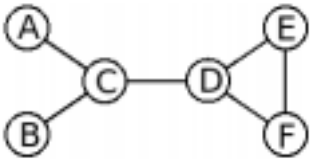
Network Science - HWK1

Diogo Bogas
up201202482

March 2019

1 Exercise 1

Regarding this exercise we consider the following network:



1.1 item a

Calculating the degree of any node in a graph, given that the edges are non directed, is reduced to counting, per node, the number of nodes it is connected to. Having said this, node A is connected to node C only, as is node B. Therefore, their degree is exactly the same, 1. Node C is connected to nodes A,B and D, which gives it a degree of 3. Node D is connected to C,E and F, matching C in degree. Nodes E and F have degree 2, because E connects to D and F, and F connects do D and E.

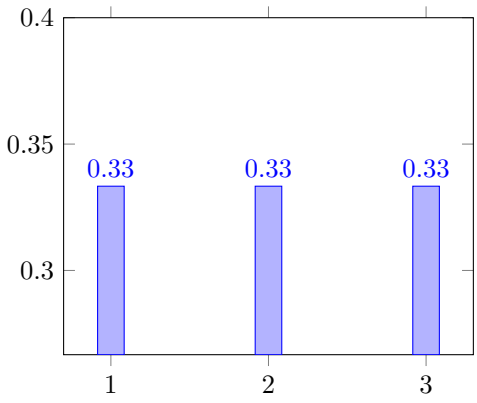
The following table shows the degree per node:

Node	Degree
A	1
B	1
C	3
D	3
E	2
F	2

Table 1: Degree per node

To plot the normalized histogram, we need to know that P(k) is the probability representing a randomly chosen node having degree k. Given that we have N nodes in the graph, and N_k represents the number of nodes that have degree k, we have $P(k) = \frac{N_k}{N}$

The normalized degree distribution for this graph is as follows:



1.2 item b

As this graph is small, we can calculate the diameter (longest shortest path) with ease, without needing to use any algorithm in particular. Distance from connected nodes is 1, so we obtain the following table (where we only use the inferior triangle, as the superior one has the exact same values, and we also disregard a node's distance to itself):

	A	B	C	D	E	F
A	-	-	-	-	-	-
B		-	-	-	-	-
C	1	1	-	-	-	-
D			1	-	-	-
E				1	-	-
F				1	1	-

Table 2: Known initial distances

The minimum distance between A and B is 2, if we follow A →C →B. Any other path has a greater distance.
Both minimum paths from A and B towards D go through C, so those distances are also 2.
Both minimum paths from A and B towards E or F have distance 2 + (distance from D to E or F), which happens to be 1, so there distances have value 3.
Both minimum path from C to either E or F go through D. We get C →D →E and C →D →F respectively. Both paths have distance value 2.

	A	B	C	D	E	F
A	-	-	-	-	-	-
B	2	-	-	-	-	-
C	1	1	-	-	-	-
D	2	2	1	-	-	-
E	3	3	2	1	-	-
F	3	3	2	1	1	-

Table 3: Final minimal distances

Now we only need to find the maximum value in the latest table, which is 3.

The diameter of the network is 3. In cases where the size of the graph doesn't allow for manual calculation, an algorithm should be used. Floyd–Warshall [4] calculates the shortest paths for every pair node possible. It does so for weighted edges, so we would only need to give every edge in this graph the weight of 1.

1.3 item c

The clustering coefficient tells us the portion of a node's neighbours that are connected, through the following formula [2]:

$$c_i = \frac{2e_i}{k_i(k_i-1)}$$

where i is the node, c_i represents the clustering coefficient for node i, e_i represents the number of edges between the neighbours of i, and k_i is node i's degree. The following table represents the above formula applied to all nodes in the network.

Coefficient	Value
$A(c_0)$	0
$B(c_1)$	0
$C(c_2)$	0
$D(c_3)$	1/3
$E(c_4)$	1
$F(c_5)$	1

Table 4: Clustering coefficient for this graph's nodes

The average clustering coefficient is then $\frac{1}{N}\sum_{n=1}^{6} c_i$ which amounts to 0.38888.

1.4 item d-i

There is only 1 triangle, composed of nodes D, E and F. Now, for every node, we check for the possibility of if being the center node of a triad, be it open or closed:

- C is in (A,C,B), (A,C,D) and (B,C,D)
- D is in (C,D,E) (C,D,F) and (E,D,F)
- E is in (D,E,F) and (F,E,D)
- and finally F is in (D,F,E) and (E,F,D)

Which gives us a grand total of 10 triads. Therefore, the global clustering coefficient is 0.3.

1.5 item d-ii

In this case, the higher degree nodes were C and D, with degree 3. The average clustering coefficient, as calculated in item c, is 0.388888 and the global clustering coefficient is 0.3. Therefore, the average clustering coefficient is the metric that gives more weigth to the nodes.

1.6 item e

For the normalized closeness centrality [6] we use the following formula:

$$C_c(i) = \frac{[\sum_{n=1}^{N} d(i,j)]^{-1}}{N-1}$$

And we can extract the following table:

Node	Normalized Closeness Centrality
A	1/55
B	1/55
C	1/35
D	1/35
E	1/45
F	1/50

Table 5: Normalized Closeness Centrality

2 Programming exercises

In the programming exercises, I used the NetworkX 2.2 for Python, only for the data structure it offers for graphs. The required solutions themselves are implemented "by hand". As for Python, i used 2.7. Other packages are random, matplotlib and __future__ (for the division).

