The neural network of C. elegans

1 Introduction

1.1 Biological context

The graph that we chose to analyze for this project is the neural network of Caenorhabditis elegans (C. elegans). It is a nematode worm made by a thousand of cell, for about $1\ mm$ in length. It is one of the most common model organism in biology, for it is convenient in many research areas. First, being transparent, it is extremely suitable for visualization. From a structural and functional point of view, it is simple enough to be well understood, but still complex enough to find interesting results. For this reason, it was the first multicellular organism to be fully sequenced and, as of today, it is still the unique for which the complete neural map (connectome) is available. Deriving the full genome of C. elegans was a crucial step before moving to the human genome. Correspondingly, the understanding of its neural network is an important step towards the major task of grasping the human brain.

The neural network of the C. elegans became a model also for the study of complex networks, thanks to the article [Watts] by Watts and Strogatz. There, the authors considered C. elegans connectome as a paradigm for biological networks, while introducing small-worldness as a general feature for real complex networks.

In the specific case of our dataset, the network is made up of 297 nodes, corresponding to as many neurons. Two nodes are connected by an edge if there is at least one synapse (or a gap junction) between the corresponding neurons. A weight is also assigned to each edge, to reflect the number of neural connections (synapses or gap junctions) between the two nodes.

The spread of a signal between connected neurons (chemically for synapses and electrically for gap junctions) occurs always in the same direction. This means the complex networks that we studied is intrinsically a directed graph. Nevertheless, most of our considerations are on the graph structure as if it was undirected. Our goal was to apply to our hundreds-of-nodes network the most suitable topics explored during the lectures.

1.2 Import libraries and data

First, we import all libraries that are needed to run all the following code.

```
In [ ]: import networkx as nx
        import numpy as np
        import matplotlib.pyplot as plt
        import matplotlib.figure as figure
```

```python
import matplotlib.figure as figure
! pip install cdlib
from cdlib import algorithms, viz
import csv
import pandas as pd
from io import BytesIO
from zipfile import ZipFile
from urllib.request import urlopen
```

```
Requirement already satisfied: cdlib in /usr/local/lib/python3.6/d
ist-packages (0.1.10)
Requirement already satisfied: pulp>=2.1 in /usr/local/lib/python3
.6/dist-packages (from cdlib) (2.4)
Requirement already satisfied: ASLPAw==2.0.0 in /usr/local/lib/pyt
hon3.6/dist-packages (from cdlib) (2.0.0)
Requirement already satisfied: chinese-whispers in /usr/local/lib/
python3.6/dist-packages (from cdlib) (0.7.3)
Requirement already satisfied: bimlpa in /usr/local/lib/python3.6/
dist-packages (from cdlib) (0.1.2)
Requirement already satisfied: future>=0.16.* in /usr/local/lib/py
thon3.6/dist-packages (from cdlib) (0.16.0)
Requirement already satisfied: shuffle-graph==1.1.1 in /usr/local/
lib/python3.6/dist-packages (from cdlib) (1.1.1)
Requirement already satisfied: networkx>=2.4 in /usr/local/lib/pyt
hon3.6/dist-packages (from cdlib) (2.5)
Requirement already satisfied: python-louvain==0.14 in /usr/local/
lib/python3.6/dist-packages (from cdlib) (0.14)
Requirement already satisfied: pquality==0.0.7 in /usr/local/lib/p
ython3.6/dist-packages (from cdlib) (0.0.7)
Requirement already satisfied: numpy>=1.15.* in /usr/local/lib/pyt
hon3.6/dist-packages (from cdlib) (1.19.4)
Requirement already satisfied: pandas>=0.24 in /usr/local/lib/pyth
on3.6/dist-packages (from cdlib) (1.1.5)
Requirement already satisfied: demon>=2.0.5 in /usr/local/lib/pyth
on3.6/dist-packages (from cdlib) (2.0.5)
Requirement already satisfied: nf1 in /usr/local/lib/python3.6/dis
t-packages (from cdlib) (0.0.3)
Requirement already satisfied: scikit-learn>=0.21.* in /usr/local/
lib/python3.6/dist-packages (from cdlib) (0.22.2.post1)
Requirement already satisfied: eva-lcd in /usr/local/lib/python3.6
/dist-packages (from cdlib) (0.1.0)
Requirement already satisfied: seaborn>=0.9.* in /usr/local/lib/py
thon3.6/dist-packages (from cdlib) (0.11.0)
Requirement already satisfied: matplotlib>=3.0.* in /usr/local/lib
/python3.6/dist-packages (from cdlib) (3.2.2)
Requirement already satisfied: scipy>=1.3.* in /usr/local/lib/pyth
on3.6/dist-packages (from cdlib) (1.4.1)
Requirement already satisfied: omega-index-py3 in /usr/local/lib/p
ython3.6/dist-packages (from cdlib) (0.3)
Requirement already satisfied: tqdm>=4.20.* in /usr/local/lib/pyth
on3.6/dist-packages (from cdlib) (4.41.1)
Requirement already satisfied: karateclub>=1.0.0 in /usr/local/lib
/python3.6/dist-packages (from cdlib) (1.0.22)
Requirement already satisfied: markov-clustering in /usr/local/lib
/python3.6/dist-packages (from cdlib) (0.0.6.dev0)
Requirement already satisfied: amply>=0.1.2 in /usr/local/lib/pyth
on3.6/dist-packages (from pulp>=2.1->cdlib) (0.1.4)
Requirement already satisfied: count-dict>=1.0.1 in /usr/local/lib
```

```
/python3.6/dist-packages (from ASLPAw==2.0.0->cdlib) (1.0.2)
Requirement already satisfied: multivalued-dict>=1.7.1 in /usr/loc
al/lib/python3.6/dist-packages (from ASLPAw==2.0.0->cdlib) (1.7.1)
Requirement already satisfied: decorator>=4.3.0 in /usr/local/lib/
python3.6/dist-packages (from networkx>=2.4->cdlib) (4.4.2)
Requirement already satisfied: pytz>=2017.2 in /usr/local/lib/pyth
on3.6/dist-packages (from pandas>=0.24->cdlib) (2018.9)
Requirement already satisfied: python-dateutil>=2.7.3 in /usr/loca
l/lib/python3.6/dist-packages (from pandas>=0.24->cdlib) (2.8.1)
Requirement already satisfied: joblib>=0.11 in /usr/local/lib/pyth
on3.6/dist-packages (from scikit-learn>=0.21.*->cdlib) (1.0.0)
Requirement already satisfied: cycler>=0.10 in /usr/local/lib/pyth
on3.6/dist-packages (from matplotlib>=3.0.*->cdlib) (0.10.0)
Requirement already satisfied: kiwisolver>=1.0.1 in /usr/local/lib
/python3.6/dist-packages (from matplotlib>=3.0.*->cdlib) (1.3.1)
Requirement already satisfied: pyparsing!=2.0.4,!=2.1.2,!=2.1.6,>=
2.0.1 in /usr/local/lib/python3.6/dist-packages (from matplotlib>=
3.0.*->cdlib) (2.4.7)
Requirement already satisfied: pygsp in /usr/local/lib/python3.6/d
ist-packages (from karateclub>=1.0.0->cdlib) (0.5.1)
Requirement already satisfied: gensim==3.8.3 in /usr/local/lib/pyt
hon3.6/dist-packages (from karateclub>=1.0.0->cdlib) (3.8.3)
Requirement already satisfied: six in /usr/local/lib/python3.6/dis
t-packages (from karateclub>=1.0.0->cdlib) (1.15.0)
Requirement already satisfied: docutils>=0.3 in /usr/local/lib/pyt
hon3.6/dist-packages (from amply>=0.1.2->pulp>=2.1->cdlib) (0.16)
Requirement already satisfied: smart-open>=1.8.1 in /usr/local/lib
/python3.6/dist-packages (from gensim==3.8.3->karateclub>=1.0.0->c
dlib) (4.0.1)
```

We can now import the dataset from Netzschleuder repository. To do so, we have to get a zipped file, containing two CSV files that are used to build the graph: `nodes.csv` and `edges.csv` .

In [ ]:
```
# get the zipped file from link below
address = "https://networks.skewed.de/net/celegansneural/files/cele
resp = urlopen(address)
zipfile = ZipFile(BytesIO(resp.read()))
```

Some preliminar treatment is necessary before using our data in `nodes.csv` . The original file has indeed a cumbersome format for nodes positions:

```
In [ ]: # load nodes.csv into a pandas DataFrame
        df = pd.read_csv(zipfile.open('nodes.csv'), index_col= False)

        # show data table
        print(df)
```

```
     # index  label                                 _pos
0          0      1  array([-7.88543459,  2.82128983])
1          1     51  array([-7.98005628,  2.73030094])
2          2     72  array([-8.17231333,  2.92122238])
3          3     77  array([-8.08359818,  2.91190722])
4          4     78  array([-8.09595906,  2.93494408])
..       ...    ...                                  ...
292      292    298  array([-8.60902516,  3.29127624])
293      293    299  array([-8.26625222,  3.44826753])
294      294    300  array([-8.48540216,  3.43185403])
295      295    301  array([-7.67134619,  2.46809062])
296      296    302  array([-7.77777291,  2.4112471 ])

[297 rows x 3 columns]
```

The following lines of code are thus used to extract positions on x and y axis and to store them into a new CSV file: `nodes_pos.csv` .

```
In [ ]: # output CSV has the three columns: label, position on x, position
        with open("nodes_pos.csv", "w") as csvfile:
                csv_writer = csv.writer(csvfile)
                csv_writer.writerow(['label', 'xpos', 'ypos'])
                for row in np.array(df):
                    csv_writer.writerow([row[0], float(row[2][7:17]), float
```

Here is the resulting data table:

```
In [ ]: # load nodes_pos.csv into a pandas DataFrame
        df = pd.read_csv("nodes_pos.csv", index_col= False)

        #show data table
        print(df)
```

```
     label      xpos      ypos
0        0 -7.885434  2.821290
1        1 -7.980056  2.730301
2        2 -8.172313  2.921222
3        3 -8.083598  2.911907
4        4 -8.095959  2.934944
..     ...       ...       ...
292    292 -8.609025  3.291276
293    293 -8.266252  3.448268
294    294 -8.485402  3.431854
295    295 -7.671346  2.468091
296    296 -7.777773  2.411247

[297 rows x 3 columns]
```

We did not manage to find the precise meaning of the coordinates above. We suppose that they come from an orthogonal projection onto a transverse plane, but we do not know what absolute positions values mean. However, we decided to use these positions for visualization and for some explorations of the role of distance in space.

The table of edges was already in a convenient form for importation:

In [ ]:
```
# load edges.csv into a pandas DataFrame
df = pd.read_csv(zipfile.open('edges.csv'), index_col= False)

#show data table
print(df)
```

```
      # source  target  value
0            0       1      1
1            0       2      2
2            0       3      1
3            0       4      2
4            0       5      1
...        ...     ...    ...
2354       292      44      1
2355       293      44      1
2356       294      44      1
2357       295     190      1
2358       296     190      1

[2359 rows x 3 columns]
```

We can now build an undirected weighted graph from `nodes_pos.csv` and `edges.csv`. The result is stored into the file `celegans.graphml`, to re-load it easily throughout the notebook. Nodes positions are saved as node attributes.

