MATH3090U: Netwrok Science
Assignment 3

Syed Naqvi
100590852

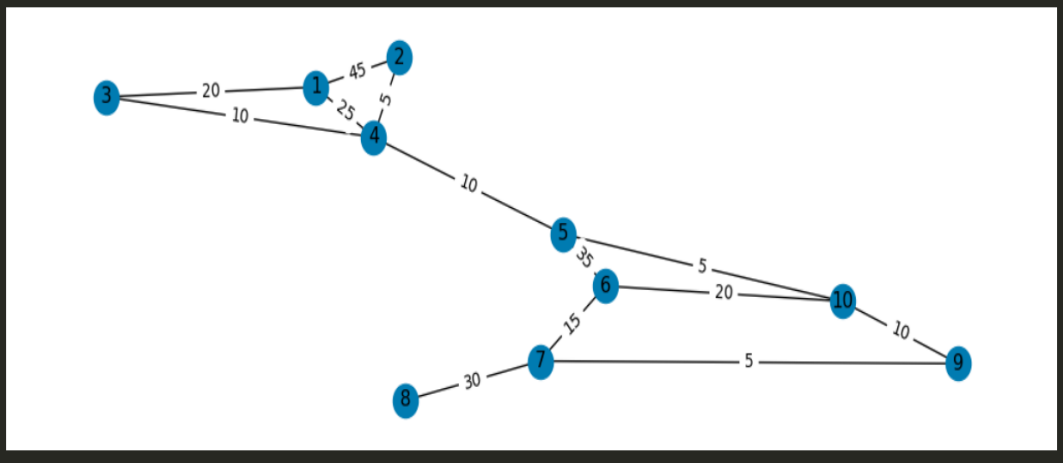
November 9, 2023

1.

I will begin by recreating the given graph using networkx:

```python
# Create an empty undirected graph
G = nx.Graph()
# Add edges along with their weights (each tuple represents an edge: (node1, node2, weight))
edges = [(1, 2, 45), (1, 3, 20), (1, 4, 25), (2, 4, 5), (3, 4, 10),
         (4, 5, 10), (5, 6, 35), (5, 10, 5), (6, 10, 20), (6, 7, 15), (7, 8, 30),
         (7, 9, 5), (9, 10, 10)]
plt.figure(figsize=(10,3))
# Add all edges to the graph at once
G.add_weighted_edges_from(edges)
# Draw the graph (with labels)
pos = nx.spring_layout(G)  # positions for all nodes
nx.draw(G, pos, with_labels=True)
# Draw the edge labels (weights)
edge_labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
plt.show()
```

```
90]  ✓ 0.0s
```



(a) The strength of a node in this network represents the total time a child has spent in close contact with other children.

(b) The node with the maximum strength is node 1 with a total incident edge weigth of 90.
The node with the minimum strength is node 9 with total incident edge weight of 15.

```python
# This function iterates through all the nodes in the network
# It then sums the weights of all incident edges for each node
# if maximum=True, return maximum of all such sums and a list of corresponding nodes
# if maximum=False, return minmum of all such sums and list of corresponding nodes
def  node_strength(G, maximum=True):

    # Get the nodes and their strengths
    strengths = G.degree(weight='weight')

    if(maximum == True):
        # Find the node(s) with the maximum strength
        max_strength = max(strengths, key=lambda x: x[1])[1]
        max_nodes = [node for node, strength in strengths if strength == max_strength]
        return (max_strength, max_nodes)

    else:
        # Find the node(s) with the minimum strength
        min_strength = min(strengths, key=lambda x: x[1])[1]
        min_nodes = [node for node, strength in strengths if strength == min_strength]
        return (min_strength, min_nodes)

print('maximum node strength: ', node_strength(G,maximum=True)[0],
      ' node(s) with maximum strength: ', node_strength(G,maximum=True)[1])
print('minimum node strength: ', node_strength(G,maximum=False)[0],
      ' node(s) with minimum strength: ', node_strength(G,maximum=False)[1])
```

[35] ✓ 0.0s

```
maximum node strength:  90  node(s) with maximum strength:  [1]
minimum node strength:  15  node(s) with minimum strength:  [9]
```

(c) **I disagree with this statement.**
If we conclude that the highest node strength indicates the individual with the highest chance of getting sick then we are assuming that total time spent with others is the biggest risk factor in catching an illness. However, I would argue that it is not the total time spent with other people, but the actual **number** of different people with whom the time is being spent that is the biggest risk factor as this increases possible exposures to the illness.
That being said, the raw node strength does not indicate the amount of different people (possible carriers of a sickness) an individual has been in close contact with since a high node strength could mean lots of time spent with a single person or little time spent with many people. Take the example of node 1 with node strength 90 and node 4 with node strength 50. Node 4 has a much lower node strength than node 1 yet has a higher unweighted degree meaning the child represented by node 4 has had more opportunities than the child represented by node 1 to catch the sickness from other children.