As for code,in a general manner, exercise k is solved in exk.py. There is some code-importing between exercises, as suggested.

The code for all exercises will appear in the Annex, as well in the final delivery.

2.1 Exercise 2

The Erdos-Renyi [3] model generates random graphs, when given an n and a p, where the n represents the number of nodes this random network has, and p represents the probability that each pair of nodes has to be connected. Knowing this, to generate 2 graphs, both with n = 1000, and p = 0.0001 and p = 0.005 respectively, the idea is to, for each possible pair of nodes, generate a random number between 0 and 1. If said number is lower than our probability p, we add an edge in the pair of nodes we are currently. As the graph is undirected, and thinking of the graph as a matrix, I only went through the triangular superior matrix. The generated files for this exercise are random1.txt and random2.txt

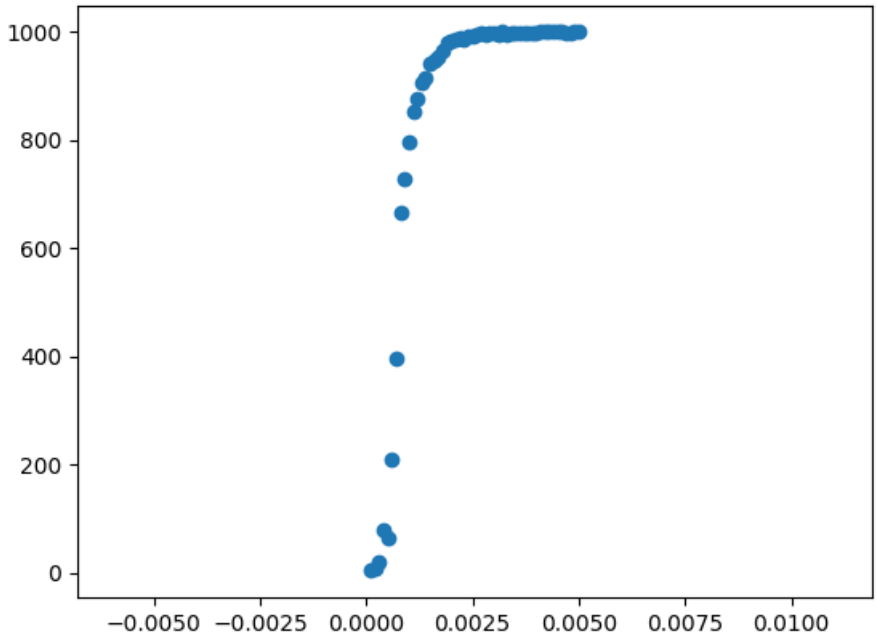
2.2 Exercise 3

To calculate the size of the giant component [5] i started by calculating all connected components there exist in the network, using Breadth-first search. Then, as the giant component is the biggest in size from all the connected components, I return its size.

For the graph where p = 0.0001, the giant component has size 5, and when p = 0.005, the size of the giant component is 1000.

2.3 Exercise 4

Calculating the giant component size, from p = 0.0001 to p = 0.005, with 0.0001 steps, and scattering the obtained points, generates the following plot, where the X axis represents p, and the Y axis represents the size of the giant component:

