Problem 1.

1. Erdos-Renyi graph

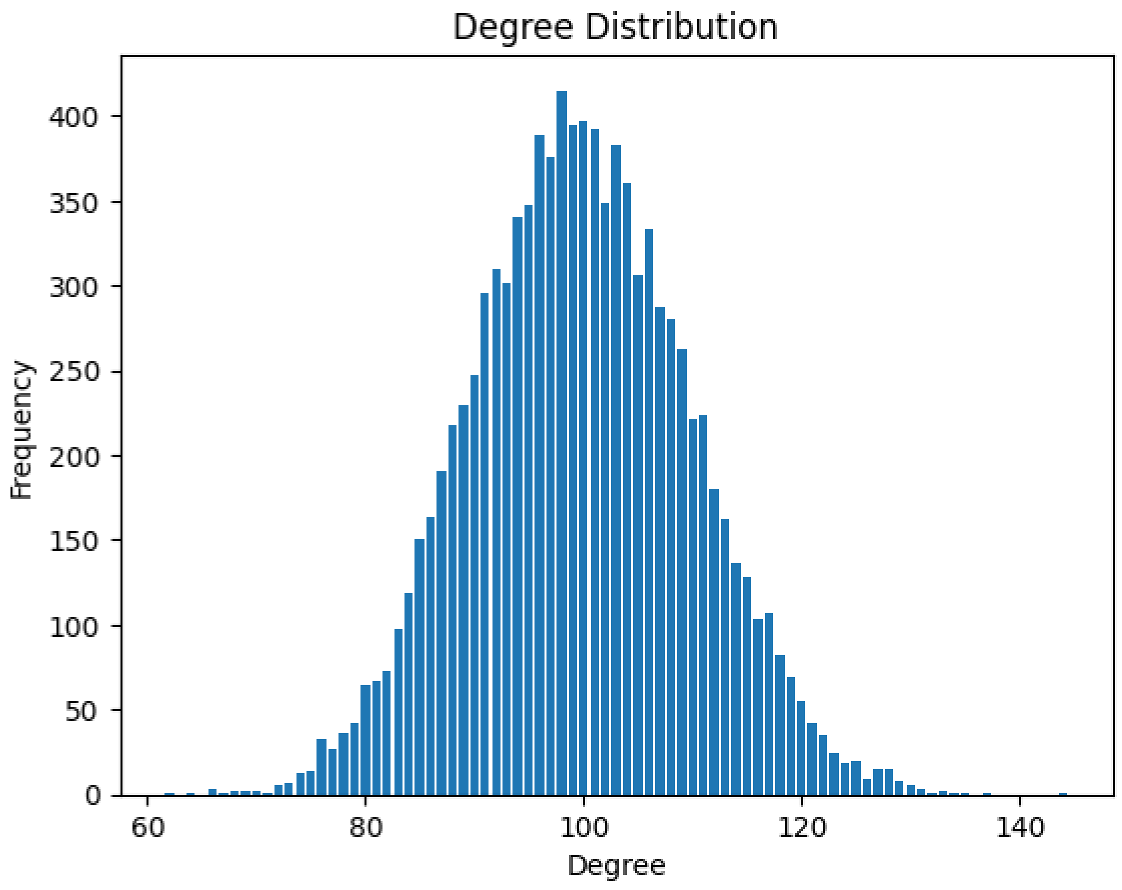
```
import networkx as nx
import matplotlib.pyplot as plt
import random as rand
import multiprocessing
from tqdm import tqdm
```

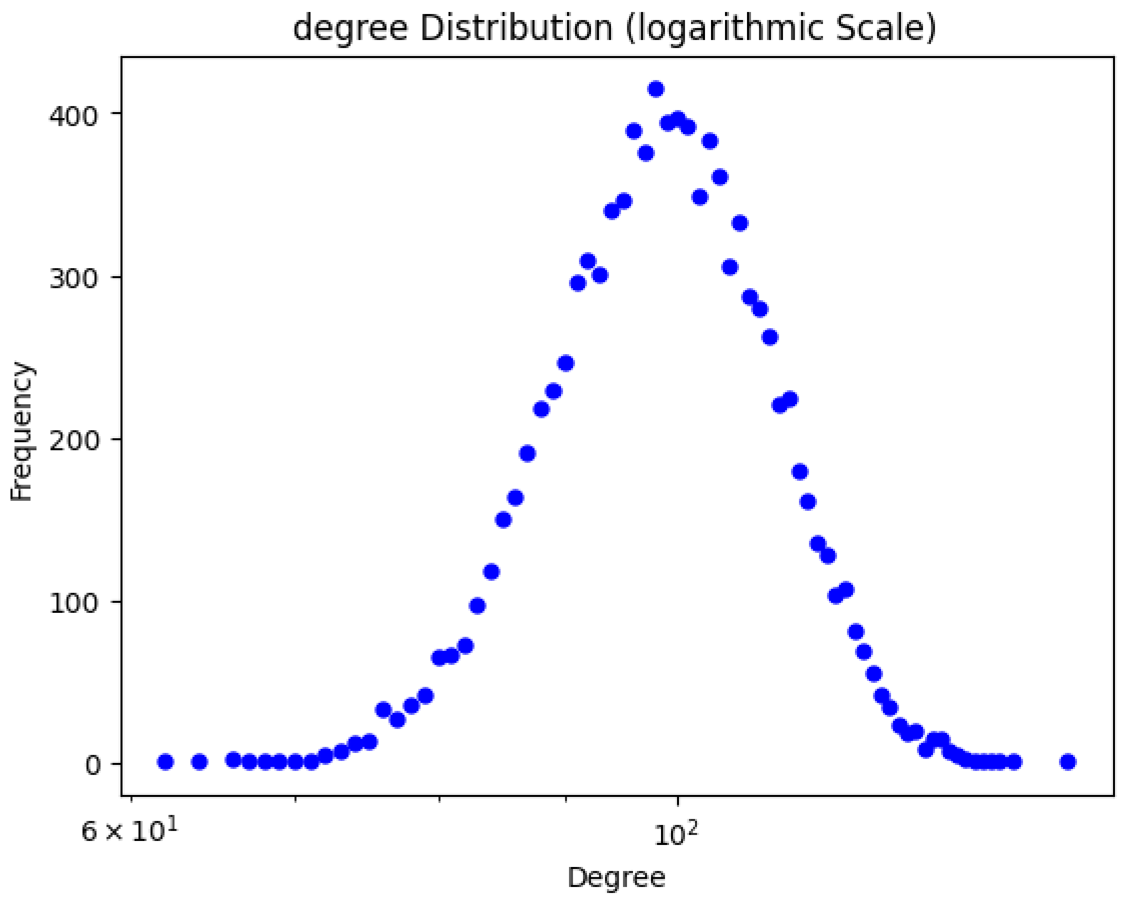
```
class Graph:
    def __init__(self):
        self.nodes = {}
        self.edges = {}

    def add_node(self, node):
        if node not in self.nodes:
            self.nodes[node] = set()

    def add_edge(self, node1, node2):
        if node1 in self.nodes and node2 in self.nodes:
            self.edges[(node1, node2)] = 1
            self.edges[(node2, node1)] = 1
            self.nodes[node1].add(node2)
            self.nodes[node2].add(node1)

    '''
    The easiest property to study is the degree distribution. A given node v is incident with
    n − 1 potential edges, and each of them exists with probability p independently of each other.
    '''
    '''
    in Section 3.2 we mentioned
    that in social networks we observe heavy-tailed degree distributions. This shows one of the
    main problems of the Erd˝os-R´enyi random graph model for modeling social networks. In other
    words in the Erd˝os-R´enyi model the nodes are very similar
    '''


    def compute_degree_distribution(self):
        degree_counts = {}
        for node in self.nodes:
            degree = len(self.nodes[node])
            degree_counts[degree] = degree_counts.get(degree, 0) + 1
        return degree_counts

    def compute_diameter(self):

    '''
    def compute_diameter(self):
        # Use breadth-first search to find the maximum shortest path length
        max_path_length = 0
        for node in self.nodes:
            queue = [(node, 0)]
            visited = set([node])

            while queue:
                current_node, path_length = queue.pop(0)
                max_path_length = max(max_path_length, path_length)

                for neighbor in self.nodes[current_node]:
                    if neighbor not in visited:
                        visited.add(neighbor)
                        queue.append((neighbor, path_length + 1))

        return max_path_length
    '''
    def compute_clustering_coefficient(self):
        if len(self.nodes) == 0:
            return 0.0  # or return a default value

        pool = multiprocessing.Pool()

        total_clustering_coefficient = 0

        nodes = list(self.nodes.keys())

        # Use multiprocessing.Pool.map to apply the compute_local_clustering_coefficient function to each node
        local_cc = pool.map(self.compute_local_clustering_coefficient, nodes)

        # Compute the total clustering coefficient
        total_clustering_coefficient = sum(local_cc)

        return total_clustering_coefficient / len(self.nodes)

    def compute_local_clustering_coefficient(self, node):
        neighbors = self.nodes[node]
        num_neighbors = len(neighbors)
        if num_neighbors < 2:
            return 0.0

        num_edges = 0
        for neighbor1 in neighbors:
            for neighbor2 in neighbors:
                if neighbor1 != neighbor2 and neighbor1 in self.nodes[neighbor2]:
                    num_edges += 1

        local_clustering_coefficient = num_edges / (num_neighbors * (num_neighbors - 1))
        return local_clustering_coefficient

    def plot_degree_distribution(self):
        degree_dist = self.compute_degree_distribution()
        degrees = list(degree_dist.keys())
        frequencies = list(degree_dist.values())

        plt.bar(degrees, frequencies)
        plt.xlabel("Degree")
        plt.ylabel("Frequency")
        plt.title("Degree Distribution")
        plt.show()
```

```python

        plt.loglog(degrees, frequencies, 'bo', markersize=5)
        plt.xlabel("Degree")
        plt.ylabel("Frequency")
        plt.title("degree Distribution (logarithmic Scale)")
        plt.show()

    def plot_graph(self):
        G = nx.Graph()
        G.add_nodes_from(self.nodes.keys())
        G.add_edges_from(self.edges.keys())

        pos = nx.spring_layout(G)
        nx.draw(G, pos, with_labels=True)
        plt.show()

class ErdosRenyiGraph(Graph):
    def generate_er_graph(self, n, p):
        self.nodes = {}
        self.edges = {}

        self.add_node(0)  # Add the first node

        for i in range(1, n):
            self.add_node(i)  # Add nodes from 1 to n-1

            for j in range(i):
                if rand.random() < p:  # Add an edge with probability p
                    self.add_edge(i, j)

class WattsStrogatzGraph(Graph):
    def generate_ws_graph(self, n, k, beta):
        self.nodes = {}
        self.edges = {}

        for node in range(n):
            self.add_node(node)

        # Create initial ring lattice
        for node in self.nodes:
            for i in range(1, k // 2 + 1):
                neighbor = (node + i) % n
                self.add_edge(node, neighbor)

        # Rewire edges
        for node in self.nodes:
            for i in range(1, k // 2 + 1):
                if rand.random() < beta:
                    neighbor = (node + i) % n
                    self.rewire_edge(node, neighbor)

    def rewire_edge(self, node, neighbor):
        # Remove existing edge
        self.remove_edge(node, neighbor)

        # Select a random node to connect
        new_neighbor = rand.choice(list(self.nodes.keys()))

        # Make sure the new neighbor is not the same as the original node or an existing neighbor
        while new_neighbor == node or new_neighbor == neighbor or new_neighbor in self.nodes[node]:
            new_neighbor = rand.choice(list(self.nodes.keys()))

        # Add the new edge
        self.add_edge(node, new_neighbor)

    def remove_edge(self, node1, node2):
        if (node1, node2) in self.edges:
            del self.edges[(node1, node2)]
            del self.edges[(node2, node1)]
            self.nodes[node1].remove(node2)
            self.nodes[node2].remove(node1)


class BarabasiAlbertGraph(Graph):
    def generate_ba_graph(self, n, l):
        self.nodes = {}
        self.edges = {}

        self.add_node(0)  # Add the first node

        for i in range(1, n):
            self.add_node(i)  # Add nodes from 1 to n-1

            # Connect the new node to l existing nodes
            selected_nodes = self.select_nodes_to_connect(l)
            for node in selected_nodes:
                self.add_edge(i, node)

    def select_nodes_to_connect(self, l):
        # Calculate the total degree of the graph
        total_degree = sum([len(neighbors) for neighbors in self.nodes.values()])

        if total_degree == 0:
            # If total_degree is zero, assign equal probabilities to all nodes
            probabilities = [1 / len(self.nodes)] * len(self.nodes)
        else:
            # Calculate the probability for each node to be selected
            probabilities = [len(neighbors) / total_degree for neighbors in self.nodes.values()]

        # Select l nodes based on the probabilities
        selected_nodes = rand.choices(list(self.nodes.keys()), weights=probabilities, k=l)
        return selected_nodes