In [ ]:
```
# import the list of edges from edges.csv
df = pd.read_csv(zipfile.open('edges.csv'))
# save a list of edges in the form: (source, target, weight)
edges_list = list(zip(df[df.columns[0]], df[df.columns[1]], df[df.c

# recover an undirected weighted graph from edges list
G = nx.Graph()
G.add_weighted_edges_from(edges_list)

# re-load positions in space, as trated above, and assign them as n
df = pd.read_csv('nodes_pos.csv', index_col= False)
xpos = dict(zip(df['label'], df['xpos']))
ypos = dict(zip(df['label'], df['ypos']))
nx.set_node_attributes(G, xpos, 'xpos')
nx.set_node_attributes(G, ypos, 'ypos')

# save the deriving graph in .graphml format
nx.write_graphml(G, "celegans.graphml")
```

We now verify that this graph is connected, i.e. that the number of connected components is 1.

```
In [ ]: # import the undirected graph
        G = nx.read_graphml('celegans.graphml')
        # display the number of connected components
        print(f'The undirected graph has {nx.number_connected_components(G)
```

```
The undirected graph has 1 connected component(s)
```

The same importation is now repeated for the directed case, where we also take care of which of the two nodes was the source and which was the target. The resulting graph is stored as `celegans_directed.graphml` .

```
In [ ]: # import the list of edges from edges.csv
        df = pd.read_csv(zipfile.open('edges.csv'))
        # save a list of edges in the form: (source, target, weight)
        edges_list = list(zip(df[df.columns[0]], df[df.columns[1]], df[df.c

        # recover an undirected weighted graph from edges list
        G = nx.DiGraph()
        G.add_weighted_edges_from(edges_list)

        # assign positions in space as node attributes
        nx.set_node_attributes(G, xpos, 'xpos')
        nx.set_node_attributes(G, ypos, 'ypos')

        # save the deriving graph in .graphml format
        nx.write_graphml(G, "celegans_directed.graphml")
```

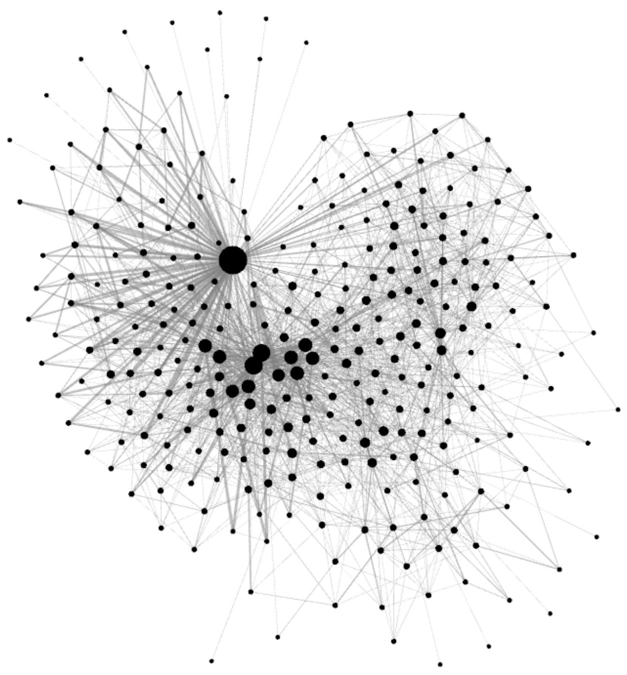
Below, we test if the directed graph is strongly connected. We observe that it is not, having nearly 60 strongly connected components.

```
In [ ]: # import the directed graph
        G = nx.read_graphml('celegans_directed.graphml')
        # display the number of connected components
        print(f'The directed graph has {nx.number_strongly_connected_compon
```

```
The directed graph has 57 connected component(s)
```

To conclude this section, we display a first image of our graph. A qualitative inspection by eye was really important for us to plan our quantitative investigations.

As for all other graph images in this notebook, it was generated via Gephi visualization software. Here, nodes size is proportional to the (unweighted and undirected) degree, as it will happen in the following if not specified. We always use node positions for visualization, but we mention here that Gephi layouts as Force Atlas 2 bring to rather similar graphs.



2 Centralities

2.1 Global coefficients

To begin with, we compute two important global parameters: the (global) clustering coefficient and the average (shortest) path length. Having a real network, we expect clustering coefficient to be high and the average path length to be rather small.

In [ ]:
```
# load the undirected graph
G = nx.read_graphml('celegans.graphml')

# display values for the two global coefficients
print('The clustering coefficient is: {:.3f}'.format(nx.average_clu
print('The average path length is: {:.3f}'.format(nx.average_shorte
```

```
The clustering coefficient is: 0.292
The average path length is: 2.455
```

The clustering coefficient is clearly higher than for a random network, as we will better see in section 3. To evaluate the average path length, we can compare it to $log(n)$, where n is the number of nodes in the graph:

In [ ]:
```
print(np.log(G.number_of_nodes()))
```

```
5.6937321388027
```

It is higher that the average shortest path length. Hence, we can say that C. elegans neural network is a small world, as Watts and Strogatz claimed in [Watts].

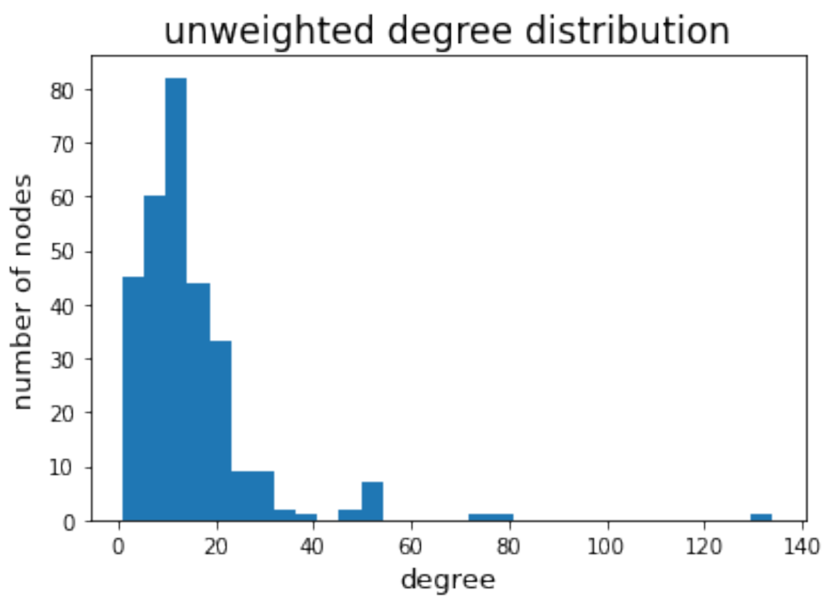
2.2 Undirected centralities

The undirected graph `celegans.graphml` will be used in this section. Next session is devoted to directedness.

In [ ]:
```
# load undirected graph
G = nx.read_graphml('celegans.graphml')
```

We start by plotting the unweighted degrees distribution.

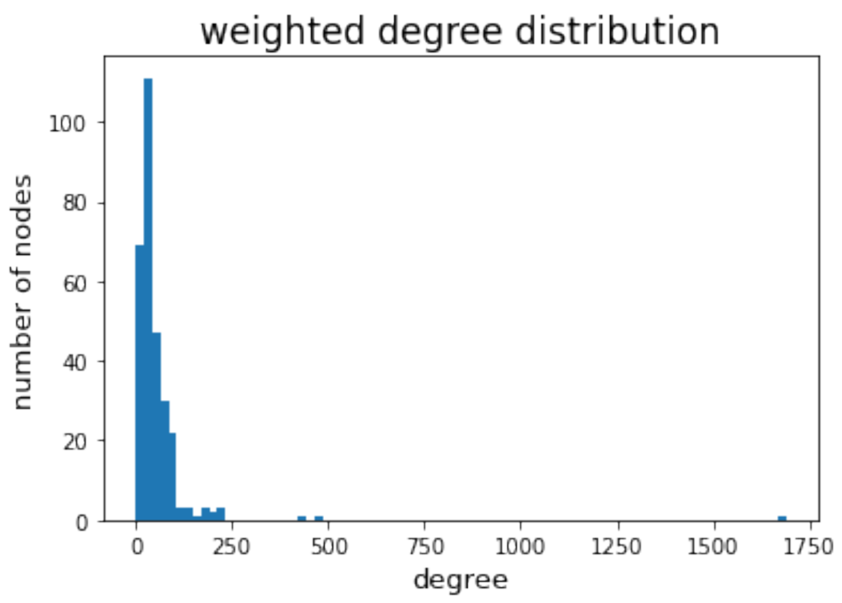
In [ ]:
```
# write a list of node degrees
degrees = [node[1] for node in G.degree()]

# plot the corresponding histogram
plt.hist(degrees, bins= 30)
plt.title("unweighted degree distribution", fontsize= 17)
plt.xlabel("degree", fontsize= 13)
plt.ylabel("number of nodes", fontsize= 13)
plt.show()
```



We can see that the distribution is quite similar to a Poisson curve, for degrees from 0 to 40. After that, the queue presents a dozen of nodes with a very high number of neighbours. This second part of the distributions reminds a power law, as it happens for scale-free networks. At a first sight, C. elegans connectome seems to combine characteristics of Erdős–Rényi model and Albert-Barabasi model. This claim will be further investigated in section 3.

We now take edge weights into account, to see how they modify the distribution above.

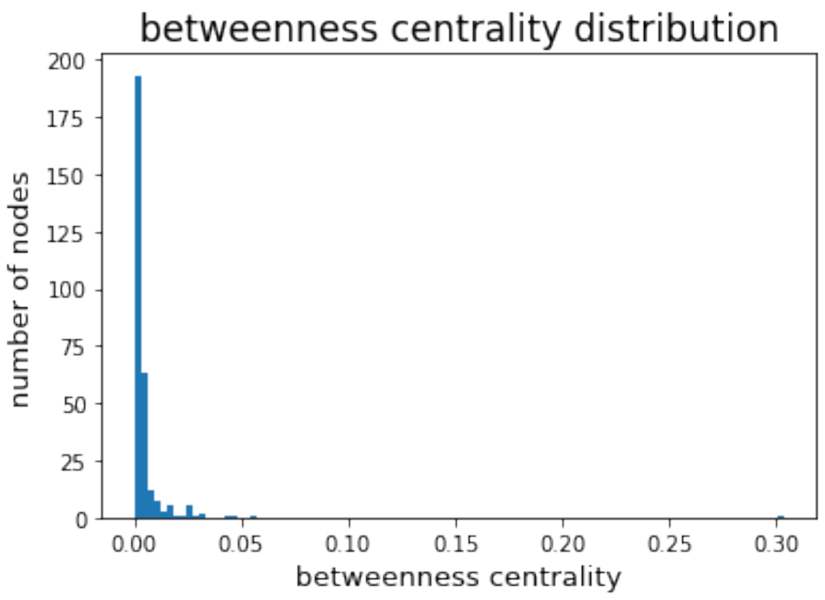
In [ ]:
```
# write a list of weighted node degrees
weighted_degrees = [node[1] for node in G.degree(weight= 'weight')]

# plot the corresponding histogram
plt.hist(weighted_degrees, bins= 80)
plt.title("weighted degree distribution", fontsize= 17)
plt.xlabel("degree", fontsize= 13)
plt.ylabel("number of nodes", fontsize= 13)
plt.show()
```



The influence of edge weights is to amplify the distance of the greatest hubs from the rest of the distributions. This suggests that nodes with higher degrees are more likely to connect with edges of high weight. A visual inspection of the graph seems to corroborate this result.

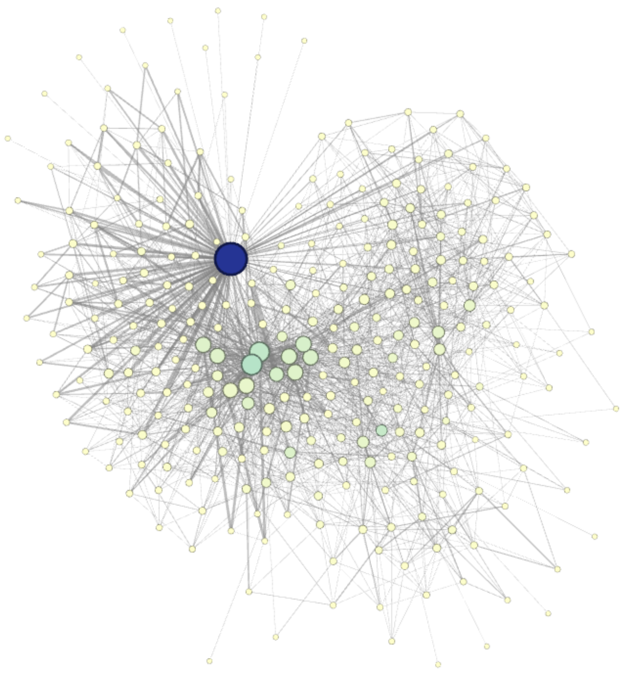
The next centrality that we analyzed is betweenness.

In [ ]:
```
# write a list of betweenness centrality values
betweenness_centrality = list(nx.betweenness_centrality(G).values()

# save betweenness centrality as a node attribute
nx.set_node_attributes(G, nx.betweenness_centrality(G), 'betweennes
# update the .graphml file for visualization
nx.write_graphml(G, 'celegans.graphml')

# plot the histogram for betweenness centrality
plt.hist(betweenness_centrality, bins= 100)
plt.title("betweenness centrality distribution", fontsize= 17)
plt.xlabel("betweenness centrality", fontsize= 13)
plt.ylabel("number of nodes", fontsize= 13)
plt.show()
```



From a Poisson-like distribution, we now shifted towards a monotonically decreasing evolution. This reflects the fact that nodes with very small degree also have a very small betweenness (it is 0 for all nodes with degree of 1).

