```
import networkx as nx
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
from scipy.stats import spearmanr, kendalltau
import seaborn as sns
from collections import Counter
import warnings
warnings.filterwarnings('ignore')


# QUESTION 1: Plot weighted directed network with proportional node size and edge width

# ===== OPTION 1: Using Random Values =====
G = nx.DiGraph()
np.random.seed(42)

# Create random weighted directed network
nodes = range(1, 16)  # 15 nodes
G.add_nodes_from(nodes)

# Add random weighted edges
for i in nodes:
    num_edges = np.random.randint(1, 4)
    targets = np.random.choice([n for n in nodes if n != i],
                               size=min(num_edges, len(nodes)-1),
                               replace=False)
    for target in targets:
        weight = np.random.uniform(0.5, 5.0)
        G.add_edge(i, target, weight=weight)

# ===== OPTION 2: Using Downloaded Dataset =====
# UNCOMMENT THE FOLLOWING TO USE YOUR DATASET
"""
# For edge list file (format: source target weight)
G = nx.read_edgelist('your_network.txt',
                     nodetype=int,
                     data=(('weight', float),),
                     create_using=nx.DiGraph())

# OR for weighted edge list CSV
# df = pd.read_csv('your_network.csv')
# G = nx.from_pandas_edgelist(df,
#                             source='source_col',
#                             target='target_col',
#                             edge_attr='weight_col',
#                             create_using=nx.DiGraph())
```

```python
# OR for GraphML format
# G = nx.read_graphml('your_network.graphml')
"""

# Calculate node degrees
degrees = dict(G.degree())

# Extract edge weights
edges = G.edges()
weights = [G[u][v]['weight'] for u, v in edges]

# Normalize sizes and widths
node_sizes = [degrees[node] * 200 for node in G.nodes()]
edge_widths = [w * 0.5 for w in weights]

# Create visualization
plt.figure(figsize=(12, 8))
pos = nx.spring_layout(G, k=2, iterations=50, seed=42)

nx.draw_networkx_nodes(G, pos, node_size=node_sizes,
                  node_color='lightblue',
                  edgecolors='black', linewidths=1.5)

nx.draw_networkx_edges(G, pos, width=edge_widths,
                  edge_color='gray',
                  arrows=True,
                  arrowsize=20,
                  arrowstyle='->',
                  connectionstyle='arc3,rad=0.1')

nx.draw_networkx_labels(G, pos, font_size=10, font_weight='bold')

# Add edge weight labels
edge_labels = {(u, v): f"{G[u][v]['weight']:.2f}" for u, v in G.edges()}
nx.draw_networkx_edge_labels(G, pos, edge_labels, font_size=8)

plt.title('Weighted Directed Network\n(Node size ∝ Degree, Edge width ∝ Weight)',
        fontsize=14, fontweight='bold')
plt.axis('off')
plt.tight_layout()
plt.savefig('q1_weighted_directed_network.png', dpi=300, bbox_inches='tight')
plt.show()

print("Q1 - Network Statistics:")
print(f"Number of nodes: {G.number_of_nodes()}")
print(f"Number of edges: {G.number_of_edges()}")
print(f"Average degree: {np.mean(list(degrees.values())):.2f}")
print(f"Average weight: {np.mean(weights):.2f}")
```

```
# QUESTION 2: Degree distribution and network properties

# ===== OPTION 1: Using Random Values (Scale-free network) =====
G = nx.barabasi_albert_graph(n=500, m=3, seed=42)

# ===== OPTION 2: Using Downloaded Dataset =====
# UNCOMMENT THE FOLLOWING TO USE YOUR DATASET
"""
# For edge list
G = nx.read_edgelist('your_network.txt', nodetype=int)

# OR for common datasets
# G = nx.karate_club_graph()  # Karate club
# G = nx.davis_southern_women_graph()  # Southern women

# OR read from CSV
# df = pd.read_csv('your_network.csv')
# G = nx.from_pandas_edgelist(df, source='source', target='target')

# OR read from GML/GraphML
# G = nx.read_gml('your_network.gml')
# G = nx.read_graphml('your_network.graphml')
"""