(d) We can use networkx to calculate the closeness centrality of all nodes diregarding weights:

```
[92]  ✓ 0.0s
```

```
import copy
H = copy.deepcopy(G)
#removing weights
for n in H.edges(data=True):
    if 'weight' in n[2]:
        del n[2]['weight']
for node in H.nodes():
    print('Node: ', node,
          ' Closeness Centrality: ', nx.closeness_centrality(G, node))
```

```
Node:  1  Closeness Centrality:  0.375
Node:  2  Closeness Centrality:  0.36
Node:  3  Closeness Centrality:  0.36
Node:  4  Closeness Centrality:  0.5
Node:  5  Closeness Centrality:  0.5625
Node:  6  Closeness Centrality:  0.5
Node:  10  Closeness Centrality:  0.47368421052631576
Node:  7  Closeness Centrality:  0.4090909090909091
Node:  8  Closeness Centrality:  0.3
Node:  9  Closeness Centrality:  0.391304347826087
```

(e) The path of fewest edges between nodes 3 and 5 is the sequence 3, 4, 5, 6. The weighted distance of this path is the sum of the weights of each edge in the sequence $10 + 10 + 35 = 55$.
$\therefore$ the weighted distance of the path of fewest edges between nodes 3 and 6 is 55.

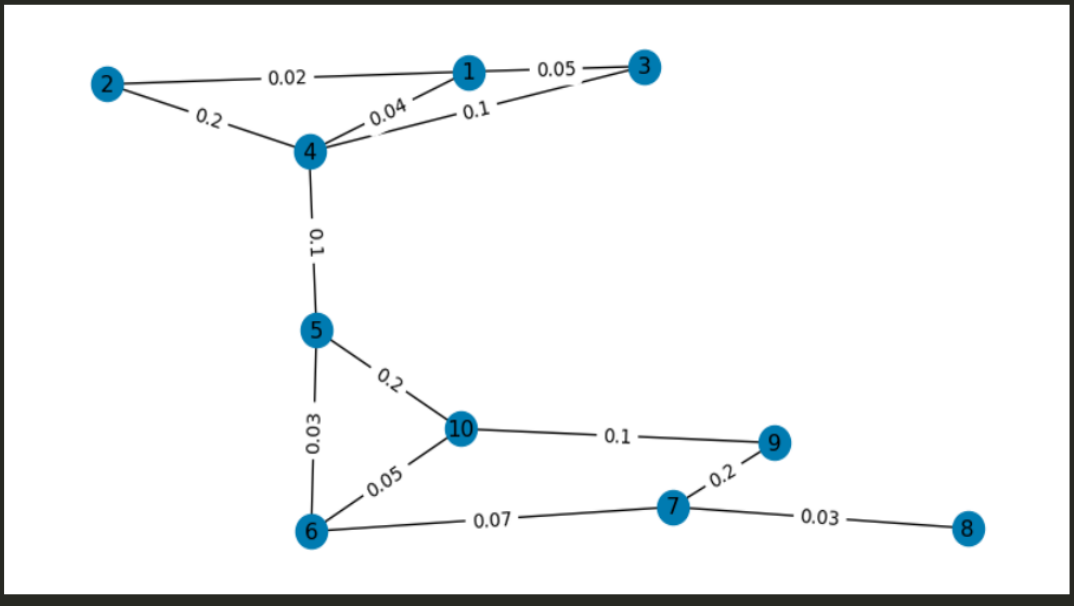
(f) The path with fewest edges between nodes 3 and 6 is P = [3, 4, 5, 6] with a total weight of 55. If we interpret weight as the length of an edge then the path Q = [3, 4, 5, 10, 6] has more edges but has a total distance (sum of all path lengths) of 45 which is less than 55 and so we would say that Q has a shorter length than P.

