Graph Theory

Armande CIROT, Titouan POQUILLON, Jiri RUZICKA

January 20, 2020

1 Introduction

Saccharomyces cerevisiae is a yeast, a small single-cell eukaryote, known for its fermentation abilities. It has been used for winemaking, baking, and brewing since first emerging civilizations. Thanks to its characteristics, *Saccharomyces cerevisiae* is examined as a convenient model organism.

We are going to study its gene regulatory network using graph theory and python **networkx** library.

2 Exploration and characterization of the gene regulatory network

[1]:
```
import networkx as ntx
import numpy as np
import pandas as pd
import seaborn as sns
import matplotlib.pyplot as plt
from networkx.drawing.nx_agraph import graphviz_layout
import pygraphviz as pgv
ntx.__version__
```

[1]:
```
'2.3'
```

The file *GRN_edges_S_cerevisiae.txt* contains the edges between transcription factors and target genes. We will use it to create our graph. We drop the first column which has no further interest.

[2]:
```
network = pd.read_csv("GRN_edges_S_cerevisiae.txt", sep = ',', header=0)
gene = pd.read_table("net4_gene_ids.tsv", sep = '\t')
transcription= pd.read_table('net4_transcription_factors.tsv')
mapping = pd.read_table('go_slim_mapping.tab.txt', header = None)
network = network.drop('Unnamed: 0', axis=1)
```

We print every dataframe to see its content. *network* contains the relations between transcription factors and target genes, *gene* the name of genes within their ids, *transcription* the list of transcription factors. In *mapping*, we'll use the gene names (first column) and the Gene Ontology annotation (sixth column).

[3]:
```
network.head()
```

[57]:
```
gene = gene.set_index('ID')
gene.head()
```

[8]:
```
transcription.head()
```

[69]:
```
mapping.head()
```

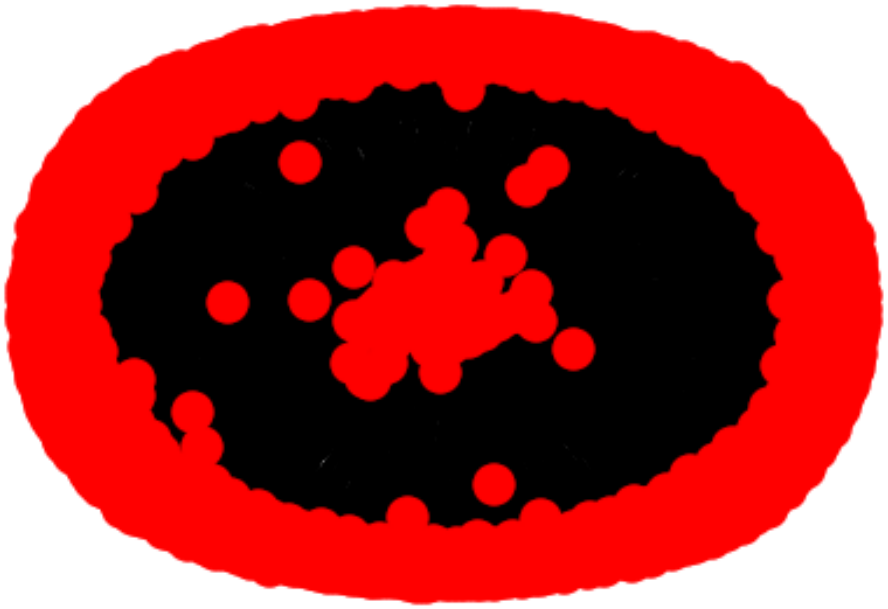
We create our directed graph. It contains 1994 nodes and 3940 edges with average in and out degree of 1.9759.

[3]:
```
G = ntx.DiGraph()
G = ntx.from_pandas_edgelist(network, 'transcription_factor', 'target_gene',␣
 ↪create_using = ntx.DiGraph())
ntx.info(G)
```

[3]:
```
'Name: \nType: DiGraph\nNumber of nodes: 1994\nNumber of edges: 3940\nAverage in
degree:   1.9759\nAverage out degree:   1.9759'
```

2.1 Plotting the network

[8]:
```
ntx.draw(G)
```

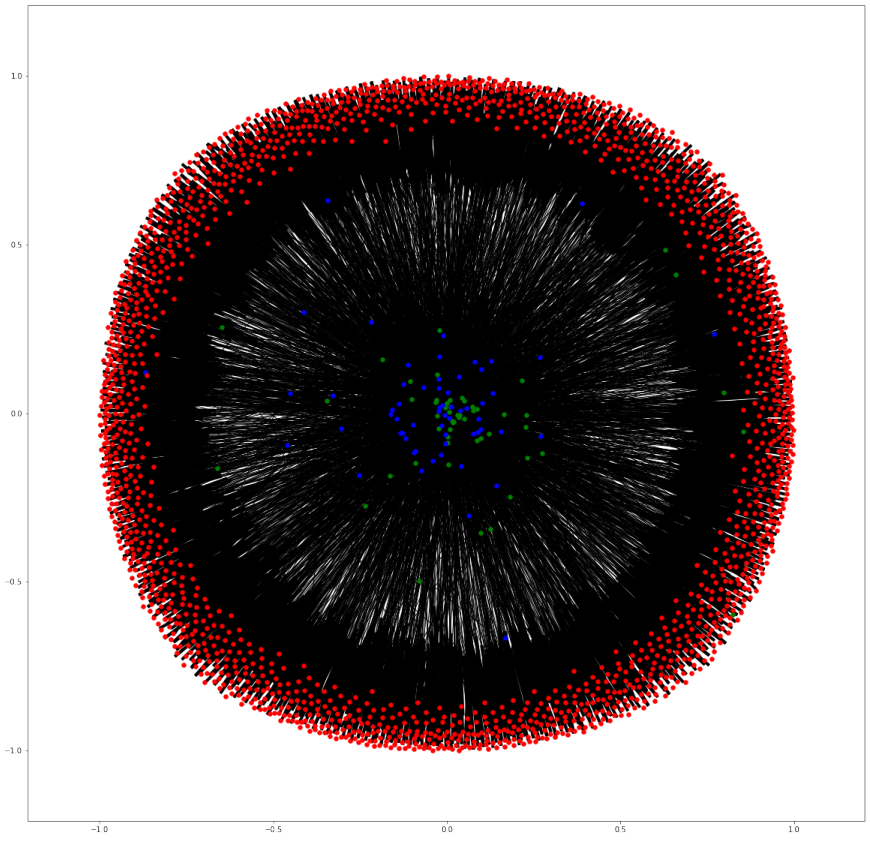


We draw the network using *draw()* method. This image is not really clear. In order to get a better representation of the graph, we have to find another method.