```

```python
graph = ErdosRenyiGraph()
graph.generate_er_graph(10000, 0.01)   # Generate an Erdos-Renyi graph with 1000 nodes and edge probability 0.01 degree
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("ER Diameter:", diameter)

```

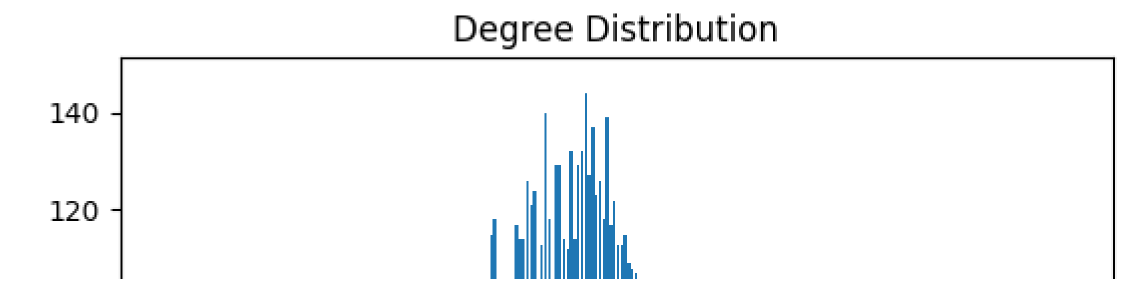
```
# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("ER Clustering Coefficient:", clustering_coefficient)