(g) The closeness centrailty of a node is the reciprocal of the average of the shortest paths between the given node and every other node in the network. Using weight of an edge as the distance between adjacent nodes means the shortest path is the one of least total weight instead of fewest edges. However, given that weight in our child network is the time two children have spent in close proximity, we can see that it is more sensible for a larger weight to represent a stronger relationship or higher "closeness" between two nodes. Thus, using weighted paths in the closeness centrality formula would actually require us to find the weakest or "least close" connections between two nodes, providing no inforamtion of a node's closeness to other nodes since we will have excluded the true closest paths from our calculations. Therefore using the weighted distance with our network's definition of weight in the closeness centrality formula would not be very useful.

(h) We can use networkx to create a deepcopy of our original weighted network and then set each weight to its reciprocal. We can then print this graph and find the shortest distance between nodes 1 and 5 in terms of cost as our new weight assignment:

```
J = copy.deepcopy(G)
for u, v, d in J.edges(data=True):
    d['weight'] = round((1/d['weight']),2)
plt.figure(figsize=(8,4))
# Draw the graph (with labels)
pos = nx.spring_layout(J)  # positions for all nodes
nx.draw(G, pos, with_labels=True)
# Draw the edge labels (weights)
edge_labels = nx.get_edge_attributes(J, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=edge_labels)
plt.show()
```

✓ 0.0s



As we can see from the graph, the sequence 1, 4, 5 provides a shortest distance of 0.14 between nodes 1 and 5.

(i) Having reproduced the graph with the weights being set to their reciprocals in the previous question, we can now find the nodes most central by closeness centrality.

```
centralities = nx.closeness_centrality(J,distance='weight')
centralities = sorted(centralities.items(), key=lambda x: x[1], reverse=True)

print('Closeness centralities of nodes in descending order:')
for i in centralities:
    print("Node: ", i[0], 'Closeness Centrailty: ', i[1])
```

[121] ✓ 0.0s

```
Closeness centralities of nodes in descending order:
Node:  5 Closeness Centrailty:  8.108108108108107
Node:  6 Closeness Centrailty:  8.108108108108107
Node:  4 Closeness Centrailty:  6.870229007633587
Node:  10 Closeness Centrailty:  6.382978723404256
Node:  1 Closeness Centrailty:  6.1224489795918355
Node:  7 Closeness Centrailty:  5.960264900662252
Node:  2 Closeness Centrailty:  5.521472392638037
Node:  8 Closeness Centrailty:  5.142857142857142
Node:  3 Closeness Centrailty:  4.81283422459893
Node:  9 Closeness Centrailty:  4.147465437788018
```

(j) The closeness centralities of nodes 5 and 6 are highest. This means that nodes 5 and 6 have the most paths of least resistance (lowest total cost, highest overall weights per edge) connecting them to every other node in the network. In the context of our network where total time spent with other children is the main disease spreading risk factor, these closeness centrailty results suggest a sickness will have the most amount of time to spread **from** every node in the network **to** nodes 5 and 6 as well as **from** nodes 5 and 6 **to** every other node in the network. This is due to interaction times being longest between nodes along paths where nodes 5 or 6 are endpoints.

2.

(a) Suppose we are given two degree sequences to apply to a network G where $|V(G)| = n$ and initially, $|E(G)| = 0$. The first sequence $I = [a_1, a_2, \dots a_k]$ consisits of *indegrees* while the second sequence $O = [b_1, b_2, \dots b_r]$ consists of *outdegrees*.

First we go through our network G and arbitrarily select nodes n_i, $n_j \in G$ where it may or may not be the case that $n_i = n_j$. We attach a_i inward-directed half-edges to n_i and b_j outward-directed half-edges to n_j. Once this process is complete, there will always exist nodes n_i, $n_j \in G$ such that $\forall a_i \in I$ and $\forall b_j \in O$, $indegree(n_i) = a_i$ and $outdegree(n_j) = b_j$.

Next, we arbitrarily connect the outward-directed half-edges to the inward-directed half-edges until there are no more half-edges remaining. This leaves us with a random directed graph, generated using an adaptation of the configuration model. Note that this process permits the existence of un-conneceted nodes, self-loops and multi-edges.

We know that for directed graphs, the total *indegree* must equal the total *outdegree*. This means that degree sequences I and O should be defined so that $\sum_i^k a_i = \sum_j^r b_j$ as this will ensure all inward-directed half-edges have a corresponding outward-directed half-edge, leaving no "dangling" edges. Another requirement that must be met for this algorithm to work is that $n \geq max(|I|, |O|)$. This is so that there are enough nodes to which we can apply all the degrees in the lengthier of sequences I and O.

3.

