Homework 4

Group 12: Mattia Di Fulvio, Esin Ildiz, Giannis Lakafosis

Part 1: Parsing Data and Creating the Graph

In [1]:

```
import json
import networkx as nx
from collections import defaultdict
```

In [2]:

```
def parseFile(path):
    json_data=open(path).read()
    dataset = json.loads(json_data)
    #Declare structures that We will use
    conferences= defaultdict(set)
    workedWith = defaultdict(set)
    workedOn   = defaultdict(set)
    #Analyze the file
    for entry in dataset:
        people={author["author_id"] for author in entry["authors"]}
        #Compute the list of people who published in a conference for e
very conference
        if entry["id_conference_int"] not in conferences:
            conferences[entry["id_conference_int"]] = people
        else:
            conferences[entry["id_conference_int"]].update(people)
        for author in entry["authors"]:
            #Compute the list of people who worked with every author
            if author["author_id"] not in workedWith.keys():
                workedWith[author["author_id"]] = people-{author["autho
r_id"]}
```

```
            else:
                workedWith[author["author_id"]].update(people-
{author["author_id"]})
            #Compute the list of publication for every author
            if author["author_id"] not in workedOn:
                workedOn[author["author_id"]]= {entry["id_publication_i
nt"]}
            else:
                workedOn[author["author_id"]].add(entry["id_publication
_int"])
    return computeGraph(workedWith, workedOn), conferences
```

In [3]:

```
def computeGraph(workedWith, workedOn):
    G= nx.Graph()
    for author1 in workedWith:
        G.add_node(author1)
        for author2 in workedWith[author1]:
            jDistance= 1-len(workedOn[author1].intersection(workedOn[au
thor2]))/len(workedOn[author1].union(workedOn[author2]))
            G.add_edge(author1,author2,weight=jDistance)
    return G
```

In [4]:

```
path="C:\\Users\\Es\\full_dblp.json"
G, conferences= parseFile(path)
print(nx.info(G))
```

```
Name:
Type: Graph
Number of nodes: 904664
Number of edges: 3679276
Average degree:   8.1340
```

Part 2: Statistics and Visualizations

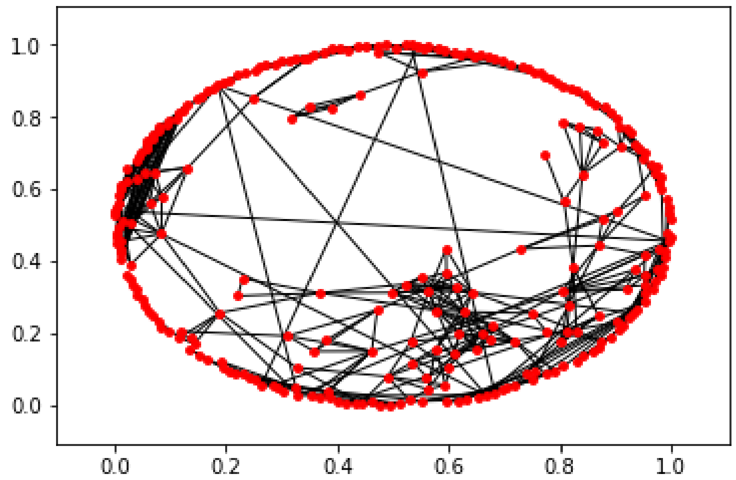
In [5]:

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

In [6]:
```
def conferenceSubgraph(graph, conferences, conferenceId):
    if conferenceId not in conferences:
        print("Error, conference not found.")
        return
    ret= graph.subgraph(conferences[conferenceId])
    print("\nThis is the subgraph induced by the set of autor who publi
shed in the input conference, We took %d as conference" %
(conferenceId))
    plt.clf()
    nx.draw_networkx(ret, node_size=15, with_labels= False)
    plt.show()
    return ret
```

In [7]:
```
conferenceId=4627
newGraph= conferenceSubgraph(G, conferences, conferenceId)
```

```
This is the subgraph induced by the set of autor who published in the i
nput conference, We took 4627 as conference
```



In [8]:
```
print(nx.info(newGraph))
```

```
Name:
```

```
Type: Graph
Number of nodes: 275
Number of edges: 539
Average degree:   3.9200
```