```



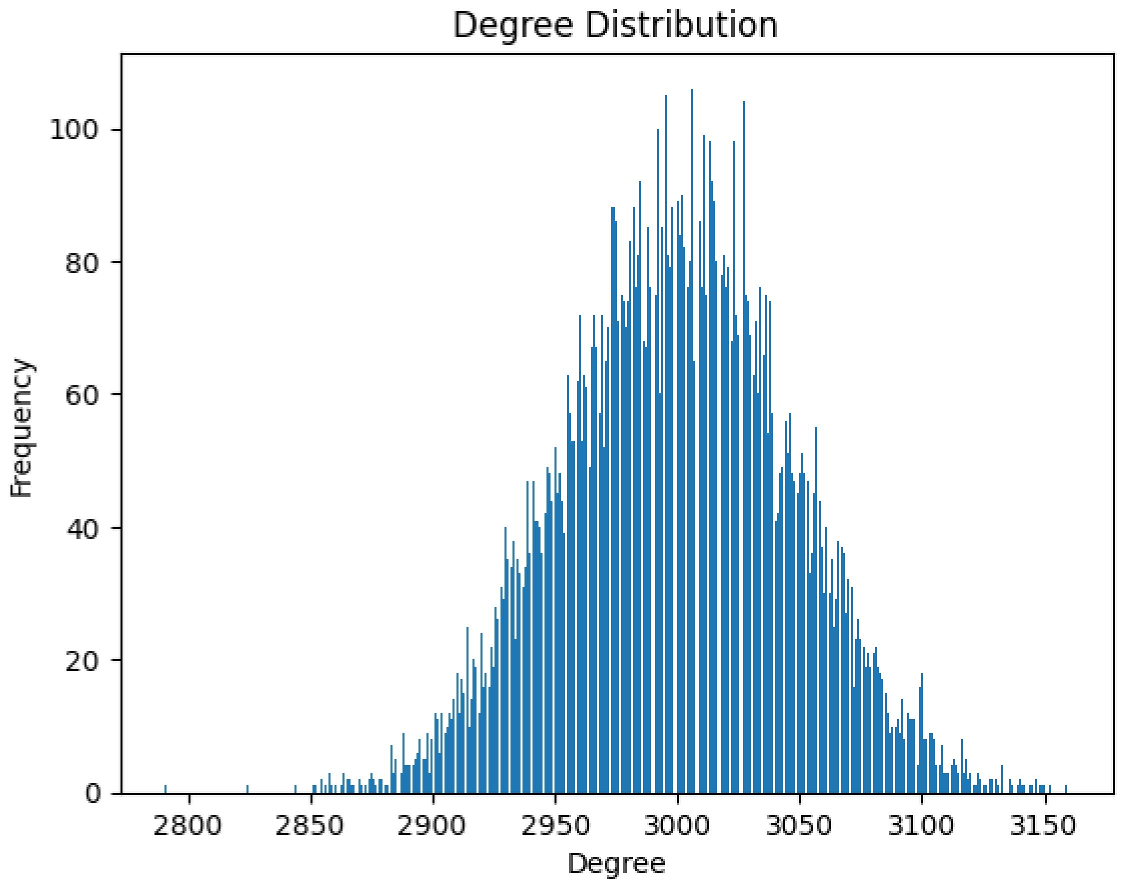


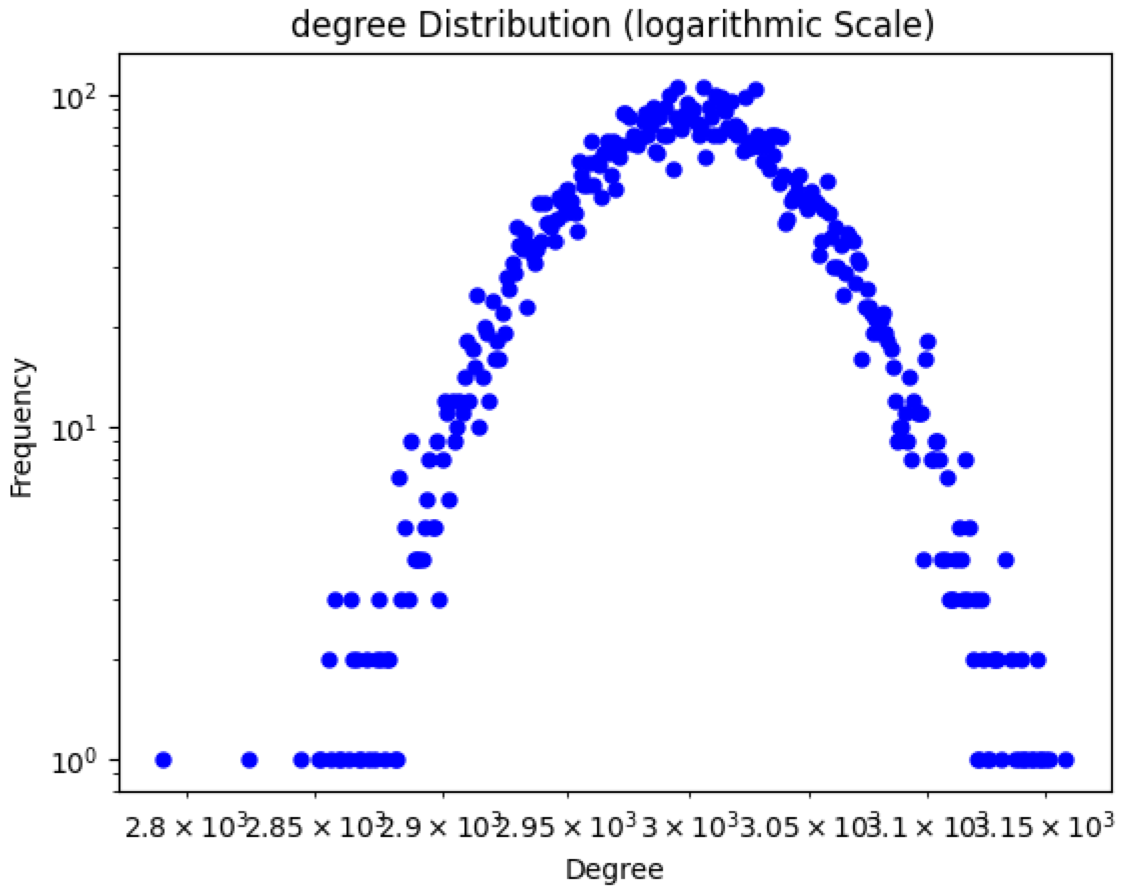

```
ER Diameter: 3
ER Clustering Coefficient: 0.00997833208934889
```

```
graph = ErdosRenyiGraph()
graph.generate_er_graph(10000, 0.11)   # Generate an Erdos-Renyi graph with 1000 nodes and edge probability 0.01 degree
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("ER Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("ER Clustering Coefficient:", clustering_coefficient)
```




```
graph = ErdosRenyiGraph()
graph.generate_er_graph(10000, 0.3)   # Generate an Erdos-Renyi graph with 1000 nodes and edge probability 0.01 degree
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("ER Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("ER Clustering Coefficient:", clustering_coefficient)
```