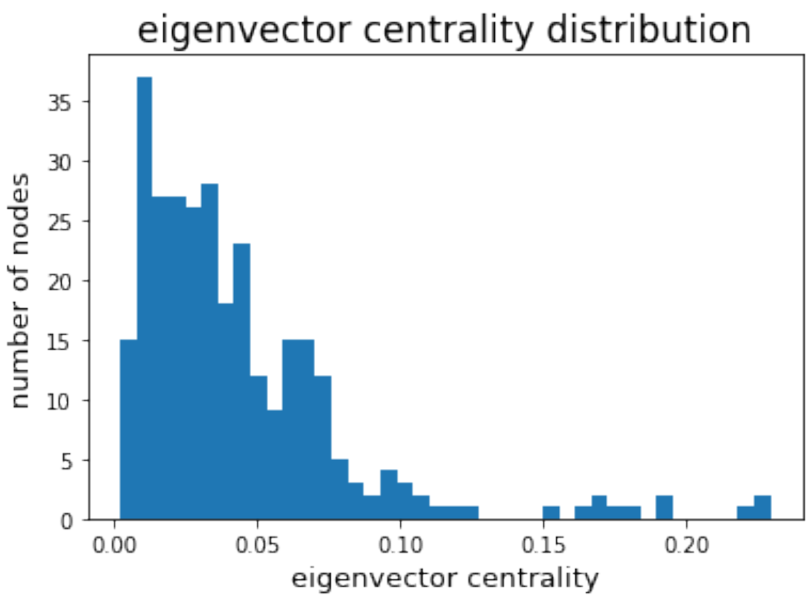
Then, we notice that a single nodes has a value that is orders of magnitude greater than most of the others'. As we immediately understand from the image below, that node is the same super-hub that emerged from the degree distribution.



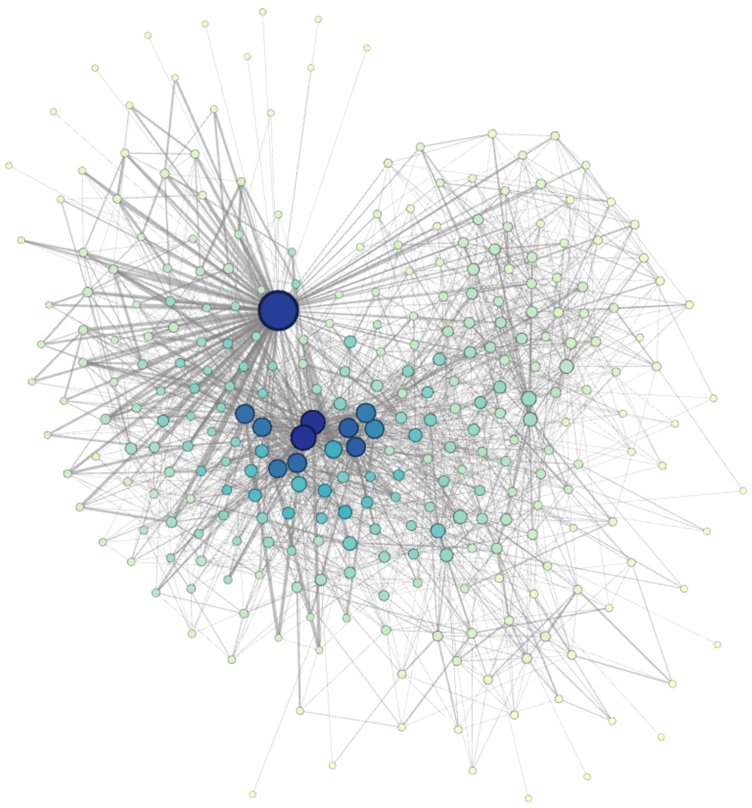
Looking at the picture, where a color gradient for betweenness centrality, it is clear why the super-hub has also a huge value of betweenness. Most of the nodes in his neighbourhood have unitary degree, hence they can only be reached by crossing the super-node.

At this point, we can consider eigenvector centrality.

```
In [ ]: # write a list of eigenvector centrality values
        eigenvector_centrality = list(nx.eigenvector_centrality(G).values()

        # plot the corresponding histogram
        plt.hist(eigenvector_centrality, bins= 40)
        plt.title("eigenvector centrality distribution", fontsize= 17)
        plt.xlabel("eigenvector centrality", fontsize= 13)
        plt.ylabel("number of nodes", fontsize= 13)
        plt.show()

        # save eigenvector centrality as a node attribute
        nx.set_node_attributes(G, nx.eigenvector_centrality(G), 'eigenvecto
        # update the .graphml file for visualization
        nx.write_graphml(G, 'celegans.graphml')
```

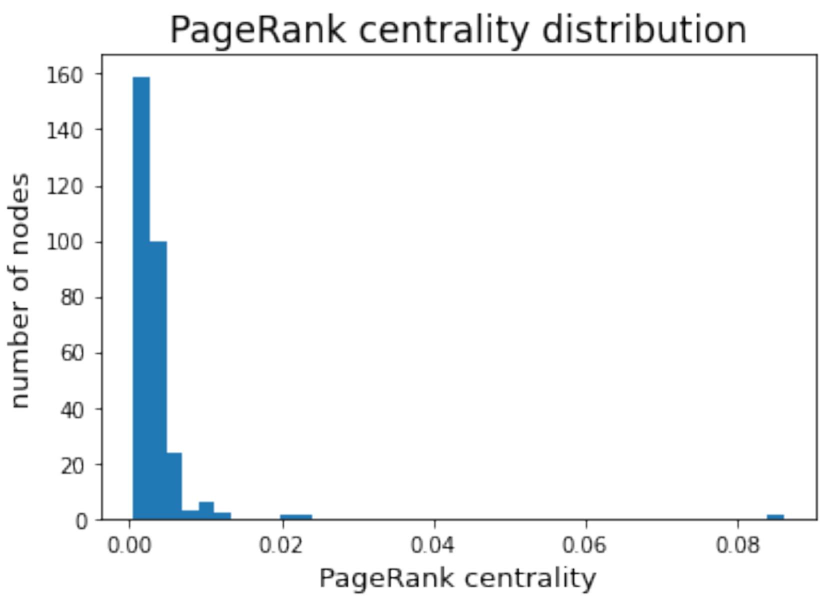


Again, the interpretation of the distribution is easier if we directly visualize the graph with a color partition. We can notice that nodes with higher degree also have higher eigenvector centrality. With respect to the degree distribution, eigenvector centrality one looks more compressed towards the x axis.



We now move to PageRank centrality. As we can see from the distribution below, it gives qualitatively similar scores with respect to betweenness centrality.

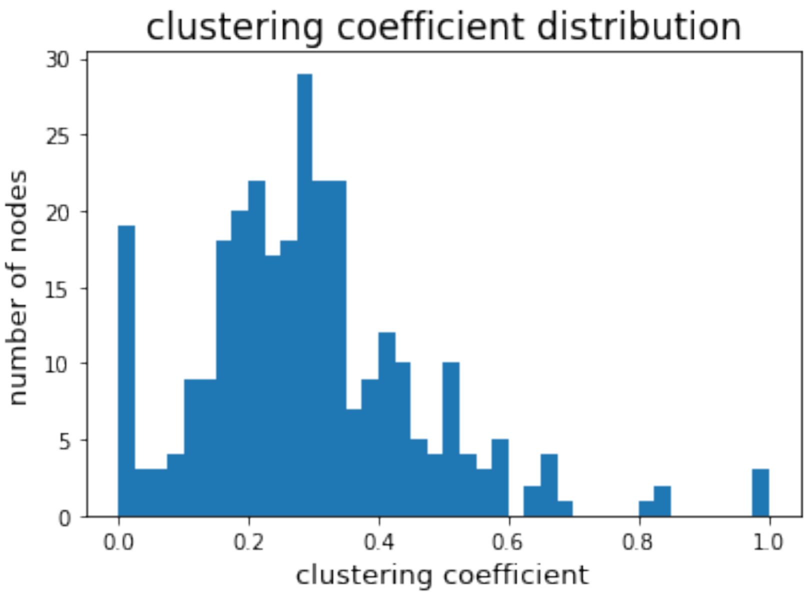
In [ ]:
```
# write a list of PageRank values
pagerank = list(nx.pagerank(G).values())

# plot the corresponding histogram
plt.hist(pagerank, bins= 40)
plt.title("PageRank centrality distribution", fontsize= 17)
plt.xlabel("PageRank centrality", fontsize= 13)
plt.ylabel("number of nodes", fontsize= 13)
plt.show()

# save PageRank as a node attribute
nx.set_node_attributes(G, nx.pagerank(G), 'pagerank')
# update the .graphml file for visualization
nx.write_graphml(G, 'celegans.graphml')
```

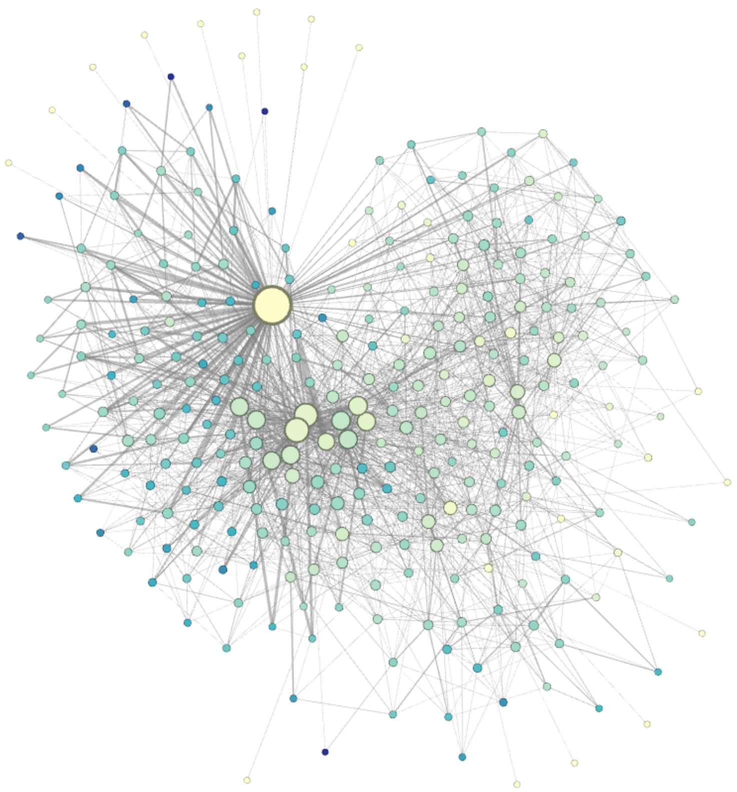


We can now make a comparison of the last three centralities. We can notice that betweenness centrality and PageRank show small scores for all nodes with respect to the largest hub. Eigenvector centrality gives relatively high scores also to the group of hubs towards the centre. Thus, the first two allow to isolate the only super-node, while eigenvector centralities has an opposite effect, bridging the gap between the same super-nodes and the smaller hubs.

The last centrality considered in this section is clustering coefficient.

```
In [ ]: # write a list of clustering coefficient values
        clustering = list(nx.clustering(G).values())

        # plot the corresponding histogram
        plt.hist(clustering, bins= 40)
        plt.title("clustering coefficient distribution", fontsize= 17)
        plt.xlabel("clustering coefficient", fontsize= 13)
        plt.ylabel("number of nodes", fontsize= 13)
        plt.show()

        # save clustering coefficient as a node attribute
        nx.set_node_attributes(G, nx.clustering(G), 'clustering')
        # update the .graphml file for visualization
        nx.write_graphml(G, 'celegans.graphml')
```

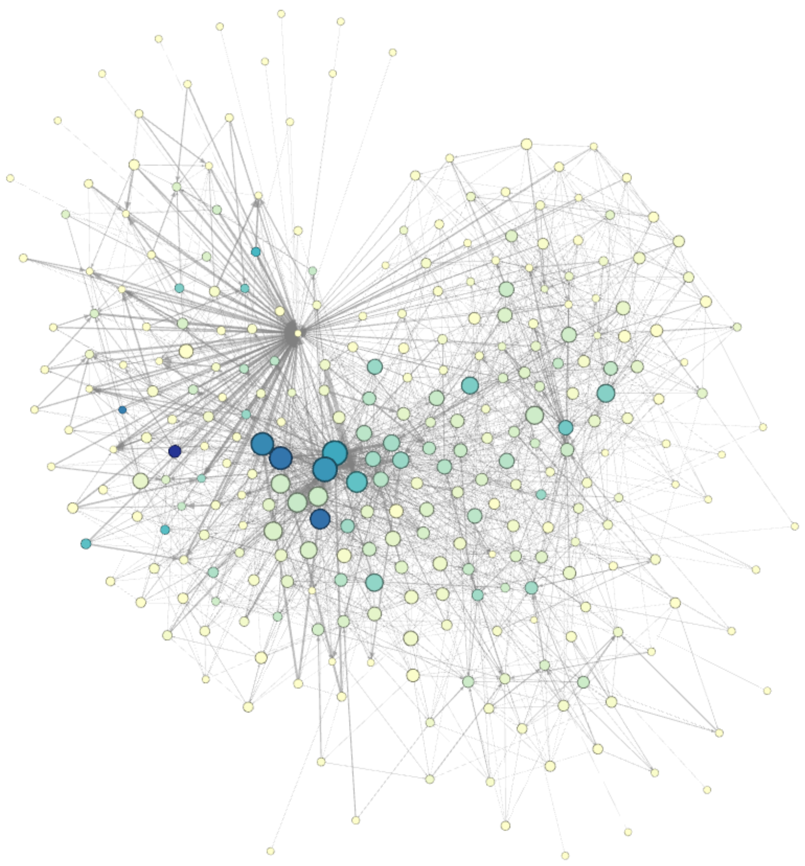




Again, graph visualization tells us much more than the scores distribution. We clearly notice that nodes with a high degree tend to have a very small clustering coefficient. This is a widely common property of real complex networks, e.g. of social networks. We will reconsider this characteristic after, in the section about homophily.