[9]:
```
both_genes = set(network['transcription_factor']).
 ↪intersection(network['target_gene'])
regulatory_genes = set(network['transcription_factor']) - set(both_genes)
target_genes = set(network['target_gene']) - set(both_genes)
```

[10]:
```
plt.figure(figsize=(20,20))
ntx.draw_networkx_edges(G, pos = ntx.spring_layout(G), edgelist = list(G.
 ↪edges()))
ntx.draw_networkx_nodes(G, nodelist= list(both_genes),
                        node_color = 'g',  pos = ntx.spring_layout(G),␣
 ↪with_labels = False, node_size = 30)
ntx.draw_networkx_nodes(G, nodelist= list(regulatory_genes),
                        node_color = 'b',  pos = ntx.spring_layout(G),␣
 ↪with_labels = False, node_size = 30)
ntx.draw_networkx_nodes(G, nodelist=list(target_genes) ,
                        node_color = 'r',  pos = ntx.spring_layout(G),␣
 ↪with_labels = False, node_size = 30)
```

[10]:
```
<matplotlib.collections.PathCollection at 0x7f91f8503310>
```

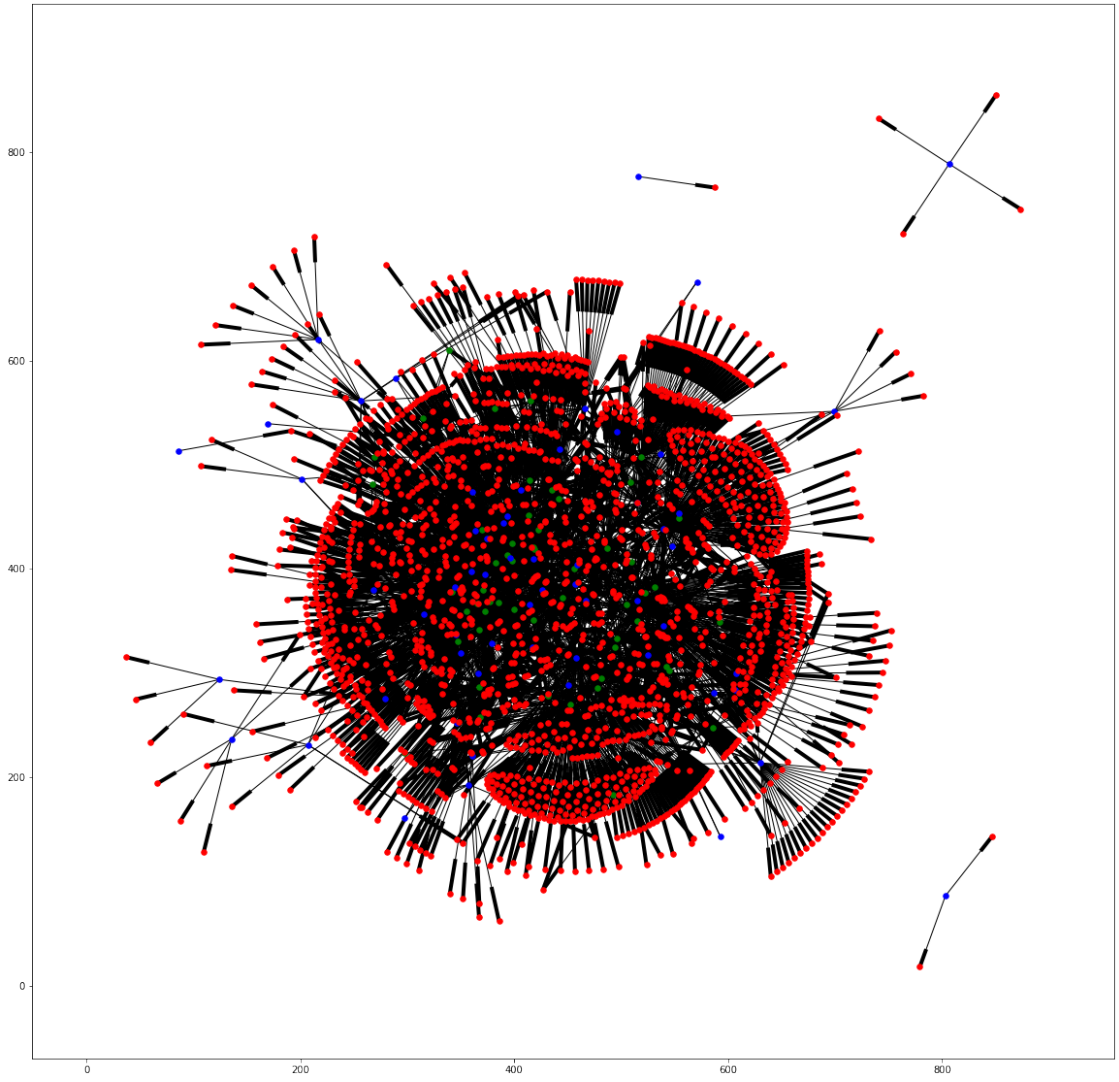


We can see that a central cluster of regulatory and both (target and regulatory) genes plays the main role in the network. Target genes form a red circle around this cluster, beeing at the end of the regulation cycle.

Even though this image, created using *ntx.spring_layout*, gives us a better idea about the disposition of factors and genes in the graph, we would like to get a shape that specifies if there are some particular parts in the graph. For that, we'll use the *nx_agraph.graphviz_layout* method.

[11]:
```
plt.figure(figsize=(20,20))
ntx.draw_networkx_edges(G, pos = ntx.nx_agraph.graphviz_layout(G), edgelist =␣
 ↪list(G.edges()))
ntx.draw_networkx_nodes(G, nodelist= list(both_genes), node_color = 'g', \
                        pos = ntx.nx_agraph.graphviz_layout(G), with_labels =␣
 ↪False, node_size = 30)
ntx.draw_networkx_nodes(G, nodelist= list(regulatory_genes), node_color = 'b',  \
                        pos = ntx.nx_agraph.graphviz_layout(G), with_labels =␣
 ↪False, node_size = 30)
ntx.draw_networkx_nodes(G, nodelist=list(target_genes) , node_color = 'r', \
                        pos = ntx.nx_agraph.graphviz_layout(G), with_labels =␣
 ↪False, node_size = 30)
```

[11]:
```
<matplotlib.collections.PathCollection at 0x7f91f87ced50>
```



This representation shows the major links between the target genes only (red), transcription factors only (blue) and the genes that accomplish both roles (green).

We see the structure of the network and distinguish the kinds of nodes. The dendogram hierarchy is also clear from this figure as we are able to see some levels of the regulatory tree. One factor often regulates a whole bunch of genes. In few cases, this relation is unconnected to the rest of the graph.

We define a new method which allows us to use simply *nx_agraph.graphviz_layout()* and distinguish certain nodes.