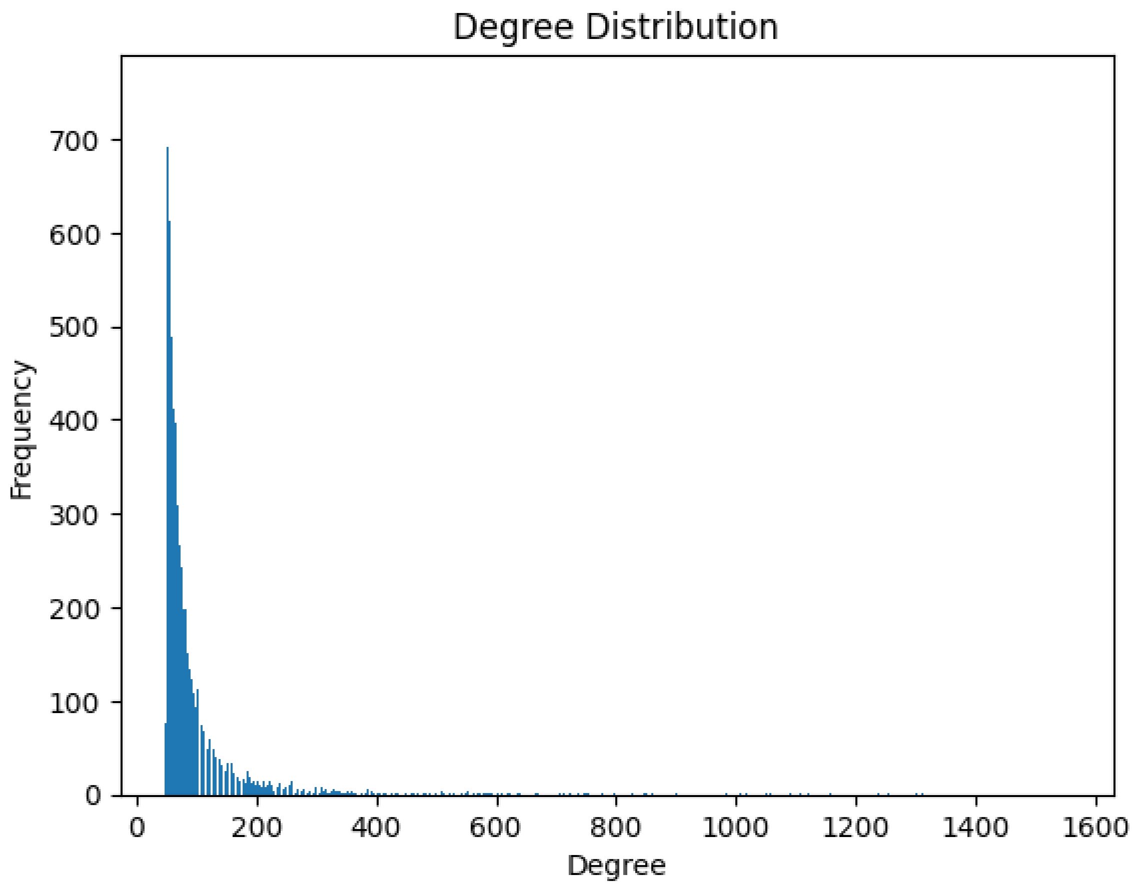


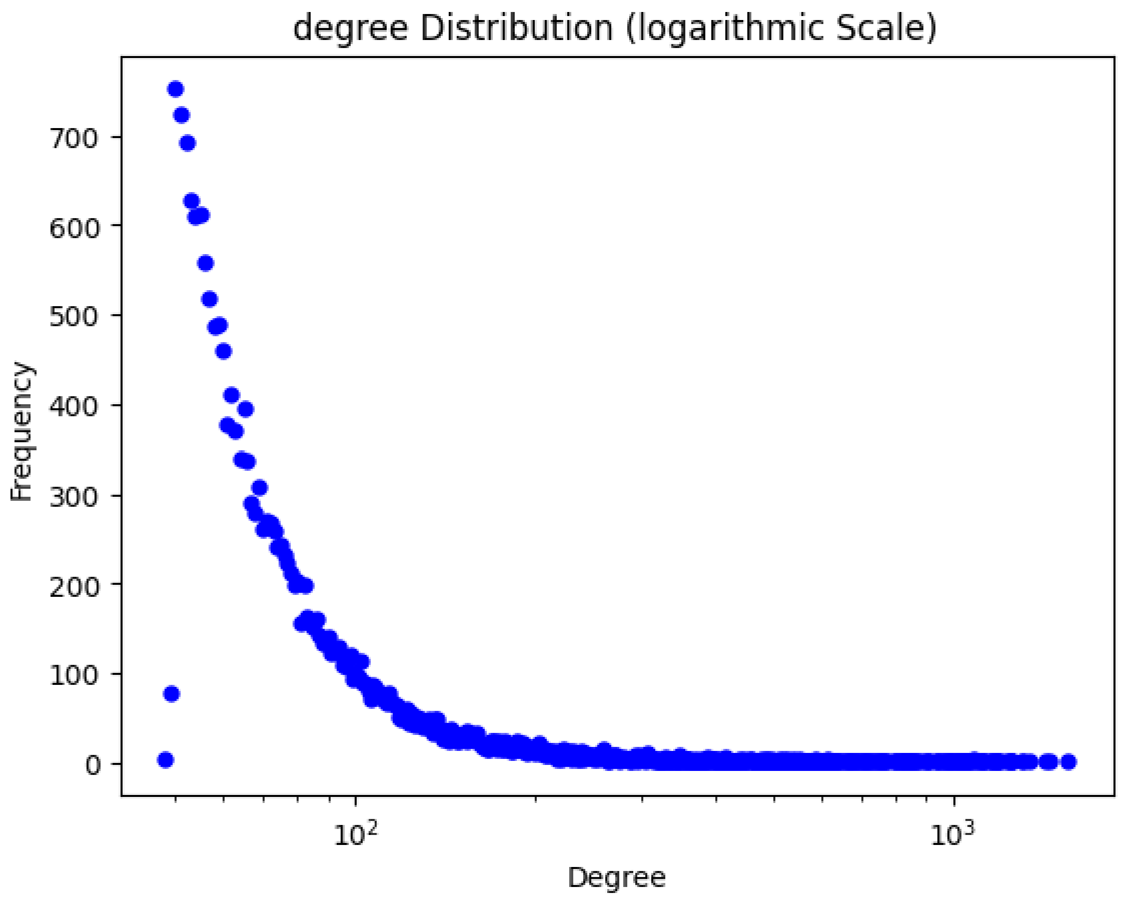

```
graph = BarabasiAlbertGraph()
graph.generate_ba_graph(10000, 5)   # Generate an
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("BA Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("BA Clustering Coefficient:", clustering_coefficient)
print("Connectedness:", graph.compute_connectedness())
print("Average Path Length:", graph.compute_average_path_length())
```

```
graph = BarabasiAlbertGraph()
graph.generate_ba_graph(20000, 5)   # Generate an
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("BA Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("BA Clustering Coefficient:", clustering_coefficient)
print("Connectedness:", graph.compute_connectedness())
print("Average Path Length:", graph.compute_average_path_length())
```

```
graph = BarabasiAlbertGraph()
graph.generate_ba_graph(20000, 50)   # Generate an
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("BA Diameter:", diameter)
```

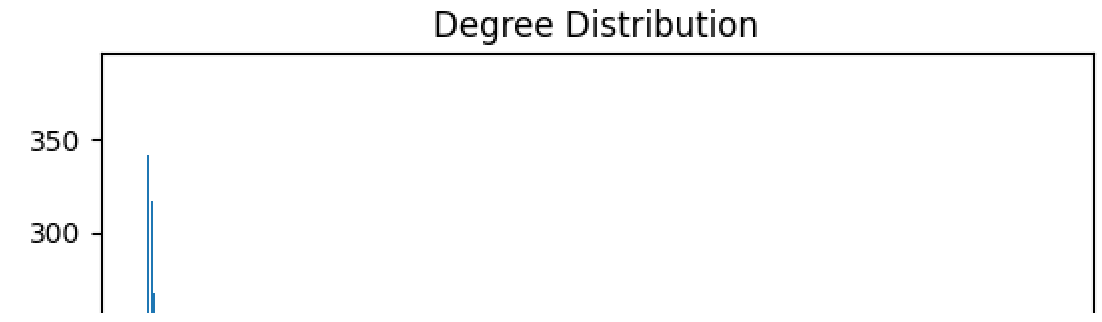
```
# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("BA Clustering Coefficient:", clustering_coefficient)