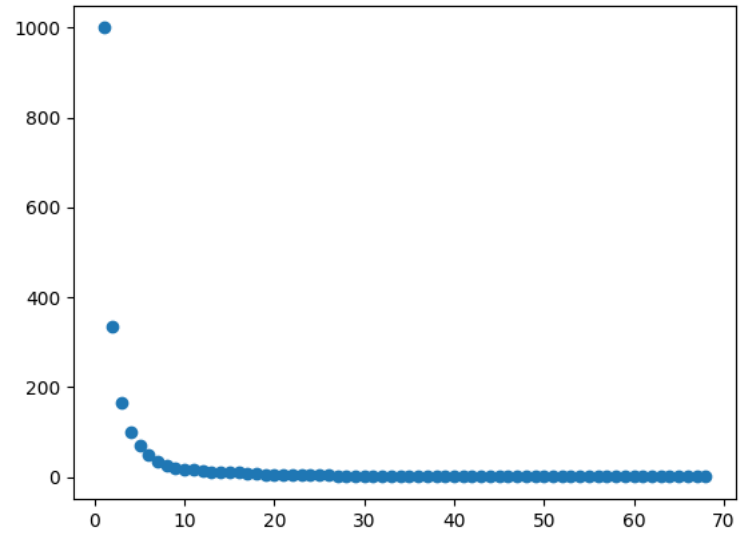


2.4 Exercise 5

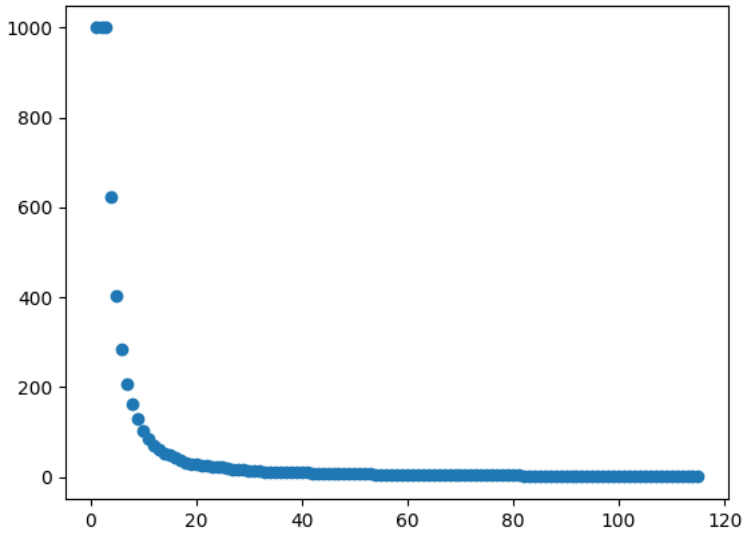
The .txt files for this exercise are ba1.txt and ba2.txt. I generate 2 graphs according to this model [1]. Both have n=1000, bu one of them has m0=3 and m=1, and the other has m0=6 and m=3.

2.5 Exercise 6

For the graph where n = 1000, m0 = 3 and m = 1 I obtained the following cumulative binning distribution:



And for the graph where n = 1000, m0 = 6 and m = 3 I got the following:



3 Gephi Exercises

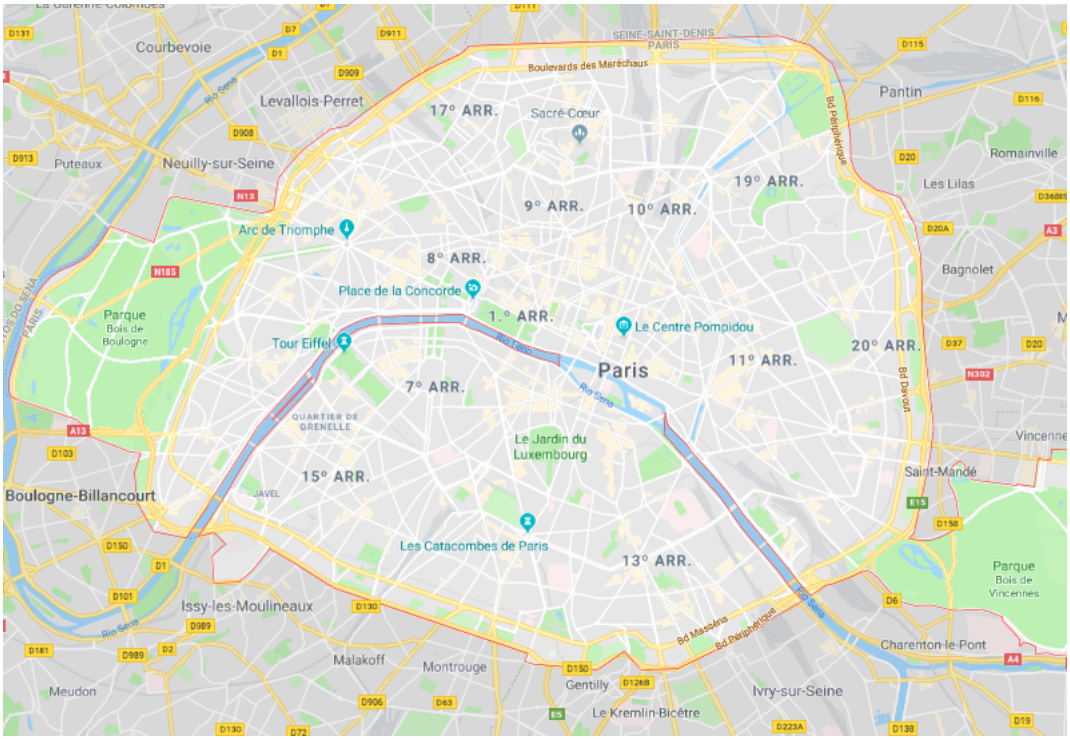
3.1 Exercise 7

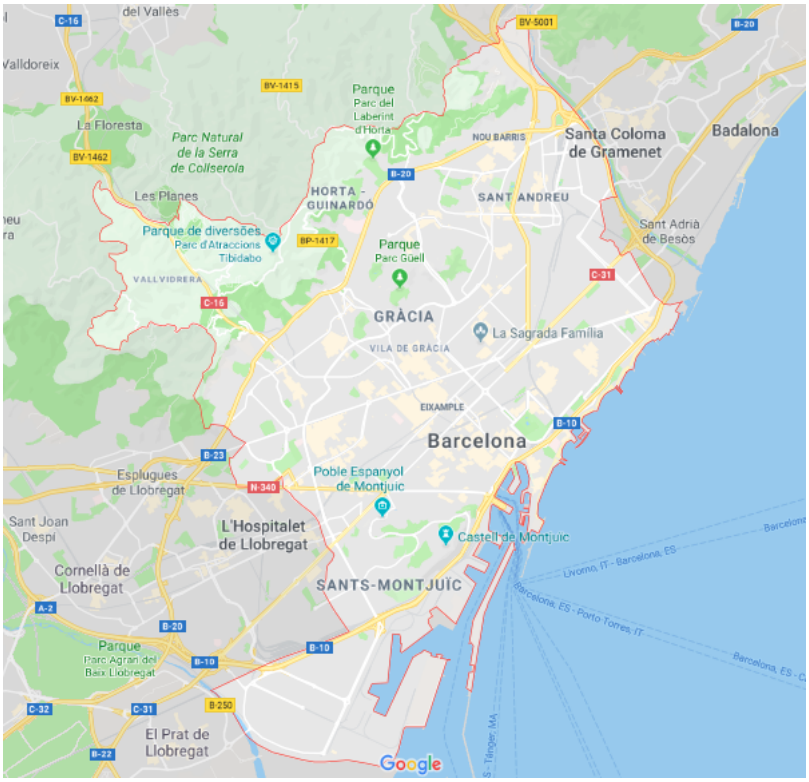
In this exercise we analyze the road network of three cities: Barcelona, Venice and Paris, not knowing which is which. To start off, city1 has 335 nodes, city2 has 210, and finally city3 has 1840 nodes.

