Examinations of network dynamics through random walks

Peter Larsen and Nathaniel Ogilvie

Complex Systems Department, University of Vermont

December 2020

1 Introduction

"A drunk man may find his way home, but a drunk bird will be lost forever."(Shizuo Kakutani) Most often, when studies of random walks are conducted, one maintains a constant degree/dimension throughout the entire process. The "drunk man", a random walk in two dimensions, stays in two dimensions. The "drunk bird", a random walk in three dimensions, similarly stays in three dimensions. But what happens when dimension is non-uniform? What happens when the dimension changes for every step? How does a random walk perform when the space it exists in is finite? Answers to these questions and more are the goal of this paper. Below, we will attempt to see whether shifts in dimension/degree every time step changes the overall distance one can travel. We will also attempt to influence network coverage by choosing specific starting points based on criteria explained later. Finally, we will attempt to guide a random walk through a network from a chosen point to a destination using a paired network.

For this project, we will be using two networks: Peer to Peer (referred to as P2P) and Group to Group (referred to as G2G). These networks were created using an bipartite adjacency matrix of characters in the Marvel Comic Book Universe drawn from the online Marvel Comics database[1] and the groups and organizations they belong to, and applying matrix multiplication to them to create unipartite adjacency lists. The matrix was constructed by hand by Peter Larsen, and the information was gathered from the online Marvel Comics Database, and the concept for creating this matrix was taken from Kieran Healy's work in June of 2013[2].

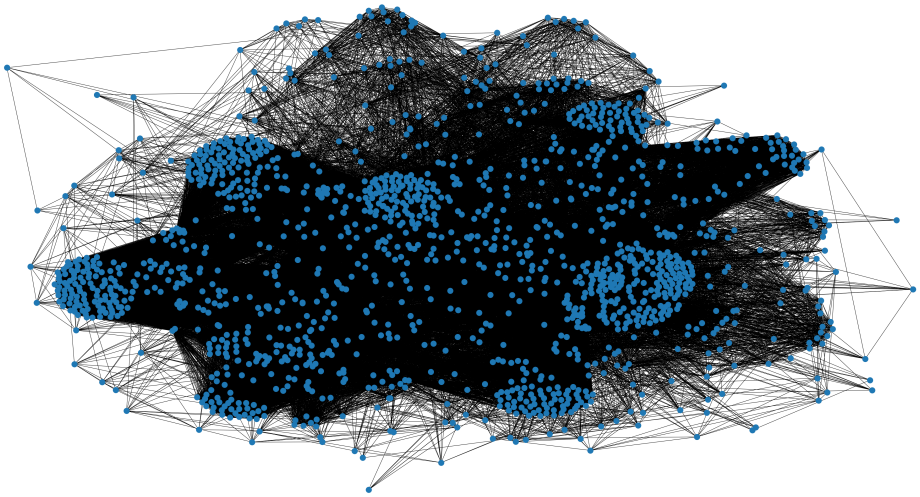

Figure 1: The Python package “networkx” representation of P2P

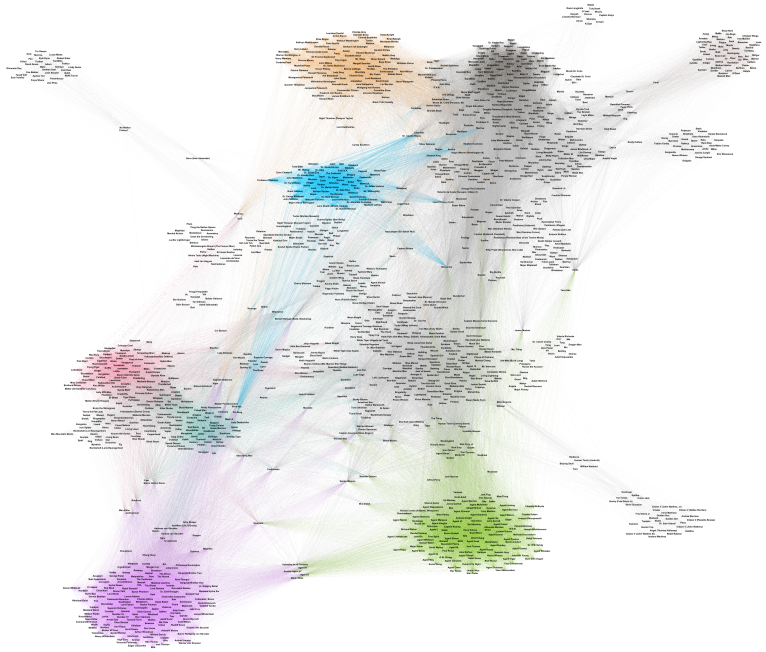

Figure 2: The Gephi representation of the P2P network

2 Comparisons Using Random Walks Between A Standard 3D Space and a Non-Uniform Dimension Network

2.1 Methods of Modeling

In order to properly examine the P2P network and any measures instituted, a control or comparison group is needed. For this project, a 3-dimensional random walk was chosen as it had elements of higher degree dynamics while still being simple to model. Although the two differ in structure, the code used on both the 3D walk[3] and the network walk is relatively the same. The code selects a random node as the current node/origin, notes its ID, and then examines its adjacency list. At random, a neighbor of the original node is chosen from the adjacency list, and it now becomes to current node, with a record of its ID. The code then selects a node at random from the new current node's adjacency list and continues on. This operation of "movement" between nodes occurs 1000 times for 1 trial. Then, the entire trial is repeated 100 times in order to smooth over any outliers or irregularities that may result from 1 individual random trial. This was the process for performing a random walk in the code for both the standard 3D walk (referred to as Uni for Uniform Degree/Dimension) and the network of characters (referred to as NonUni for NonUniform Degree/Dimension).