2.3 Influence of directedness

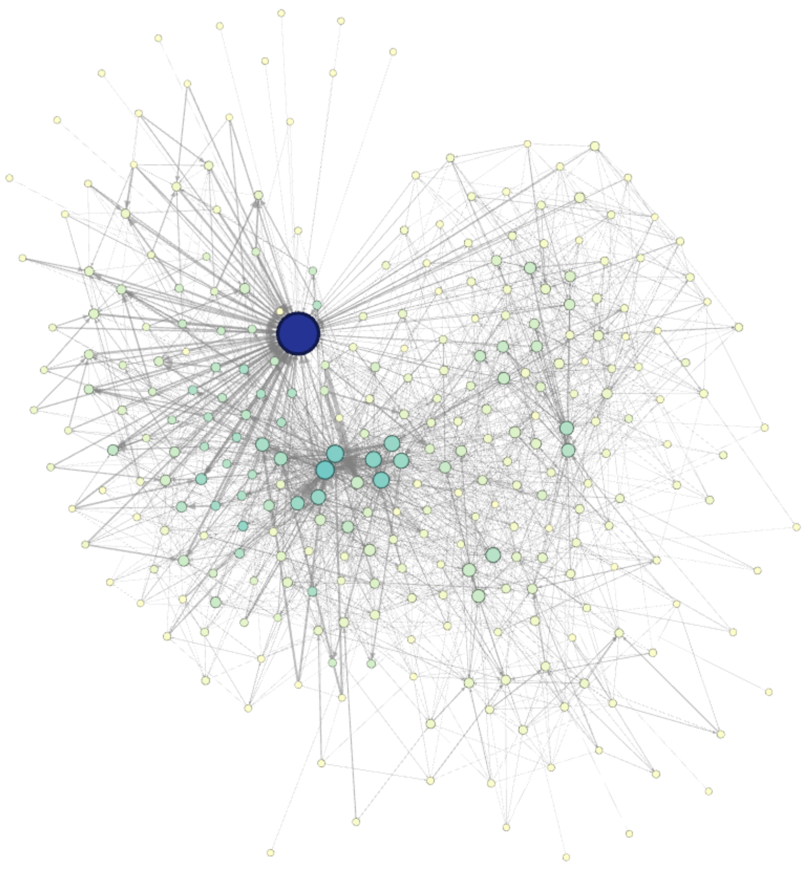
In this section, we briefly discuss how some of the centralities above get modified if we consider edges direction. We begin by showing betweenness centrality directly on the graph, with a color gradient (from ivory to blue). Nodes size is now given by the out-degree.



The first remark concerns the largest hub on the top-left part of the graph. Nearly all its edges are incoming, hence its betweenness centrality becomes very low. Then, we can ask who is the new node with the highest betweenness centrality (in the darkest shade of blue). Surprisingly, we find that it is not among the central group of hubs. Looking at its neighborhood, we discover a crucial property of our network: that node acts as a bridge between the super-hub and the littler hubs at the centre. It is indeed the most important among the very few node which connect the two areas, despite having a rather small degree. It is really important for the rest of the network because there are no direct edges between the super-hub and the littler hubs at the centre.

We will see in section 4 that the two key areas above correspond to two distinct communities. So, the node with the highest directed betweenness centrality seems to play the role of a bridge between the two communities.

The second centrality that we use to investigate the influence of directedness is eigenvector centrality. Again, we directly show the graph visualization, obtained from Gephi. Nodes size is now for the in-degree.



Comparing with the undirected analogue, we notice a stronger convergence toward the largest hub. It is precisely what we expect from the observation that most of its edges are incoming.

2.4 Spatial dependence

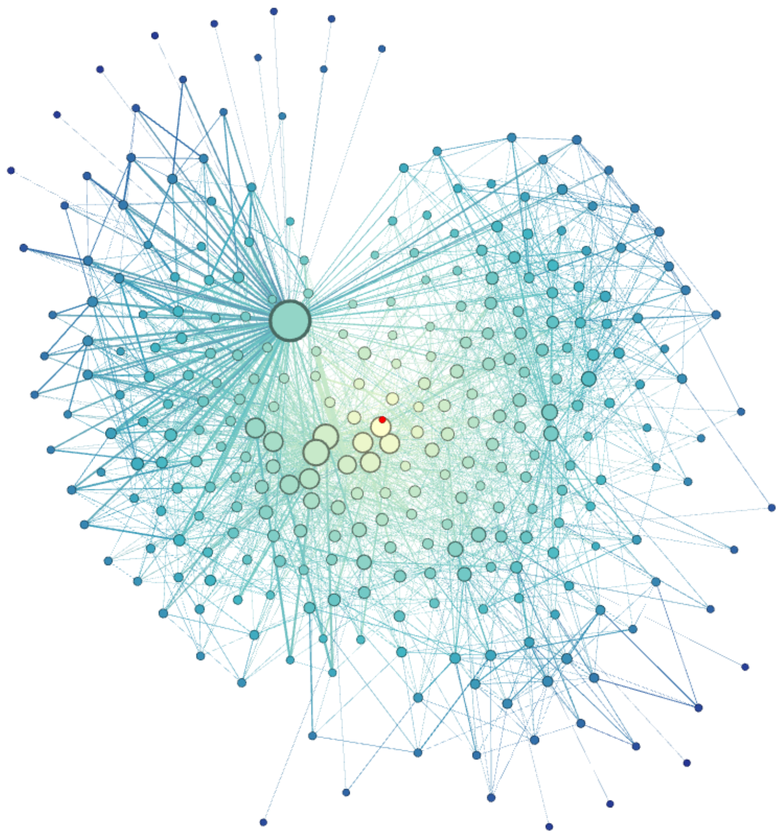
In this section, we will explore the effect of spatial distances on the graph. To begin with, we notice that the group of hubs is more or less at the centre of the graph. Hence, we would like to test a correlation between node degrees and the distance from the centre. Clearly, the super-hub falls out from this supposed correlation, we rather refer to the more homogeneous part of the graph. The idea is that the group of nodes at the centre could have higher degrees simply because, being far from periphery, they have more nodes around.

We first compute distances from positions on x and y. As "graph centre" we considered the centroid of all node positions.

In [ ]:
```
# load positions xpos and ypos from nodes_pos.csv
df = pd.read_csv('nodes_pos.csv', index_col= 'label')

# find graph centre and rescale node positions with respect to it
centre = np.mean(df, axis= 0)
df['xpos'] = df['xpos'] - centre[0]
df['ypos'] = df['ypos'] - centre[1]

# store distances from centre into a list and a dictionary
distances = np.sqrt(df['xpos']**2 + df['ypos']**2)
distances_dict = dict(zip([str(i) for i in range(df.shape[0])], dis
```

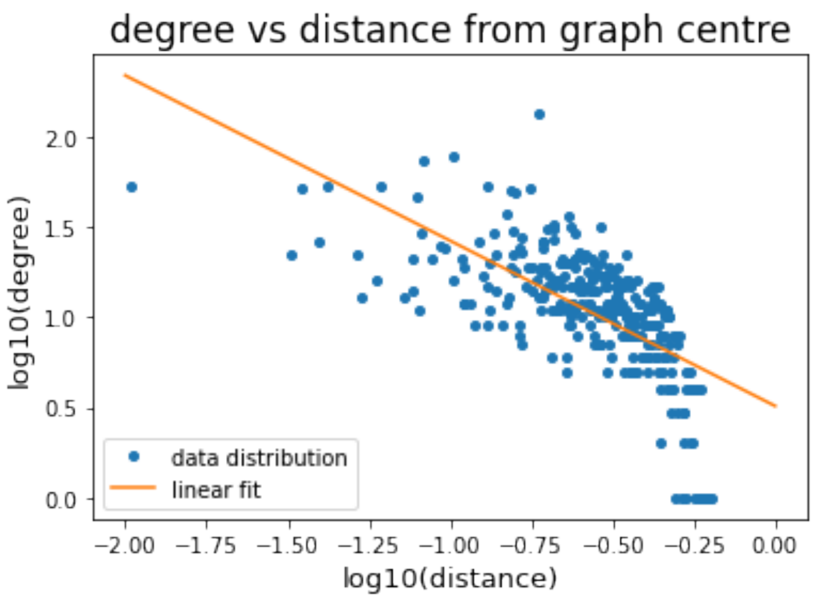
Next, we can visualize distances on the graph. Thus, we added a fictious node to represent the centre (in red in picture below) and displayed a color gradient for increasing distances from it.

In [ ]:
```
# load undirected graph
G = nx.read_graphml('celegans.graphml')

# add a node to represent graph centre, for visualization
G.add_nodes_from([("centre", {'xpos': centre[0], 'ypos': centre[1]}

# store distances as node attributes
distances_dict["centre"] = 0
nx.set_node_attributes(G, distances_dict, "distance")

# save a new graph with the additions above
nx.write_graphml(G, 'celegans_with_centre.graphml')
```



Qualitatively, we see what we mentioned above: node degree decreases from centre to periphery. For a more quantitative analysis, we plot below degree against distance in logarithmic scale.

```
In [ ]: # write lists for distances and degrees
        distance = [distances_dict[str(i)] for i in range(len(distances) -
        degree = [G.degree[str(i)] for i in range(len(distances) - 1)]

        # plot degree vs distance, both as logarithms to base 10
        plt.plot(np.log10(distance), np.log10(degree), 'o', markersize= 4,

        # plot a linear fit of the distribution
        m, b = np.polyfit(np.log10(distance), np.log10(degree), 1)
        plt.plot(np.linspace(-2., 0.), m * np.linspace(-2., 0.) + b, label=
        plt.legend()
        plt.title("degree vs distance from graph centre", fontsize= 17)
        plt.xlabel("log10(distance)", fontsize= 13)
        plt.ylabel("log10(degree)", fontsize= 13)
        plt.show()

        # return the slope of the linear regression
        print("\nLinear regression slope is: ", np.round(m, 3))
```



```
Linear regression slope is:  -0.915
```

We find interesting that the slope of the linear fit is near to -1, suggesting a power law as $\frac{1}{x}$. Clearly, this result has to be taken with a grain of salt, because we are dealing with a rather scattered distributions. Nevertheless, it could be interesting to further investigate on this issue.

Correlation between the two variables is explicitly computed below.

```
In [ ]: # return correlation coefficient for the distribution plotted above
        print("The correlation cofficient is: ", np.round(np.corrcoef(dista
```

```
The correlation cofficient is:  -0.565
```

This suggests that a real correlation exist. We can thus infer that the group of hubs at the centre has a structural origin: high degrees are due mainly to their position in the graph. On the contrary, the largest and isolated hub stands out from this correlation, suggesting that its high degree has a functional origin.

3 Graph Models

In order to characterize the C.Elegans network and identify its main characteristics, we will generate randomized version of the graph. We consider some of the most known models, i.e. the ones that we saw during the lectures. Our goal is to compare the real biological network to its randomized versions. As we explained in section 2.1, relevant parameters for the C.Elegans network are the average path length and the clustering coefficient.

For each model, we fix some starting parameters, and we build the graph through iteration of random operations. At the end we can evaluate if the creation of the network is more ruled by some kind of random phenomena or by some ordered phenomena.

Two useful functions for following analysis are defined below.

In [ ]:
```
# Computing the average degree, average path length and clustering
def sw_param(G):
    path_length= nx.average_shortest_path_length(G)
    CC=nx.average_clustering(G)
    return path_length, CC
```

In [ ]:
```
def degree_distribution(G):
    degree=dict(G.degree())
    degree_dist=list(degree.values())
    H=np.histogram(degree_dist, bins=25)
    return H[1][1:], H[0], degree_dist
```

ER random graph

The first model analysed is the Erdős–Rényi model. In this case we chose a **number of nodes** equal to the original graph, and progressively add edge between two nodes with **probability p**, equal to the density of the original graph.

