project_2.1

March 10, 2023

In this project, you will be working with the protein interaction network (PIN) of *Saccharomyces cerevisiae* from the BioGRID Multi-Validated (MV) Datasets. The nodes and edges of this PIN denote proteins and their physical interactions, respectively. These interactions have been identified from large-scale protein interaction experiments such as yeast two-hybrid screenings and affinity purification with mass spectrometry. Additionally, they pass a specific set of criteria as defined here (BioGRID MV Datasets). See the following link to familiarize yourself with the contents of this BioGRID file. You will be analyzing the yeast PIN using k-core analysis, and further use the network as a scaffold for incorporating gene expression data, allowing you to perform s-core analysis using pairwise gene expression correlations as weights.

0.0.1 1.1 Yeast protein interaction network

Download the file containing all multi-validated protein-protein interactions from BioGRID found on Blackboard `BioGRID_PIN.txt`. Create a network with NetworkX using the systematic names to set source and target nodes. These corresponds to the columns named `Systematic Name Interactor A` and `Systematic Name Interactor B`. Remove all self-edges from the network.

NB! I have created my own module with functions that I have been using several times, both in project 1 and 2. The content of the module is printed out in the Appendix of this document. It is likely to change from project to project.

[ ]:
```
# Imports
import pandas as pd
import numpy as np
import networkx as nx
from pyvis.network import Network
import sys
import os
if sys.path[-1] != "..":
    sys.path.append("..")
from SysBio_fun import *
from tbtFunctions import *
```

[ ]:
```
# path_ = sys.path[0].split("/")
# os.chdir("/".join(path_[:-1]))
# os.getcwd()
# os.chdir("/".join(path_))
```

```
[ ]: # Read file:
df = pd.read_table("BioGRID_PIN.txt", sep="\t", low_memory=False)

# Define source and target column:
source = "Official Symbol Interactor A"
target = "Official Symbol Interactor B"
# Rename columns:
df.rename(columns={source:"source",
                   target:"target"},
                   inplace=True)
# Drop rows where column "source" value is identical to column "target" value.
# I.e., drop self loops:
# **NB!** This the number of rows left in the df may vary depending on which
# column-pairs were treated as source-target. E.g. source:"-" and target:"-"
# will be removed, even though they may have different Entrez ID.
df = df[df["source"] != df["target"]]

# Generate the graph:
# **NB!** I treat the graph as multi directed, i.e. each row in the df
# amounts to a single edge in the network!
PIN = nx.from_pandas_edgelist(df, edge_attr=True, create_using=nx.
 ↪MultiDiGraph())

# Make a copy
PIN_copy = PIN.copy()
```

(i) How large is the network (i.e. number of nodes and edges), and what is the average node degree?

```
[ ]: N = PIN.number_of_nodes()
E = PIN.number_of_edges()
mean_k = (E*2)/N  # Average node degree. Each link contributes a degree "point"␣
 ↪for each node it connects. Therefore 2*E

print(f"""
The number of nodes are: {N}
The number of edges are: {E}
The average node degree is: {mean_k}
""")
```

```
The number of nodes are: 20550
The number of edges are: 245288
The average node degree is: 23.872311435523116
```

(ii) Look up *S. cerevisiae* S288C in the NCBI Taxonomy Database here and take note of the taxonomy ID. Using the `Organism Interactor A` and `Organism Interactor B` edge attributes in the network, use this taxonomy ID to filter out all non-*S. cerevisiae* S288C nodes from the network. Also, remove all self-loops and isolated nodes. How many nodes and edges remain? Create a plot of its degree distribution. How would you characterize this network and its degree distribution?