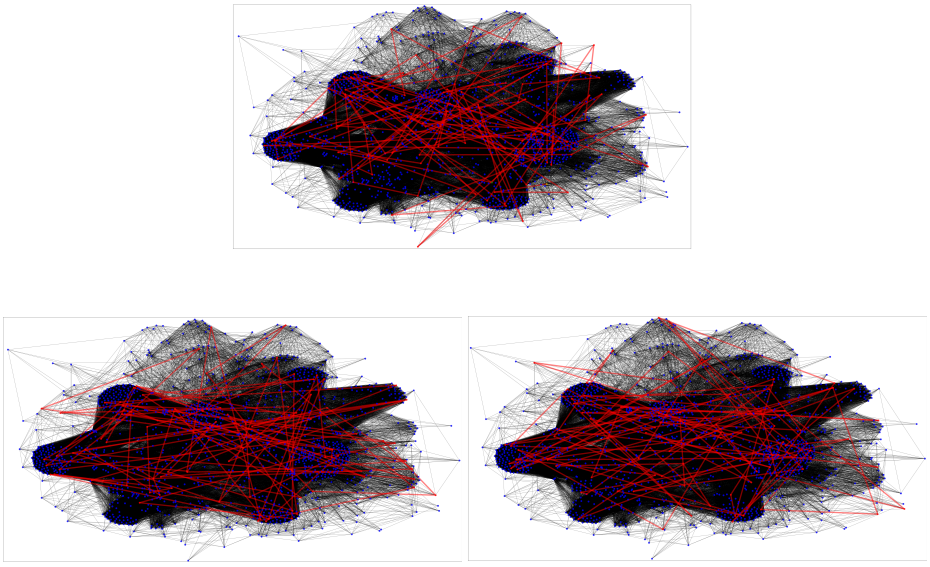

Figure 3: Three different example trials captured at 100 time steps

We performed 3 test runs to determine if the code properly moved from node to node, shown above in Figure 3. Here, the red nodes represent nodes that have been visited at some time step in the trial, and the red edges represent the path taken from some node A to a neighboring node B. This visual representation allows us to determine that the process works, and we can continue on to more time steps.

In order to examine the results from the random walks, we will be utilizing two metrics. The first is Coverage Fraction (referred to as CF), which measures the number of unique nodes visited in any trial. It is calculated as $CF = \frac{N_U}{N_T}$, where N_U is the number of unique nodes visited in a trial, and N_T is the total number of nodes visited in a trial. As the code keeps N_T constant, $N_T = 1001$, as the origin is also a potential unique node visited. Since this is true, CF may be referred to solely as N_U at some points, since CF is directly tied to N_U by a factor of $N_T = 1001$. The second metric is Rate of Return (referred to as RoR), which measures the average likelihood of a trial returning to the origin. The RoR for a simple 3D random walk was calculated by Glasser and Zucker (1977)[4] and references therein as:

$$P(3) = 1 - \frac{1}{u(3)} = 0.340537\ldots \tag{1}$$

where $u(3)$ is calculated like so:

$$u(3) = \frac{3}{(2\pi)^3} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} \int_{-\pi}^{\pi} \frac{dxdydz}{3 - \cos(x) - \cos(y) - \cos(z)} \approx 1.516386 \tag{2}$$

where $P(3)$ is the probability in $[0, 1]$ of a 3-dimensional random walk returning to the origin.

2.2 Results

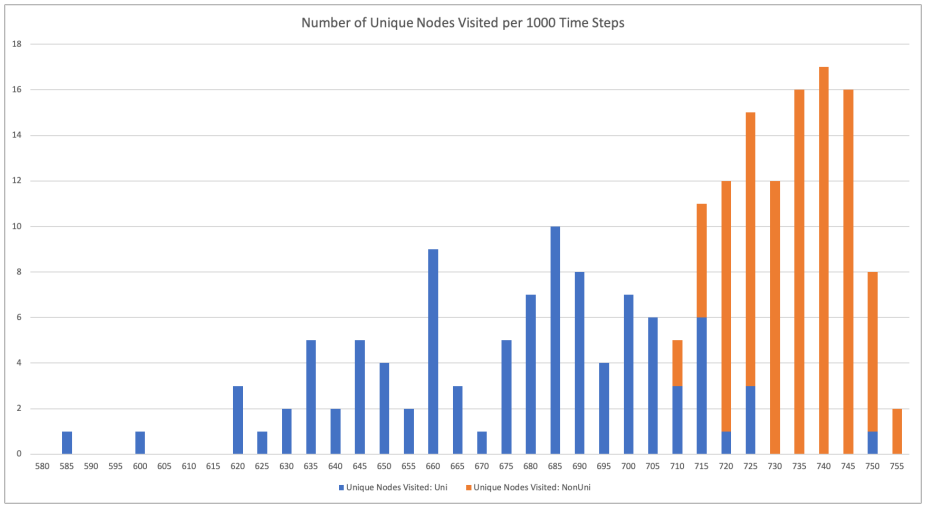

Figure 4: A graphical comparison of the 100 trials for Uni and NonUni models

As can be seen in Figure 4, where the blue bars represent the Uni trials and the orange bars represent the NonUni trials, the Uni trials possess a wider range of values, while the NonUni trials appear more clustered on the higher end of the scale. We can see further evidence of this in a mathematical analysis of the data. The CF for the Uni runs was 674.10, or 67.41%, with a standard deviation of 30.54. The CF for the NonUni runs was 732.8, or 73.13%, with a standard deviation of 10.69. This means that on average, the random walk in the P2P network was reaching approximately 59 more unique nodes per run than the 3D random walk. Using these values, a two-sample t-test assuming unequal variance can be conducted, and, with $\alpha = 0.05$, we find that our t-statistic is -17.91, and our p-value is 4.31×10^{-36}. Since this number is less than α, the difference between the two is statistically significant. Therefore, it can be assumed that other sets of trials among the two models would produce similar if not identical results.