[12]:
```
def draw_agraph(G, nodes):
    plt.figure(figsize=(20,20))
    col = ['#084594', '#2171b5', '#4292c6', '#6baed6', '#9ecae1', '#c6dbef',␣
 ↪'#deebf7', '#f7fbff']
```

```
    ntx.draw_networkx_edges(G, pos = ntx.nx_agraph.graphviz_layout(G), edgelist␣
↪= list(G.edges()))
    for i in range(len(nodes)):
        ntx.draw_networkx_nodes(G, pos = ntx.nx_agraph.graphviz_layout(G), \
                                nodelist = nodes[i], node_color = col[i], \
                                with_labels = False, node_size = 30)
```

2.2 Network metrics

Network metrics help us characterize the main features of the graph.

Clustering coefficient To calculate the clustering coefficient of a node, we use the following formula :

$$C = \frac{\text{number of closed triplets}}{\text{number of all triplets (open and closed)}}$$

[13]:
```
clust_coef = ntx.average_clustering(G.to_undirected())
clust_coef
```

[13]:
```
0.04436303023690503
```

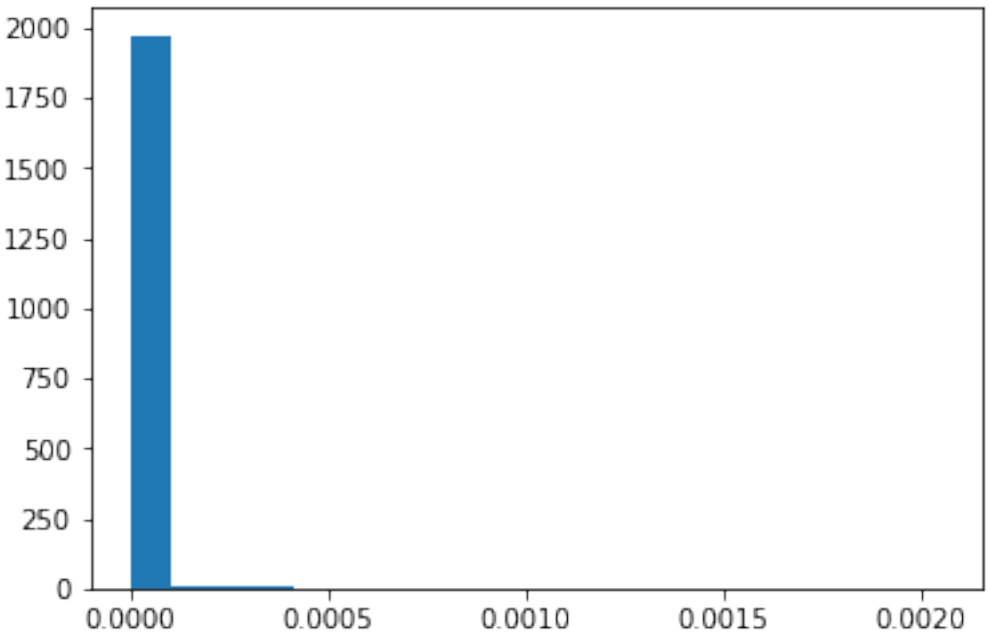
The clustering coefficient measures the degree to which nodes in a graph tend to cluster together. In this graph, it is very low : around 0.044. The nodes are not very connected to each other to make clusters. This value matches the figure of our graph. We can see most of the nodes are leaves.

The brute method to calculate the clustering coefficient is for every node to find the neighboroud, and to find every neighbor's neighbor, to see if bucle up in a triangle. The complexity of this algorithm would be of $O(n^3)$.

Betweenness centrality

[26]:
```
btw_ctr = ntx.betweenness_centrality(G)
plt.hist(btw_ctr.values(), bins=20)
```

[26]:
```
(array([1.973e+03, 4.000e+00, 4.000e+00, 3.000e+00, 2.000e+00, 2.000e+00,
        2.000e+00, 1.000e+00, 0.000e+00, 0.000e+00, 0.000e+00, 0.000e+00,
        0.000e+00, 0.000e+00, 1.000e+00, 1.000e+00, 0.000e+00, 0.000e+00,
        0.000e+00, 1.000e+00]),
 <a list of 20 Patch objects>)
```



Betweeness centrality is defined as the number of shortest paths, among any pair of nodes, passing by a given vertex (node).

We can see that in our case it is often very low. Majority of nodes (cca 1973 of 1994) figure in a very small number of shortest paths (0 or 1).

According to the networkx documentation, the betweenness centrality complexity of the python algorithm is $O(nm + n^2 log(n))$

(https://www.tandfonline.com/doi/abs/10.1080/0022250X.2001.9990249).

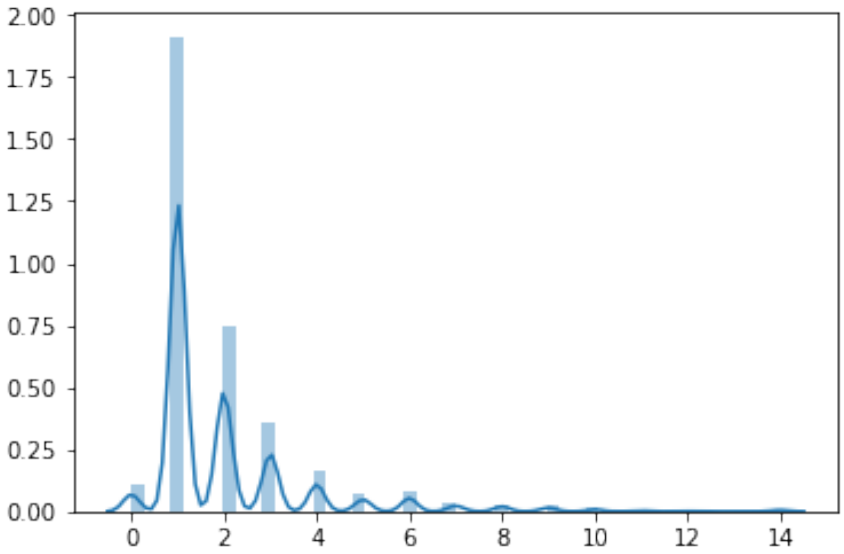
Degrees in and out

[35]:
```
adj_G = ntx.to_numpy_matrix(G, nodelist=G.nodes())
adjacency = pd.DataFrame(adj_G)
in_degree = adjacency.sum(axis=0).sort_values()
out_degree = adjacency.sum(axis=1).sort_values()
```