[ ]:
```python
PIN = PIN_copy.copy()

# The taxonomic ID of S. cerevisiae S288C is: 559292

# Generate a list of edges that does not fit:
r1 = [edge for edge,tax \
      in nx.get_edge_attributes(PIN, "Organism Interactor A").items() \
          if tax != 559292]  # The edge information is a tuple and the key
r2 = [edge for edge,tax \
      in nx.get_edge_attributes(PIN, "Organism Interactor B").items() \
        if tax != 559292]
rem = set(r1+r2)  # set() is used to remove potential duplicates
# Use the list to remove all edges
PIN.remove_edges_from(rem)
# In pandas:
# len(df[(df["Organism Interactor A"] == 559292) & (df["Organism Interactor B"]␣
 ↪== 559292)])



# Remove nodes with a node degree of 0 (i.e. isolated nodes):
PIN.remove_nodes_from(  # Remove nodes from list ...
    [nodes for nodes,degree \
     in dict(PIN.degree()).items() if degree == 0]  # nodes with degree 0
)

N = PIN.number_of_nodes()  # New number of nodes
E = PIN.number_of_edges()  # New number of edges

print(f"""
The number of nodes is now {N},
while number of edges is {E}
""")
```

```
The number of nodes is now 4104,
while number of edges is 68704
```

(iii) Compare the yeast PIN with Barabási–Albert (BA) and Erdős–Rényi (ER) networks with approximately the same number of nodes and edges using a few selected network measurements and the degree distributions. Do these models do a good job in describing the yeast

PIN? Discuss why/why not.

```
[ ]: l_comp = nx.Graph(PIN.subgraph(
         sorted(
             nx.connected_components(
                 nx.to_undirected(PIN.copy())
                 #PIN.copy().to_undirected(as_view=True)
                 ),
             reverse=True
         )[0]
     ).copy())

     n = l_comp.number_of_nodes()
     e = l_comp.number_of_edges()

     ER = connected_ER(n, e)
     BA = connected_BA(n, round(((e*2)/n)))
```

```
[ ]: #type(PIN.subgraph(nx.connected_components(PIN.to_undirected())[0]))
     type(
         nx.Graph(PIN.subgraph(
             sorted(
                 nx.connected_components(
                     nx.Graph(PIN.copy())
                 ), reverse=True)[0]
             ))
         )
     type(PIN)

     # PIN2 = nx.Graph(PIN.copy())
     # print(type(PIN2))
     # print(PIN == PIN2)
     # PIN2 = PIN2.to_undirected()
     # print(type(PIN2))
```

```
[ ]: networkx.classes.multidigraph.MultiDiGraph
```

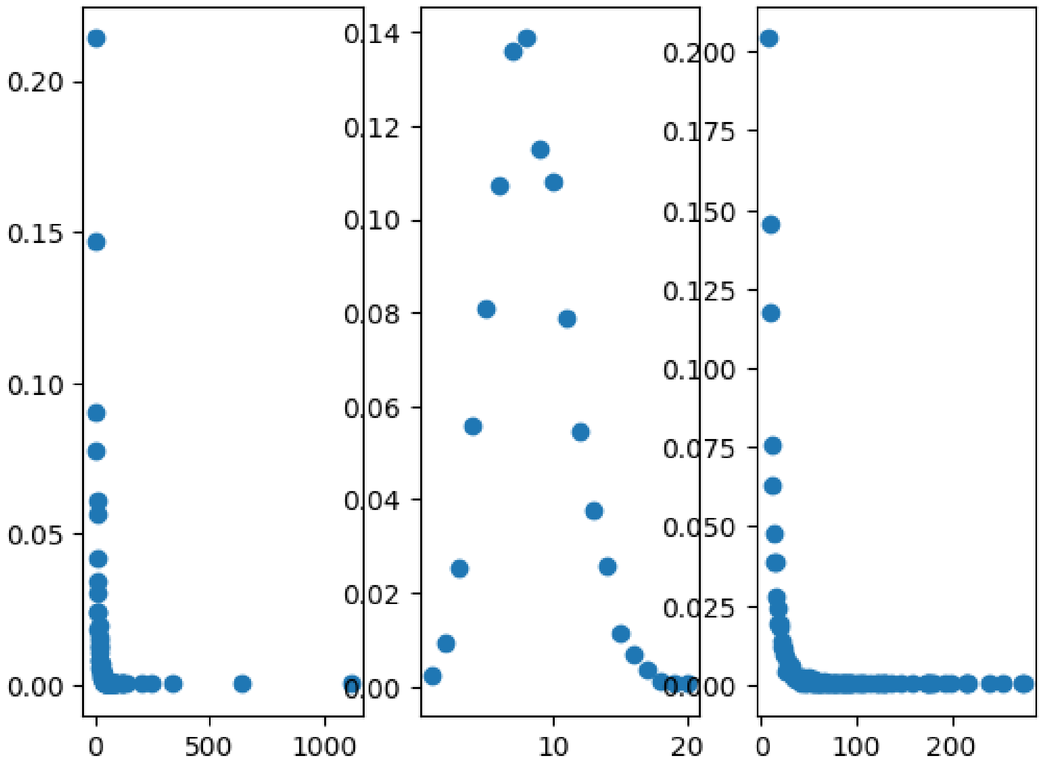
```
[ ]: graph_info(l_comp)
     graph_info(ER)
     graph_info(BA)
```

```
Number of nodes: 4022
Number of edges: 16700
Is connected:    True

Number of nodes: 4022
Number of edges: 16700
```

```
Is connected:    True

Number of nodes: 4022
Number of edges: 32112
Is connected:    True
```

```
[ ]: def degree_distribution(graph):
         """
         Function that returns a scatter plot of the degree
         distribution of a graph.
         """
         N = graph.number_of_nodes()  # Total number of nodes
         y = np.array(nx.degree_histogram(graph))/N  # All occurrences of "n"␣
      ↪degree, divided by total number of nodes
         x = np.arange(len(y))[y != 0]  # x values
         y = y[y != 0]  # Remove the values = 0 from the array

         return x, y

     x_l_comp, y_l_comp = degree_distribution(l_comp)
     x_ER, y_ER = degree_distribution(ER)
     x_BA, y_BA = degree_distribution(BA)


     figure, axis = plt.subplots(1, 3)
     axis[0].scatter(x_l_comp, y_l_comp)
     axis[1].scatter(x_ER, y_ER)
     axis[2].scatter(x_BA, y_BA)
```

```
[ ]: <matplotlib.collections.PathCollection at 0x7f6e295f1340>
```



```
[ ]: #plt.scatter(nx.clustering(BA).keys(), nx.clustering(BA).values())
```