The RoR for the Uni runs was 34%, which is very close to the mathematical percentage produced by the equation above of 34.053%. In comparison, RoR for the NonUni runs was 44%. This indicates that although the NonUni network is more likely to visit more nodes, it is also more likely to return to the origin node by almost 10%. The reason may be that it is possible to begin in a cluster in the NonUni network, meaning that the statisical likelihood of a return is reasonably high simply due to the large number of time steps and the tendency to structural trapping that occurs in flows through social-type networks like P2P, meaning that it could have travelled around the cluster and returned before ever leaving.

3 Random Walks Starting at Specific Origins Based on Centralities

3.1 Methods of Modeling

Now that the dynamics of the general NonUni network are established, more properties of networks can be examined.In network dynamics, centralities enable one to determine the most important vertices/nodes in a network based on different criteria. Instead of selecting nodes at random and moving from that origin, each trial will begin at a specified origin node, which is determined by a ranking of nodes and their centralities in the NonUni network. Three centralities, described below, will be used, and the top 5, middle 5, and bottom 5 of each ranking will have 10 trials each to smooth over irregularities, for a total of 450 trials.

Betweenness Centrality is a measure of a node's placement on the shortest paths between any two nodes in the system, and is calculated by the following:

$$C_B(v) = \sum_{s \neq v \neq t \in V} \frac{\sigma_{s,t}(v)}{\sigma_{s,t}}$$

where $\sigma_{s,t}$ represents the total number of shortest paths from node s to node t and $\sigma_{s,t}(v)$ is the number of those paths that pass through v. Closeness Centrality is a measure of the average shortest distance (path length) to every other node in the network, and is calculated by the following:

$$C(x) = \frac{1}{\sum_y d(y,x)}$$

where $d(y,x)$ is the length of the shortest path between nodes x and y. Eigenvector Centrality is a measure of a node's relationship to other highly central nodes, and is calculated by the following:

$$x_v = \frac{1}{\lambda} \sum_{t \in G} a_{v,t} x_t$$

where $a_{v,t} = 1$ if nodes x and t are neighbors and 0 otherwise, x_t represents the eigenvector centrality score of t, and λ is a constant.

3.2 Results

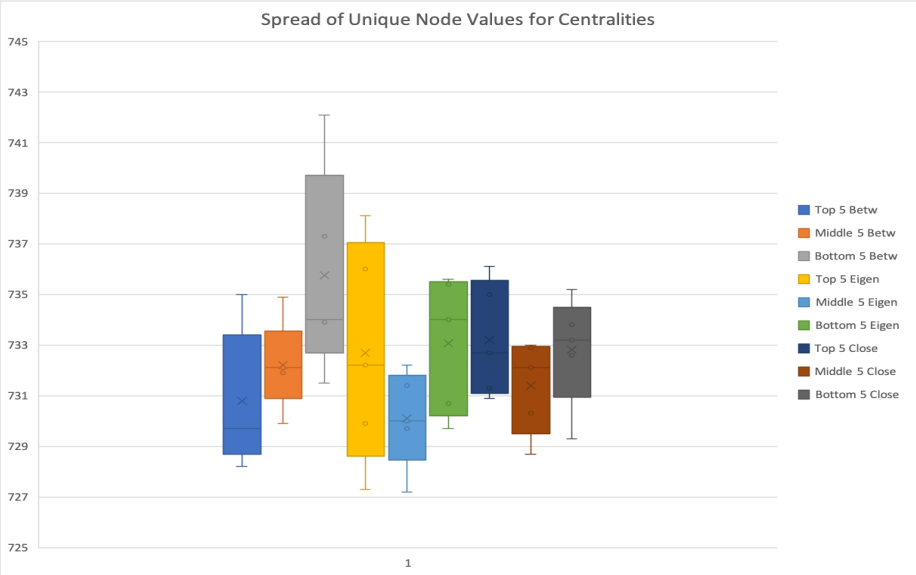

Figure 5: CF Box-and-whisker plot of all 3 centralities and the category rankings

As we can see in the plot above, it appears that there is no real correlation between any of the centralities and the CF, save for perhaps a small correlation in the Betweenness Centrality rankings (the three leftmost box-and-whiskers), as they are the only set that appears to follow a trend. For Betweenness, the average CF was 730.78 ± 2.70 for the top 5, 732.20 ± 1.78 for the middle 5, and 735.76 ± 4.10 for the bottom 5. However, as we can see from those standard deviations, the overlap was pretty significant. This would lead us to believe that there is not a statistically significant difference between the two of them, and the math would agree; for $\alpha = 0.05$, the two-sample t-test assuming unequal variances gave the following p-values: Top 5 and Middle 5 = 0.3589, Top 5 and Bottom 5 = 0.0576, and Middle 5 and Bottom 5 = 0.1351. None of these are less than α, and so there is no statistically significant difference between the CF values for the top 5, middle 5, or bottom 5 betweenness centrality rankings.

In terms of RoR, we have something more interesting:

	Top 5	Middle 5	Bottom 5
Betweenness	0.48	0.54	0.54
Eigenvector	0.42	0.44	0.44
Closeness	0.5	0.52	0.58

The table above would indicate that as the rankings decrease, the RoR increases. Again, it is reasonable to assume that, due to the nature of clustering, the lower ranked nodes would be more likely to be inside of a cluster, and thus the rate of return is higher because of the tendency to structural trapping that occurs in flows through social-type networks like P2P. So while centrality ranking does not affect the coverage of a random walk, they certainly affect the ability to return to the origin node.