Unlike the other two, Venice will have the smallest network in terms of number of nodes. This is because Venice has cannals throughout the city, which compose another network of transporting paths. This being said, Venice is city2.

Now, to distinguish between Barcelona and Paris, we can compare the area that each one occupies and attribute the graph with most nodes to the city that occupies the largest land area. According to cite1 and cite2, Paris occupies $105.40km^2$ while Barcelona occupies $101.4km^2$. The problem with this approach is that, while the areas don't differ much themselves, the number of nodes between the graphs does. This makes this approach unreliable.

A better approach is, as suggested, to look at the cities in Google Maps, from where I took 1 screenshot for each city, capturing the road networks:





Given the massive difference between the remaining graph sizes, and the massive difference between road networks present in the screenshots above, it is safe to conclude that city3 is Paris, and city1 is Barcelona.

3.2 Exercise 8

- a) The number of nodes and edges can be consulted in the top right corner of the overview. This graph has 3121 nodes and 18734 edges.
- b) The diameter and average path length can be calculated in the statistics view, to the right of the overview screen. We need simply to press a button to compute these two values. According to the report, this graph has diameter with value 12. The average path length has been determined to be 3.925411192993699.
- c) Each edge represents a flight from point A to point B. Its weigth represents the number of trips between A and B. Therefore, to find the pair of airports with more flights between each other, we need to skip over to the data laboratory, and in the data table select the Edges view. Then, we sort the edges by weigth, descending. The source is 3682 and the destination is 3830, with weigth 39. This means that, this graph has recorded 39 flights between Chicago Ohare Intl and Hartsfield Jackson Atlanta Intl.

- d) To list the top-5 of the airports that fly to the highest number of other airports, we need to compute the degree for all nodes, and then sort by degree, descending:

Name	City	Degree
Schiphol	Amsterdam	248
Frankfurt Main	Frankfurt	244
Charles De Gaulle	Paris	239
Ataturk	Istanbul	234
Hartsfield Jackson Atlanta Intl	Atlanta	211

Table 6: Top 5 airports by degree

- e) To answer this item, we need to compute the normalized betweenness centrality first, then sort descending. The dataset already had a column with values for the betweenness centrality, but those aren't normalized. To fix this, I ran the average path length again, but this time, i ticked the box that appears in the dialog to normalize the centralities. The result follows:

Airport	Betweenness Centrality	City and Country
Ted Stevens Anchorage Intl	0.06855652585360995	Anchorage, US
Los Angeles Intl	0.06574842820615719	Los Angeles, US
Charles De Gaulle	0.06256174667558051	Paris, France
Dubai Intl	0.05963127927902291	Dubai, United Arab Emirates
Frankfurt Main	0.054017294541720745	Frankfurt, Germany

Table 7: Top 5 airports, sorted by descending betweenness centrality, and their respective countries.

The number one airport was in fact, surprising. The other 4 make absolute sense in terms of centrality, given their size, and the cities they are in. According to the Ted Stevens Anchorage airport website, it is less than 9.5 hours away from 90% of the world, and ranks among the top 5 in the world for cargo throughput, as well as serving over 5 million passengers annually. Among other facts, this is enough to justify the values we see above.

- f) The normalized closeness centrality originated in item e) of this question, since I normalized all centralities. We need only to extract the table from the data laboratory:

Airport	Normalized Closeness Centrality
Frankfurt Main	0.4169450755044768
Charles de Gaulle	0.41395780814647737
Heathrow	0.41133816743572843
Dubai Intl	0.4074702886247878
Schiphol	0.40715124624820564

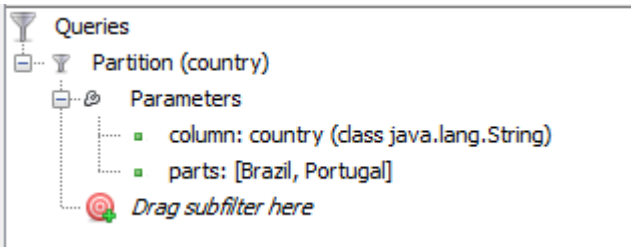
Table 8: Normalized Closeness Centrality

- g) The top-5 of countries with the highest number of airports, by percentage, can be calculated by partitioning the nodes by country:

