Solutions to Assignment 1
Network Science
PHYS 7332, Fall 2024

Chelsea Ajunwa*

1. *Question 1.*

(a) Csv files are easy for humans to read and edit. Pickle files are light and even faster to read and write dataframes to. Files in .gml format can store nodes, edges, and their attributes all in place. Elasticsearch enables the storage and analysis of very large graphs. It is designed for fast information retrieval.

(b) Csv files can easily store edge attributes but not node attributes. Another separate file is required to do so. Pickle is python specific. Files in .gml format are large. Additionally, the format seems to be less widely used than others. Elasticsearch is resource-intensive and is not meant to be used for primary storage.

(c) Pickle poses a security risk. A malicious actor could modify the serialized data and cause you to execute whatever code they've added. Therefore, one should only open pickle files from trusted sources.

(d) Csv is best when only edge information needs to be stored and you are working with a graph that is relatively small and tabular in nature. It is also useful for storing network data that will be shared with others that may not be using the same analysis software as you. For example, you could use .csv to store brain network connectivity data that will be shared across researchers or institutions. Pickle is best when you are working in python with python-specific data structures and are not receiving data from untrustworthy sources. For example, you might use pickle to store an intermediate version of a graph that you are altering. The .gml format should be used for storing graphs with edge and node attributes. It is best for small but complex graphs, like a social network graph in which nodes are divided into different groups and therefore have group membership attributes. Elasticsearch should be used when you have a very large dataset that you would like to perform searches and queries on. For example, you could use Elasticsearch to store internet data in which you would like to search for occurrences of a particular word or phrase.

2. *Question 2.*

(a) You will never be able to find an allowable edge exchange if you input a complete graph. This is because in such a graph, all possible node pairs already have edges between them. This is not allowed by the algorithm

* **ADD CLASS REPO FORK** https://github.com/cajunwa/phys7332_fa25

(b) 2b) Modified algorthm (in python):

```
def edge_exchange(G,t,n_failed_attempts_allowed=1000):

G_random=G.copy()
edges = list(G_random.edges())

n_failed_attempts =0
s=0
while s<t:
    if n_failed_attempts < n_failed_attempts_allowed:
        edge1_id = np.random.choice(list(range(len(edges))))
        i,j = edges[edge1_id]
        #print(i,j)
        edge1_id = np.random.choice(list(range(len(edges))))
        u,v = edges[edge1_id]
        #print(u,v)
        if i==u or i==v or j==u or j==v:
            print("self-loop error")
            n_failed_attempts+=1
            continue
        elif G_random.has_edge(i,v) or G_random.has_edge(j,u):
            print("edge exists error")
            n_failed_attempts+=1
            continue
        G_random.remove_edge(i,j)
        G_random.remove_edge(u,v)
        G_random.add_edge(i,v)
        G_random.add_edge(j,u)
        edges=list(G_random.edges())

        s+=1
        #print(n_failed_attempts)
    else:
        raise Exception("too many failed attempts")


return G_random
```

(c) 2c) modified algorthm (in python):

```
    def
        edge_exchange_preserve_conn(G,t,n_failed_attempts_allowed=1000):
"""
```

```
Bagrow & Ahn
Basic edge exchange algorithm
But preserves connectivity.
If there was a path between two nodes in the original graph,
there will also be such a path after edge exchange is run.

G = graph
t = number of exchanges
n_failed_attempts_allowed = number of failed exchanges allowed
    before termination
"""
G_random=G.copy()
edges = list(G_random.edges())
nodes = list(G_random.nodes())

#lists all paths
connected_nodes_orig = connected_nodes_list(G_random)



n_failed_attempts =0
s=0
while s<t:
    if n_failed_attempts < n_failed_attempts_allowed:

        edge1_id = np.random.choice(list(range(len(edges))))
        i,j = edges[edge1_id]
        #print(i,j)
        edge1_id = np.random.choice(list(range(len(edges))))
        u,v = edges[edge1_id]
        #print(u,v)
        if i==u or i==v or j==u or j==v:
            #print("self-loop error")
            n_failed_attempts+=1
            continue
        elif G_random.has_edge(i,v) or G_random.has_edge(j,u):
            #print("edge exists error")
            n_failed_attempts+=1
            continue


        G_random.remove_edge(i,j)
        G_random.remove_edge(u,v)
        G_random.add_edge(i,v)
        G_random.add_edge(j,u)
```

```
            if connected_nodes_list(G_random) != connected_nodes_orig:
                #reverse
                G_random.remove_edge(i,v)
                G_random.remove_edge(j,u)
                G_random.add_edge(i,j)
                G_random.add_edge(u,v)

                print("connectivity change reversed")

            edges=list(G_random.edges())
            s+=1
            #print(n_failed_attempts)
        else:
            raise Exception("too many failed attempts")


    return G_random
```