4 Navigating a Network with Random Walks While Including Bias from a Paired Network

4.1 Methods of Modeling

The goal of this final part is to choose 2 groups, an origin group and a destination group, and then choose a node in the origin group such that it does not belong to the destination group and denote it as $v*$. We will then take the adjacency list of $v*$, and weight each of its neighbors based upon their interaction with the destination group. The G2G network provides us with the strength of interaction between each group. Using these weights, we will alter the adjacency lists, and then randomly walk until we hit the node selected within the target group. We will then compare this to the shortest path calculation between the nodes from the differing groups, and see how alike they are.

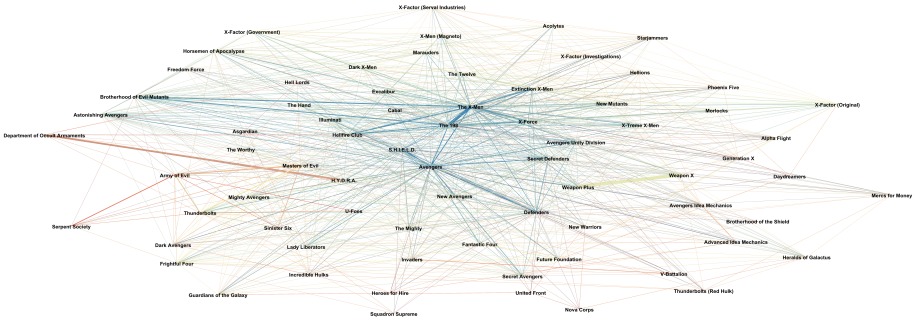

Figure 6: An image of the G2G network

For this part, we utilized the CF metric again while implementing two new ones: success rate (SR), which measured the overall rate of success of the weighted random network reaching the destination node faster (in fewer time steps), and find rate (FR), which measured the overall likelihood of an individual method (Weighted Random or Random) finding the destination node.

4.2 Results

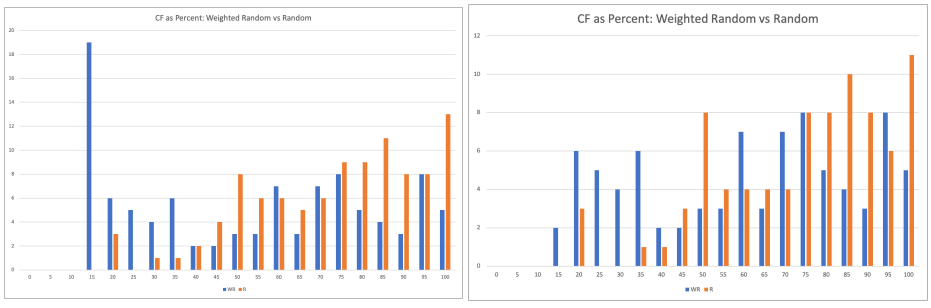

Figure 7: Graphical comparison between the Weighted Random (blue) and the True Random Walks (orange) with and without skew

In 100 paired runs, the weighted random trials had an FR of 83%, while the true random trials had an FR of 100%. However, because of this, 17 of the weighted random trials never reached the destination, and thus their weights are potentially skewing the results for the other metrics. This skew will be examined below.

The average CF for the weighted random trials was 0.5097 ± 0.2948, while the average CF for the true random trials was 0.7076 ± 0.2070. This would indicate that the true random trials covered more of the network as opposed to the weighted random, but this is skewed potentially by the 17 trials where the weighted random never reached the destination, thus creating error terms by the fact that, with only 1511 total nodes in the network, 10000 timesteps without reaching the destination is going to create very small fractions. We can examine the significance of this. In counting all 100 pairs, the difference between the two means using a two-sample t-test assuming unequal variances at $\alpha = 0.05$ gives a p-value of 1.3458×10^{-7}. This would indicate a significant difference, but because of the 17 trials that never arrived for the weighted random, we can instead examine the 83 trials that did to determine the CF. In those 83 pairs, the average CF for the weighted random trials was 0.5869 ± 0.2636, while the average for the true random was 0.7156 ± 0.2057, with p-value of 0.0005 at $\alpha = 0.05$. This means that, although it is less significant than the comparison including the skew, it still is significant enough that we can say there is a definite difference between the two means.

The SR overall was 0.35, or 35%, meaning that in 35 of the paired trials, the weighted random walk reached the destination node in fewer time steps than the true random walk. We even broke this down for the true distance (shortest path length) between the two nodes: for a shortest path length of 2, the SR was 0.4 or 40%, for 3, SR = 0.39344 = 39.34%, and for 4, SR = 0.26471 = 26.47%. This would indicate that the weighted random had more success when the path length was shorter, but this could be due to the nature of random, and the fact that in 100 trials, only 5 trials had a shortest apth length of 2.

Overall, it appears that weighted random walks are not as efficient as true random walks at finding a target node. This could be due to the nature of random, or something in our code, but as of now, it appears that bias in the network does not truly affect the flow in the case of random walks.

4.3 Further Analysis

Upon further analysis of the code and the weights in the G2G network adjacency matrix, it became clear why the weighted random did not perform better than the true random, as was expected: the adjacency matrix contains no self-referential term for groups, meaning a group isn't associated with itself. Therefore, when attempting to add the weights to influence movement in adja-

cency lists, the network would only see groups associated with the target node's group, and not the target node's group. An original list of $\{A : B, C, D, E\}$ would become $\{A : B, B, B, B, C, D, E\}$ when C was in the target node's group and D was our target node (here, the association between the group of B and the group of D was 4). This was an error we did not catch until the very end.