The density is equal to the fraction of edge over all possible edge of the graph:

$d = \frac{2m}{n(n-1)}$ for undirected graph, where m the number of edges and n the number of nodes

In [ ]:
```
G = nx.read_graphml('celegans.graphml')
n_conn=nx.algorithms.components.number_connected_components(G)
print('The number of connected component is: {:3d}'.format(n_conn))
# Randomized version of the original graph, with same density and s
N=len(G.nodes())
p=nx.classes.function.density(G)
G_ER=nx.gnp_random_graph(N,p)
```

```
The number of connected component is:   1
```

In [ ]:
```
n_conn=nx.algorithms.components.number_connected_components(G_ER)
print('The number of connected component is: {:3d}'.format(n_conn))

path_length_ER, clustering_c_ER = sw_param(G_ER)
print('The average path length is {:.4f}'.format(path_length_ER))
print('The clustering coefficient is {:.4f}'.format(clustering_c_ER
```

```
The number of connected component is:   1
The average path length is 2.4123
The clustering coefficient is 0.0483
```

Configuration Model

In the configuration model the fixed parameter is the **number of nodes** and the **degree distribution**. Starting from this the graph model is elaborated.

In [ ]:
```
degree_list=G.degree()
w=list(degree_list)
d=[]
for i in range(0, N,1):
    d.append(w[i][1])
G_conf_model=nx.expected_degree_graph(d)
```

In [ ]:
```
n_conn=nx.algorithms.components.number_connected_components(G_conf_
print('The number of connected component is: {:3d}'.format(n_conn))
#Considering the largest connected component of the Graph
G_conf_model=G_conf_model.subgraph(max(nx.connected_components(G_co

path_length_CM, clustering_c_CM = sw_param(G_conf_model)
print('The average path length is {:.3f}'.format(path_length_CM))
print('The clustering coefficient is {:.3f}'.format(clustering_c_CM
```

```
The number of connected component is:  10
The average path length is 2.398
The clustering coefficient is 0.147
```

Barabasi Albert Graph

In the Barabàsi-Albert model we fix the number of nodes and the density for generating a initial random graph with 30 nodes. Later we iterate in order to add 270 nodes one by one, and at each iteration we add 5 random edge with a probability that makes more likely to create a **link with nodes of high degree**, according to the preferential attachement principle.

```
In [ ]: # load the undirected graph
        G = nx.read_graphml('celegans.graphml')

        # create an inital random graph with 30 nodes
        # density is chosen to be the same as for our original graph
        G_BA = nx.gnp_random_graph(30, nx.density(G))

        # add other 270 nodes with preferential attachement
        for i in range(270):
            new_node = len(G_BA)
            degrees = G_BA.degree()
            tot_degree = sum([node[1] for node in degrees])
            prob = [node[1] / tot_degree for node in degrees]
            new_edges = np.random.choice(len(G_BA), size= 5, replace= False
            for edge in new_edges:
                G_BA.add_edge(new_node, edge)
```

```
In [ ]: n_conn=nx.algorithms.components.number_connected_components(G_BA)
        print('The number of connected component is: {:3d}'.format(n_conn))
        #Considering the largest connected component of the Graph
        G_BA=G_BA.subgraph(max( nx.connected_components(G_BA), key=len))
        path_length_BA, clustering_c_BA= sw_param(G_BA)
        print('The average path length is {:.3f}'.format(path_length_BA))
        print('The clustering coefficient is {:.3f}'.format(clustering_c_BA
```

```
The number of connected component is:  10
The average path length is 2.644
The clustering coefficient is 0.074
```

Watts-Strogatz

The last random graph model that we considered is the Watts-Strogatz one. We expect it to predict well graph's clustering coefficient and average path length. In the original version introduced in [Watts], the authors started from a regular lattice and rewired edges with a certain probability, to see when a small world network was obtained. Our idea is to reproduce the same procedure but starting from C. elegans neural network positions. Instead of assigning a certain probability to rewire each edge, we fix the percentage of edges to be rewired.

For the starting regular lattice, we consider two possibilities. The first is to connect all nodes inside of a sphere with a fixed radius, as in the random geometric graph (RGG) model. The second possibility is to connect a fixed number k of the nearest neighbors. We will refer to the latter as the knn lattice.

RGG

We start by building the starting network. The fixed radius to connect nearby nodes was taken to get a number of edges that is close to the original graph one.

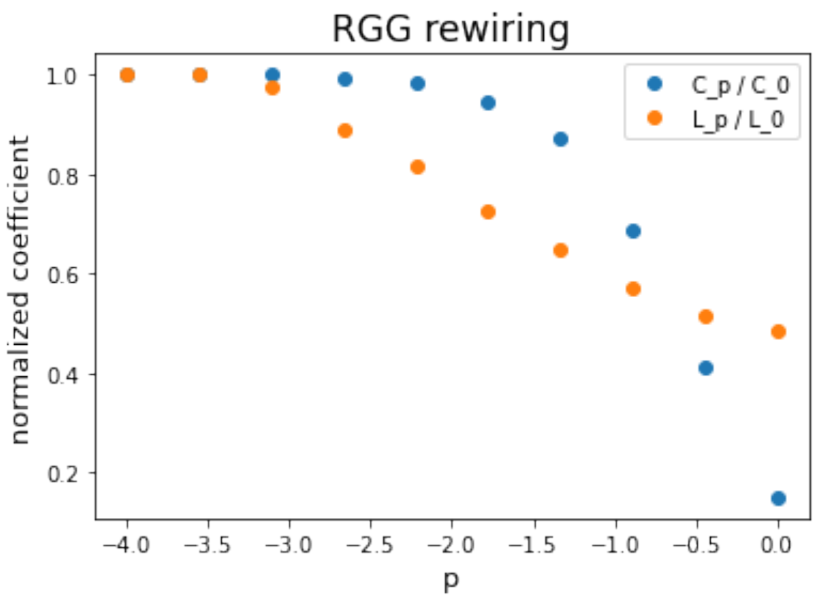
In [ ]:
```
# load nodes positions from nodes_pos.csv and store them in a dicti
df = pd.read_csv('nodes_pos.csv', index_col= False)
positions = dict(zip([i for i in range(df.shape[0])], zip(df['xpos'
# generate the RGG
RGG = nx.random_geometric_graph(df.shape[0], radius= 0.11, pos= pos

# define a new graph to insert RGG edges
G = nx.Graph()
edges_list = list(RGG.edges())
G.add_edges_from(edges_list)

# re-load positions as node attributes
xpos = dict(zip([i for i in range(df.shape[0])], df['xpos']))
ypos = dict(zip([i for i in range(df.shape[0])], df['ypos']))
nx.set_node_attributes(G, xpos, 'xpos')
nx.set_node_attributes(G, ypos, 'ypos')

# store the obtained graph in a new .graphml file
nx.write_graphml(G, "celegans_RGG.graphml")
```

We can now rewire edges, using parameter `p` for the percentage of edges to be rewired. The lines of code below repeat the procedure for logarithmically spaced values for `p`. For each generated graph, the (global) clustering coefficient and the average (shortest) path length are stored into a list.

In [ ]:
```python
# load RGG graph and store the number of edges
G = nx.read_graphml('celegans_RGG.graphml')
m = G.number_of_edges()

# initialize lists to collect clustering coefficient and average pa
C_p = []
L_p = []
# compute values for the starting lattice
C_0 = nx.average_clustering(G)
L_0 = nx.average_shortest_path_length(G)

# write the array of p values to be considered
p_array = np.logspace(-4,0,10)

# repeat the procedure explained above for all values of p
for p in p_array:
    G = nx.read_graphml('celegans_RGG.graphml')
    # the total number of rewirings is p*m
    for i in range(np.int(p*m)):
        edge_index = np.random.choice(G.number_of_edges())
        edge = list(G.edges())[edge_index][0], list(G.edges())[edge_
        G.remove_edge(edge[0], edge[1])
        node = np.random.choice(G.nodes())
        # repeat edge choice in the unlucky case that the chosen ta
        while node in nx.all_neighbors(G, edge[0]):
            node = np.random.choice(G.nodes())
        G.add_edge(edge[0], node)
    # append coefficients to the corresponding lists
    C_p.append(nx.average_clustering(G))
    if nx.is_connected(G) == True:
        L_p.append(nx.average_shortest_path_length(G))
```

We plot the results as it is done in paper [Watts]: clustering coefficient and average path length are displayed against `p`. All values are normalized with respect to the starting regular lattice, whose coefficients are indicated as `C_0` and `L_0`.

```
In [ ]: # plot C_p / C_0 vs p and L_p / L_0 vs p
        plt.plot(np.log10(p_array), np.array(C_p) / C_0, 'o', label= "C_p /
        plt.plot(np.log10(p_array), np.array(L_p) / L_0, 'o', label= "L_p /
        plt.legend()
        plt.title("RGG rewiring", fontsize= 17)
        plt.xlabel("p", fontsize= 13)
        plt.ylabel("normalized coefficient", fontsize= 13)
        plt.show()
```



The small network regime is smaller than for original Watts and Strogatz simulations. This means that, at least from the RGG lattice, we do not easily reach small-world-ness by rewiring stochastically.

K nearest nodes

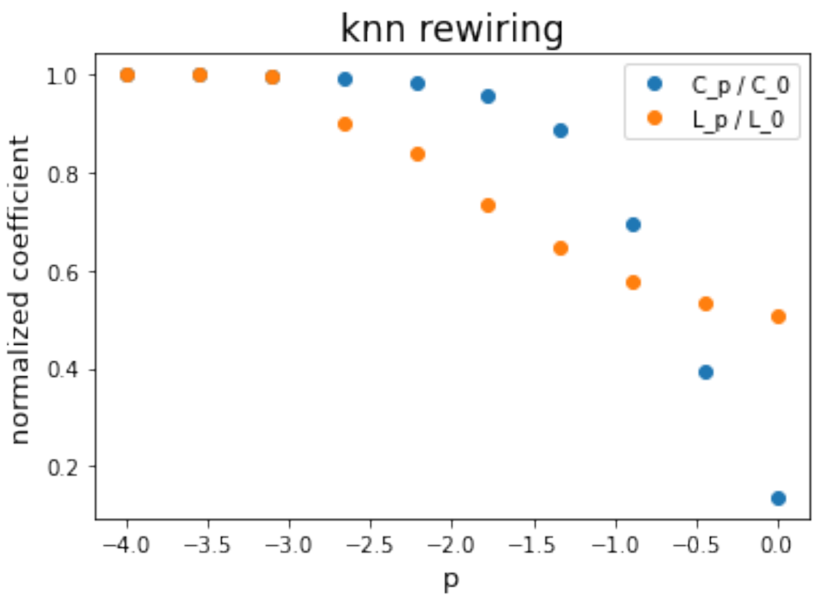
We now move to the second model considered for the generation of regular lattice. Again, the parameter for edges creation was tuned to get almost the same number of edges than in the original graph.

```
In [ ]: # lode node positions and save them into dictionaries
        df = pd.read_csv('nodes_pos.csv', index_col= False)
        xpos = dict(zip([i for i in range(df.shape[0])], df['xpos']))
        ypos = dict(zip([i for i in range(df.shape[0])], df['ypos']))

        # iniziate the list of edges
        edges_list = []
        # the degree k of each node is fixed here
        edges_per_node = 14
        # connect all k nearest nieghbors to each node
        for i in range(df.shape[0]):
            distances = {j: np.sqrt((xpos[j]-xpos[i])**2 + (ypos[j]-ypos[i]
            nearest = sorted(distances.keys(), key = lambda j: distances[j]
            edges_list += zip([i for k in range(edges_per_node)], nearest[1

        # define a new graph to insert knn edges
        G = nx.Graph()
        G.add_edges_from(edges_list)

        # re-load positions as node attributes
        nx.set_node_attributes(G, xpos, 'xpos')
        nx.set_node_attributes(G, ypos, 'ypos')

        # store the obtained graph in a new .graphml file
        nx.write_graphml(G, "celegans_knn.graphml")
```

```
In [ ]: # load knn graph and store the number of edges
        G = nx.read_graphml('celegans_knn.graphml')
        m = G.number_of_edges()

        # initialize lists to collect clustering coefficient and average pa
        C_p = []
        L_p = []
        # compute values for the starting lattice
        C_0 = nx.average_clustering(G)
        L_0 = nx.average_shortest_path_length(G)

        # write the array of p values to be considered
        p_array = np.logspace(-4,0,10)

        # repeat the procedure explained above for all values of p
        for p in p_array:
            G = nx.read_graphml('celegans_knn.graphml')
            # the total number of rewirings is p*m
            for i in range(np.int(p*m)):
                edge_index = np.random.choice(G.number_of_edges())
                edge = list(G.edges())[edge_index][0], list(G.edges())[edge_
                G.remove_edge(edge[0], edge[1])
                node = np.random.choice(G.nodes())
                # repeat edge choice in the unlucky case that the chosen ta
                while node in nx.all_neighbors(G, edge[0]):
                    node = np.random.choice(G.nodes())
                G.add_edge(edge[0], node)
            # append coefficients to the corresponding lists
            C_p.append(nx.average_clustering(G))
            if nx.is_connected(G) == True:
                L_p.append(nx.average_shortest_path_length(G))
```