(iv) You will now start peeling away layers of the yeast PIN by k-core analysis. In your own words, describe/define k-core analysis, and explain how it works on a network. What network does the 1-core correspond to?

k-core analysis is a method to identify importance of nodes by how connected they are to other nodes. Nodes with few links are removed stepwise until you end up with a core of nodes that have many connections in the original network, in addition to being tightly connected between themselves. For each iteration of the method a layer of nodes are removed. The first core (i.e. the 1-core) consists of nodes that have a single link or more, i.e. a node degree \$ \$1. This core is therefore the entire network. The 2-core is the network after we have removed all nodes with a degree of 1, in addition to the nodes that now ends up with a degree of 1 following the first removal. In the 2-core network all nodes will have a node degree \$ \$2. When all nodes with a degree of 2, and those who suddenly get a node degree $\leq$, are removed we are left with the 3 core and the next iteration can begin.

Pseudo code: k-core = 1 while number of nodes in network > 0: while any node with degree == k-core: Remove nodes with node degree $\leq$ k-core

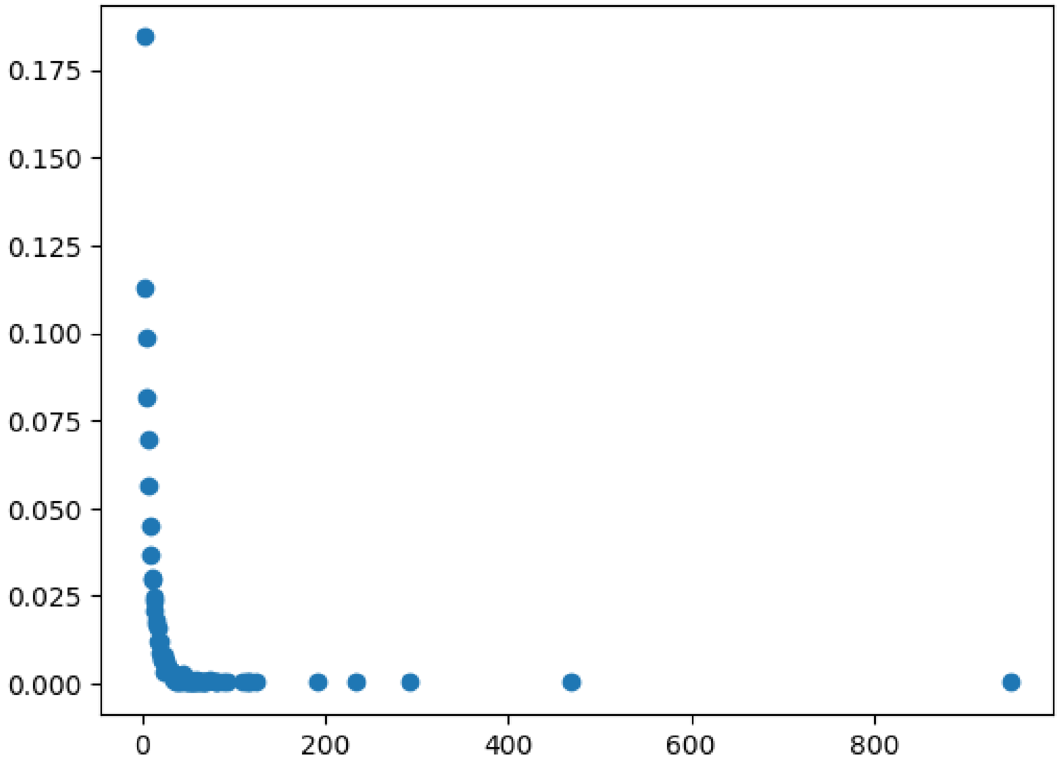
```
return network
k-core = k-core + 1
```

(v) Calculate the 2-core of the yeast PIN. How large is this network? Plot and characterize its degree distribution and compare it to the original yeast PIN.

[ ]:
```
l_comp_2_core = nx.k_core(l_comp, k=2)
graph_info(l_comp_2_core)

x, y = degree_distribution(l_comp_2_core)
plt.scatter(x,y)
```

```
Number of nodes: 3102
Number of edges: 15780
Is connected:    True
```

[ ]:
```
<matplotlib.collections.PathCollection at 0x7f6e2bea3dc0>
```



(vi) Find the innermost k-core (the last k before the network is empty). How many edges and nodes are there in the innermost k-core? What is the absolute maximal amount of edges that there could be between this number of nodes, and how does the innermost core compare to that?

[ ]:
```
current_k = 0
g = l_comp.copy()
while nx.number_of_nodes(g) > 0:
    current_k += 1
    g = nx.k_core(g, k=current_k)
innermost_k_core = current_k-1

print(f"The innermost k-core is: {innermost_k_core}")

graph_info(nx.k_core(l_comp.copy(), k=innermost_k_core))
```

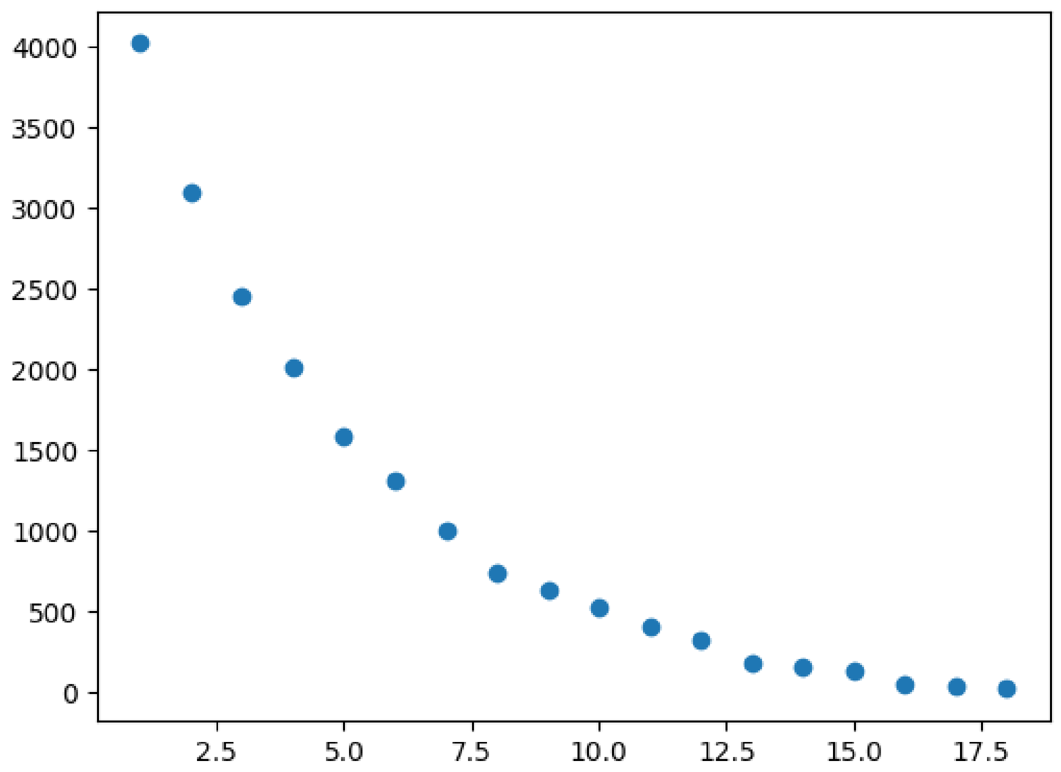
```
The innermost k-core is: 18

Number of nodes: 19
Number of edges: 171
Is connected:    True

Number of nodes: 19
Number of edges: 171
Is connected:    True
```