A solution that could implemented later would be to alter the adjacency group like so: a tiered "if" system. Tier 4 would be nodes in groups with no association with the target node's group, and they would stay constant (in the above example, E would be a Tier 4). Tier 3 would be nodes in groups with an association with the target node's group, and they would have W counts of N, where N is that node, and W is the weight from the G2G adjacency matrix. Tier 2 would be nodes who are members of the target node's group (C in the above example) and they would be weighted like so: $W_{TNG} = 2 * max(W)$, where W is all the weights with the target group in the adjacency matrix. This would make the likelihood of moving to a Tier 2 node twice as likely as any Tier 3 node. Finally, Tier 1 would be the target node itself, with a weight of $W_{TN} = 3 * max(W)$. This would mean that the likelihood of moving to the target node is 3 times as likely as a Tier 3 node. Thus, for a $max(W) = 5$, the adjacency list of A would go from $\{A : B, C, D, E\}$ to $\{A : B, B, B, B, C, C, C, C, C, C, C, C, C, C, D, D, D, D, D, D, D, D, D, D, D, D, D, D, D, E\}$. This would mean that, although random is still in play, the weights are having an effect on the walk.

5 Code

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation
import random
import csv

class travelor:
    def __init__(self, start_n):
        self.nodes_traveled = [start_n]
        self.current_node = start_n
        self.start_node = start_n
        self.return_start = False
        self.iter_return = 0
        self.number_nodes = 0
        self.unique_nodes = 0
        self.CF = 0

    def check_start(self, new_node):
        if (self.return_start == False):
            if (self.start_node == new_node):
```

```
                self.iter_return = len(self.nodes_traveled) + 1
                return True
            else:
                return False
        else:
            return True

    def move_2_node(self, new_node):
        self.nodes_traveled.append(new_node)
        self.current_node = new_node
        self.return_start = self.check_start(new_node)

    def get_path(self):
        return self.nodes_traveled

    def get_edge_list(self):
        edge_list = []
        for iter in range(len(self.nodes_traveled) - 1):
            edge_list.append((self.nodes_traveled[iter], self.
    nodes_traveled[iter+1]))
        return edge_list

    def get_return_status(self):
        return self.return_start

    def get_return_move(self):
        return self.iter_return

    def get_current_node(self):
        return self.current_node

    def calculate_stats(self):
        self.number_nodes = len(self.nodes_traveled)
        self.unique_nodes = len(set(self.nodes_traveled))
        self.CF = float(self.unique_nodes) / self.number_nodes
        return [self.CF, self.unique_nodes, self.number_nodes, self
    .return_start, self.iter_return]

def run_simulation(iterations, moves, G):
    #create list of travelors
    travelors = []
    for iter in range(iterations):
        #Create new simulation
        my_travelor = travelor(random.choice(list(G.nodes())))
        for i in range(moves):
            new_node = random.choice(list(nx.
    node_connected_component(G, my_travelor.get_current_node())))
            my_travelor.move_2_node(new_node)
            #print(new_node)
        travelors.append(my_travelor)
    return travelors

def display_graph(display, Graph):
    if (display):
        pos = nx.spring_layout(Graph,k=0.25,iterations=25,seed=17)
        nx.draw(Graph, pos)
        plt.show()
```

```
77
78  def display_graph_paths(display, Graph, my_travelor):
79      if (display):
80          pos = nx.spring_layout(Graph,k=0.25,iterations=25,seed=17)
81          no_traveled_nodes = [node for node in list(G.nodes()) if
    node not in my_travelor.get_path()]
82          options = {"node_size": 100, "alpha": 0.8}
83          nx.draw_networkx_nodes(Graph, pos, nodelist=
    no_traveled_nodes, node_color="b", **options)
84          nx.draw_networkx_nodes(Graph, pos, nodelist=my_travelor.
    get_path(), node_color="r", **options)
85          nx.draw_networkx_edges(G, pos, width=1.0, alpha=0.5)
86          nx.draw_networkx_edges(G, pos, edgelist=my_travelor.
    get_edge_list(), width=8, alpha=0.5, edge_color="r",)
87          plt.show()
88
89  if __name__ == "__main__":
90      iterations = 10
91      moves = 100
92      #reading in csv file
93      mydata = np.genfromtxt('MARVEL P2P.csv', delimiter=',')
94      p2p_matrix = mydata[1:,1:]
95      G = nx.from_numpy_matrix(p2p_matrix)
96      travelors = run_simulation(iterations, moves, G)
97      #move above to sperate function
98      #color eges that have been traversed / get edges that have been
         travered probably through list of nodes just tuple at every
        step
99      #color nodes that have been visited
100     #functionafy to make it happen for many iterations
101     display_graph(False, G)
102     display_graph_paths(False, G, travelors[8])
103     with open('p2p_marvelCF_part1.csv', mode='w', newline='') as
    csv_file:
104         csv_writer = csv.writer(csv_file, delimiter=',', quotechar=
    '"', quoting=csv.QUOTE_MINIMAL)
105         csv_writer.writerow(['CF','Unique Nodes','Total Nodes','
    Returned','Return Iteration'])
106         for travelor in travelors:
107             csv_writer.writerow(travelor.calculate_stats())
108             if (travelor.get_return_status()):
109                 print ("travelor returned on move: " + str(travelor
    .get_return_move()))
```

Listing 1: Part One Code Marvel Random Walk

```
1  import numpy as np
2  import matplotlib.pyplot as plt
3  from mpl_toolkits.mplot3d import Axes3D
4  import csv
5
6  def output_lines(x, y, z):
7      start_node = [x[0], y[0], z[0]]
8      node_path = []
9      returned = False
10     return_iter = -1
11     for node in range(1,len(x)):
12         node_path.append([x[node], y[node], z[node]])
```