# Compute degrees
degrees = [G.degree(n) for n in G.nodes()]
degree_count = Counter(degrees)

# Degree distribution
deg, cnt = zip(*sorted(degree_count.items()))

# Plot degree distribution
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

# Linear scale
axes[0].bar(deg, cnt, color='steelblue', alpha=0.7, edgecolor='black')
axes[0].set_xlabel('Degree', fontsize=12)
axes[0].set_ylabel('Frequency', fontsize=12)
axes[0].set_title('Degree Distribution (Linear Scale)', fontsize=13, fontweight='bold')
axes[0].grid(alpha=0.3)

# Log-log scale
axes[1].loglog(deg, cnt, 'o-', color='darkred', markersize=8, linewidth=2)
axes[1].set_xlabel('Degree (log)', fontsize=12)
axes[1].set_ylabel('Frequency (log)', fontsize=12)
axes[1].set_title('Degree Distribution (Log-Log Scale)', fontsize=13, fontweight='bold')
axes[1].grid(alpha=0.3)
```

```python
plt.tight_layout()
plt.savefig('q2_degree_distribution.png', dpi=300, bbox_inches='tight')
plt.show()

# ===== GLOBAL PROPERTIES =====
print("\n" + "="*60)
print("Q2 - GLOBAL NETWORK PROPERTIES")
print("="*60)

print(f"Number of nodes: {G.number_of_nodes()}")
print(f"Number of edges: {G.number_of_edges()}")
print(f"Network density: {nx.density(G):.4f}")
print(f"Average degree: {np.mean(degrees):.2f}")
print(f"Degree variance: {np.var(degrees):.2f}")

# Clustering coefficient
avg_clustering = nx.average_clustering(G)
print(f"Average clustering coefficient: {avg_clustering:.4f}")

# Transitivity
transitivity = nx.transitivity(G)
print(f"Transitivity: {transitivity:.4f}")

# Connected components
if nx.is_connected(G):
    print("Network is connected")
    diameter = nx.diameter(G)
    avg_path_length = nx.average_shortest_path_length(G)
    print(f"Diameter: {diameter}")
    print(f"Average shortest path length: {avg_path_length:.4f}")
else:
    num_components = nx.number_connected_components(G)
    print(f"Number of connected components: {num_components}")
    largest_cc = max(nx.connected_components(G), key=len)
    G_largest = G.subgraph(largest_cc)
    diameter = nx.diameter(G_largest)
    avg_path_length = nx.average_shortest_path_length(G_largest)
    print(f"Diameter of largest component: {diameter}")
    print(f"Average path length of largest component: {avg_path_length:.4f}")

# Assortativity
assortativity = nx.degree_assortativity_coefficient(G)
print(f"Degree assortativity coefficient: {assortativity:.4f}")

# ===== LOCAL PROPERTIES (Sample nodes) =====
print("\n" + "="*60)
print("LOCAL PROPERTIES (Sample of 5 nodes)")
print("="*60)
```

```python
sample_nodes = list(G.nodes())[:5]
clustering_coeffs = nx.clustering(G)

for node in sample_nodes:
    print(f"\nNode {node}:")
    print(f"  Degree: {G.degree(node)}")
    print(f"  Clustering coefficient: {clustering_coeffs[node]:.4f}")

    # Ego network size
    ego = nx.ego_graph(G, node, radius=1)
    print(f"  Ego network size (1-hop): {ego.number_of_nodes()}")


# QUESTION 3: Generate and compare network models

n = 1000  # Number of nodes

# Generate networks
# 1. Random Network (Erdős-Rényi)
p = 0.005  # Probability for edge creation
G_random = nx.erdos_renyi_graph(n, p, seed=42)

# 2. Small World Network (Watts-Strogatz)
k = 6  # Each node connected to k nearest neighbors
p_rewire = 0.1  # Rewiring probability
G_small_world = nx.watts_strogatz_graph(n, k, p_rewire, seed=42)

# 3. Preferential Attachment (Barabási-Albert)
m = 3  # Number of edges to attach from new node
G_preferential = nx.barabasi_albert_graph(n, m, seed=42)

# ===== OPTION: Using Downloaded Dataset for comparison =====
# UNCOMMENT TO ADD YOUR REAL NETWORK TO COMPARISON
"""
G_real = nx.read_edgelist('your_network.txt', nodetype=int)