Again, we plot the results in the same way as in [Watts].

```
In [ ]: # plot C_p / C_0 vs p and L_p / L_0 vs p
        plt.plot(np.log10(p_array), np.array(C_p) / C_0, 'o', label= "C_p /
        plt.plot(np.log10(p_array), np.array(L_p) / L_0, 'o', label= "L_p /
        plt.legend()
        plt.title("knn rewiring", fontsize= 17)
        plt.xlabel("p", fontsize= 13)
        plt.ylabel("normalized coefficient", fontsize= 13)
        plt.show()
```



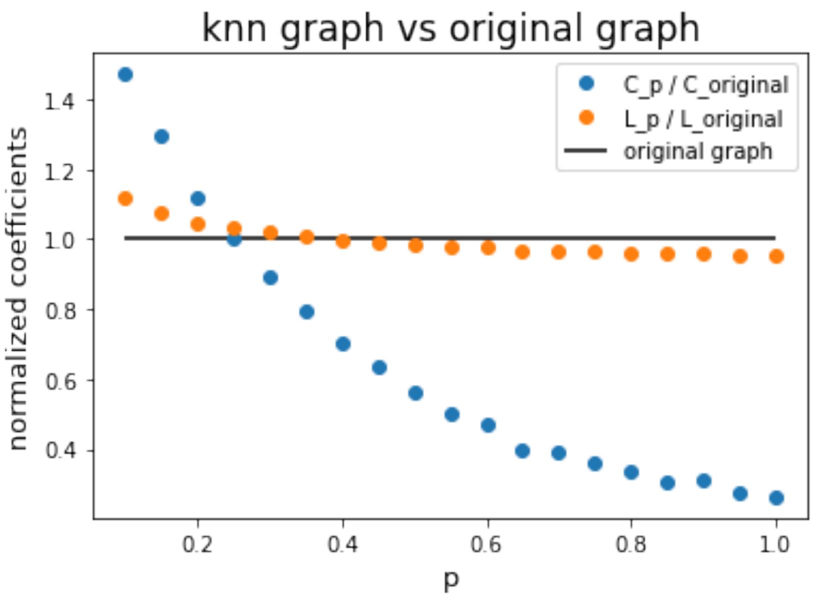
The situation is quite similar to the RGG case, but we can appreciate an earlier decrease for shortest path length. This is what we would like to have for a small network.

Hence, we will go further with our analysis for this second case. In particular, we would like to generate the graph with the closest values for clustering coefficient and average path length to the original graph, in order to make a comparison. To begin with, we repeat the test above for multiple values of parameter p , but linearly spaced for this time.

In [ ]:
```python
# load knn graph and store the number of edges
G = nx.read_graphml('celegans_knn.graphml')
m = G.number_of_edges()

# initialize lists to collect clustering coefficient and average pa
C_p = []
L_p = []
# compute values for the starting lattice
C_0 = nx.average_clustering(G)
L_0 = nx.average_shortest_path_length(G)

# write the array of p values to be considered
p_array = np.linspace(0.1,1,19)

# repeat the procedure explained above for all values of p
for p in p_array:
    G = nx.read_graphml('celegans_knn.graphml')
    # the total number of rewirings is p*m
    for i in range(np.int(p*m)):
        edge_index = np.random.choice(G.number_of_edges())
        edge = list(G.edges())[edge_index][0], list(G.edges())[edge
        G.remove_edge(edge[0], edge[1])
        node = np.random.choice(G.nodes())
        # repeat edge choice in the unlucky case that the chosen ta
        while node in nx.all_neighbors(G, edge[0]):
            node = np.random.choice(G.nodes())
        G.add_edge(edge[0], node)
    # append coefficients to the corresponding lists
    C_p.append(nx.average_clustering(G))
    L_p.append(nx.average_shortest_path_length(G))
```

To directly compare the Watts-Strogatz-like random graphs to the original graph, we now plot the two considered coefficients, normalized with respect to original graph values. Hence, we can evaluate for which value of `p` the two curves are closer to $y = 1$, corresponding to the original graph.

In [ ]:
```
# load undirected original graph
G = nx.read_graphml('celegans.graphml')
# compute the two coefficients
C_original = nx.average_clustering(G)
L_original = nx.average_shortest_path_length(G)

# plot normalized C_p and L_p as explained above
plt.plot(p_array, np.array(C_p) / C_original, 'o', label= "C_p / C_
plt.plot(p_array, np.array(L_p) / L_original, 'o', label= "L_p / L_
plt.hlines(1, p_array[0], p_array[-1], label= "original graph")
plt.legend()
plt.title("knn graph vs original graph", fontsize= 17)
plt.xlabel("p", fontsize= 13)
plt.ylabel("normalized coefficients", fontsize= 13)
plt.show()
```



It seems that the fourth value considered for p gives the best outcome. Converting into the number of edges to be rewired, we get:

In [ ]:
```
# m was the number of edges of the knn graph
p_array[4] * m
```

Out[100]:
```
734.7000000000002
```

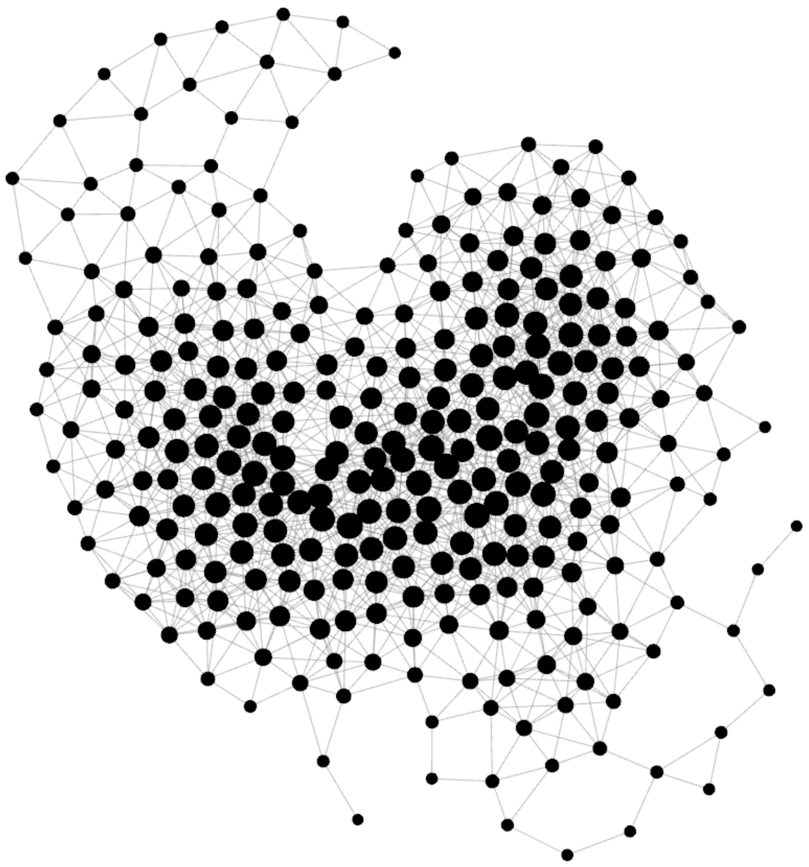
We can thus generate the "optimal" graph, with approximately the same number of rewirings above. the resulting graph is then displayed below, to compare it visually to the real graph.

In [ ]:
```python
# load the knn lattice graph
G = nx.read_graphml('celegans_knn.graphml')

# simulate rewirings
for i in range(730):
    edge_index = np.random.choice(G.number_of_edges())
    edge = list(G.edges())[edge_index][0], list(G.edges())[edge_ind
    G.remove_edge(edge[0], edge[1])
    node = np.random.choice(G.nodes())
    # repeat edge choice in the unlucky case that the chosen target
    while node in nx.all_neighbors(G, edge[0]):
        node = np.random.choice(G.nodes())
    G.add_edge(edge[0], node)

# save resulting graph as a new .graphml file
nx.write_graphml(G, "celegans_knn_rewired.graphml")

# print values for the two important coefficients
print("Clustering coefficient for knn lattice W-S graph is:", np.ro
print("Average path length for knn lattice W-S graph is:", np.round
```

```
Clustering coefficient for knn lattice W-S graph is: 0.265
Average path length for knn lattice W-S graph is: 2.502
```

We immediately notice that, even if the global clustering coefficient and the average path length are similar, this randomized version of the graph is really different, visually. Node degrees distribution is more homogeneous here, starting from a constant value for all nodes.

By comparison, the RGG lattice gives a more realistic degree distribution, as we can see below. But clearly, its average path length is way bigger than in the original graph.



Final comparison

First of all, the Watts-Strogatz graph is loaded, to store its clustering coefficient and average path length.

In [ ]:
```
G_WS= nx.read_graphml('celegans_knn_rewired.graphml')
path_length_WS=nx.average_shortest_path_length(G_WS)
clustering_c_WS=nx.average_clustering(G_WS)
```

Average path length and clustering coefficient

The two key global parameters are compared on the table below, for all the model considered.

```
In [ ]: print('The average path length: \t L \nThe clustering coefficient: '
        print('The Erdős–Rényi model model \n \t\t\t\t\t\t L: {:.3f} \t\t C
        print('\n\nThe Conformational Model \n \t\t\t\t\t\t L: {:.3f} \t\t 
        print('\n\nThe Barabàsi–Albert \n \t\t\t\t\t\t L: {:.3f} \t\t CC: {
        print('\n\nThe Watts–Strogatz Model \n \t\t\t\t\t\t L: {:.3f} \t\t 
        print('\n\nThe Original Graph \n \t\t\t\t\t\t L: {:.3f} \t\t CC: {:
```

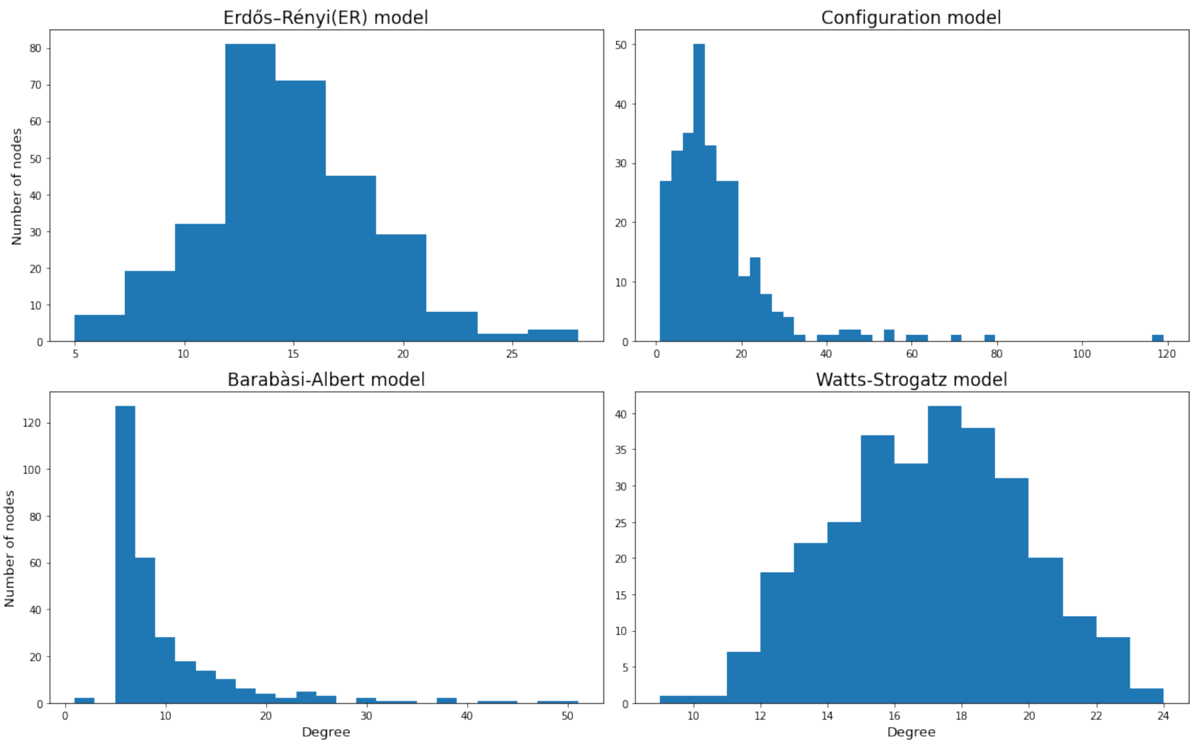
```
The average path length:        L
The clustering coefficient:     CC

The Erdős–Rényi model model
                                                L: 2.412

CC: 0.048


The Conformational Model
                                                L: 2.398

CC: 0.147


The Barabàsi–Albert
                                                L: 2.644

CC: 0.074


The Watts–Strogatz Model
                                                L: 2.502

CC: 0.265


The Original Graph
                                                L: 2.455

CC: 0.292
```

Most of the randomized graphs that we considerd have a low clustering coefficient with respect to the dataset analysed. On the contrary, the low average path length is quite well predicted by all.