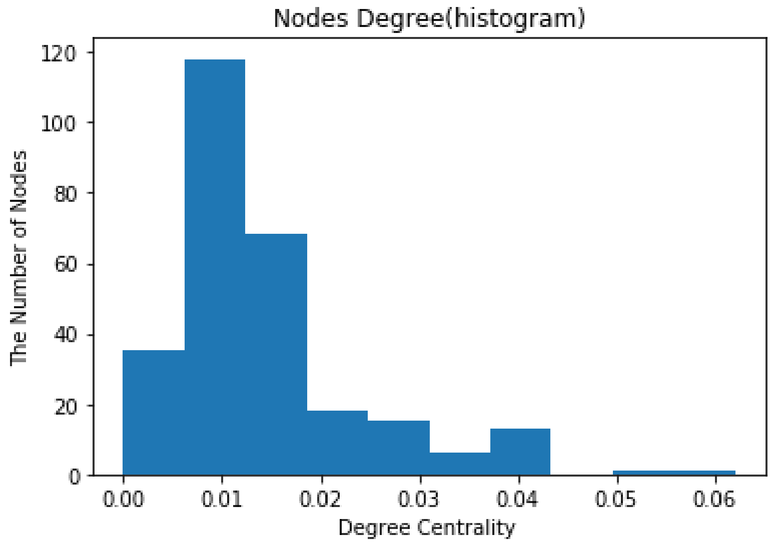
In [9]:

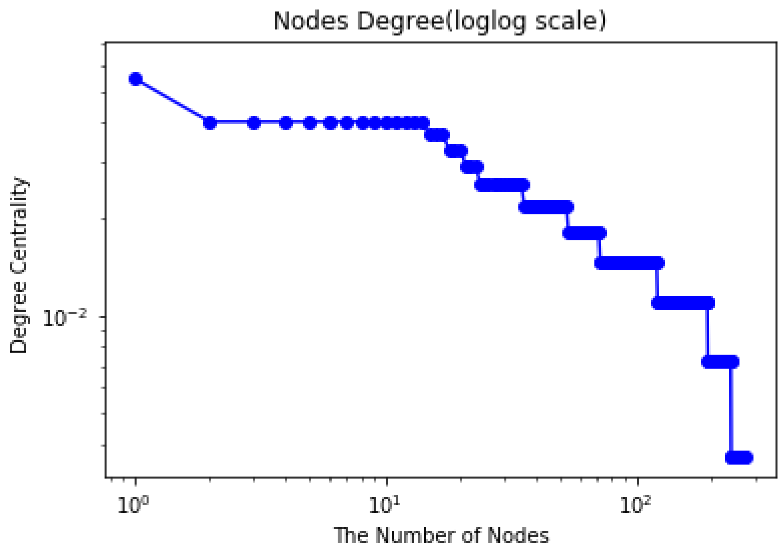
```
def degree(graph):
    degree= sorted(list(nx.degree_centrality(graph).values()))[::-1]
    #histogram
    plt.hist(degree)
    plt.title("Nodes Degree(histogram)")
    plt.xlabel('Degree Centrality ')
    plt.ylabel('The Number of Nodes')
    plt.show()
    #loglog
    plt.loglog(degree, 'b', marker = 'o')
    plt.title("Nodes Degree(loglog scale)")
    plt.xlabel('The Number of Nodes')
    plt.ylabel('Degree Centrality ')
    plt.show()
    print("In the next graph node's size depends by their Dregree centr
ality")
    plt.figure(figsize=(6,6))
    pos_c= nx.spring_layout(graph, iterations = 1000)
    nsize = np.array([v for v in (list(nx.degree_centrality(graph).valu
es()))])
    nsize = 600*(nsize  - min(nsize))/(max(nsize) - min(nsize))
    nodes=nx.draw_networkx_nodes(graph, pos = pos_c, node_size = nsize)
    nodes.set_edgecolor('b')
    nx.draw_networkx_edges(graph, pos = pos_c, alpha = .1)
    plt.axis('off')
    plt.show()
```

$degreecentrality = \dfrac{deg(v)}{m-1}$, where m is number of nodes

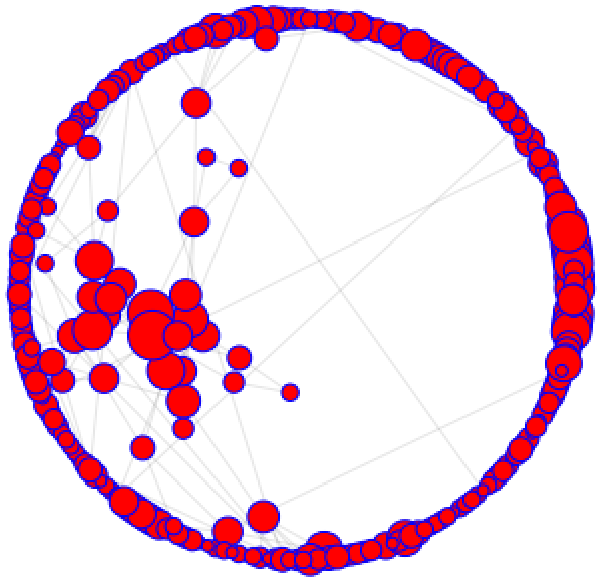
Degree centrality of a node refers to the number of edges attached to the node. In order to know the standardized number of edges attached to the node, we need to divide number of edges attached to the node by n-1 (n = the number of nodes) for each node. Since the graph has 275 nodes, 275-1=274 is the denominator of each node. Plots show us that large set of nodes corresponds to low degrees(0-0.02 degree centrality)