(vii) Create a plot of the number of nodes and the number of edges in a k-core against k. How would you characterize this plot? What does it tell you?

```
[ ]: y = [nx.number_of_nodes(nx.k_core(l_comp.copy(), k=k_)) for k_ in range(1,␣
      ↪innermost_k_core+1)]
     x = range(1, innermost_k_core+1)
```

```
[ ]: plt.scatter(x, y)
```

```
[ ]: <matplotlib.collections.PathCollection at 0x7f6e35a4d6d0>
```



(viii) Generate the second innermost k-core and visualize it using pyvis. Describe the network.

[ ]:
```
# def show_html(graph, name="nx", show=True, size="small"):
#     """
#     Generate and display the graph through pyvis
#     """

#     if size == "small":
#         nt = Network("500px", "500px")
#     else:
#         nt = Network("1080px", "1920px")

#     nt.from_nx(graph)
#     if show == True:
#         nt.show(f"{name}.html")

s = nx.k_core(l_comp.copy(), k=innermost_k_core-1)
show_html(s, name=f"{innermost_k_core-1}-core_network", show=True, size="large")
```

(ix) We will now load gene expression data onto our yeast PIN. Go to this url containing the gene expression data set from *Transcription profiling by array of yeast to investigate expression of the beta-subunit of Snf1 kinase*. For those interested, you can find detailed information about the dataset in the original publication here. Download `normalized_expressions.tsv` from Blackboard and import the list of gene names, as well as corresponding expression data from 24 experimental conditions. For all pairwise gene combinations, calculate the Pearson correlation coefficients and associated p-values for the N genes present in the yeast PIN (both as N x N Numpy arrays). Out of the gene pairs with positive correlation values, how many of them have a significance value below 0.0001?

```
[ ]: norm_ex = pd.read_table("normalized_expressions.tsv", sep="\t")
     PIN_genes = list(PIN.nodes())
     norm_ex = norm_ex[norm_ex["Gene ID"].isin(PIN_genes)]

     gene_exp = norm_ex.iloc[:, 1:].to_numpy()
     corrmat, pmat = faster_corrcoef(gene_exp)

     # ge_data = pd.read_csv(...appropriate values...)  #creates pandas DataFrame␣
      ↪object, Read gene expression data

     # #Filter out PIN-genes not in the yeast PIN - important to ensure that you're␣
      ↪not wasting lots of compute time on irrelevant proteins
     # PIN_genes = list(yeast_pin.nodes())
     # ge_data = ge_data[ge_data['Gene ID'].isin(PIN_genes)]

     # gene_exp = ge_data.iloc[:, 1:].to_numpy()   # convert expression data to␣
      ↪NumPy array

     # corrmat, pmat = tbt.faster_corrcoef(gene_exp)  # Calculate Pearson␣
      ↪correlations and associated p-values

     # And now, you have the correlation matrix as well as the related p-value␣
      ↪matrix. Necessary to loop through the correlation matrix and only count the␣
      ↪correlations that have P-values < 1E-4.
```

(x) Download `s_core.py`. This is similar to the NetworkX function `k_core`, but uses node strength instead of node degree. Strength is defined as the sum of the weights of all the links attached to a node (i.e. Person correlations). With negative weights, the absolute values of the weights are used to calculate the strength. In this case, however, we are only looking at positively correlated nodes. Do you think it would make sense to include negative correlations in this particular case? Discuss.