[37]:
```
print(max(in_degree))
sns.distplot(in_degree)
```

```
14.0
```

[37]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f91f4635a10>
```

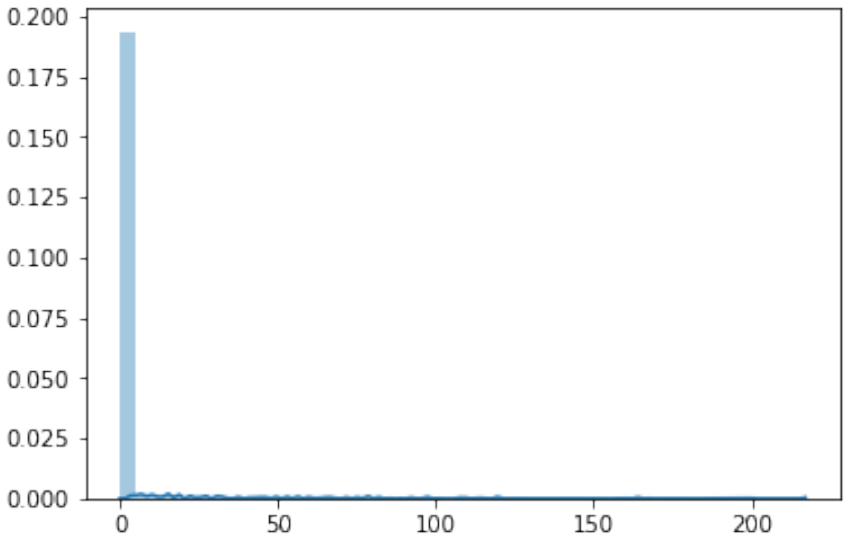


The histogram of in-degree seems to follow the Chi-squared distribution. Most of the nodes have only one incoming edge (final point of the regulatory network). Quantity of nodes having others incoming edges decrease with the maximum of 14.

[39]:
```
print(max(out_degree))
sns.distplot(out_degree)
```