In [10]:
```
degree(newGraph)
```





In the next graph node's size depends by their Dregree centrality



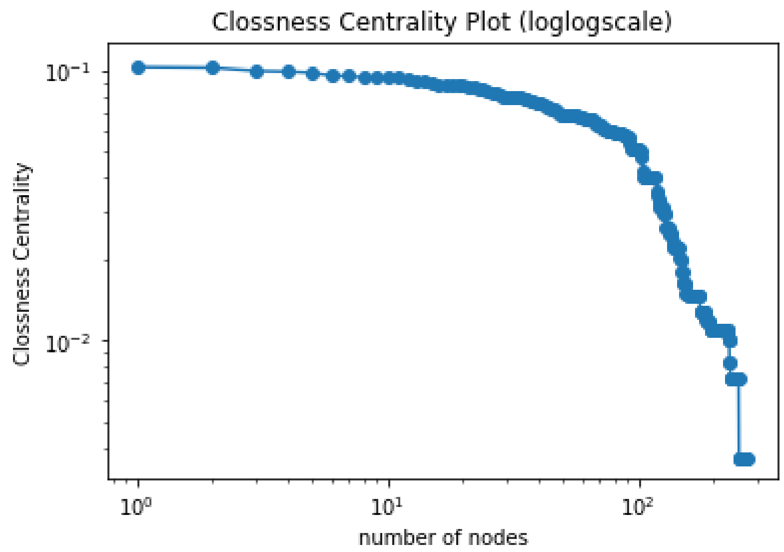
From the previous plots and the graph We can visualize that for the conference We had as input, there are few nodes with high values of Degree centrality so we checked again our data and discovered that few author(14%) who cooperated with an high values of with many different authors(more than 10) and an higher number of author(around 55%) who published alone or maybe published always with the same colleagues because they have 4 or less nodes

In [11]:

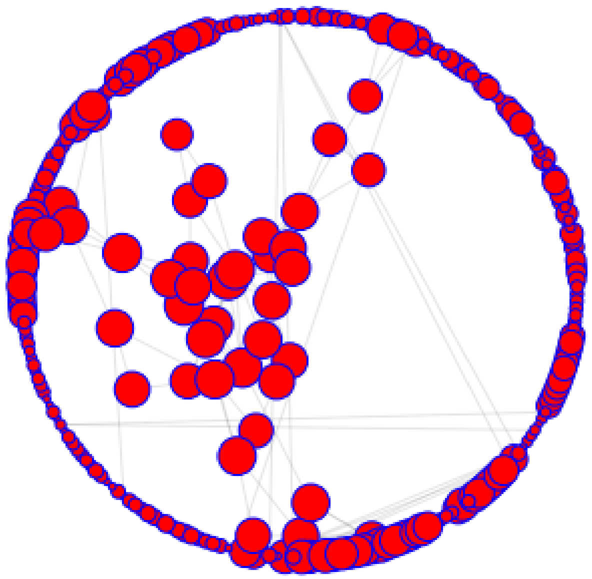
```
def closeness(graph):
    closeness  = sorted(list(nx.closeness_centrality(graph).values()))
[::-1]
    plt.title("Clossness Centrality Plot (loglogscale)")
    plt.xlabel('number of nodes')
    plt.ylabel('Clossness Centrality')
    plt.loglog(closeness,marker='o')
    plt.show()
    print("In the following graph node's size depends by their Closenes
s centrality")
    plt.figure(figsize=(6,6))
    pos_c= nx.spring_layout(graph, iterations = 1000)
    nsize = np.array([v for v in list(nx.closeness_centrality(graph).va
lues())])
    nsize = 400*(nsize  - min(nsize))/(max(nsize) - min(nsize))
    nodes=nx.draw_networkx_nodes(graph, pos = pos_c, node_size = nsize)
    nodes.set_edgecolor('b')
    nx.draw_networkx_edges(graph, pos = pos_c, alpha = .1)
    plt.axis('off')
    plt.show()
```