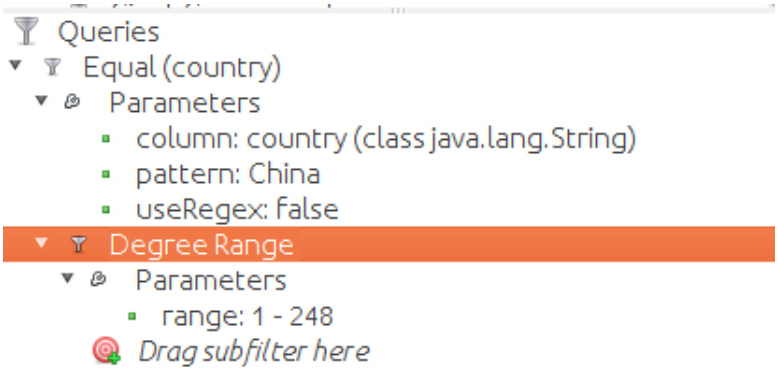
Country	% of world airports
USA	17.62%
China	5.29%
Canada	4.52%
Brazil	3.68%
Australia	3.56%

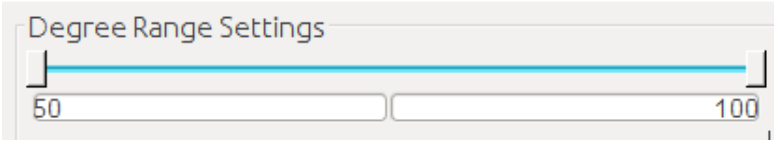
Table 9: Percentage of world airports by country

- h) To find the value of flights between Brazil and Portugal, I applied a filter to the graph. That filter acts according to the partitioning we can do by country. Inside the filter I then selected the possible values for country, which are Brazil and Portugal. The resulting filtered graph has 129 nodes and 398 edges, which means that this graph has recorded 398 flights between Brazil and Portugal. Follows the used filter:



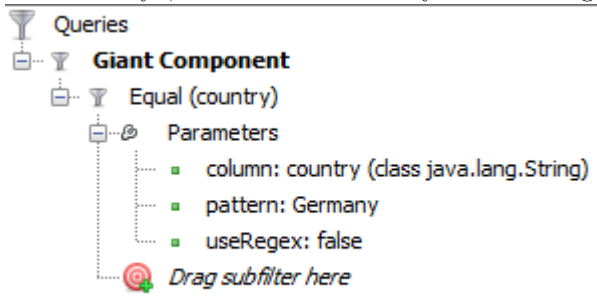
- i)To know how many airports in China have between 50 and 100 connecting flights, the first step is to filter all nodes, working only with Airports in China. Then, we need to filter those airports by degree, so it is between 50 and 100, like so:



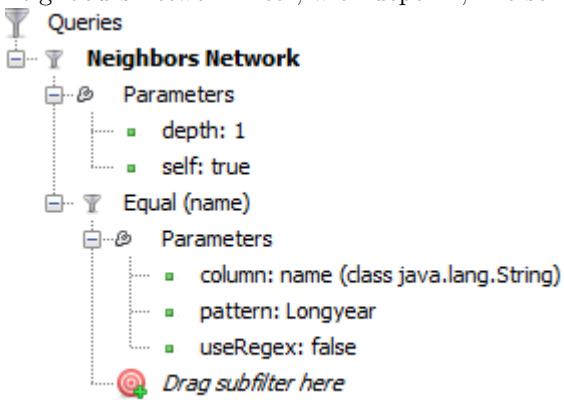


And looking at the top right corner, we see that we have 165 airports with 50 to 100 flights.

- j) To answer this item, we simply need to filter the giant component, for the nodes with country = "Germany", which can be done by the following filter:



- k) The airport that is further to the north is the one with the highest latitude, in this case it is Longyear, in Norway. In only 1 flight, we can reach its neighbours, which we do using the neighbours network filter, with depth 1, like so:

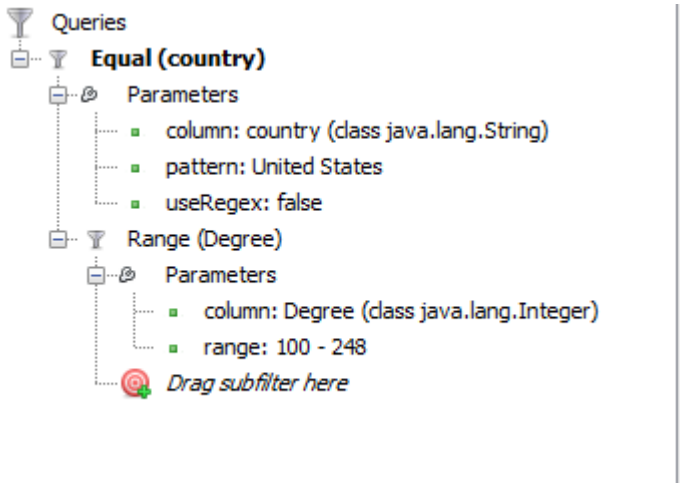


And we conclude that Longyear has 2 neighbours.

With 2 flights, we change the depth in the query above to 2, and get 113 nodes, so we can reach 113 destinations.

With 3 flights, again changing the depth, we get 986 nodes, and 986 reachable destinations.