# Make sure it has ~1000 nodes or sample it:
# if G_real.number_of_nodes() > 1000:
#     G_real = G_real.subgraph(list(G_real.nodes())[:1000])
"""

networks = {
    'Random Network': G_random,
    'Small World': G_small_world,
    'Preferential Attachment': G_preferential
    # 'Real Network': G_real  # Uncomment if using real data
}
```

```
# Compute properties
properties = {}

for name, G in networks.items():
    props = {}

    # Basic properties
    props['Nodes'] = G.number_of_nodes()
    props['Edges'] = G.number_of_edges()
    props['Density'] = nx.density(G)
    props['Avg Degree'] = np.mean([d for n, d in G.degree()])
    props['Avg Clustering'] = nx.average_clustering(G)
    props['Transitivity'] = nx.transitivity(G)

    # Path length (on largest component if disconnected)
    if nx.is_connected(G):
        props['Avg Path Length'] = nx.average_shortest_path_length(G)
        props['Diameter'] = nx.diameter(G)
    else:
        largest_cc = max(nx.connected_components(G), key=len)
        G_cc = G.subgraph(largest_cc)
        props['Avg Path Length'] = nx.average_shortest_path_length(G_cc)
        props['Diameter'] = nx.diameter(G_cc)

    props['Assortativity'] = nx.degree_assortativity_coefficient(G)

    # Degree distribution properties
    degrees = [d for n, d in G.degree()]
    props['Max Degree'] = max(degrees)
    props['Degree Std'] = np.std(degrees)

    properties[name] = props

# Create comparison table
df = pd.DataFrame(properties).T
print("\n" + "="*80)
print("Q3 - NETWORK MODEL COMPARISON")
print("="*80)
print(df.to_string())

# Visualization: Degree distributions
fig, axes = plt.subplots(1, 3, figsize=(18, 5))

for idx, (name, G) in enumerate(networks.items()):
    degrees = [d for n, d in G.degree()]
    degree_count = Counter(degrees)
    deg, cnt = zip(*sorted(degree_count.items()))
```

```
    axes[idx].loglog(deg, cnt, 'o-', markersize=6, linewidth=2)
    axes[idx].set_xlabel('Degree (log)', fontsize=11)
    axes[idx].set_ylabel('Frequency (log)', fontsize=11)
    axes[idx].set_title(name, fontsize=12, fontweight='bold')
    axes[idx].grid(alpha=0.3)

plt.suptitle('Degree Distribution Comparison', fontsize=14, fontweight='bold', y=1.02)
plt.tight_layout()
plt.savefig('q3_model_comparison.png', dpi=300, bbox_inches='tight')
plt.show()

# Bar plot comparison
metrics = ['Avg Clustering', 'Avg Path Length', 'Assortativity', 'Density']
fig, axes = plt.subplots(2, 2, figsize=(14, 10))
axes = axes.ravel()

for idx, metric in enumerate(metrics):
    values = [properties[name][metric] for name in networks.keys()]
    axes[idx].bar(networks.keys(), values, color=['steelblue', 'orange', 'green'],
                  alpha=0.7, edgecolor='black', linewidth=1.5)
    axes[idx].set_ylabel(metric, fontsize=11)
    axes[idx].set_title(metric, fontsize=12, fontweight='bold')
    axes[idx].grid(axis='y', alpha=0.3)
    axes[idx].tick_params(axis='x', rotation=15)

plt.tight_layout()
plt.savefig('q3_metrics_comparison.png', dpi=300, bbox_inches='tight')
plt.show()


# QUESTION 4: Centrality measures and rank correlation

top_n = 10  # Change this to get more or fewer top nodes

# ===== OPTION 1: Using Random Values =====
G = nx.barabasi_albert_graph(n=200, m=3, seed=42)

# ===== OPTION 2: Using Downloaded Dataset =====
# UNCOMMENT THE FOLLOWING TO USE YOUR DATASET
"""
# For undirected network
G = nx.read_edgelist('your_network.txt', nodetype=int)

# For directed network
# G = nx.read_edgelist('your_network.txt', nodetype=int, create_using=nx.DiGraph())

# From CSV
# df = pd.read_csv('your_network.csv')
```

```
# G = nx.from_pandas_edgelist(df, source='source', target='target')
"""

print(f"\nQ4 - Computing centrality measures for {G.number_of_nodes()} nodes...")

