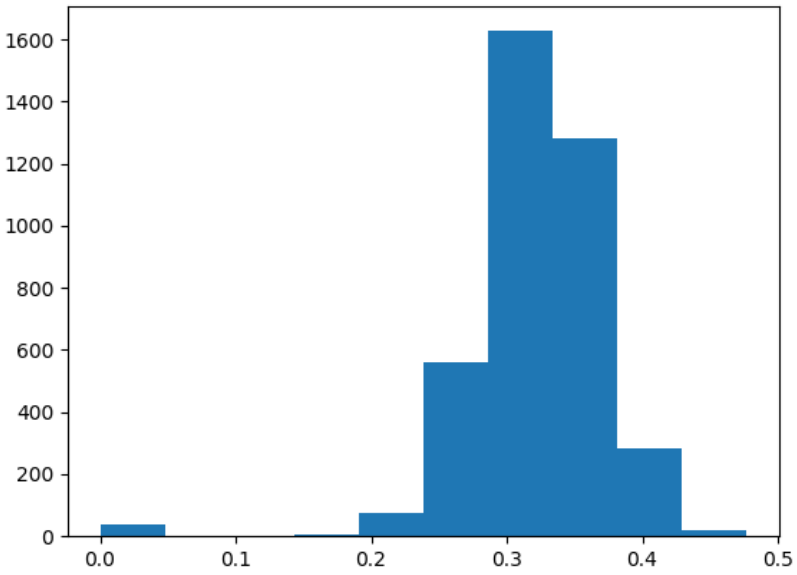

Figure 14: CLoseness Centrality output