```



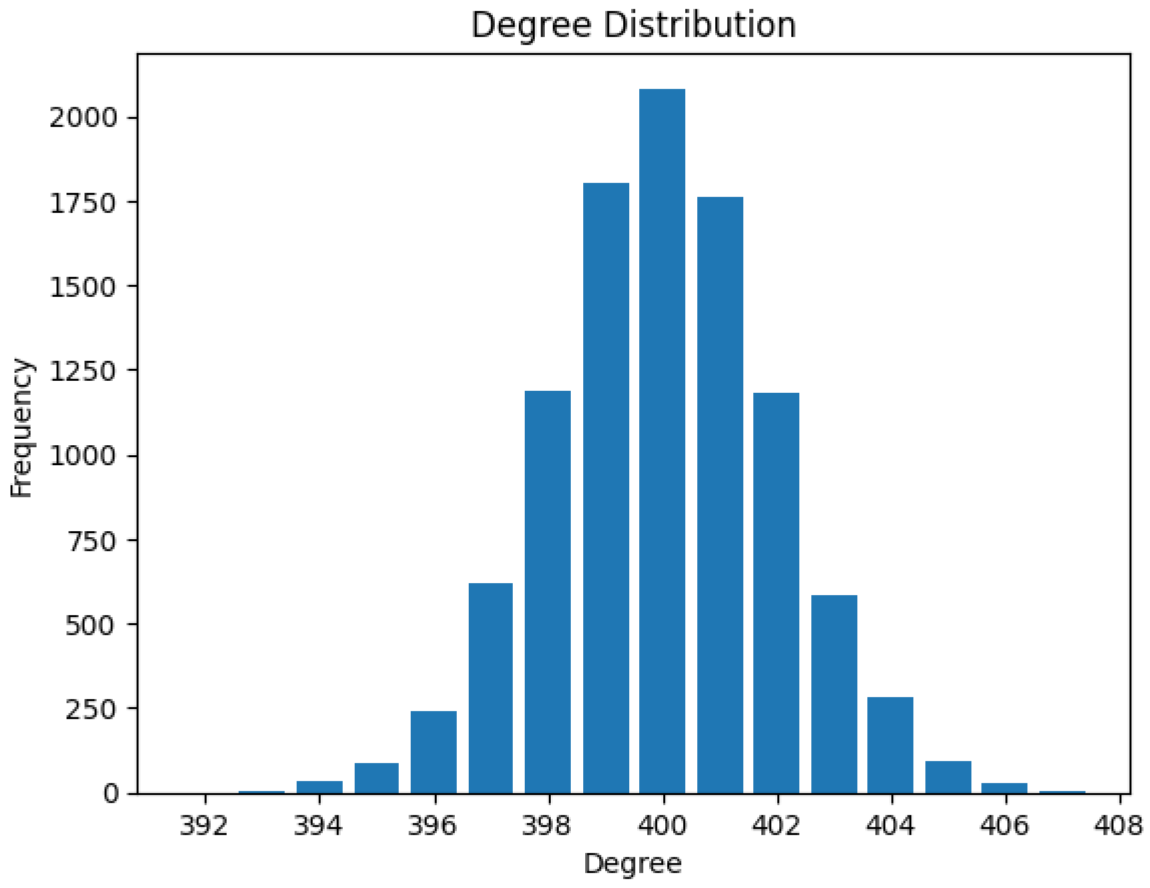


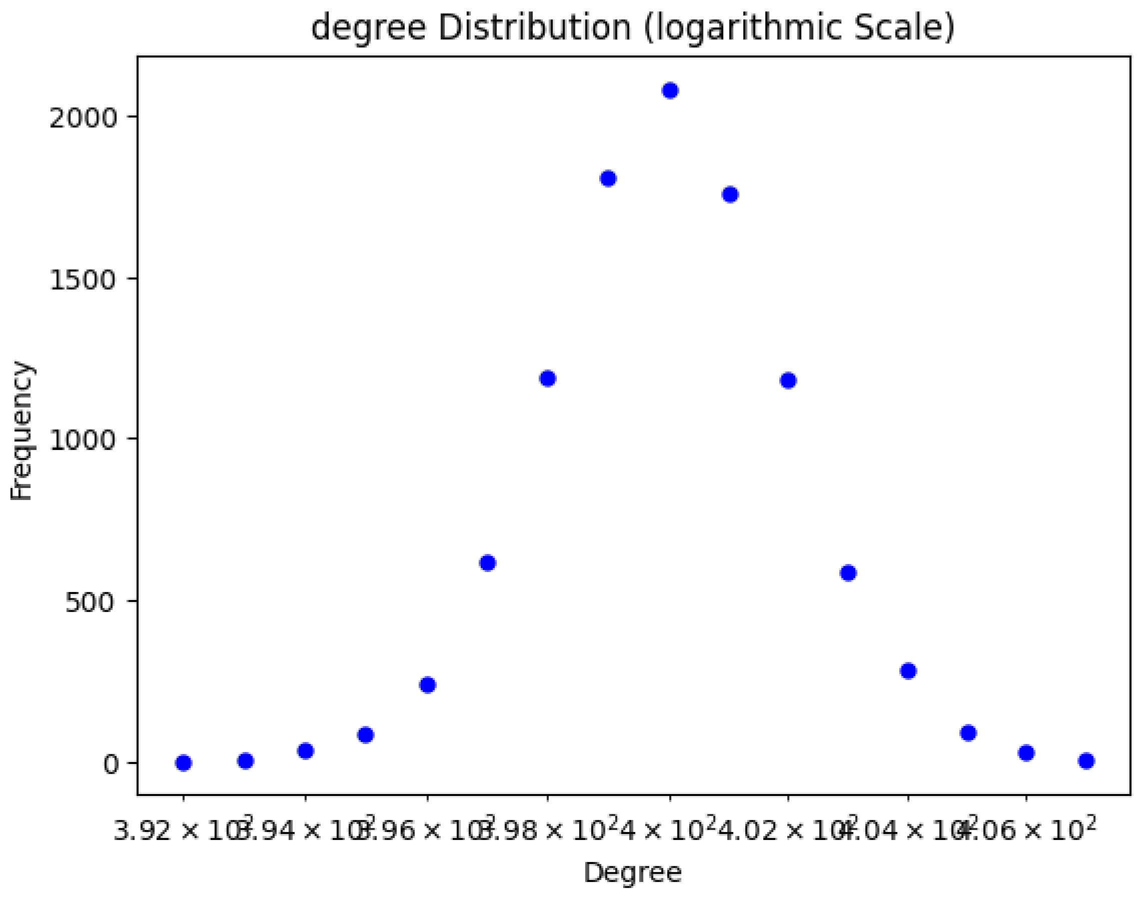

```
BA Diameter: 3
BA Clustering Coefficient: 0.019026751954177537
```

```
graph = BarabasiAlbertGraph()
graph.generate_ba_graph(20000, 100)   # Generate an
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("BA Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("BA Clustering Coefficient:", clustering_coefficient)


```




```
graph = WattsStrogatzGraph()
graph.generate_ws_graph(10000,400, 0.01)
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("WS Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("WS Clustering Coefficient:", clustering_coefficient)

#graph.plot_graph()
```

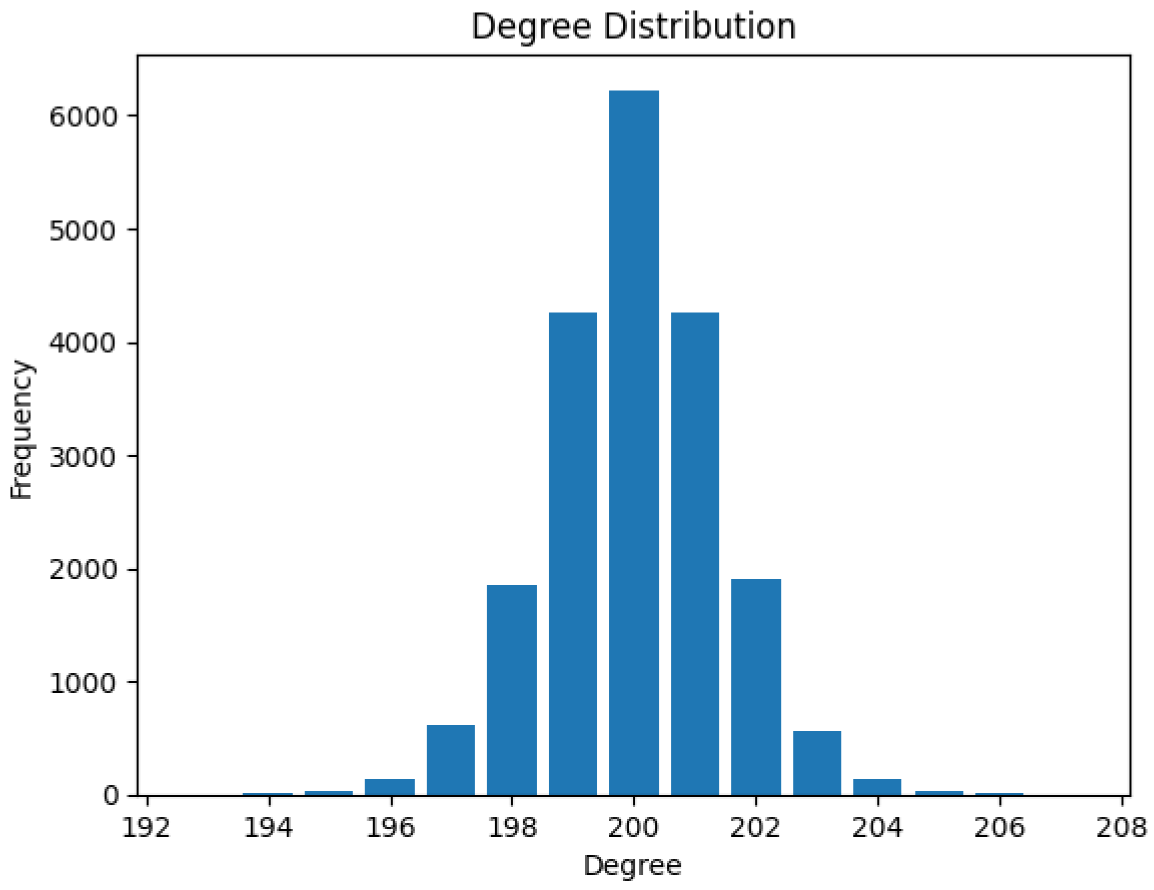


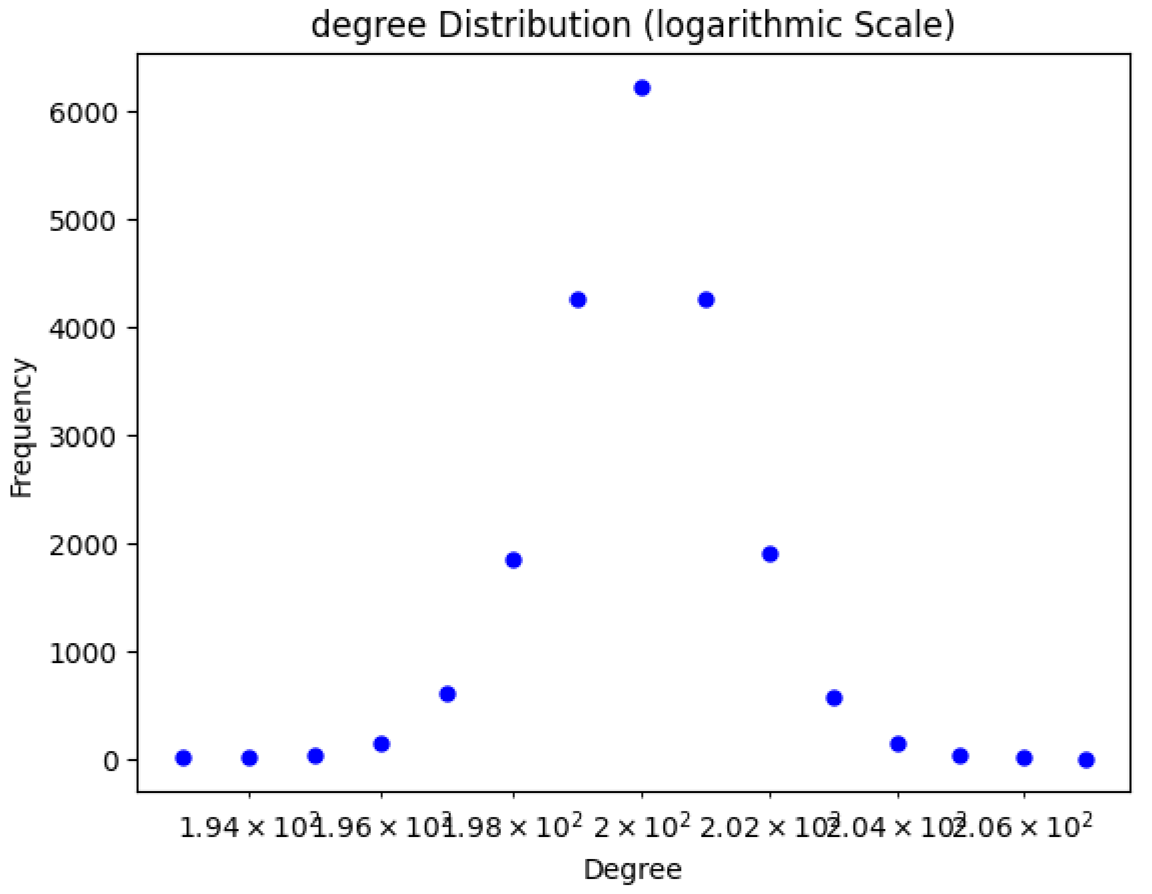



```
WS Diameter: 3
WS Clustering Coefficient: 0.7263757595527361
```

```
graph = WattsStrogatzGraph()
graph.generate_ws_graph(20000,200, 0.01)
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("WS Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("WS Clustering Coefficient:", clustering_coefficient)
```