- l) Here we have to make all calculations before any filtering or partitioning. Load up the entire network, and hit, by that order, Geo Layout, Avg. Degree, Network Diameter (which computes centralities). Know we use a filter, by country, like so:



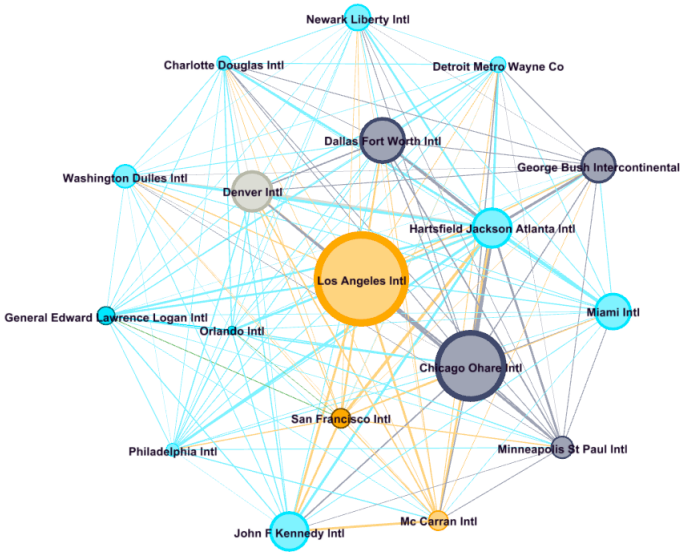
So the size of the node reflects the betweenness centrality, which we already computed, we go to the right side of the screen in Gephi, click on nodes, select size, and set the minimum to 4, and max to 100.

Then, to colour the nodes by timezone (labeled as tz in the dataset), we go over to Color, select partition, select tz, create a pallete that has as many colors as needed, and apply said pallete to the graph. The pallette itself is displayed in then Annex.

We then label the nodes by name.

The final step is getting an acceptable layout, so the nodes and their labels dont overlap. I used Fruchterman Reingold.

The result is as follows (full size in the annex):



4 Annex

Exercise 2:

```
import random
import networkx as nx

p1 = 0.0001
p2 = 0.005


def generate_erdos_renyi_graph(size, prob):
    graph = nx.Graph()
    nodes = list(x for x in range(1,size+1))
    graph.add_nodes_from(nodes)
    # triangular superior matrix (undirected graph)
    for i in graph.nodes:
        for j in graph.nodes:
            # not sure if < or <= here
            if i != j and random.uniform(0, 1) < prob:
                graph.add_edge(i,j)

    return graph


def print_graph_to_file(filename,graph):

    file = open(filename, "w+")

    file.write("%d\n" % len(graph.nodes))

    for edge in graph.edges:
        file.write("%d %d\n" % (edge[0], edge[1]))

    file.close()

    return


if __name__ == '__main__':

    size = 1000
    erdos_renyi_1 = generate_erdos_renyi_graph(size, p1)
    erdos_renyi_2 = generate_erdos_renyi_graph(size, p2)

    print_graph_to_file("random1.txt", erdos_renyi_1)
    print_graph_to_file("random2.txt", erdos_renyi_2)
```

Exercise 3:

```
    # compute the size of the random networks we generated

import networkx as nx
import time

def read_graph_from_file(filename):

    #read from file
    file = open(filename, "r")
    lines = file.readlines()
    file.close()

    size = int(lines[0])
    nodes = (x for x in range(0, size))
    graph = nx.Graph()

    #add nodes
    for node in nodes:
        graph.add_node(node,visited=False)


    # read and add edges
    for i in range(1, len(lines)):
        x,y = lines[i].split()
        graph.add_edge(int(x),int(y))

    return graph


# implementing BFS here
# sources:
#   https://en.wikipedia.org/wiki/Breadth-first_search
#   https://www.geeksforgeeks.org/connected-components-in-an-undirected-graph/
def calculate_giant_component_size(graph):

    time_start = int(round(time.time() * 1000))

    components = list()
    node_queue = list()
    visitedNodes = set()
    nodes_to_visit = list(graph.nodes)
    root = nodes_to_visit.pop(0)
    currNode = root
    currComp = list()
    sizes = list()
    node_queue.append(currNode)

    while len(visitedNodes) < len(list(graph.nodes)):

        currNode = node_queue.pop(0)
        currComp.append(currNode)
        visitedNodes.add(currNode)
```

```
            neighbours = nx.all_neighbors(graph,currNode)

            for n in neighbours:
                if n not in visitedNodes and n not in node_queue
                    and n in nodes_to_visit:
                        node_queue.append(n)
                        nodes_to_visit.remove(n)

            # component end
            if len(node_queue) == 0:
                if len(nodes_to_visit) > 0:
                    node_queue.append(nodes_to_visit.pop(0))
                components.append(currComp)
                sizes.append(len(currComp))
                currComp = list()

        time_end = int(round(time.time() * 1000))
        return max(sizes)

if __name__ == '__main__':
    graph1 = read_graph_from_file("random1.txt")
    print "Graph 1 has probability 0.0001
        and giant component size: %d " % calculate_giant_component_size(graph1)

    graph2 = read_graph_from_file("random2.txt")
    print "Graph 2 has probability 0.005
        and giant component size: %d " % calculate_giant_component_size(graph2)
```