```
217.0
```

[39]: <matplotlib.axes._subplots.AxesSubplot at 0x7f91f42f1210>



As expected, majority of the network has no out-coming edge (corresponds to the target genes only). The regulation is thus compised in a small comunity of genes and factors (this was also clear in the plots). Maximum of the out-degree is 217, so there is a gene (factor) regulating 217 other genes!

The complexity of this algorithm is O(M), the number of edges. Indeed, to solve this algorithm, we start with null degree for every node. We have to cover every edge of the graph and for each one upgrade the degree of the each node (0(1)).

Density $D = \frac{\text{nb of edges}}{\text{max nb of edges}} = \frac{E}{N(N-1)}$

[69]:
```
ntx.density(G)
```

[69]:
```
0.0009914339103612895
```

As we can see, our graph is not dense at all, this is due to the vast majority of target genes that have only one incoming edge.

Because we have direct access to both the number of nodes and the number of edges, we have a complexity of O(1).

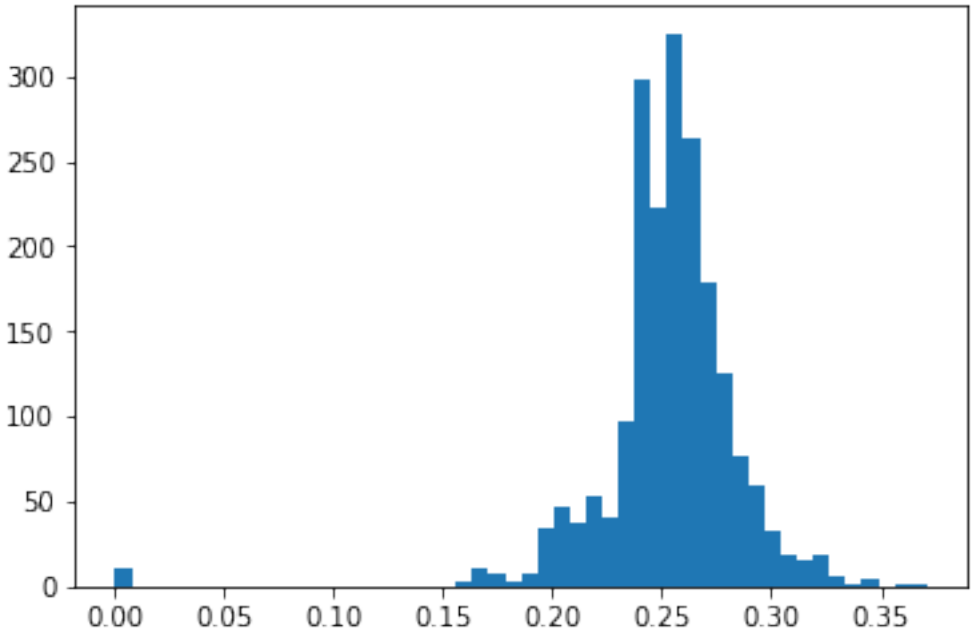
Closeness centrality $H(X) = \Sigma_{x \neq y} \frac{1}{d(y,x)}$

It is, for directed graphs, the sum of the reciprocal length of the shortest paths between the node and all other nodes in the graph.

To compute the closeness centrality of a node, we just have to run Djikstra's algorithm once to find the distance from a node to all other nodes. Thus, its complexity is O(E + Nln(N)).

[71]:
```
cc_cnt = ntx.closeness_centrality(G.to_undirected())
plt.hist(cc_cnt.values(),bins = 50)
```

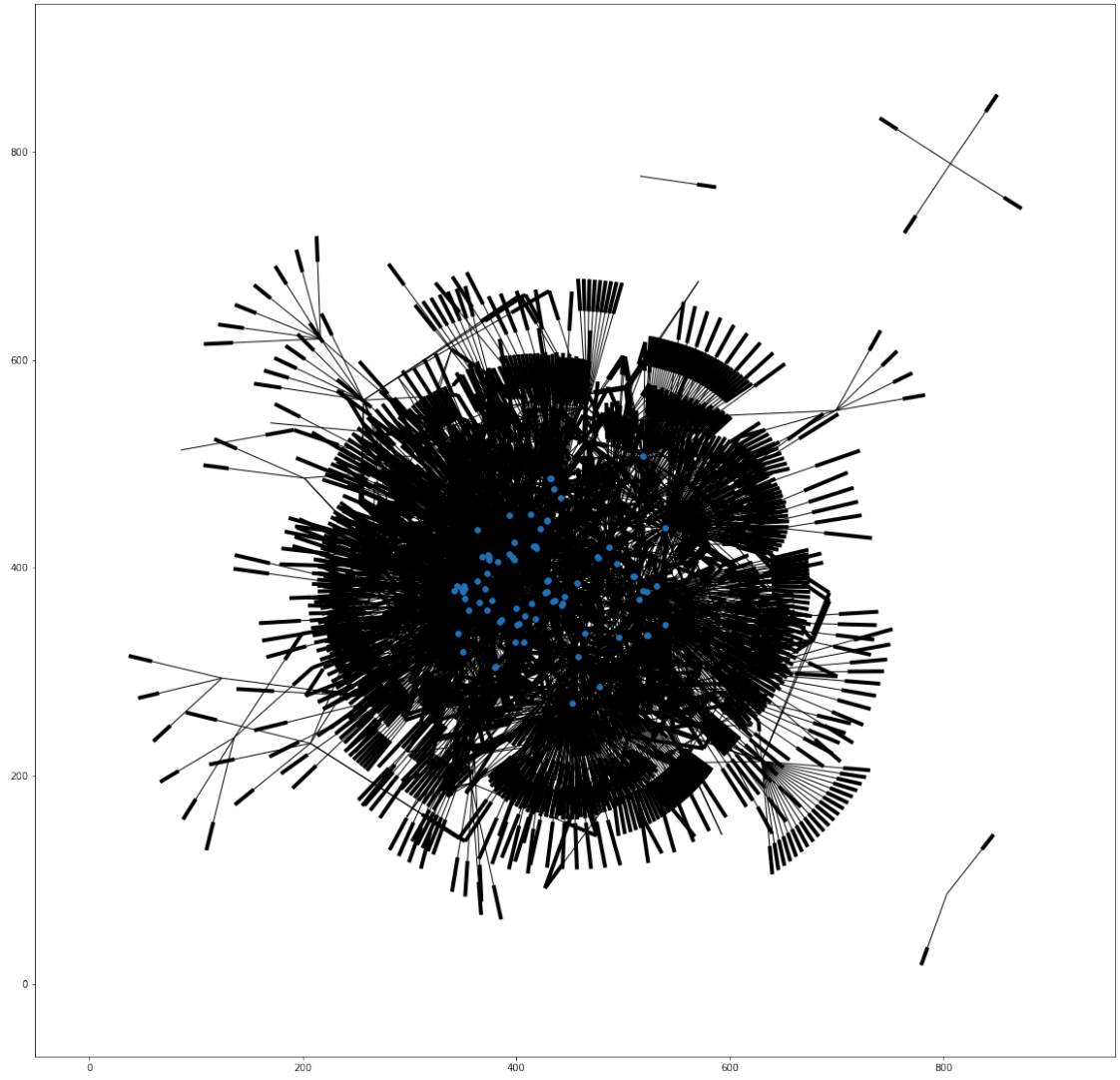
[71]:
```
(array([ 10.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,
          0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   0.,   3.,
         10.,   8.,   3.,   8.,  34.,  46.,  37.,  53.,  40.,  97., 298.,
        222., 325., 264., 179., 125.,  76.,  60.,  32.,  18.,  15.,  18.,
          6.,   1.,   4.,   0.,   1.,   1.]),<a list of 50 Patch objects>)
```



Closeness centrality is a measure of centrality in a network. That means, the more central a node is, the closer it is to all other nodes.

We observe a histogram of clossenes centralities of our graph. Few nodes have a CC of 0 - they are not connected to the rest. Majority of nodes have a CC about 0.25 so we need approximately 4 steps to pass from one node to any other in the network. Several nodes are more central - we need only about 3 edges to get to any other node. We will show these nodes on a figure.

[72]:
```
nodes_cc = [[]]*2
for n in G.nodes():
    if ntx.closeness_centrality(G.to_undirected(), n) > 0.32:
        nodes_cc[0].append(n)
    elif ntx.closeness_centrality(G.to_undirected(), n) > 0.30:
        nodes_cc[1].append(n)
draw_agraph(G, nodes_cc)
```



As presumed, these nodes are in the center of the network, regulating the rest of the genes.

The metrics allowed us to conclude about the general connectivity and closeness in the graph - the nodes are not densily connected but pretty close to each other. We can explain this thanks to the structure of regulation where only a bunch of nodes influences all the network.

2.3 K-shell decomposition

K–shell (or k–core) decomposition is an algorithm for analyzing the structure of large–scale graphs. It provides a method for identifying hierarchies in a network.

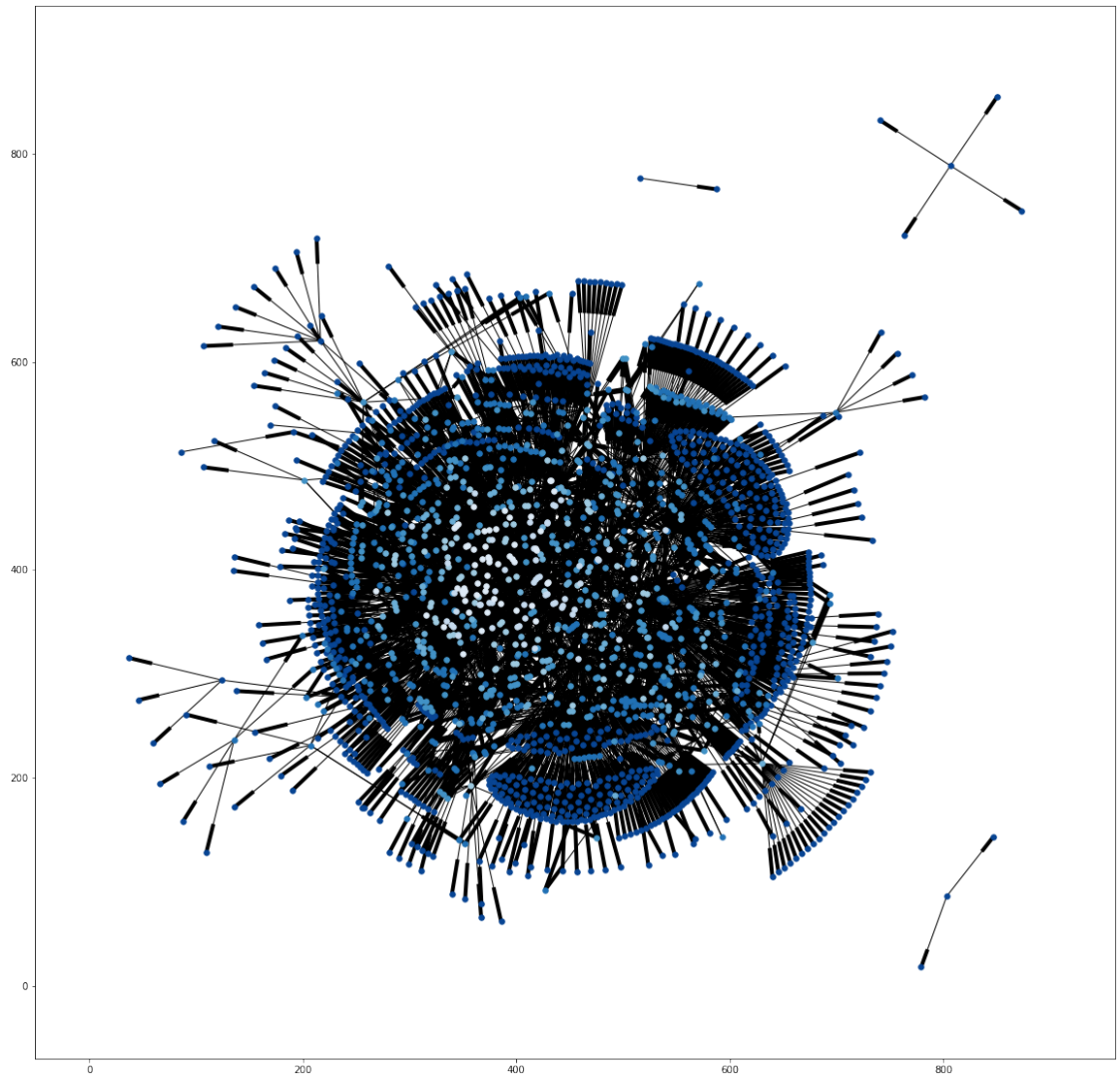
The *k-shell* instance provides us with a list of lists where the nodes in every layer has a degree corresponding to its index (and the nodes getting this degree by removing other nodes recursively).

The *k-core* instance is a list of lists containing the nodes surviving in the graph after the removal.

[38]:
```
def k_shell(G):
    k = 1
    G_ = G.copy()
    k_shell = []
    k_core = []