The Watts-Strogatz Model (also called small world) is the one that solve this problem, and best reproduce both clustering coefficient and average path length parameters of the original graph. In biological network, the small-worldness suggests local computation of the signal due to its high value of the clustering coefficient, and a rapid communication across the network, due to its low average path length [Jarrell].

Degree distribution

```
In [ ]: G = nx.read_graphml('celegans.graphml')
        X_o, Y_o, degree_dist = degree_distribution(G)
        X_ER, Y_ER , degree_dist_ER= degree_distribution(G_ER)
        X_CM, Y_CM , degree_dist_CM = degree_distribution(G_conf_model)
        X_BA, Y_BA, degree_dist_BA = degree_distribution(G_BA)
        X_WS, Y_WS, degree_dist_WS = degree_distribution(G_WS)

        fig, ax= plt.subplots(nrows=2, ncols=2, figsize=(16,10), sharex=Fal

        ax[0][0].hist(degree_dist_ER)
        ax[0][0].set_title('Erdős-Rényi(ER) model',fontsize=17)
        ax[0][0].set_ylabel('Number of nodes', fontsize=13)

        ax[0][1].hist(degree_dist_CM, bins=45)
        ax[0][1].set_title('Configuration model',fontsize=17)

        ax[1][0].hist(degree_dist_BA, bins=25)
        ax[1][0].set_ylabel('Number of nodes', fontsize=13)
        ax[1][0].set_xlabel('Degree',fontsize=13)
        ax[1][0].set_title('Barabàsi-Albert model', fontsize=17)

        ax[1][1].hist(degree_dist_WS, bins=15)
        ax[1][1].set_xlabel('Degree',fontsize=13)
        ax[1][1].set_title('Watts-Strogatz model',fontsize=17)

        fig.tight_layout()

        plt.show()
```



Nevertheless the Watts-Strogatz model doesn't describe well the degree distribution of the real network. It is a bell-curved function similar to the the random ER model. The Barabàsi-Albert degree distribution follows a power law distribution, and this is what we often observe in biological networks.

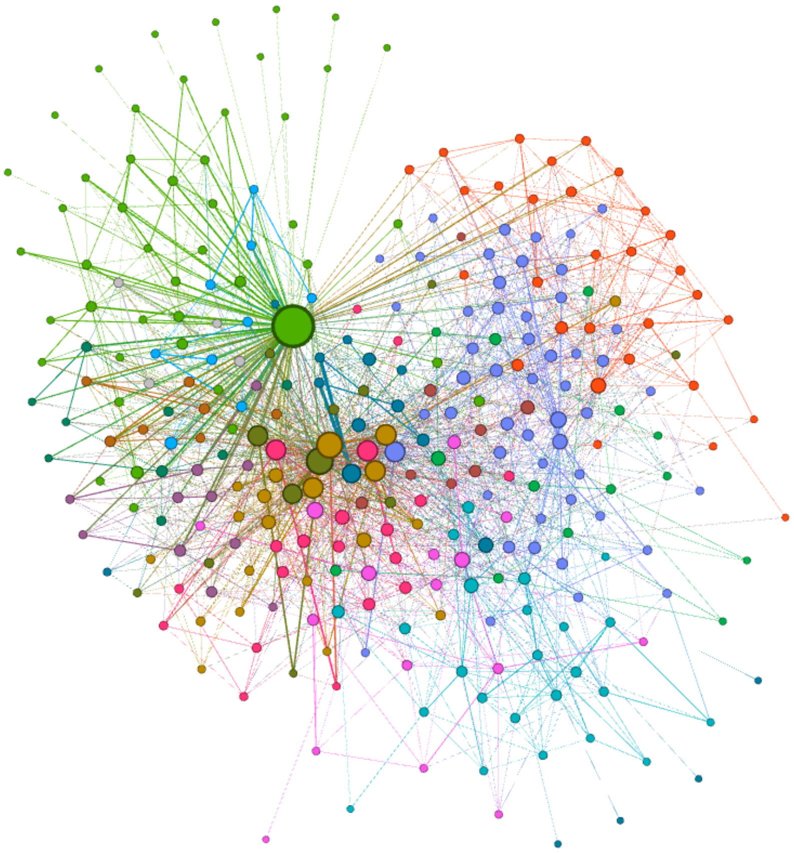
4 Community Detection

In this last section, we apply Louvain algorithm for community detection to our graph. We found interesting to compare the resulting communities for different values of the resolution parameter. The comparison will be qualitative at first sight, than quantitative, by looking at modularity.

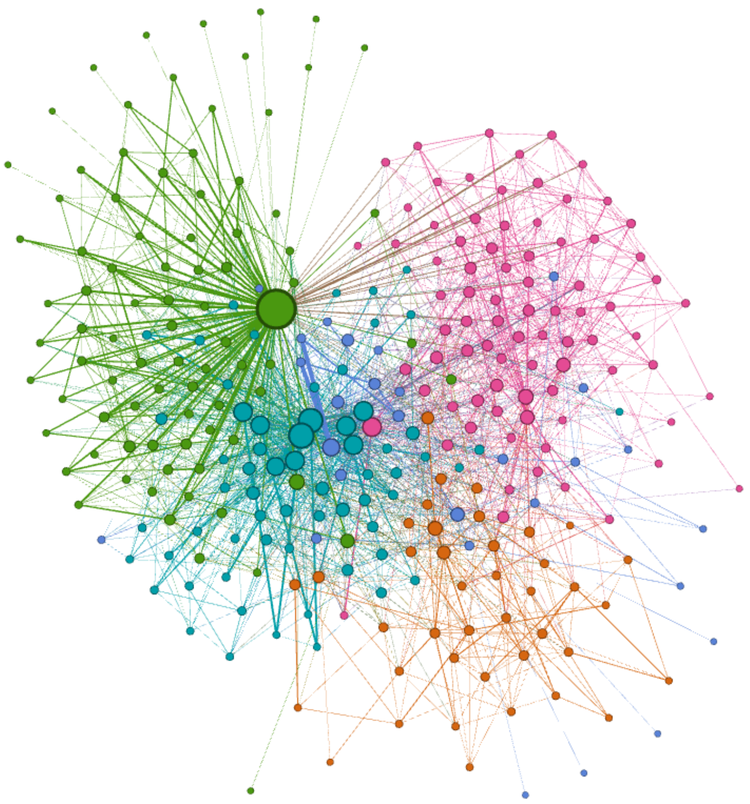
The lines of code below were run multiple times, with different choices for the resolution parameter `res` . The partition into communities is saved as node attributes and the resulting graph is saved as a new dedicated `.graphml` file.

In [ ]:
```
# load the undirected graph
G = nx.read_graphml('celegans.graphml')

# resolution can be chosen below to tune communities size
res = 1
coms = algorithms.louvain(G, resolution= res)

# community belonging is stored in the graph as a node attribute
nx.set_node_attributes(G, coms.to_node_community_map(), name= "comm
for node in G:
    G.nodes[node]["community"] = G.nodes[node]["community"][0]

# the new graph is saved in a new .graphml file
nx.write_graphml(G, "celegans_comms_res" + str(res) + ".graphml")
```

We now display graphs with a color coding for communities, again via Gephi software. The first one below is for `res = 0.5` . At this communities size, we have a rather useless classification.

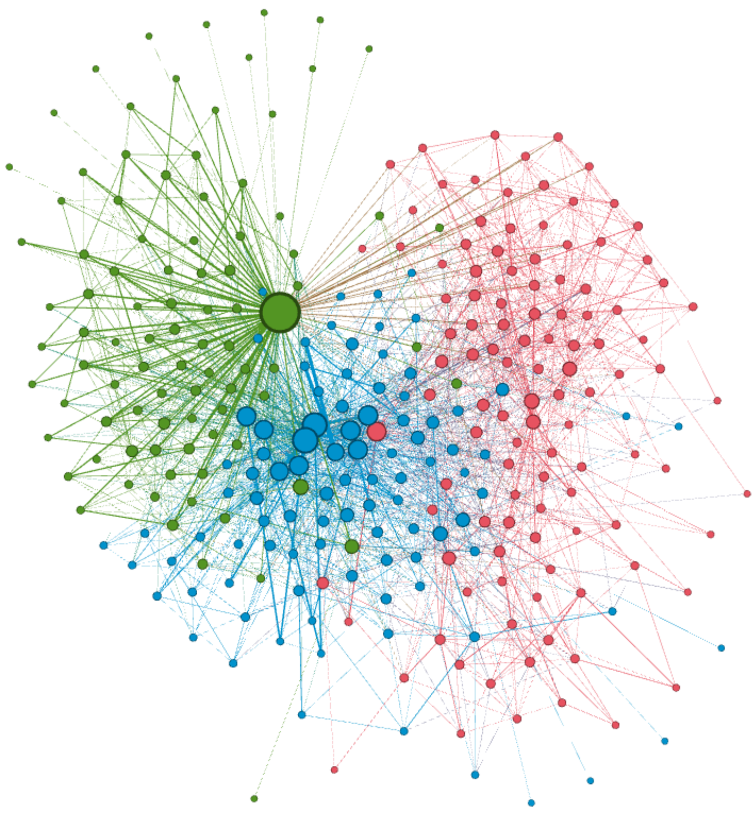


Going up to `res = 1`, we begin to see a spatial based clustering. Interestingly, the green community accounts for most of the super-hub neighborhood, while the light blue community correspond to the group of littler hubs towards the centre. To better visualize how the communities evolve with respect to resolution, we kept colors for largest communities as near as possible from one picture to the other.

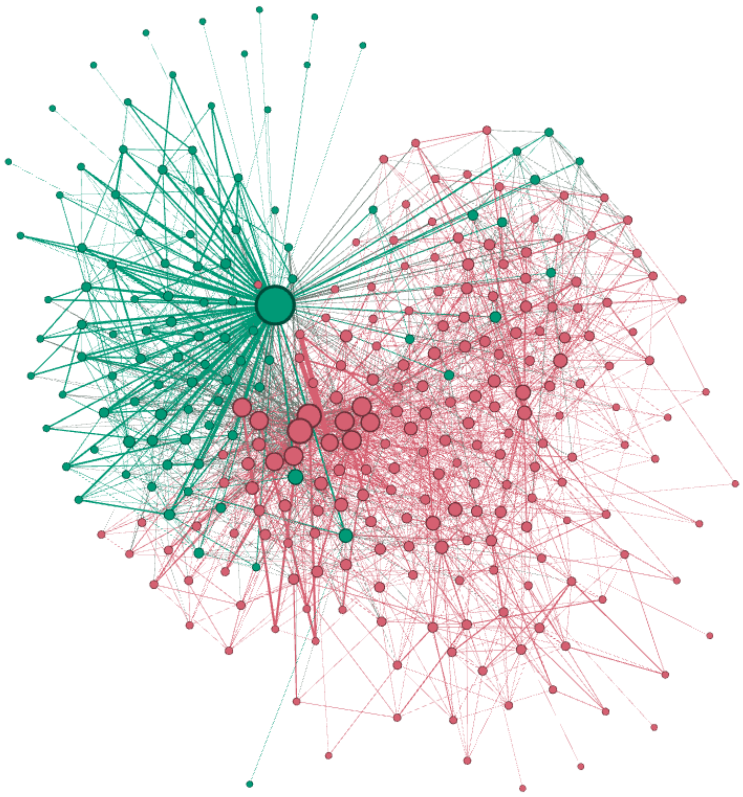


At `res = 2` we definitely observe a rather meaningful clustering. The two communities above for the largest hubs (in green) and the central group of hubs (in light blue) did not vary much. Most of the nodes outside this two communities grouped to form a unique peripheral community (in pink). This trifold clustering could reflect a functional subdivision of the roles: a huge neurons collects from a hundred of nodes and send this information to the group of littler hubs at the centre; that group of hubs, probably interneurons, forwards the signal to the peripheral area, corresponding more or less to the third community detected.