(d) 2d) See code below. This additional constraint makes edge exchange more difficult because it leads to more invalid exchanges and therefore more iterations of the algorithm

```
    def edge_exchange_binary(G,t,n_failed_attempts_allowed=1000):
"""
Bagrow & Ahn
Basic edge exchange algorithm
But for binary attribute graphs.


G = graph with edge weights 1 or -1
t = number of exchanges
n_failed_attempts_allowed = number of failed exchanges allowed
    before termination



"""
G_random=G.copy()
edges = list(G_random.edges())

n_failed_attempts =0
s=0
while s<t:
```

```
        if n_failed_attempts < n_failed_attempts_allowed:
            edge1_id = np.random.choice(list(range(len(edges))))
            i,j = edges[edge1_id]
            #print(i,j)
            edge1_id = np.random.choice(list(range(len(edges))))
            u,v = edges[edge1_id]
            #print(u,v)
            if i==u or i==v or j==u or j==v:
                print("self-loop error")
                n_failed_attempts+=1
                continue
            elif G_random.has_edge(i,v) or G_random.has_edge(j,u):
                print("edge exists error")
                n_failed_attempts+=1
                continue

            w1=G_random[i][j]["weight"]
            w2 =G_random[u][v]["weight"]

            init_attributes = set([w1,w2])
            G_random.remove_edge(i,j)
            G_random.remove_edge(u,v)
            G_random.add_edge(i,v,weight=w1)
            G_random.add_edge(j,u,weight=w2)
            end_attributes =
                set([G_random[i][v]["weight"],G_random[j][u]["weight"]])

            if init_attributes != end_attributes:
                print("init and end attributes do not match")
                G_random.remove_edge(i,v)
                G_random.remove_edge(j,u)
                G_random.add_edge(i,j,weight=w1)
                G_random.add_edge(u,v,weight=w2)

            edges=list(G_random.edges())



            s+=1
            #print(n_failed_attempts)
        else:
            raise Exception("too many failed attempts")


    return G_random
```

3. *Question 3.*

(a) Betweenness centrality function:

```
    # Supplement your written text with code, if you'd like!
    import networkx as nx

    #Question 3a

#Write function to calculate betweenness centrality
def betweenness_centrality(G,node):
    """
    Calculates the betweenness centrality for a node in a graph.

    Betweenness centrality of a node is the sum of the fraction of
        shortest paths that pass through it.
    For each s and t nodes in graph, calculates proportion of shortest
        paths that pass through node
    of interest. These proportions are then summed across all
        combinations of s and t.

    Parameters
    ----------
    G: networkx.Graph
        The input graph on which betweenness centrality is calculated.
    node:
        Node of the input graph that you would like to calculate the
            betweenness centrality

    Returns
    -------
    bc: float
        The betweenness centrality of the node.

    Example
    -------

    """
    nodes = list(G.nodes())

    ratio_sum = 0
    for s in nodes:
```

```
        #print(s)
        for t in nodes:
            if s !=t:
                #print(t)
                shortest_paths_st = list(nx.all_shortest_paths(G,s,t))

                num_sp = len(shortest_paths_st)
                paths_collapse = [item for sublist in shortest_paths_st
                    for item in sublist]
                paths_collapse.remove(s)
                paths_collapse.remove(t)

                num_v_in_sp = paths_collapse.count(node)

                ratio = num_v_in_sp/num_sp

                ratio_sum = ratio_sum + ratio
    return ratio_sum
```

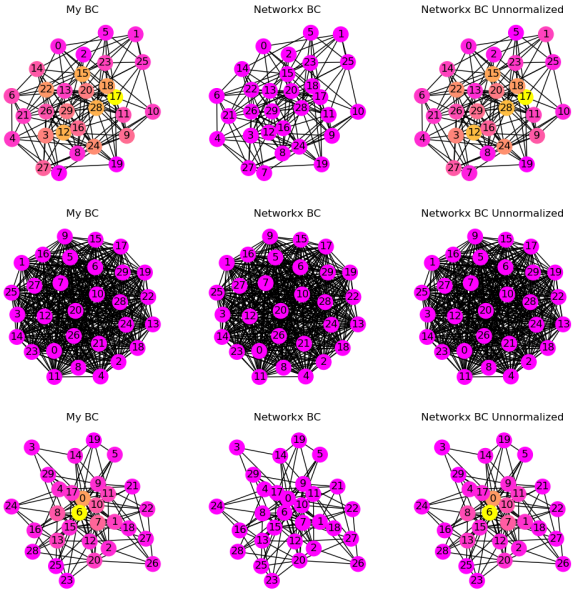
(b) Code with error handling:

```
        def betweenness_centrality(G,node):
    """
    Calculates the betweenness centrality for a node in a graph.

    Betweenness centrality of a node is the sum of the fraction of
        shortest paths that pass through it.
    For each s and t nodes in graph, calculates proportion of shortest
        paths that pass through node
    of interest. These proportions are then summed across all
        combinations of s and t.

    Parameters
    ----------
    G: networkx.Graph
        The input graph on which betweenness centrality is calculated.
            Should be unweighted and
        undirected.
    node:
        Node of the input graph that you would like to calculate the
            betweenness centrality

    Returns
    -------
    bc: float
```

```
    The betweenness centrality of the node.

Example
-------

"""
#Making sure graph is networkx object
if isinstance(G,nx.classes.graph.Graph)==False:
    raise Exception("Graph must be networkx object")

#Making sure node of interest is in graph
nodes = list(G.nodes())
if node not in nodes:
    raise Exception("Node not in graph")

#Making sure graph has connectivity
edges = list(G.edges())
if edges==[]:
    raise Exception("No connectivity")

ratio_sum = 0
for s in nodes:
    #print(s)
    for t in nodes:
        if s !=t:
            #print(t)
            shortest_paths_st = list(nx.all_shortest_paths(G,s,t))

            num_sp = len(shortest_paths_st)

            paths_collapse = [item for sublist in shortest_paths_st
                for item in sublist]
            paths_collapse.remove(s)
            paths_collapse.remove(t)

            num_v_in_sp = paths_collapse.count(node)

            try:
                ratio = num_v_in_sp/num_sp
            except:
                print(f"No shortest paths between {s} and {t}.
                    Continuing")
                continue

            ratio_sum = ratio_sum + ratio
```

```
    return ratio_sum
```