Exercise 4:

```
    import networkx
import matplotlib.pyplot as plt
import numpy

import ex2
import ex3


n_nodes = 1000
s_prob = 0.0001
f_prob = 0.005
step_prob = 0.0001

res_x = list()
res_y = list()


# plotting code adapter from here:
#https://stackoverflow.com/questions/21519203/
plotting-a-list-of-x-y-coordinates-in-python-matplotlib

#Show a plot of your results, with the X axis representing p
#and the Y axis representing the size of the giant component.
if __name__ == '__main__':

    steps = numpy.arange(s_prob, f_prob+step_prob, step_prob)
    print "Using %d steps." % len(steps)
    print "Working..."
    for step in steps:
        graph = ex2.generate_erdos_renyi_graph(n_nodes,step)
        giant_comp_size = ex3.calculate_giant_component_size(graph)
        res_x.append(step)
        res_y.append(giant_comp_size)
    print "Done, plotting."
    plt.scatter(res_x,res_y)
    #plt.plot(res_x,res_y)
    plt.show()
```

Exercise 5:

```
    from __future__ import division
import networkx as nx
import random
from ex2 import print_graph_to_file

def get_sum_all_degrees(graph):
    sum = 0;
    for node in graph.nodes:
        sum += graph.degree(node,1)

    return sum

def generate_barabasi_albert_graph(n,m0,m):
    graph = nx.Graph()

    #graph starts with m0 nodes (1 to m0 included)
    for i in range(1,m0+1):
        graph.add_node(i)

    # make graph fully connected
    for n1 in graph.nodes:
        for n2 in graph.nodes:
            if n1 != n2 and not graph.has_edge(n1,n2):
                graph.add_edge(n1,n2)

    #add nodes until graph has # nodes
    while len(graph.nodes) < n:

        curr = len(graph.nodes) + 1
        # get m random nodes that already exist,
        # and add an edge between them and curr
        nodes = list()
        sum_all_degrees =  get_sum_all_degrees(graph)
        while len(nodes) < m:
            j = random.randint(1,len(graph.nodes))
            p_i = graph.degree(j,1) / sum_all_degrees
            target = random.uniform(0,1)
            if not j in nodes and p_i > target:
                nodes.append(j)
        for node in nodes:
            graph.add_edge(node,curr)

    return graph

if __name__ == '__main__':

    g1 = generate_barabasi_albert_graph(1000,3,1)
    print_graph_to_file("ba1.txt", g1)
    g2 = generate_barabasi_albert_graph(1000,6,3)
    print_graph_to_file("ba2.txt", g2)
```

Exercise 6:

```
    import ex3
import matplotlib.pyplot as plt

def degree_dist_cummulative_binning(graph):
    dgs = list()
    for node in graph.nodes:
        dgs.append((node,graph.degree(node,1)))

    cummulative_dgs = list()
    for i in range(1, len(graph.node) + 1):
        sum = 0
        for pair in dgs:
            if(pair[1] >= i):
                sum += 1
        cummulative_dgs.append((i,sum))

    # filter pairs (dg,v) where the number of nodes v with degree dg is 0
    return [x for x in cummulative_dgs if not x[1] == 0]


if __name__ == '__main__':

    #g1 = ex3.read_graph_from_file("ba1.txt")
    g2 = ex3.read_graph_from_file("ba2.txt")

    #dgs = degree_dist_cummulative_binning(g1)
    dgs = degree_dist_cummulative_binning(g2)

    dgs_x = list()
    dgs_y = list()
    for d in dgs:
        dgs_x.append(d[0])
        dgs_y.append(d[1])
    plt.scatter(dgs_x,dgs_y)
    plt.show()
```