    degrees = list(G_.degree())
    only_deg = []
    for i in range(len(degrees)):
        only_deg.append(degrees[i][1])

    while G_.number_of_nodes() > 0:
        while True in [True for c in range(k+1) if c in only_deg]:
            for n, d in list(G_.degree()):
                if d <= k :
                    if len(k_shell) >= k:
                        k_shell[k-1].append(n)
                    else:
                        k_shell.append([n])
                    G_.remove_node(n)

            degrees = list(G_.degree())
            only_deg = []
            for i in range(len(degrees)):
                only_deg.append(degrees[i][1])

        k_core.append(list(G_.nodes()))
        print("k", k)
        print("nb of nodes", G_.number_of_nodes())
        k = k+1
    return k_core, k_shell

G_core = k_shell(G)[1]
draw_agraph(G, G_core)
```

```
('k', 1)
('nb of nodes', 941)
('k', 2)
('nb of nodes', 524)
('k', 3)
('nb of nodes', 293)
('k', 4)
('nb of nodes', 184)
('k', 5)
('nb of nodes', 112)
('k', 6)
('nb of nodes', 55)
```

```
('k', 7)
('nb of nodes', 0)
```



Nodes colored in blue are in the first layers of the k-shell, they have been removed in first so they are not in the core of the graph. Nodes in white form the true heart of the graph - they have been removed in the last moments of the decomposition.

We can see that at every layer, the number of nodes is roughly divided by two. In the end, we get 55 nodes (in white) that form the true core of the graph. Blue nodes are the leafs of the dendogram.

3 Community detection

3.0.1 Girvan–Newman algorithm

The Girvan-Newman algorithm is a method used to find communities in complex graphs. The goal of this method is to identify the edges probably binding communities, and to remove them in order to get individual communities.

The algorithm works this way: - calculate the betweenness of all edges in the graph - remove the edges having the highest betweenness - calculate again the betweenness of the edges - get back to the second and third step

Indeed, the first edges to be removed will have the highest betweenness, and so will be used a lot to make a shortest path between two nodes of the graph. We can say then that the edges are used as bridges between the different communities, and many are crossed to go from one side of the cluster to another.

[5]:
```
import itertools

k = 1
comp = ntx.algorithms.community.centrality.girvan_newman(G)
```

[7]:
```
for communities in itertools.islice(comp, k):
    print(tuple(sorted(c) for c in communities))
    list_comm = tuple(sorted(c) for c in communities)
```

Time Complexity:

- number of steps: Each step of the algorithm continues until all edges have been removed. The number of steps is the number of edges E.
- calculating the highest betweenness centrality for a graph with E nodes and N edges. Calculating all the shortest paths ($C = O(N * (N + E)ln(N))$) and counting for every edge the number of paths, keeping track of the highest one. N is the number of nodes and $(N + E)ln(N)$ the complexity of Djikstra algorithm to find all shortest paths starting from the same node.

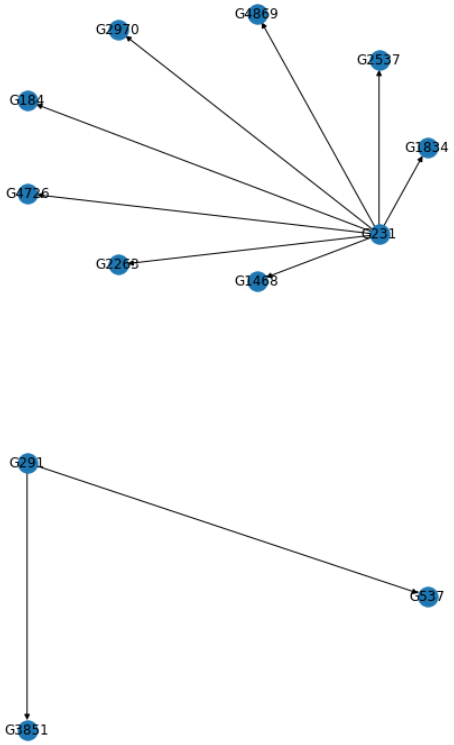
Conclusion: $C = O(EN(N + E)ln(N))$ in the worst case: $E = N^2$.$C = O(N^6 ln(N))$

However, we found in literature (*L.Despalatovi, T.Vojkovie and Damir Vukicevie "Community structure in networks:Girvan-Newman algorithm improvement",MIPRO 2014, 26-30 May 2014, Opatija, Croatia*) that calculating the betweenness centrality for every edge has actually a complexity $C = O(EN) < O(N(N + E)ln(N)))$. The time complexity of Girvan–Newman algorithm is therefore :$C = O(E^2N)$ and in the worse case: $C = O(N^6)$.

This finding corresponds to the reality - the execution of the algorithm is very long. At the end, we could find a partition from this method. We draw a subgraph for every community.

[13]:
```
commun1 = G.subgraph(['G1375', 'G220', 'G2303', 'G4322', 'G745'])
ntx.draw_circular(commun1, with_labels= True)
for comm in list_comm:
    plt.figure()
```