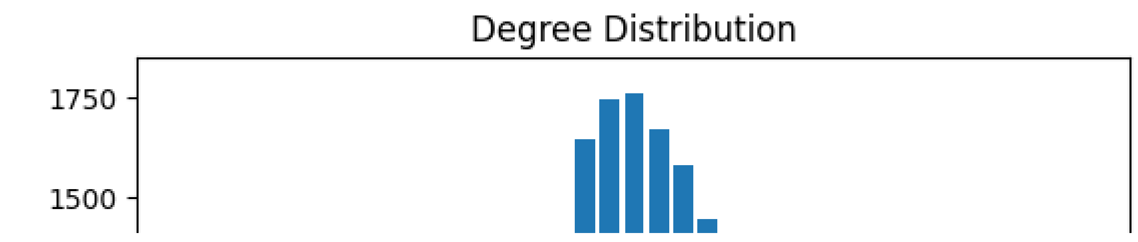
```

#graph.plot_graph()
```



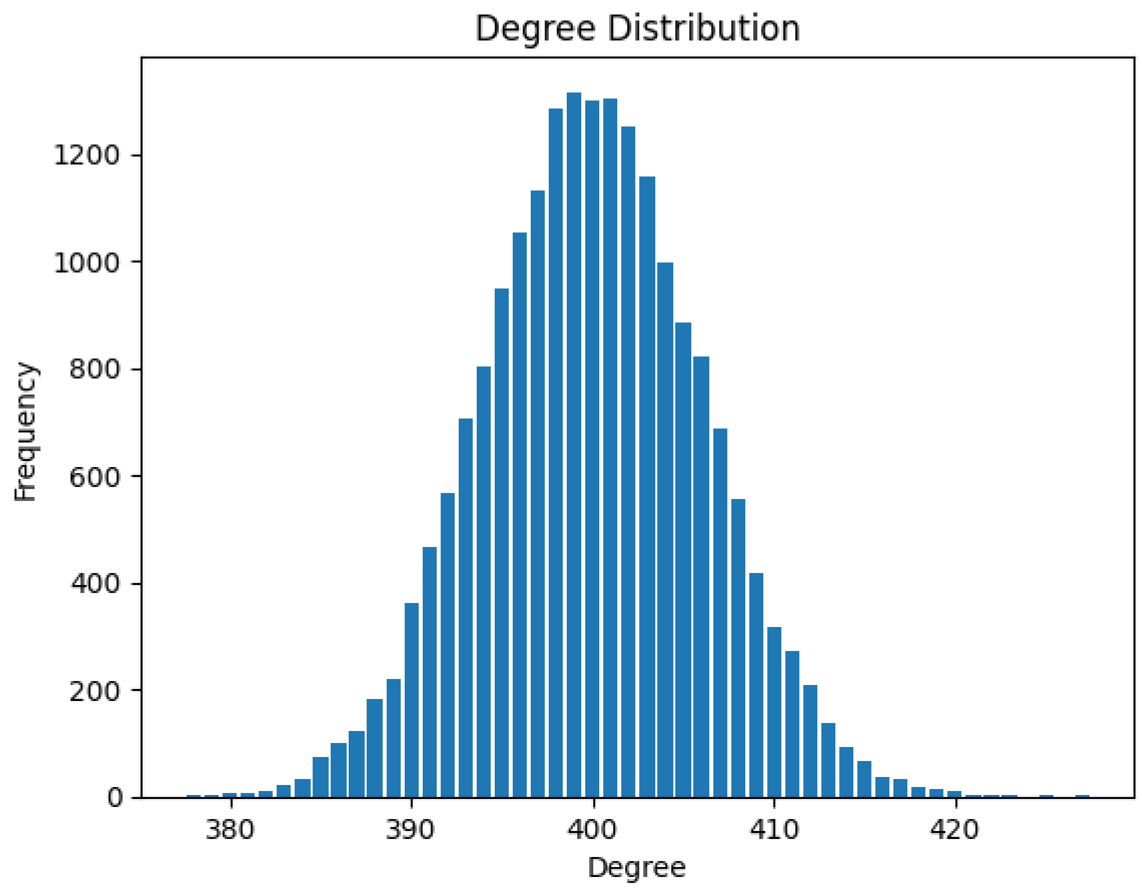


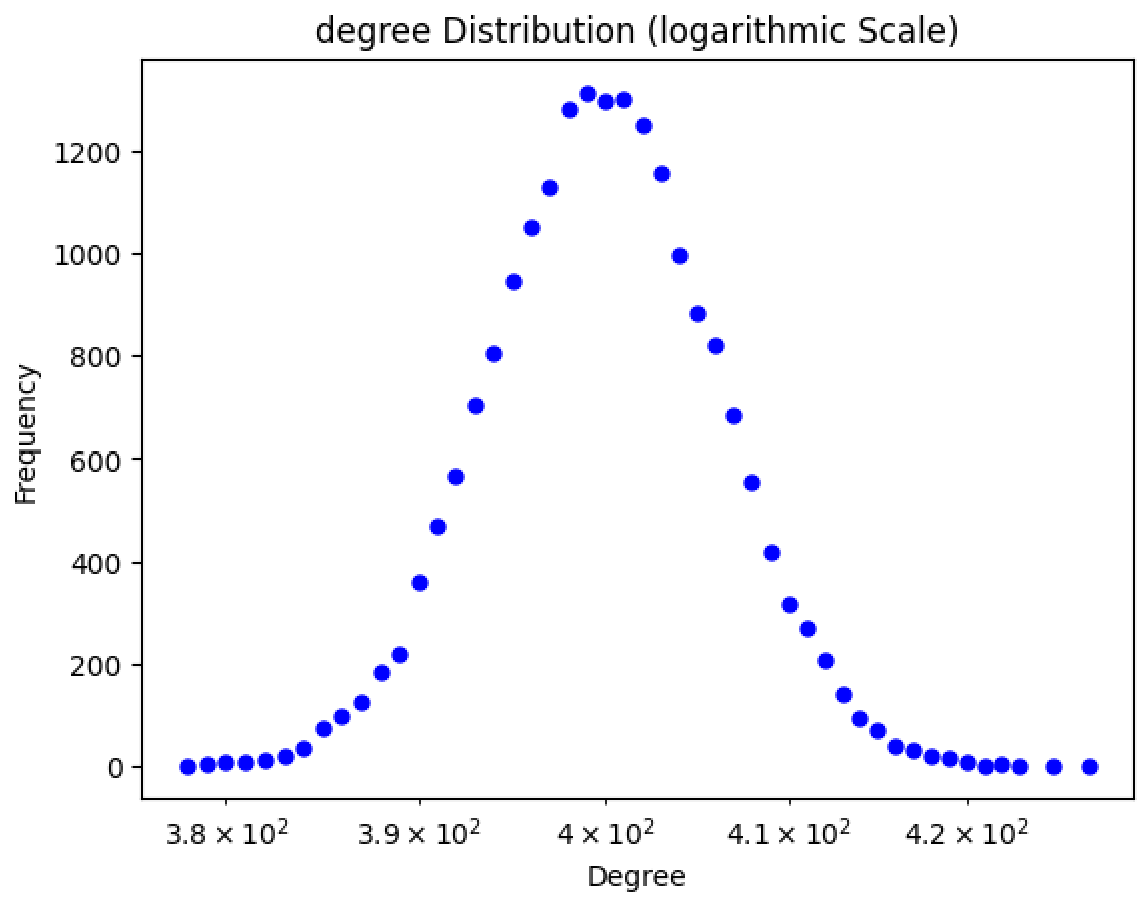

```
WS Diameter: 4
WS Clustering Coefficient: 0.7237925804327776
```

```
graph = WattsStrogatzGraph()
graph.generate_ws_graph(20000,200, 0.11)
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("WS Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("WS Clustering Coefficient:", clustering_coefficient)
#graph.plot_graph()
```