In [12]:

```
closeness(newGraph)
```



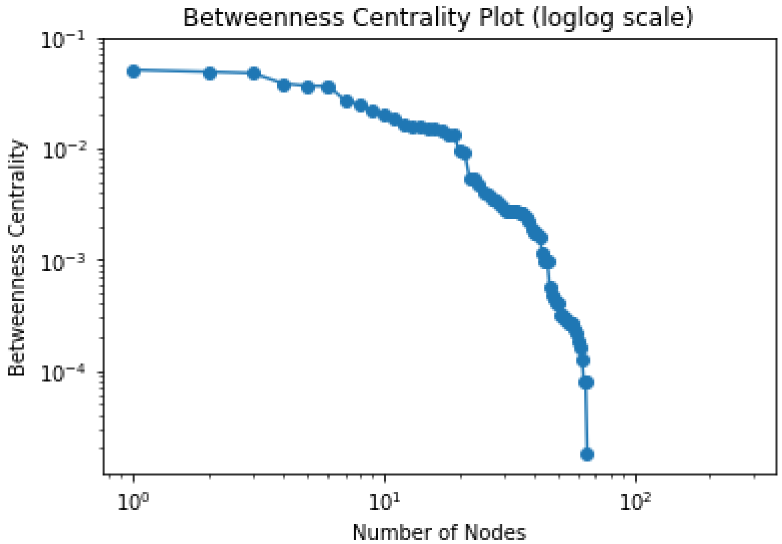
In the following graph node's size depends by their Closeness central ity



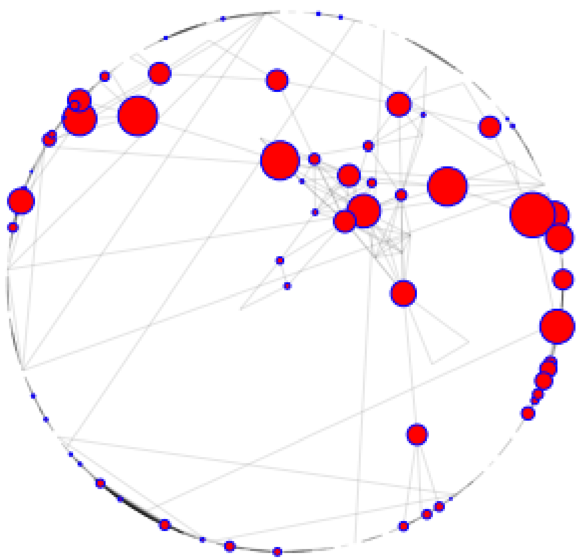
Clossness centrality is calculated as the average length of the shortest paths between the node and all other nodes in the graph. Thus the more central a node is, the closer it is to all other nodes

In [13]:
```
def betweenness(graph):
    betweenness=
sorted(list(nx.betweenness_centrality(graph).values()))[::-1]
    plt.loglog(betweenness, marker = 'o')
    plt.title("Betweenness Centrality Plot (loglog scale)")
    plt.xlabel('Number of Nodes')
    plt.ylabel('Betweenness Centrality')
    plt.show()
    print("In the next graph node's size depends by their Betweenness c
entrality")
    plt.figure(figsize=(6,6))
    pos_c= nx.spring_layout(graph, iterations = 1000)
    nsize = np.array([v for v in
list(nx.betweenness_centrality(graph).values())])
    nsize = 500*(nsize  - min(nsize))/(max(nsize) - min(nsize))
    nodes=nx.draw_networkx_nodes(graph, pos = pos_c, node_size = nsize)
    nodes.set_edgecolor('b')
    nx.draw_networkx_edges(graph, pos = pos_c, alpha = .1)
    plt.axis('off')
    plt.show()
```

In [14]:
```
betweenness(newGraph)
```



In the next graph node's size depends by their Betweenness centrality



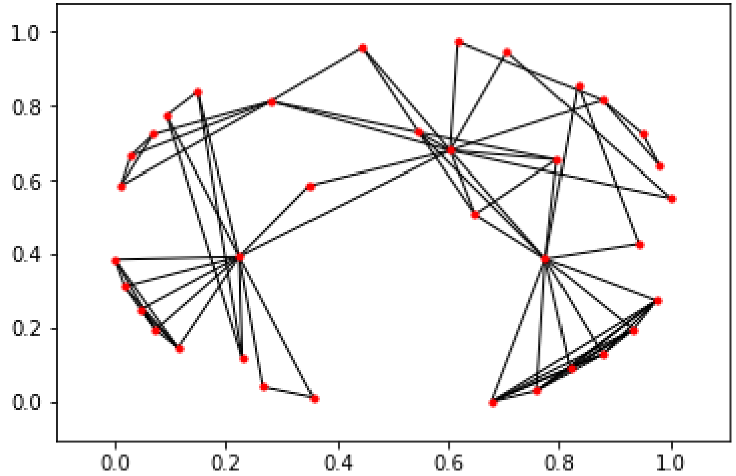
We can see that the betweenness centrality has a similar behaviour to the degree centrality, we expected this cause the denominator depends by the nodes in the graph so it is the same for every author but those with an high number of nodes will have an higher chance to be part of a shortest path so values of degree and betweennes centraly are in some way related