(c) My betweenness centrality results are quite different from the network function's results. It looks like networkx uses some algorithm to approximate betweenness centrality. It also performs some normalization. When I remove the normalization, the networkx graph does look more like mine despite the differing absolute betweenness centrality values.



(d) I chose three test cases. The first was an empty graph with no nodes. We might expect the betweenness centrality function to have an issue with this graph due to the lack of paths to calculate betweenness centrality from. Running this graph with my betweenness centrality function generates a "no connectivity" error message, as I intended. The centrality function test indicates this as well. The issue of not having paths from which to calculate betweenness centrality also applies to my next test case, the null graph, which has no nodes and no edges. My third test case was a directed graph. As indicated in the docstring, my betweenness centrality function was meant for undirected graphs only. In addition to these test cases, I also included a path graph as a valid example. As indicated by centrality function test, the function ran successfully on this graph.

4. *Question 4.*

(a) Add! Your! Answer! Here!

(b) Add! Your! Answer! Here!

(c) Add! Your! Answer! Here!

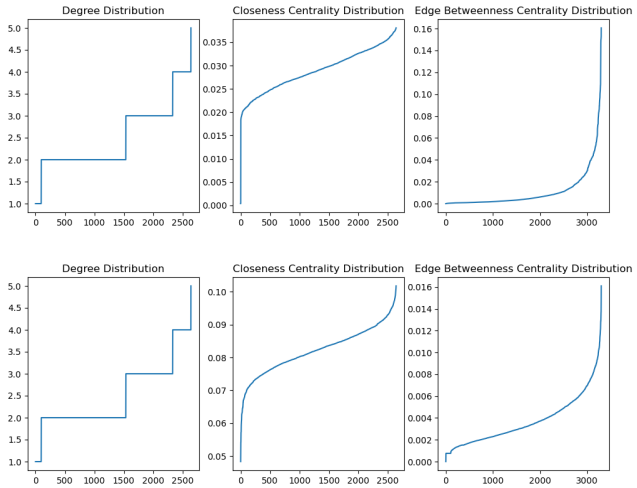
(d) Add! Your! Answer! Here!

(e) Add! Your! Answer! Here!

5. *Question 5.*

(a) My network is a Minnesota road network. Presumably, the nodes are places and the edges are roads connecting them. There isn't much other information about this network on networkrepository.

(b) The degree distribution of my Minnesota road graph is quite restricted, ranging from 1 to 5. Closeness centrality ranges mostly from 0.02 to 0.035, with fairly even distribution between those points. Edge betweenness centrality ranges from 0 to 0.16. Most values are low. The degree distribution of my random graph is the same, but closeness centrality has a larger range, from 0.05 to 0.10. Edge betweenness centrality has a much smaller range, from 0 to 0.016. Within that range, the values are slightly more distributed than in the Minnesota road graph.



(c) This exercise is useful because it provides insight into what sorts of attribute distributions we are likely to find by chance and which we are not. If an attribute distribution differs significantly from that of a null model, this may indicate some sort of underlying network organization scheme, such as community structure.

6. *(Optional) Question 6 (5 bonus points).*

For every problem set this semester, there will be an opportunity to earn extra credit. Since this is a new course, we are eager to receive genuine feedback on the material, pace, instruction style, etc. of the course so far.

I think the course has gone well so far. My experience of the class may be different from others due to my lack of knowledge of network science. There are some things I am able to grasp and others that go over my head.

I would personally always prefer shorter homework, but I understand that, without tests, homework is the only way to gain further familiarity with the material.

If you reference anything, add it `main.bib` and cite using `\cite{Hartle2020wegd}`.