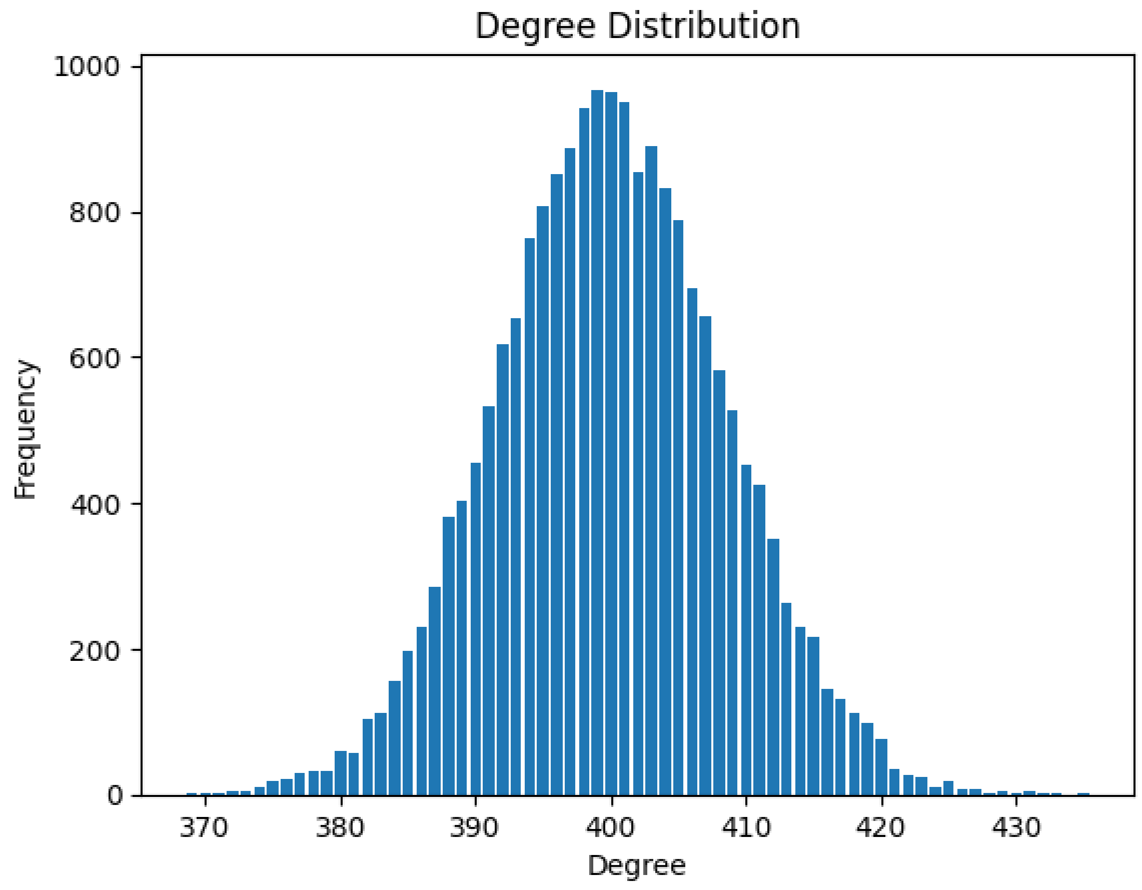

```
graph = WattsStrogatzGraph()
graph.generate_ws_graph(20000,400, 0.1)
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("WS Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("WS Clustering Coefficient:", clustering_coefficient)

#graph.plot_graph()
```






```
WS Diameter: 3
WS Clustering Coefficient: 0.5478430167077372
```

```
graph = WattsStrogatzGraph()
graph.generate_ws_graph(20000,400, 0.2)
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("WS Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("WS Clustering Coefficient:", clustering_coefficient)
print("Connectedness:", graph.compute_connectedness())
print("Average Path Length:", graph.compute_average_path_length())
#graph.plot_graph()
```




```
graph = WattsStrogatzGraph()
graph.generate_ws_graph(20000,800, 0.2)
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("WS Diameter:", diameter)

# Compute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("WS Clustering Coefficient:", clustering_coefficient)
print("Connectedness:", graph.compute_connectedness())
print("Average Path Length:", graph.compute_average_path_length())
#graph.plot_graph()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-3-7f0417b47873> in <cell line: 1>()
----> 1 graph = WattsStrogatzGraph()
      2 graph.generate_ws_graph(20000,800, 0.2)
      3 graph.compute_degree_distribution()
      4 graph.plot_degree_distribution()
      5

NameError: name 'WattsStrogatzGraph' is not defined
```

SEARCH STACK OVERFLOW

```
graph = WattsStrogatzGraph()
graph.generate_ws_graph(20000,800, 0.5)
graph.compute_degree_distribution()
graph.plot_degree_distribution()

# Compute and print the diameter
diameter = graph.compute_diameter()
print("WS Diameter:", diameter)

# ompute and print the clustering coefficient
clustering_coefficient = graph.compute_clustering_coefficient()
print("WS Clustering Coefficient:", clustering_coefficient)
print("Connectedness:", graph.compute_connectedness())
print("Average Path Length:", graph.compute_average_path_length())
#graph.plot_graph()
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
<ipython-input-1-11b5b807c5fe> in <cell line: 1>()
----> 1 graph = WattsStrogatzGraph()
      2 graph.generate_ws_graph(20000,800, 0.5)
      3 graph.compute_degree_distribution()
      4 graph.plot_degree_distribution()
      5

NameError: name 'WattsStrogatzGraph' is not defined
```

SEARCH STACK OVERFLOW

Problem 2

[ ] ↳ 1 cell hidden

Problem 5

Files:

nodeId.edges : The edges in the ego network for the node 'nodeId'. Edges are undirected. The 'ego' node does not appear, but it is assumed that they follow every node id that appears in this file.

nodeId.circles : The set of circles for the ego node. Each line contains one circle, consisting of a series of node ids. The first entry in each line is the name of the circle.

nodeId.feat : The features for each of the nodes that appears in the edge file.

nodeId.egofeat : The features for the ego user.

nodeId.featnames : The names of each of the feature dimensions. Features are '1' if the user has this property in their profile, and '0' otherwise. This file has been anonymized for facebook users, since the names of the features would reveal private data.

```
import pandas as pd
import numpy as np
```

```
import codecs
import csv

```

```
from google.colab import drive
drive.mount('/content/drive')
```

```
Mounted at /content/drive
```