Hop Distance

In [15]:
```
def ego_graph(graph, author, hops):
    if author not in graph.nodes():
        print("The author is not in the graph")
        return
    print("The graph above is the ego graph of author with hop distance at most %d from author %d" %(hops, author))
    ego= nx.ego_graph(graph, author, radius=hops, center=True, undirected=False, distance=None)
    plt.clf()
    nx.draw_networkx(ego, node_size=10, with_labels= False)
    plt.show()
    return ego
```

In [25]:
```
ego= ego_graph(G, 20405,3)
#Our code is working but we had some issues while plotting full dataset
```

```
The graph above is the ego graph of author with hop distance at most 3
 from author 20405
```



Part 3: Shortest Path

In [17]:
```
import numpy as np
import heapq

def dijkstra(graph,start, end):
    heap= []
    nodes= graph.nodes()
    if start not in nodes or end not in nodes:
        print("The node in input is not in the graph")
        return {}
    explored={start:0}
    unexplored = {key: np.Inf for key in nodes if key!=start}
    current= start
    while current!=end:
        for edge in graph.edges(current):
            other= edge[1]
            edgeWeight= graph.get_edge_data(current,other)["weight"]
            try:
                if explored[current]+edgeWeight < unexplored[other]:
```

```
                    unexplored[other]= explored[current]+edgeWeight
                    heapq.heappush(heap, (unexplored[other], other))
            except: continue #avoiding keyerror when a node is already
 explored
        while True:
            if len(heap)==0: return explored
            cWeight, current= heapq.heappop(heap)
            if current not in explored: break
        if cWeight== np.Inf: break
        else:
            unexplored.pop(current, None)
            explored[current]= cWeight
    return explored
```

In [18]:

```
def shortestPathWeight(graph, author):
    print("Trying to reach Aris from author: %d" %(author))
    distances = dijkstra(graph, author, 256176)
    if 256176 not in distances:
        print("You can't reach Aris starting from this author.")
        return
    print("The weight of the shortest path between Aris and the other a
uthor is:"+ str((distances[256176])))
    return distances[256176]
```

In [24]:

```
spweight= shortestPathWeight(G, 365452)
```

```
Trying to reach Aris from author: 365452
The node in input is not in the graph
You can't reach Aris starting from this author.
```

In the previous blocks We simply use the funcion We already talked about.

In [20]:

```
def multisourceDijkstra(graph,startList):
    heap= []
    nodes= graph.nodes()
    explored={key:0 for key in startList if key in nodes}
    unexplored = {key: np.Inf for key in nodes if key not in startList}
```

```
    #mod the first part to make the algorithm multisource
    for starter in explored.keys():
        for edge in graph.edges(starter):
            other= edge[1]
            edgeWeight= graph.get_edge_data(starter,other)["weight"]
            try:
                if edgeWeight < unexplored[other]:
                    unexplored[other]= edgeWeight
                    heapq.heappush(heap, (unexplored[other], other))
            except: continue #avoiding keyerror when a node is already
 explored
    #select the starter
    while True:
        if len(heap)==0: return explored
        cWeight, current= heapq.heappop(heap)
        if current not in explored: break
    if cWeight== np.Inf: return explored     #maybe useless
    else:
        unexplored.pop(current, None)
        explored[current]= cWeight

    #classic Dijkstra
    while len(unexplored.keys())>0:
        for edge in graph.edges(current):
            other= edge[1]
            edgeWeight= graph.get_edge_data(current,other)["weight"]
            try:
                if explored[current]+edgeWeight < unexplored[other]: #c
Weight instead of explored[current]
                    unexplored[other]= explored[current]+edgeWeight
                    heapq.heappush(heap, (unexplored[other], other))
            except: continue #avoiding keyerror when a node is already
 explored
        while True:
            if len(heap)==0: return explored
            cWeight, current= heapq.heappop(heap)
            if current not in explored: break
        if cWeight== np.Inf: break
        else:
```