First we create 100 $G(n, p)$ graphs with $n = 60$ and $p = (1/25)$. We then calculate the required graph statistics for each graph and store the values in a list for later analysis.

```
# Creating 100 G(n,p) random graphs for some fixed n, p values
# and storing the number of edges, avg degree, transitivity and
# diameter of each graph in a list of tuples

n = 60
p = (1/25)
graphs = []

for i in range (1,101):
    # creating a new random G(n,p) graph
    G = nx.gnp_random_graph(n,p)
    # storing the number of edges
    edges = len(G.edges)
    # storing the average degree of each graph
    avg_deg = sum(dict(G.degree()).values())/len(G)
    # storing the transitivity of each graph
    transitivity = nx.transitivity(G)
    # storing the maximum of the diameters of each component of the graph
    diameter = max([nx.diameter(G.subgraph(c)) for c in nx.connected_components(G)])
    # appending all graph statistics to our tuple list
    graphs.append((edges, avg_deg, transitivity, diameter))
```

[340] ✓ 0.7s

(a) Comparing the average values with each parameter we see that they are almost identical. The expected diameter is also of the correct order of magnitude.

```
cal_avg_edges = round(sum([graphs[i][0] for i in range(0, len(graphs))]) / len(graphs),2)
print('expected number of edges: ', round(((n*(n-1))/2)*p,2),
      ', calculated average number of edges: ', cal_avg_edges)

cal_avg_deg = round(sum([graphs[i][1] for i in range(0, len(graphs))]) / len(graphs),2)
print('expected average degree: ', round(((n-1)*p),2),
      ', calculated average degree: ', cal_avg_deg)

cal_avg_transitvity = round(sum([graphs[i][2] for i in range(0, len(graphs))]) / len(graphs),2)
print('expected transitivity: ', round(p,2),
      ', calculated average transitivity: ', cal_avg_transitvity)

cal_avg_diameter = round(sum([graphs[i][3] for i in range(0, len(graphs))]) / len(graphs),2)
print(f'expected diameter is on the order of ln({n}): ', round(math.log(n),2),
      ', calculated average diameter: ', cal_avg_diameter)
```

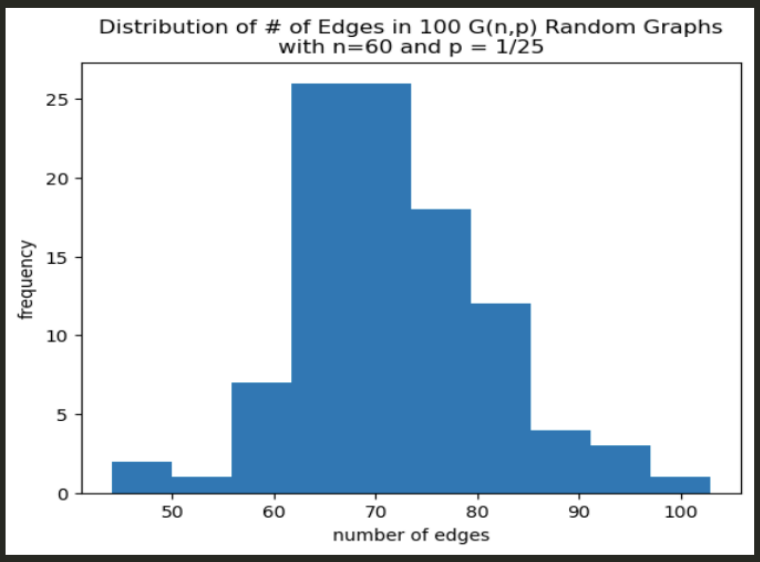
[349] ✓ 0.0s

```
expected number of edges:  70.8 , calculated average number of edges:  71.64
expected average degree:  2.36 , calculated average degree:  2.39
expected transitivity:  0.04 , calculated average transitivity:  0.04
expected diameter is on the order of ln(60):  4.09 , calculated average diameter:  9.84
```

(b) We can see that the number of edges and the average degree are both roughly normally distributed, with most graphs having average degree and number of edges near our earlier calculated values. As we move towards the tails, there are fewer graphs.