Data Preprocessing

```
import os
import numpy as np

data_dir = '/content/drive/MyDrive/Social Networks Homework_Vano Mazashvili/facebook'
node_ids = ['0', '107', '348', '414', '686', '698', '1684', '1912', '3437', '3980']

def process_feat_files(data_dir, node_id):
    feat_file_path = os.path.join(data_dir, f'{node_id}.feat')
    featname_file_path = os.path.join(data_dir, f'{node_id}.featnames')

    with open(feat_file_path, 'r') as feat_file, open(featname_file_path, 'r') as featname_file:
        featnames = featname_file.read().strip().split('\n')
        arrays = []

        for line in feat_file:
            line = line.strip()
            if line:
                feat = list(map(int, line.split()[1:]))
                selected_featnames = [featnames[i-1].split()[-1] for i, val in enumerate(feat, start=1) if val == 1]

                new_array = np.zeros(1283, dtype=int)
                for featname in selected_featnames:
                    new_array[int(featname)] = 1

                arrays.append(new_array)

        arrays = np.array(arrays)
        return arrays


edge_files= [f for f in os.listdir(data_dir) if f.endswith('.edges')]

edge_file_path = os.path.join(data_dir, f'{node_ids}.edges')

edge_files = [f for f in os.listdir(data_dir) if f.endswith('.edges')]

edges = []
for edge_file in edge_files:
    with open(os.path.join(data_dir, edge_file), 'r') as file:
        for line in file:
            edge = line.strip().split()
            edge = list(map(int, edge))  # Convert edges to integers
            edges.append(edge)

unique_nodes = set()
for edge in edges:
    unique_nodes.add(edge[0])
    unique_nodes.add(edge[1])



features = None

for node_id in node_ids:
    result = process_feat_files(data_dir, node_id)
    if features is None:
        features = result
    else:
        features = np.concatenate((features, result), axis=0)

circles_files = [f for f in os.listdir(data_dir) if f.endswith('.circles')]
l = {}
for circles_file in circles_files:
    with open(os.path.join(data_dir, circles_file), 'r') as file:
        for line in file:
            circle = line.strip().split()
            circle_name = circle[0]
            circle_nodes = circle[1:]

            if circle_name in l:
                l[circle_name].extend(circle_nodes)
            else:
                l[circle_name] = circle_nodes

# Assign labels to nodes
node_ids = set()
for node_list in l.values():
    node_ids.update(node_list)

max_label_index = len(l) - 1

label_vectors = []
for node_id in range(1, 4168):  # Assuming 4167 nodes in total
    label_vector = [0] * (max_label_index + 1)
    for label_index, node_list in enumerate(l.values()):
        if str(node_id) in node_list:
            label_vector[label_index] = 1
    label_vectors.append(label_vector)

labels = []
labels = np.sum(label_vectors, axis=1)

num_nodes = len(unique_nodes)
lb = np.zeros((num_nodes, 46), dtype=int)

for node_id, node_list in enumerate(l.values()):
    for circle_name, circle_nodes in l.items():
        circle_index = int(circle_name.replace("circle", "")) - 1
        if str(node_id + 1) in circle_nodes:
```

```
            lb[node_id][circle_index] = 1


#print(len(labels))
#print(max(labels))
#print(labels)
#print(features)
#print(features[345][2])
#print(len(features))
print(len(edges))
print(len(features))
print(l)
print(labels)
print(len(labels))
num_unique_nodes = len(unique_nodes)
print(f"Number of unique nodes: {num_unique_nodes}")
```

```
170174
4167
{'circle0': ['71', '215', '54', '61', '298', '229', '81', '253', '193', '97', '264', '29', '132', '110', '163', '259', '183', '334', '245', '222', '475', '373', '461', '39
[1 1 1 ... 0 0 0]
4167
Number of unique nodes: 3959
```

```
import torch
import torch.nn as nn
import torch.optim as optim
from torch_geometric.data import Data
from torch_geometric.nn import GCNConv
from torch.nn import Linear

# Convert the edges list to a PyTorch Geometric data object
edge_index = torch.tensor(edges, dtype=torch.long).t().contiguous()

# Convert the features and labels to PyTorch tensors
x = torch.tensor(features, dtype=torch.float)
y = torch.tensor(labels, dtype=torch.long)


# Create a PyTorch Geometric data object
data = Data(x=x, edge_index=edge_index, y=y)

# Define the GNN model
class GNNModel(nn.Module):
    def __init__(self, input_dim, hidden_dim, output_dim):
        super(GNNModel, self).__init__()
        self.conv1 = GCNConv(input_dim, hidden_dim)
        self.conv2 = GCNConv(hidden_dim, hidden_dim)
        self.conv3 = GCNConv(hidden_dim, hidden_dim)
        self.conv4 = GCNConv(hidden_dim, hidden_dim)
        self.conv5 = GCNConv(hidden_dim, hidden_dim)
        self.out = Linear(hidden_dim, output_dim)


    def forward(self, x, edge_index):

        # First Message Passing Layer (Transformation)
        x = self.conv1(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)

        # Second Message Passing Layer
        x = self.conv2(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)

        x = self.conv3(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv4(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)
        x = self.conv5(x, edge_index)
        x = x.relu()
        x = F.dropout(x, p=0.5, training=self.training)
        # Output layer
        x = F.softmax(self.out(x), dim=1)
        return x

# Set the dimensions for the GNN model
input_dim = x.shape[1]
hidden_dim = 128
output_dim = len(l)

# Create an instance of the GNN model
model = GNNModel(input_dim, hidden_dim, output_dim)

# Define the loss function and optimizerfrom torch.nn import Linear
criterion = nn.CrossEntropyLoss()
optimizer = optim.Adam(model.parameters(), lr=0.001)

# Train the GNN model
def train(model, data, optimizer, criterion, num_epochs):
    model.train()

    for epoch in range(num_epochs):
        optimizer.zero_grad()
        out = model(data.x, data.edge_index)
        loss = criterion(out[data.train_mask], data.y[data.train_mask])
        loss.backward()
        optimizer.step()

        # Perform evaluation on the validation set
        model.eval()
        with torch.no_grad():
            logits = model(data.x, data.edge_index)
            pred = logits.argmax(dim=1)
            correct = pred[data.val_mask] == data.y[data.val_mask]
            val_acc = int(correct.sum()) / int(data.val_mask.sum())

        print(f'Epoch: {epoch+1}, Loss: {loss.item()}, Val Acc: {val_acc}')

# Split the data into training, validation, and testing masks
```

```
num_nodes = len(data.y)
train_mask = torch.zeros(num_nodes, dtype=torch.bool)
val_mask = torch.zeros(num_nodes, dtype=torch.bool)
test_mask = torch.zeros(num_nodes, dtype=torch.bool)

# Assuming 70% training, 15% validation, and 15% testing split
train_mask[:int(num_nodes*0.7)] = True
val_mask[int(num_nodes*0.7):int(num_nodes*0.85)] = True
test_mask[int(num_nodes*0.85):] = True

data.train_mask = train_mask
data.val_mask = val_mask
data.test_mask = test_mask

# Train the model
num_epochs = 100
train(model, data, optimizer, criterion, num_epochs)

# Evaluate the model on the testing set
model.eval()
with torch.no_grad():
    logits = model(data.x, data.edge_index)
    pred = logits.argmax(dim=1)
    correct = pred[data.test_mask] == data.y[data.test_mask]
    test_acc = int(correct.sum()) / int(data.test_mask.sum())

print(f'Test Acc: {test_acc}')
```

```
Epoch: 44, Loss: 3.3399364948272705, Val Acc: 0.8128
Epoch: 45, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 46, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 47, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 48, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 49, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 50, Loss: 3.3399367332458496, Val Acc: 0.8128
Epoch: 51, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 52, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 53, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 54, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 55, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 56, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 57, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 58, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 59, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 60, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 61, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 62, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 63, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 64, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 65, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 66, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 67, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 68, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 69, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 70, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 71, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 72, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 73, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 74, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 75, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 76, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 77, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 78, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 79, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 80, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 81, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 82, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 83, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 84, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 85, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 86, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 87, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 88, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 89, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 90, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 91, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 92, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 93, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 94, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 95, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 96, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 97, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 98, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 99, Loss: 3.339937210083008, Val Acc: 0.8128
Epoch: 100, Loss: 3.339937210083008, Val Acc: 0.8128
Test Acc: 0.13258785942492013
```

```
from sklearn.metrics import accuracy_score, precision_score, recall_score, f1_score

# Evaluate the model on the testing set
model.eval()
with torch.no_grad():
    logits = model(data.x, data.edge_index)
    pred = logits.argmax(dim=1)
    pred_test = pred[data.test_mask].cpu().numpy()
    labels_test = data.y[data.test_mask].cpu().numpy()

accuracy = accuracy_score(labels_test, pred_test)
precision = precision_score(labels_test, pred_test, average='macro')
recall = recall_score(labels_test, pred_test, average='macro')
f1 = f1_score(labels_test, pred_test, average='macro')

print(f'Test Accuracy: {accuracy:.4f}')
print(f'Precision: {precision:.4f}')
print(f'Recall: {recall:.4f}')
print(f'F1-score: {f1:.4f}')
```

```
Test Accuracy: 0.1326
Precision: 0.0221
Recall: 0.1667
F1-score: 0.0390
/usr/local/lib/python3.10/dist-packages/sklearn/metrics/_classification.py:1344: UndefinedMetricWarning: Precision is ill-defined and being set to 0.0 in labels with no pre
  _warn_prf(average, modifier, msg_start, len(result))
```

So, our feature array will have length of 1283 and will be the same for all nodes to avoid jagged matrix code given below creates feature vectors with the length of 1283 for each node

Obtaining Labels from the .circles files