```
        if (start_node == [x[node], y[node], z[node]]):
            print("Returned at iter: " + str(node))
            return_iter = node
            returned = True
    total_nodes = len(x)
    unique_nodes = len(set(map(tuple,node_path)))
    CF = float(unique_nodes) / total_nodes
    #unique_nodes = len(map(list,unique_nodes_tuple))
    return [CF, unique_nodes, total_nodes, returned, return_iter]

N = 1000
iterations = 100
csv_lines = []
with open('3d_randomWalkCF_part1.csv', mode='w', newline='') as
    csv_file:
    csv_writer = csv.writer(csv_file, delimiter=',', quotechar='"',
     quoting=csv.QUOTE_MINIMAL)
    csv_writer.writerow(['CF','Unique Nodes','Total Nodes','
    Returned','Return Iteration'])
    for iter in range(iterations):
        R = (np.random.rand(N)*6).astype("int")
        x = np.zeros(N)
        y = np.zeros(N)
        z = np.zeros(N)
        x[ R==0 ] = -1; x[ R==1 ] = 1
        y[ R==2 ] = -1; y[ R==3 ] = 1
        z[ R==4 ] = -1; z[ R==5 ] = 1
        x = np.cumsum(x)
        y = np.cumsum(y)
        z = np.cumsum(z)
        csv_writer.writerow(output_lines(x,y,z))

plt.figure()
ax = plt.subplot(1,1,1, projection='3d')
ax.plot(x, y, z,alpha=0.6)
ax.scatter(x[-1],y[-1],z[-1])
plt.show()
```

Listing 2: Part One Code 3D random Walk

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation
import random
import csv
import sys

class travelor:
    def __init__(self, start_n):
        self.nodes_traveled = [start_n]
        self.current_node = start_n
        self.start_node = start_n
        self.return_start = False
        self.iter_return = 0
        self.number_nodes = 0
        self.unique_nodes = 0
```

```
        self.CF = 0

    def check_start(self, new_node):
        if (self.return_start == False):
            if (self.start_node == new_node):
                self.iter_return = len(self.nodes_traveled) + 1
                return True
            else:
                return False
        else:
            return True

    def move_2_node(self, new_node):
        self.nodes_traveled.append(new_node)
        self.current_node = new_node
        self.return_start = self.check_start(new_node)

    def get_path(self):
        return self.nodes_traveled

    def get_edge_list(self):
        edge_list = []
        for iter in range(len(self.nodes_traveled) - 1):
            edge_list.append((self.nodes_traveled[iter], self.
    nodes_traveled[iter+1]))
        return edge_list

    def get_return_status(self):
        return self.return_start

    def get_return_move(self):
        return self.iter_return

    def get_current_node(self):
        return self.current_node

    def get_start_node(self):
        return [self.start_node]

    def get_stop_node(self):
        return [self.current_node]

    def calculate_stats(self):
        self.number_nodes = len(self.nodes_traveled)
        self.unique_nodes = len(set(self.nodes_traveled))
        self.CF = float(self.unique_nodes) / self.number_nodes
        return [self.CF, self.unique_nodes, self.number_nodes,
    float(self.return_start), self.iter_return]

def run_simulation(moves, G, start_nodes):
    #create list of travelors
    travelors = []
    my_travelor = None
    #random.seed(datetime.now())
    for iter in range(len(start_nodes)):
        #Create new simulation
        my_travelor = travelor(start_nodes[iter])
```

```
74          for i in range(moves):
75              new_node = random.choice(list(nx.
    node_connected_component(G, my_travelor.get_current_node())))
76              my_travelor.move_2_node(new_node)
77          travelors.append(my_travelor)
78      return travelors
79
80  def display_graph(display, Graph):
81      if (display):
82          pos = nx.spring_layout(Graph,k=0.25,iterations=25,seed=17)
83          nx.draw(Graph, pos)
84          plt.show()
85
86  def display_graph_paths(display, Graph, my_travelor):
87      if (display):
88          pos = nx.spring_layout(Graph,k=0.25,iterations=25,seed=17)
89          no_traveled_nodes = [node for node in list(G.nodes()) if
    node not in my_travelor.get_path()]
90          options = {"node_size": 100, "alpha": 0.8}
91          nx.draw_networkx_nodes(Graph, pos, nodelist=
    no_traveled_nodes, node_color="b", **options)
92          nx.draw_networkx_nodes(Graph, pos, nodelist=my_travelor.
    get_path(), node_color="r", **options)
93          nx.draw_networkx_nodes(Graph, pos, nodelist=my_travelor.
    get_start_node(), node_color="g", **options)
94          nx.draw_networkx_nodes(Graph, pos, nodelist=my_travelor.
    get_stop_node(), node_color="y", **options)
95          nx.draw_networkx_edges(G, pos, width=1.0, alpha=0.5)
96          nx.draw_networkx_edges(G, pos, edgelist=my_travelor.
    get_edge_list(), width=8, alpha=0.5, edge_color="r",)
97          plt.show()
98
99  if __name__ == "__main__":
100     start_nodes = [133, 988, 1283, 236, 830, 587, 634, 662, 663,
    667, 1504, 1510, 1055, 1143, 1309, 1481, 344, 1283, 1135, 240,
    791,
101     799, 825, 842, 851, 1037, 1226, 1055, 1143, 1309, 1481, 236,
    1283, 344, 1135, 587, 634, 662, 663, 667, 1504, 1510, 1055,
    1143, 1309]
102     moves = 1000
103     iters = 100
104     #reading in csv file
105     mydata = np.genfromtxt('MARVEL P2P.csv', delimiter=',')
106     p2p_matrix = mydata[1:,1:]
107     G = nx.from_numpy_matrix(p2p_matrix)
108     #for some reason doesn't work
109     travelors_total = []
110     for i in range(len(start_nodes)):
111         travelors_total.append([0,0,0,0,0])
112
113     for iter in range(iters):
114         output_str = "iteration: " + str(iter) + " started"
115         sys.stdout.write(output_str)
116         sys.stdout.flush()
117         travelors = []
118         travelors = run_simulation(moves, G, start_nodes)
119         current_travelor = 0
```

```
        for travelor_1 in travelors:
            addition = travelor_1.calculate_stats()
            for item in range(len(addition)):
                travelors_total[current_travelor][item] =
    travelors_total[current_travelor][item] + addition[item]
            current_travelor = current_travelor + 1

    with open('p2p_marvelCF_part2_1000_teststep_100iter.csv', mode=
    'w', newline='') as csv_file:
        csv_writer = csv.writer(csv_file, delimiter=',', quotechar=
    '"', quoting=csv.QUOTE_MINIMAL)
        csv_writer.writerow(['CF','Unique Nodes','Total Nodes','
    Returned','Return Iteration'])
        print(travelors_total)
        for i in range(len(travelors_total)):
            travelors_total[i][:] = [float(x) / iters for x in
    travelors_total[i]]
        for travelor in travelors_total:
            csv_writer.writerow(travelor)
```