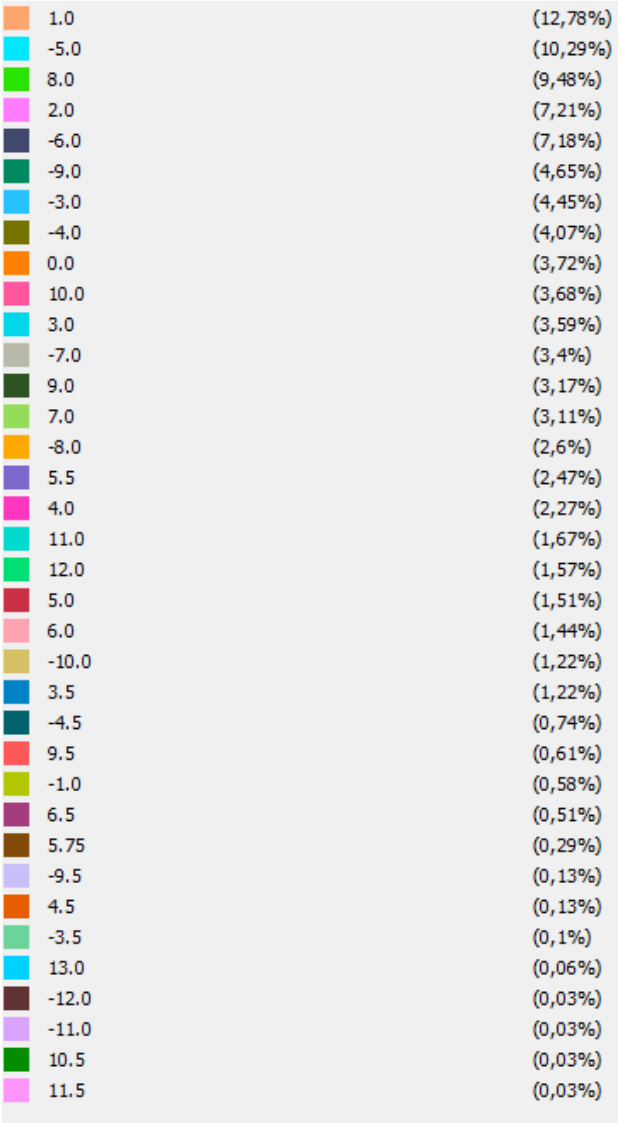


Figure 1: Palete used to color the nodes for item l in Exercise 8

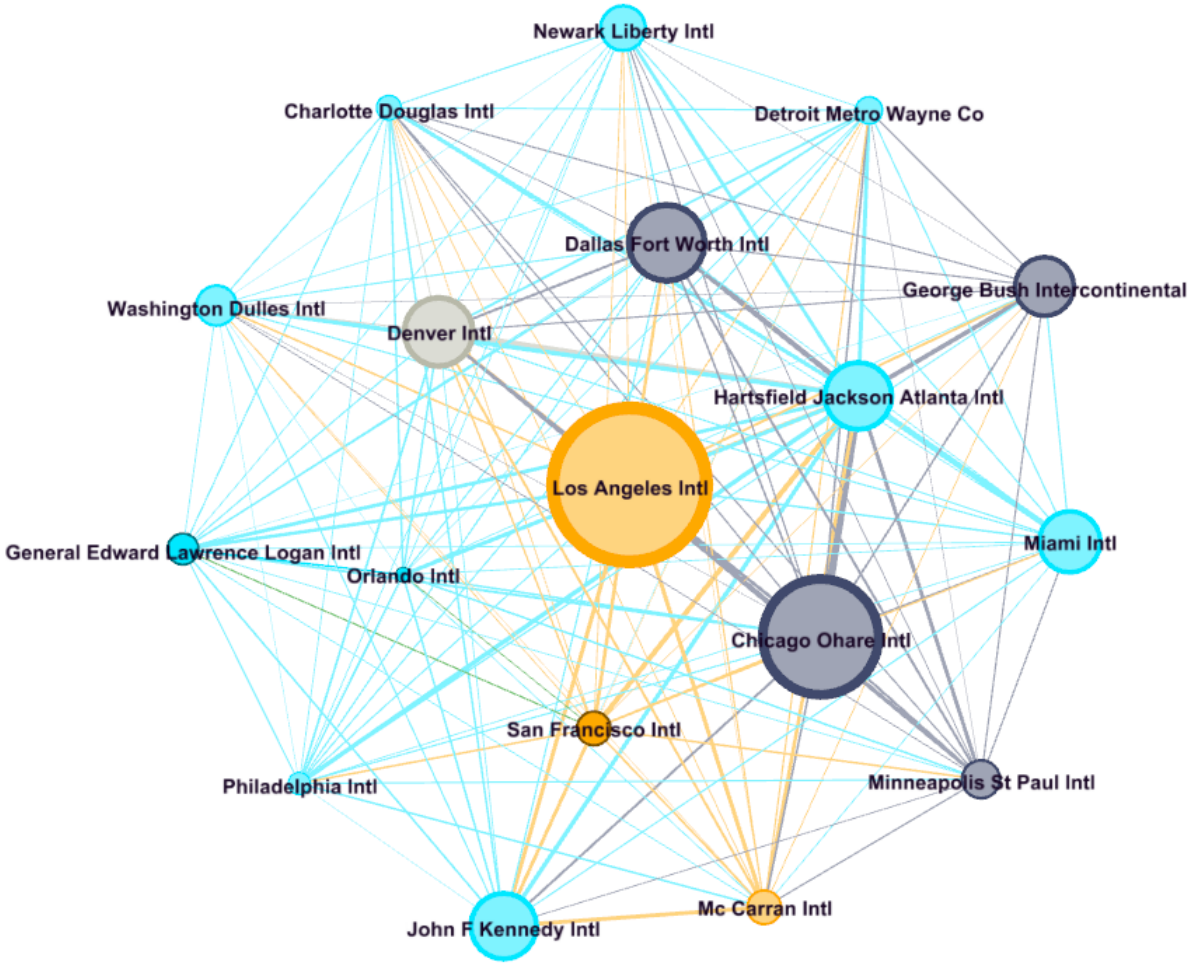


Figure 2: Final network as an answer to Exercise 8

References

[1] Barabasi-albert model tips. http://www.dcc.fc.up.pt/~pribeiro/aulas/ns1819/homework/bamodel.pdf. Accessed:2019-03-22.

[2] Clustering coefficient (slide 8). http://www.dcc.fc.up.pt/~pribeiro/aulas/ns1819/2_graphmodels.pdf. Accessed:2019-03-22.

[3] Erdos-renyi wiki page. https://en.wikipedia.org/wiki/Erd%C5%91s%E2%80%93R%C3%A9nyi_model. Accessed:2019-03-22.

[4] Floyd–warshall algorithm. https://en.wikipedia.org/wiki/Floyd%E2%80%93Warshall_algorithm. Accessed:2019-03-22.

[5] Giant component (slide 10). http://www.dcc.fc.up.pt/~pribeiro/aulas/ns1819/2_graphmodels.pdf. Accessed:2019-03-22.

[6] (normalized) closeness centrality (slide 28). http://www.dcc.fc.up.pt/~pribeiro/aulas/ns1819/3_centrality.pdf. Accessed:2019-03-22.