```
    ntx.draw_circular(G.subgraph(comm), with_labels= True)
```



We can see 6 communities given by Girvan–Newman method. Sadly, it's mostly the unconnected nodes as the Girvan–Newman is sensible to these kinds of nodes. Still, these types of communities are present everywhere in the graph.

Because of the complexity, we'll use the Louvain algorithm for our analysis. It provides much simplier and faster manipulation, even though we have to install **python-louvain** package on our computers.

3.0.2 Louvain algorithm

Louvain algorithm is a hierarchic method used for community detection based on optimization of modularity. Modularity is a value between -1 and 1 that measures the density of edges inside communities to edges outside communities.

In the Louvain Method, at first, small communities are found by optimizing modularity locally on all nodes, then each small community is grouped into one node and the first step is repeated.

[106]:
```
import community
undi_G = G.to_undirected()
partition = community.best_partition(undi_G)
nb_com = max(partition.values())+1
```

Time complexity If data structures are maintained such that each calculus of the modularity gain for one node when changing community can be computed in $O(1)$ time, the algorithm's time complexity per iteration is $O(E)$, where E is the number of edges because each community variation corresponds to one edge, and every edge corresponds to two community variations (one for each of its nodes).

We don't know how many iterations the algorithm needs to end (the modularity gain in null for each edge at the last iteration), but for most real network (including ones with billions of edges) this number is inferior to 10 so we can assume that the louvain algorithme is $O(E)$.

ref: *Lu Hao, Mahantesh Halappanavar, and Ananth Kalyanaraman "Parallel heuristics for scalable community detection*

[107]:
```
d_names = {}
for key in partition:
    for g in list(gene.index):
        if g == key:
            d_names[gene.get_value(g,'Name')] = partition[key]
len(d_names)
```

```
C:\Computations\Anaconda3\lib\site-packages\ipykernel_launcher.py:5:
FutureWarning: get_value is deprecated and will be removed in a future release.
Please use .at[] or .iat[] accessors instead
  """
```

[107]: 1994

We create a copy of our dictionary *partition* but with gene names as keys. This is necessary as only names are present in the *mapping* file.

We create an empty (containing only zeros) dataframe that we well use as our counting matrix M. We define its dimensions as the number of GO annotations (got thanks to a set of *mapping[5]*) and the number of communities (here 18, but it varies with every execution).

[172]:
```
df = pd.DataFrame(np.zeros((len(set(mapping[5])), nb_com)), \
                  index = list(set(mapping[5])), columns=[i for i in␣
 ↪range(nb_com)])
df = df.drop(df.index[0], axis=0)
df.head()
```

[172]:

	0	1	2	3	4	5	6	7	8	9	10	11	12	\
GO:0016050	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
GO:0006970	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
GO:0003735	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
GO:0016197	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	
GO:0016798	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	0.0	

	13	14	15	16	17
GO:0016050	0.0	0.0	0.0	0.0	0.0

```
GO:0006970  0.0  0.0  0.0  0.0  0.0
GO:0003735  0.0  0.0  0.0  0.0  0.0
GO:0016197  0.0  0.0  0.0  0.0  0.0
GO:0016798  0.0  0.0  0.0  0.0  0.0
```

We fill the dataframe with our results (expression of GO annotations by communities).

[204]:
```
for i in range(len(mapping[5])):
    if mapping[0][i] in d_names:
        if isinstance(mapping[5][i], str):
            j = d_names[mapping[0][i]]
            df[j][mapping[5][i]] += 1

df.head()
```

[204]:

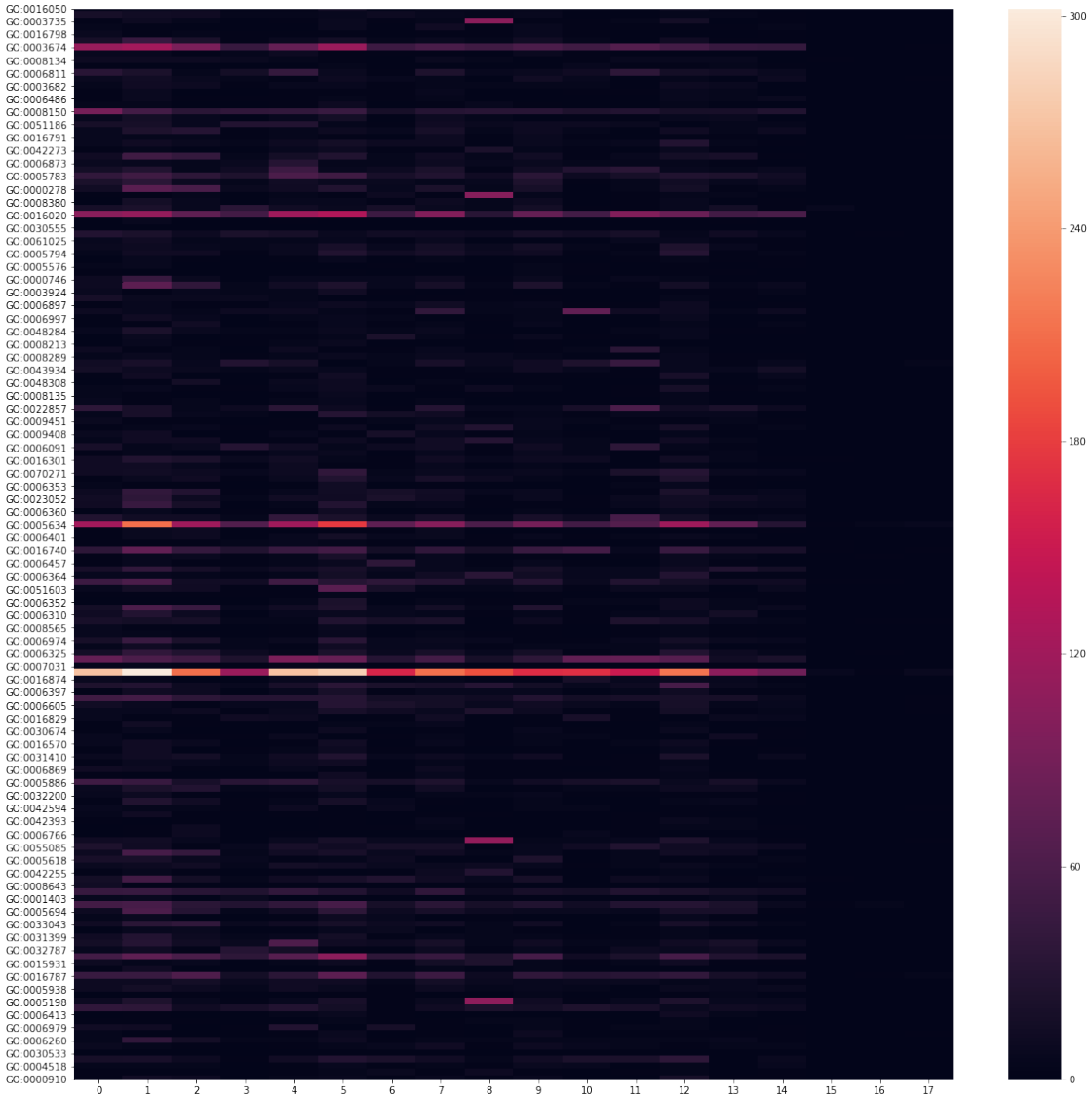
	0	1	2	3	4	5	6	7	8	9	10	\
GO:0016050	2.0	6.0	8.0	0.0	2.0	12.0	0.0	4.0	4.0	4.0	0.0	
GO:0006970	18.0	14.0	12.0	4.0	4.0	6.0	10.0	6.0	4.0	6.0	0.0	
GO:0003735	2.0	10.0	4.0	0.0	2.0	8.0	2.0	4.0	106.0	2.0	2.0	
GO:0016197	4.0	2.0	0.0	0.0	0.0	8.0	0.0	12.0	2.0	6.0	2.0	
GO:0016798	8.0	2.0	4.0	2.0	2.0	2.0	0.0	2.0	2.0	6.0	2.0	

	11	12	13	14	15	16	17
GO:0016050	0.0	8.0	2.0	0.0	0.0	0.0	0.0
GO:0006970	0.0	4.0	4.0	2.0	0.0	0.0	0.0
GO:0003735	4.0	16.0	0.0	2.0	0.0	0.0	0.0
GO:0016197	2.0	2.0	2.0	4.0	0.0	0.0	0.0
GO:0016798	0.0	0.0	0.0	2.0	0.0	0.0	0.0

We have our dataframe, we'll use a heatmap to represent the data.

[207]:
```
plt.figure(figsize=(20,20))
sns.heatmap(df)
```

[207]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1ece9c85bc8>
```