(xi) Calculate the s-core with $s = 2.5$, 3, 3.5, and 4, and characterize their degree distributions. Discuss how the distribution changes for increasing values of s.

```
[ ]: from s_core import *
```

(xii) Find an integer value for s that gives as good match as possible relative to the second innermost k-core in terms of number of distinct modules and number of nodes. Visualize this s-core with pyvis and describe the network.

```
[ ]:
```

(xiii) Using the full yeast PIN network (network from (ii)), extract the nodes that are either part of the second innermost *k*-core, the *s*-core in (xii), or part of both cores. How many nodes and edges are there in this new network? Create a visualization of this subnetwork using pyvis, and color the nodes according to the following classifications:

- Red if it was only found in the second innermost *k*-core.
- Green if it was only found in the *s*-core.
- Blue if it was found in both.

Describe the network and discuss what you observe. Is it a single connected component? What do the differences between the *s*- and *k*-cores tell you?

1 Appendix

[ ]:
```
file_path = "/".join(sys.path[0].split("/")[:-1])

with open(file_path+"/SysBio_fun.py") as file:
    print(file.read())
```

```
#############################################
#   This is a module created to hold        #
#   all custom functions used in multiple   #
#   project throughout the TBT4165 course   #
#############################################



## Imports:
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
from pyvis.network import Network


## Functions:

def connected_ER(N, E):
    """
    N : int, Number of nodes
    E : int, Number of edges
    """
    g = nx.gnm_random_graph(N, E)
    while nx.is_connected(g) == False:
        g = nx.gnm_random_graph(N, E)
    return g



def connected_BA(N, m):
    """
    N : int, Number of nodes
    m : int, Number of edges to attach from a new node to existing nodes
    """
    g = nx.barabasi_albert_graph(n=N, m=m)
    while nx.is_connected(g) == False:
        g = nx.barabasi_albert_graph(n=N, m=m)
    return g


def get_largest_component(g, is_undirected=True):
    if is_undirected:
        l_comp = g.subgraph(
            sorted(
                nx.connected_components(
                    nx.to_undirected(g.copy())
                    ),
                reverse=True
            )[0]
        ).copy()
    else:
        l_comp = g.subgraph(
            sorted(
                nx.connected_components(g),
                reverse=True
            )[0]
        ).copy()
    return l_comp


def graph_info(g):
    print(f"\nNumber of nodes: {g.number_of_nodes()}")
    print(f"Number of edges: {g.number_of_edges()}")
    try:
        print(f"Is connected:    {nx.is_connected(g)}")
    except:
        print("""\
The graph is not undirected. Therefore .is_connected() does not work.
        """)


def show_html(graph, name="nx", show=True, size="small"):
    """
    Generate and display the graph through pyvis
    """

    if size == "small":
        nt = Network("500px", "500px")
    else:
        nt = Network("1080px", "1920px")

    nt.from_nx(graph)
    if show == True:
        nt.show(f"{name}.html")



def nice_plot(title, xlab, ylab, show=True):
    """
    Function to generate title and axis labels to a plot.
    The argument "show" is by default True.
    """
    plt.title(title)
    plt.xlabel(xlab)
    plt.ylabel(ylab)
    if (show == True):
        plt.show()


def plot_degree_distribution(graph):
    """
    Function that returns a scatter plot of the degree
    distribution of a graph.
    """
    N = graph.number_of_nodes()  # Total number of nodes
    y = np.array(nx.degree_histogram(graph))/N  # All occurrences of "n" degree,
divided by total number of nodes
    x = np.arange(len(y))[y != 0]  # x values
    y = y[y != 0]  # Remove the values = 0 from the array

    return [x, y]
    #return plt.scatter(x,y)
```