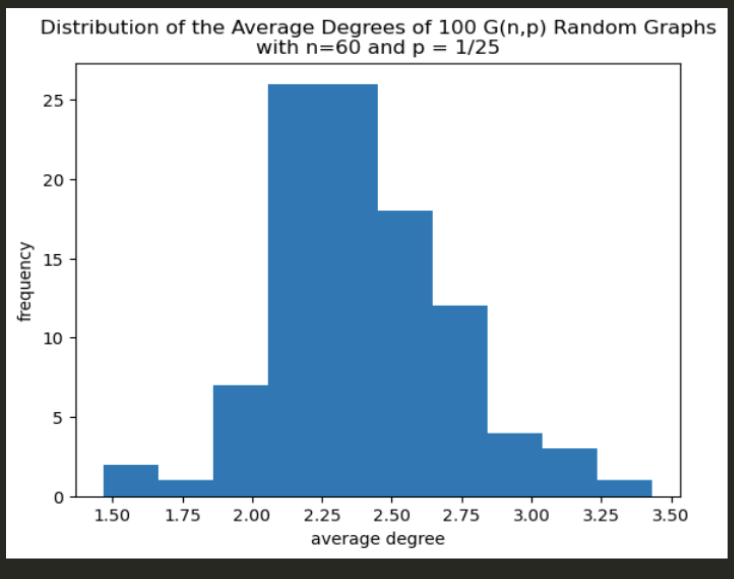
```
# histogram plotting the distrbution of the number of edges
plt.hist(y_E)
plt.xlabel('number of edges')
plt.ylabel('frequency')
plt.title('Distribution of # of Edges in 100 G(n,p) Random Graphs\nwith n=60 and p = 1/25')
plt.show()
```

[343] ✓ 0.1s




```
# histogram plotting the distrbution of the average degrees
plt.hist(y_AD)
plt.xlabel('average degree')
plt.ylabel('frequency')
plt.title('Distribution of the Average Degrees of 100 G(n,p) Random Graphs\nwith n=60 and p = 1/25')
plt.show()
```

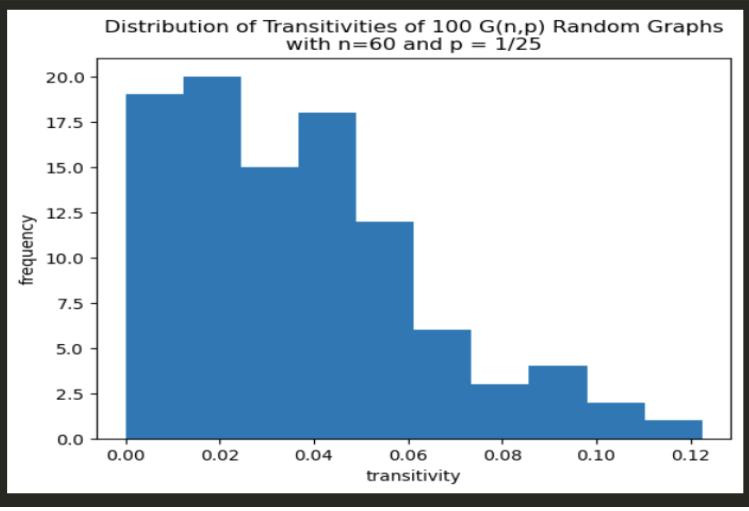
[344] ✓ 0.1s



The distribution of transitivities exists within a range of 0 to just above 0.1 indicating that the probabiltiy of forming trinagles within these graphs is relatively low which makes sense given the low p value. The graph is right skewed suggesting that the number of graphs decreases as the transitivity increases. This is again as expected because the probability of there being an edge between **any** pair of nodes, let alone those with a common neighbor is low: 1/25. We can also see that there is a large number of graphs with a transitivity around 0.2 to 0.4 which does roughly align with our calculated and expected transitivity averges of 0.4.

```
# histogram plotting the distrbution of the transitivities
plt.hist(y_T)
plt.xlabel('transitivity')
plt.ylabel('frequency')
plt.title('Distribution of Transitivities of 100 G(n,p) Random Graphs\nwith n=60 and p = 1/25')
plt.show()
```

[345] ✓ 0.1s



By convention, we considered the largest of the diameters of each connected component as the diameter of the random graphs. With this definition, we see that the diameter distribution seems roughly normally distributed, with a slightly right skew. It seems that the majority of the random graphs had a diameter close to 10 with a non-insignificant amount having diameter above above 10. This shows the random nature of the graphs and the rough diameter predition being on the order of ln(n) which these results do seem to to demonstrate.

```
# histogram plotting the distrbution of the diameters
plt.hist(y_D)
plt.xlabel('diameter')
plt.ylabel('frequency')
plt.title('Distribution of Diameters of 100 G(n,p) Random Graphs\nwith n=60 and p = 1/25')
plt.show()
```

[346] ✓ 0.1s