Listing 3: Part Two Code

```
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt
import mpl_toolkits.mplot3d.axes3d as p3
import matplotlib.animation as animation
import random
import csv

class travelor:
    def __init__(self, start_n, stop_n, max_m):
        self.nodes_traveled_rnd = [start_n]
        self.nodes_traveled_wrd = [start_n]
        self.current_node = start_n
        self.start_node = start_n
        self.return_start = False
        self.iter_return = 0
        self.number_nodes_rnd = 0
        self.unique_nodes_rnd = 0
        self.CF_rnd = 0
        self.number_nodes_wrd = 0
        self.unique_nodes_wrd = 0
        self.CF_wrd = 0
        self.groups = []
        self.max_moves = max_m
        self.destination = stop_n
        self.find_dest_rnd = False
        self.find_dest_wrd = False
        self.iter_dest_rnd = -1
        self.iter_dest_wrd = -1
        self.shortest_path = []

    def check_start_rnd(self, new_node):
        if (self.find_dest_rnd == False):
            if (self.destination == new_node):
                self.iter_dest_rnd = len(self.nodes_traveled_rnd) +
     1
```

```
                return True
            else:
                return False
        else:
            return True

    def check_start_wrd(self, new_node):
        if (self.find_dest_wrd == False):
            if (self.destination == new_node):
                self.iter_dest_wrd = len(self.nodes_traveled_wrd) +
 1
                return True
            else:
                return False
        else:
            return True

    def move_2_node_wrd(self, new_node):
        self.nodes_traveled_wrd.append(new_node)
        self.current_node = new_node
        self.find_dest_wrd = self.check_start_wrd(new_node)

    def move_2_node_rnd(self, new_node):
        self.nodes_traveled_rnd.append(new_node)
        self.current_node = new_node
        self.find_dest_rnd = self.check_start_rnd(new_node)

    def get_path(self):
        return self.nodes_traveled

    def get_edge_list(self):
        edge_list = []
        for iter in range(len(self.nodes_traveled) - 1):
            edge_list.append((self.nodes_traveled[iter], self.
    nodes_traveled[iter+1]))
        return edge_list

    def get_return_status(self):
        return self.return_start

    def get_return_move(self):
        return self.iter_return

    def get_current_node(self):
        return self.current_node

    def calculate_stats(self):
        self.number_nodes_wrd = len(self.nodes_traveled_wrd)
        self.unique_nodes_wrd = len(set(self.nodes_traveled_wrd))
        self.CF_wrd = float(self.unique_nodes_wrd) / self.
    number_nodes_wrd
        self.number_nodes_rnd = len(self.nodes_traveled_rnd)
        self.unique_nodes_rnd = len(set(self.nodes_traveled_rnd))
        self.CF_rnd = float(self.unique_nodes_rnd) / self.
    number_nodes_rnd
        shortest_path_len = len(self.shortest_path)
        return [self.CF_wrd, self.unique_nodes_wrd, self.
```

```
    number_nodes_wrd, self.find_dest_wrd, self.iter_dest_wrd,
    shortest_path_len, self.CF_rnd, self.unique_nodes_rnd, self.
    number_nodes_rnd, self.find_dest_rnd, self.iter_dest_rnd]

    def update_groups(self, group_in):
        self.groups = group_in

    def done_traveling(self, moves):
        if (self.destination == self.current_node):
            return True
        if (moves > self.max_moves):
            return True
        return False

    def shortest_path_1(self, path):
        self.shortest_path = path

    def update_rand_moves(self,rnd_moves):
        self.num_random_moves = rnd_moves

    def reset(self):
        self.current_node = self.start_node

def find_stop_n(G, groups_dict, start_node):
    #find start node group
    remove_groups = groups_dict[start_node]
    stop_nodes = []
    print(remove_groups)
    #eliminate all nodes with those groups
    for key, value in groups_dict.items():
        if (not any(item in value for item in remove_groups)):
            stop_nodes.append(key)
    #randomly select from nodes left
    return random.choice(stop_nodes)

def get_weighted_moves(start_node, possible_nodes, g2g, groups_dict
    ):
    possible_nodes_out = []
    start_groups = groups_dict[start_node]
    for node in possible_nodes:
        multiply_node_by = 0
        end_groups = groups_dict[node]
        for group_start in start_groups:
            for group_end in end_groups:
                multiply_node_by = multiply_node_by + g2g[
    group_start][group_end]
        for i in range(int(multiply_node_by)):
            possible_nodes_out.append(node)
    #get possible end_nodes
    #find groups of end_nodes
    #look up in matrix [start_node_group][end_node_group]
    #add multiple times if different groups
    #add to possible nodes repeated a bunch then select a random
    one from the list
    return possible_nodes_out