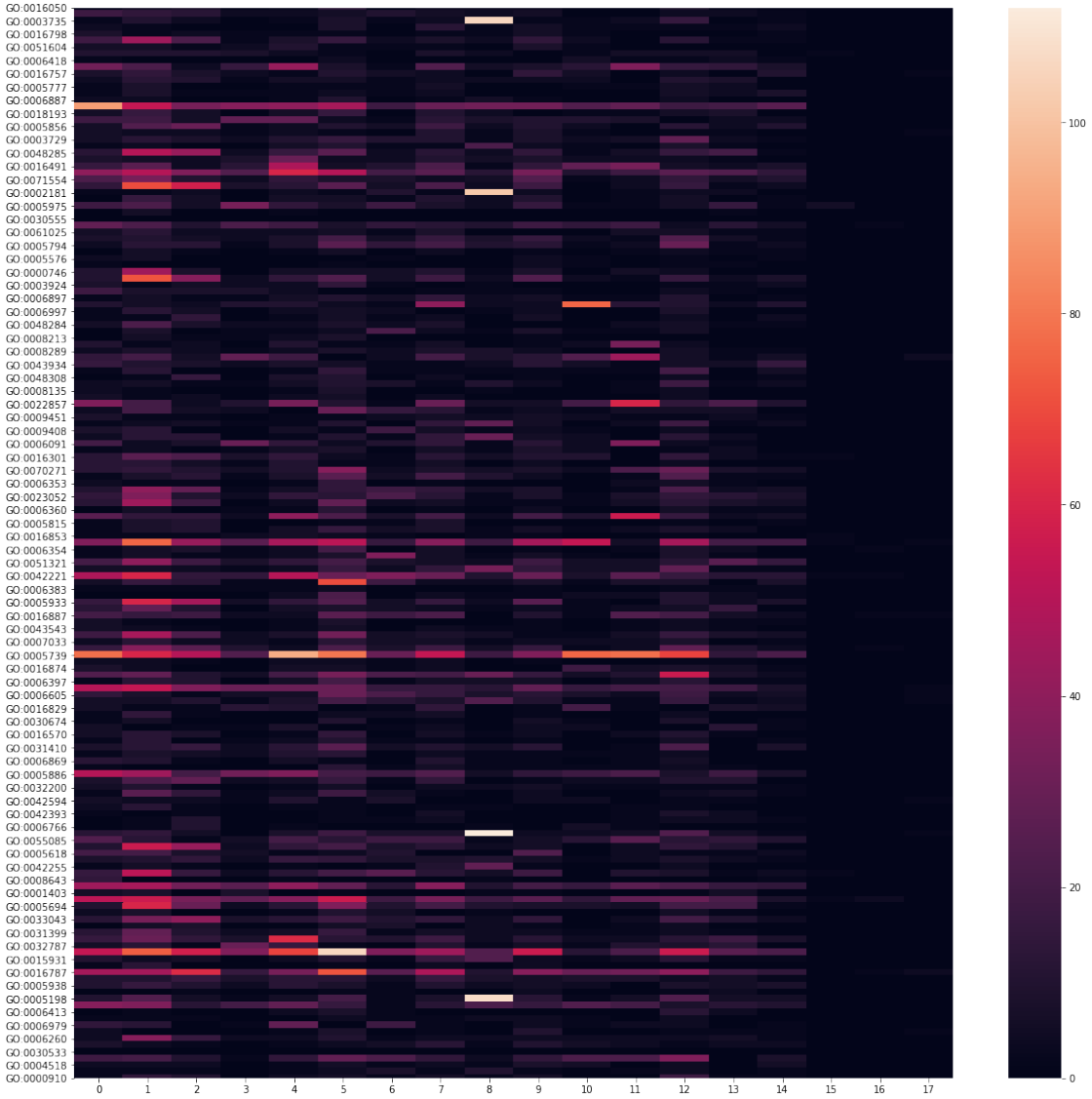


For the sake of clarity, we exclude the rows associated to GO having too many counts, as they denote too general features. Concretely, it is the rows GO:0005634, GO:0005737, GO:0003674, GO:0016020. This will give us a much clearer representation.

[209]:
```
df.loc['GO:0005634']
df = df.drop('GO:0005634')
df.loc['GO:0005737']
df = df.drop(['GO:0005737', 'GO:0003674', 'GO:0016020'])
```

[210]:
```
plt.figure(figsize=(20,20))
sns.heatmap(df)
```

[210]:
```
<matplotlib.axes._subplots.AxesSubplot at 0x1eceb964248>
```



This heatmap provides an interesting inside into the gene expression. Certain GOs are still more global than anothers, but the difference is not so important - in general, the roxs are homogenic.

The communities, also, share a different level of expression. The communities n. 15, 16, 17 have very small number of corresponding GOs. On the other side, communities n. 1, 5 and 12 are strongly linked to a important number of GOs.

Interesting case is the community n. 8. It contains very strongly only 4 GOs! We will look up more closely these annotations.

- G0-0003735 : cAMP-dependent protein kinase catalytic subunit, WD-repeat protein involved in ubiquitin-mediated protein degradation
- G0-0002181 : WD-repeat protein involved in ubiquitin-mediated protein degradation, cAMP-dependent protein kinase catalytic subunit

- G0-0006766 : negative regulation of cytoplasmic translation
- G0-0005198 : double-strand break repair via nonhomologous end joining P-body

The annotations share common features, for example cAMP-dependent protein kinase. We can conclude the the genes in the communities share some cell functions and they are involved in particular cell mechanisms.

4 Conclusion

We can see that in a yeast, only a few genes regulate all other genes. This is why our graph is really not dense, with a structure that can look like a tree.

The communities analysis allowed us to discover shared cell functions of genes in some communities.