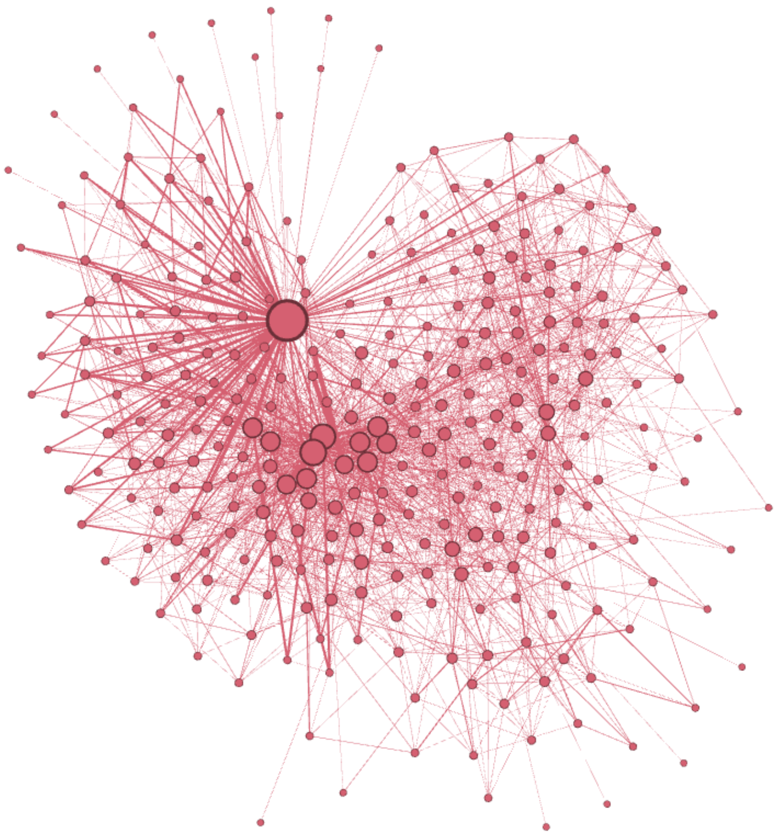
As we already mentioned in section 2, this interpretation is supported by our observations on edges directions and nodes neighborhoods.



At `res = 3` , there are two communities left. The green one corresponds now to the whole neighborhood of the largest hub, more or less. All the rest belong to the pink community.



From `res = 4` we get the trivial case of a unique community, i.e. without any clustering.



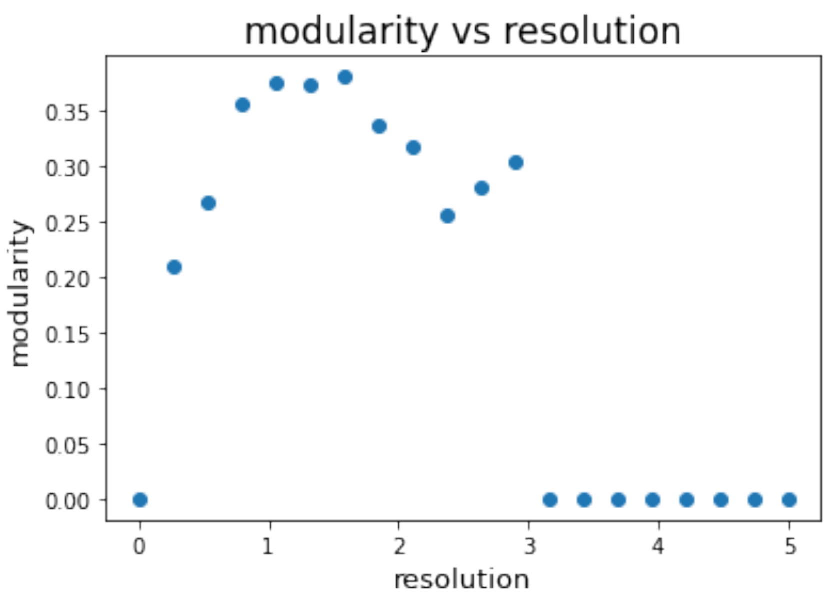
At this point we can compare our qualitative inspection with modularity score. Our eye suggests that more meaningful partitions should be around `res = 2`. We will see if modularity curve is in agreement with our eyes or not.

The function `cdlib.evaluation.erdos_renyi_modularity` was used to compute modularity scores.

```
In [ ]: # initialize lists for resolution and modularity
res_vec = []
mod_vec = []

# compute modularity for linearly spaced values for res
for res in np.linspace(0, 5, 20):
    coms = algorithms.louvain(G, resolution= res)
    mod = coms.erdos_renyi_modularity().score
    res_vec.append(res)
    mod_vec.append(mod)
```

Below, modularity is plotted against resolution parameter. We can see that, according to modularity, best partitions are obtained between `res = 1` and `res = 1.5`. It is close to what we claimed by eye. Interestingly, the modularity rises just before `res = 3`, i.e. when we find a nice subdivision for only two communities. After, modularity drops to zero because there is only one single community left.

```
In [ ]: # plot modularity against resolution
plt.plot(res_vec, mod_vec, 'o')
plt.title("modularity vs resolution", fontsize= 17)
plt.xlabel("resolution", fontsize= 13)
plt.ylabel("modularity", fontsize= 13)
plt.show()
```



5 Homophily

Homophily, also referred to as assortativity, is the tendency of having a higher number of edges between similar nodes. To quantify similarity, different choices are possible. Usually, node degree is considered, but we could choose every possible node attribute. For the case of our graph, as anticipated in section 2, it is interesting to test degree homophily for our graph.

In order to evaluate globally the degree homophily of the graph , we used the formula defined in [Newman]:

$$r = \frac{\sum_i e_{ii} - \sum_i a_i^2}{1 - \sum_i a_i^2}$$

where e_{ii} is the fraction of edges between nodes with attribute value i and a_i is the fraction of edges that have at least one end with the same attribute value i. Below, we compute degree assortativity for our graph.

```
In [ ]: # load the undirected graph
G = nx.read_graphml('celegans.graphml')

# compute degree assortativity
r=nx.degree_assortativity_coefficient(G)
print('The degree assortativity coefficient is: {:.3f}'.format(r))
```

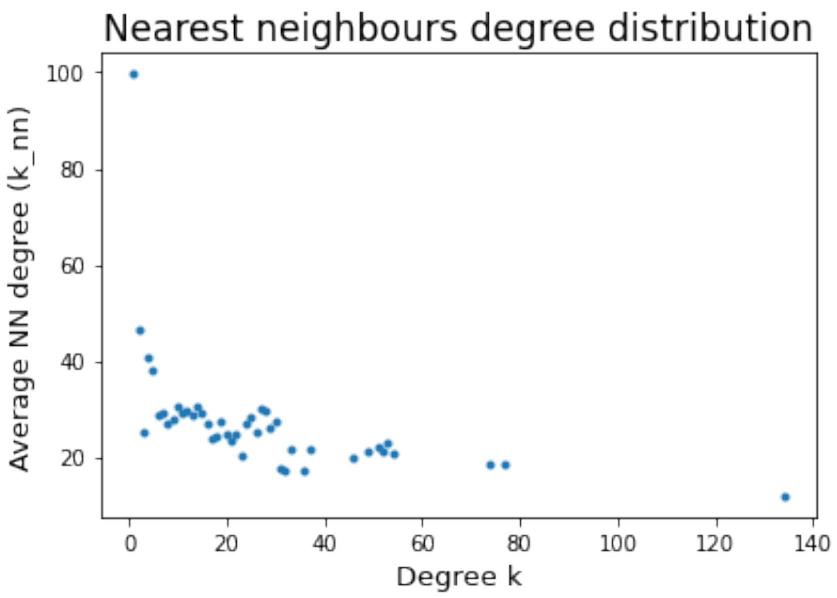
```
The degree assortativity coefficient is: -0.163
```

The coefficient of assortativity is smaller than zero, meaning that the graph globally shows dis-assortativity. As we expected, nodes tends to have edges with nodes of different degree value.

Average nearest neighbors degree k_{nn}

Now we study the curve of $k_{nn}(k)$, the average degree connectivity, that is the average nearest neighbors (NN) degree for nodes of degree k.

```
In [ ]: # compute average degree connectivity
        degree_connectivity=nx.average_degree_connectivity(G)

        # plot its distribution
        plt.plot(list(degree_connectivity.keys()), list(degree_connectivity
        plt.title("Nearest neighbours degree distribution", fontsize= 17)
        plt.xlabel('Degree k', fontsize=13)
        plt.ylabel('Average NN degree (k_nn)', fontsize=13)
        plt.show()
```



We observe that the function $k_{nn}(k)$ decreases with respect to k. Thus, nodes with small degree tend to be connected to nodes with high degree. As we already noticed in the previous sections, hubs tend to have few reciprocal connections. Again, the result is coherent what we expect for a biological neural network.

6. Conclusions

Many tools were used along this notebook to analyze the neural network of the C. elegans. As stated in the introduction, the goal was to understand it as much as possible, using the notions that we acquired during lessons. Some results that we got can be applied to many real networks, while some others were specific of C. elegans connectome. The major example for the first category is the small-world-ness, that is the fact of having large clustering coefficient and small average path length.

Then, with node properties as degrees, betweenness centrality and eigenvector centrality, we quantitatively explored some specificities of our graph. We discovered an inhomogeneous distribution of the three, with few nodes dominating the distributions. This often happens when a signal has to spread rapidly, but the number of edges has to be kept low. This brings to the development of few very large hubs to which many peripheral nodes of small degree are connected. We explored this degree disassortativity with a dedicated section, showing that degree and average degree of the neighbors are indeed inversely correlated.

We noticed that hubs are not homogeneously distributed in space. On the contrary, there is a node with a very large degree (around 100) in a rather peripheral region. This fact, together with the fact that nearly all its edges are incoming, suggested us that it should have a precise functional role. Probably, it receives information from a lot of directly connected sensorial neurons.

All other nodes with a high degree are concentrated toward the centre of the graph. This made as think that their high degree could simply derive from their position in space: being at the centre, they find more nodes to connect to. Combining what we found for community detection and directed centralities distributions, we assigned the role of interneurons to this central group: they connect the largest hub to the opposite periphery of the graph. Interestingly, the super-node and the central group were in two distinct communities until high values for Louvain algorithm resolution parameter. Indeed, there are no direct edges between the super-hub and the littler hubs, but communication between the two has to go from little bridge-nodes, well enlightened by directed betweenness centrality.

We compared the most common graph models, to see which one corresponded better to C.elegans connectome. We found that many of them captured some of its properties, but none of them went really close to the original graph. For example, node degree distribution was found to be a combination of a Poisson curve and a power law, mixing what we have for an Erdös–Rényi random network and for a scale-free network, respectively. Clustering coefficient and average path length were both well reproduced by a Watts-Strogatz model graph that we built. In turn, the degree distributions becomes completely different from the original graph one.

7. Future Perspectives

We think to have grasped many properties of the neural network that we analyzed. To verify that our claims make sense, a more detailed data for the nodes will be needed. In particular, it would be interesting to make a bijection between communities or single nodes and the corresponding biological function. We predicted the roles of some neurons that we considered the most relevant, and we expect them to also be considered the most important by biologists.

Only some of the simplest centralities and graph models were considered. The analysis in this notebook could does be enriched by applying more complex tools, as long as the limited number of nodes allows to do that. For instance, we did not explore much the influence of edge weights and directionality. A deeper study of these characteristics could bring to new relevant considerations. Spreading on the graph could also be studied. Having a small world network, we expect diffusion to be very fast, as we wish for a neural network.

Spatial influence could also be further investigated, for instance with a gravity model or a radiation law. However, it would be important to know the exact meaning of the 2D positions before, that is we should know the embedding procedure.

8. References

[Jarrel] Travis A. Jarrell, Yi Wang, Adam E. Bloniarz, Christopher A. Brittin, MengXu, J. Nichol Thomson, Donna G. Albertson, David H. Hall, and Scott W.Emmons. "The connectome of a decision-making neural network". Science, 337(6093):437–444 (2012).

[Newman] Newman, M. E. J. "Mixing patterns in networks". Phys. Rev. E, 67:026126, (2003).

[Watts] Watts, D., Strogatz, S. "Collective dynamics of ‘small-world’ networks". Nature 393, 440–442 (1998).

[Képès] Képès, F. "Biological networks". World Scientific, 2007.

The Wikipedia page: https://en.wikipedia.org/wiki/Caenorhabditis_elegans (https://en.wikipedia.org/wiki/Caenorhabditis_elegans) was our main reference for general consideration in the Introduction.