def run_simulation(iterations, max_moves, G, g2g_matrix,
```

```
    groups_dict):
    #create list of travelors
    travelors = []
    for iter in range(iterations):
        start_node = random.choice(list(G.nodes()))
        stop_node = find_stop_n(G, groups_dict, start_node)
        #Create new simulation
        my_travelor = travelor(start_node, stop_node, max_moves)
        my_travelor.update_groups(groups_dict[start_node])
        #test with random moves how long it takes
        moves = 0
        while (not my_travelor.done_traveling(moves)):
            new_node = random.choice(list(nx.
node_connected_component(G, my_travelor.get_current_node())))
            my_travelor.move_2_node_rnd(new_node)
            moves = moves + 1
        my_travelor.update_rand_moves(moves)
        #test with weird walk how long it takes
        moves = 0
        my_travelor.reset()
        while (not my_travelor.done_traveling(moves)):
            unweighted_moves = list(nx.node_connected_component(G,
my_travelor.get_current_node()))
            possible_moves = get_weighted_moves(my_travelor.
get_current_node(), unweighted_moves, g2g_matrix, groups_dict)
            new_node = random.choice(possible_moves)
            my_travelor.move_2_node_wrd(new_node)
            moves = moves + 1
        #find shortest path for comparison add to travelors
        path = nx.shortest_path(G, source=start_node, target=
stop_node)
        my_travelor.shortest_path_1(path)
        travelors.append(my_travelor)
    return travelors

def display_graph(display, Graph):
    if (display):
        pos = nx.spring_layout(Graph,k=0.25,iterations=25,seed=17)
        nx.draw(Graph, pos)
        plt.show()

def display_graph_paths(display, Graph, my_travelor):
    if (display):
        pos = nx.spring_layout(Graph,k=0.25,iterations=25,seed=17)
        no_traveled_nodes = [node for node in list(G.nodes()) if
node not in my_travelor.get_path()]
        options = {"node_size": 100, "alpha": 0.8}
        nx.draw_networkx_nodes(Graph, pos, nodelist=
no_traveled_nodes, node_color="b", **options)
        nx.draw_networkx_nodes(Graph, pos, nodelist=my_travelor.
get_path(), node_color="r", **options)
        nx.draw_networkx_edges(G, pos, width=1.0, alpha=0.5)
        nx.draw_networkx_edges(G, pos, edgelist=my_travelor.
get_edge_list(), width=8, alpha=0.5, edge_color="r",)
        plt.show()

if __name__ == "__main__":
```

```
    iterations = 100
    moves = 10000
    #reading in csv file
    mydata = np.genfromtxt('MARVEL P2P.csv', delimiter=',')
    p2p_matrix = mydata[1:,1:]
    #groups
    marvalG2G_in = np.genfromtxt('MARVEL G2G.csv', delimiter=',')
    g2g_matrix = marvalG2G_in[1:,1:]

    proj_in = np.genfromtxt('MARVEL Project.csv', delimiter=',')
    proj_matrix = proj_in[1:,1:]

    travelor_groups = {}
    for idx, x in np.ndenumerate(proj_matrix):
        if (x == 1):
            if (idx[0] in travelor_groups.keys()):
                list_g = travelor_groups[idx[0]]
                list_g.append(idx[1])
            else:
                list_g = [idx[1]]
            travelor_groups[idx[0]] = list_g

    #update travelors
    G = nx.from_numpy_matrix(p2p_matrix)
    travelors = run_simulation(iterations, moves, G, g2g_matrix,
    travelor_groups)
    #move above to sperate function
    #color eges that have been traversed / get edges that have been
     travered probably through list of nodes just tuple at every
    step
    #color nodes that have been visited
    #functionafy to make it happen for many iterations
    display_graph(False, G)
    display_graph_paths(False, G, travelors[3])
    with open('p2p_marvelCF_part3_10000step.csv', mode='w', newline
    ='') as csv_file:
        csv_writer = csv.writer(csv_file, delimiter=',', quotechar=
    '"', quoting=csv.QUOTE_MINIMAL)
        csv_writer.writerow(['Wrd CF','Wrd Unique Nodes','Wrd Total
     Nodes','Wrd Fnd Dest','Wrd Dest Iteration','Shortest Path
    Length','Rnd CF','Rnd Unique Nodes','Rnd Total Nodes','Rnd Fnd
    Dest','Rnd Dest Iteration'])
        for travelor in travelors:
            csv_writer.writerow(travelor.calculate_stats())
            if (travelor.get_return_status()):
                print ("travelor returned on move: " + str(travelor
    .get_return_move()))
```

Listing 4: Part Three Code

6 Works Cited

[1] *Marvel Comics Database.* https://marvel.fandom.com/wiki/Marvel_Database

[2] K. Healy, *Using Metadata to Find Paul Revere* (2013). https://kieranhealy.org/blog/archives/2013/06/09/using-metadata-to-find-paul-revere/

[3] Hogeo, *Simple way to draw 3D random walk using Python and Matplotlib.pyplot* (2017).https://pythonmatplotlibtips.blogspot.com/2017/11/simple-way-to-draw-3d-random-walk-matplotlib.html

[4] Glasser, M. L. and Zucker, I. J. *Extended Watson Integrals for the Cubic Lattices.* Proc. Nat. Acad. Sci. (U.S.A.) 74, 1800-1801 (1977).