# Compute centrality measures
degree_cent = nx.degree_centrality(G)
betweenness_cent = nx.betweenness_centrality(G)
closeness_cent = nx.closeness_centrality(G)
eigenvector_cent = nx.eigenvector_centrality(G, max_iter=1000)
pagerank = nx.pagerank(G)

# Get top-N nodes for each measure
top_degree = sorted(degree_cent.items(), key=lambda x: x[1], reverse=True)[:top_n]
top_betweenness = sorted(betweenness_cent.items(), key=lambda x: x[1],
reverse=True)[:top_n]
top_closeness = sorted(closeness_cent.items(), key=lambda x: x[1], reverse=True)[:top_n]
top_eigenvector = sorted(eigenvector_cent.items(), key=lambda x: x[1],
reverse=True)[:top_n]
top_pagerank = sorted(pagerank.items(), key=lambda x: x[1], reverse=True)[:top_n]

# Print top nodes
print(f"\nTop-{top_n} nodes by different centrality measures:")
print("="*80)

measures = {
    'Degree': top_degree,
    'Betweenness': top_betweenness,
    'Closeness': top_closeness,
    'Eigenvector': top_eigenvector,
    'PageRank': top_pagerank
}

for measure_name, top_nodes in measures.items():
    print(f"\n{measure_name} Centrality:")
    for rank, (node, value) in enumerate(top_nodes, 1):
        print(f"  {rank}. Node {node}: {value:.6f}")

# Rank correlation with PageRank
print("\n" + "="*80)
print("RANK CORRELATION WITH PAGERANK")
print("="*80)

# Create ranking dictionaries
all_nodes = list(G.nodes())

def get_ranks(cent_dict):
    sorted_nodes = sorted(cent_dict.items(), key=lambda x: x[1], reverse=True)
```

```
    return {node: rank for rank, (node, _) in enumerate(sorted_nodes, 1)}

ranks_degree = get_ranks(degree_cent)
ranks_betweenness = get_ranks(betweenness_cent)
ranks_closeness = get_ranks(closeness_cent)
ranks_eigenvector = get_ranks(eigenvector_cent)
ranks_pagerank = get_ranks(pagerank)

# Compute correlations
correlations = {}

for name, ranks in [('Degree', ranks_degree),
                    ('Betweenness', ranks_betweenness),
                    ('Closeness', ranks_closeness),
                    ('Eigenvector', ranks_eigenvector)]:

    rank_list1 = [ranks[n] for n in all_nodes]
    rank_list2 = [ranks_pagerank[n] for n in all_nodes]

    spearman_corr, spearman_p = spearmanr(rank_list1, rank_list2)
    kendall_corr, kendall_p = kendalltau(rank_list1, rank_list2)

    correlations[name] = {
        'Spearman': spearman_corr,
        'Kendall': kendall_corr
    }

    print(f"\n{name} vs PageRank:")
    print(f"  Spearman correlation: {spearman_corr:.4f} (p-value: {spearman_p:.4e})")
    print(f"  Kendall correlation: {kendall_corr:.4f} (p-value: {kendall_p:.4e})")

# Visualization
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

measure_names = list(correlations.keys())
spearman_vals = [correlations[m]['Spearman'] for m in measure_names]
kendall_vals = [correlations[m]['Kendall'] for m in measure_names]

x = np.arange(len(measure_names))
width = 0.35

axes[0].bar(x - width/2, spearman_vals, width, label='Spearman',
           color='steelblue', alpha=0.8, edgecolor='black')
axes[0].bar(x + width/2, kendall_vals, width, label='Kendall',
           color='coral', alpha=0.8, edgecolor='black')
axes[0].set_ylabel('Correlation Coefficient', fontsize=12)
axes[0].set_title('Rank Correlation with PageRank', fontsize=13, fontweight='bold')
axes[0].set_xticks(x)
```

```
axes[0].set_xticklabels(measure_names, rotation=15)
axes[0].legend()
axes[0].grid(axis='y', alpha=0.3)
axes[0].axhline(y=0, color='black', linestyle='-', linewidth=0.5)

# Scatter plot: Degree vs PageRank ranks
axes[1].scatter([ranks_degree[n] for n in all_nodes],
               [ranks_pagerank[n] for n in all_nodes],
               alpha=0.5, s=30, color='darkgreen')
axes[1].set_xlabel('Degree Centrality Rank', fontsize=11)
axes[1].set_ylabel('PageRank Rank', fontsize=11)
axes[1].set_title('Degree vs PageRank Ranking', fontsize=13, fontweight='bold')
axes[1].grid(alpha=0.3)

# Add diagonal line
max_rank = max(max(ranks_degree.values()), max(ranks_pagerank.values()))
axes[1].plot([0, max_rank], [0, max_rank], 'r--', linewidth=2, alpha=0.5, label='Perfect
correlation')
axes[1].legend()

plt.tight_layout()
plt.savefig('q4_centrality_correlation.png', dpi=300, bbox_inches='tight')
plt.show()


# QUESTION 5: Community detection and comparison

# ===== OPTION 1: Using Karate Club (Built-in) =====
G = nx.karate_club_graph()

# ===== OPTION 2: Using Downloaded Dataset =====
# UNCOMMENT THE FOLLOWING TO USE YOUR DATASET
"""
# Small network from edge list
G = nx.read_edgelist('your_small_network.txt', nodetype=int)

# From CSV
# df = pd.read_csv('your_network.csv')
# G = nx.from_pandas_edgelist(df, source='source', target='target')

# Other built-in small networks you can use:
# G = nx.davis_southern_women_graph()
# G = nx.florentine_families_graph()
# G = nx.les_miserables_graph()
"""

print(f"\nQ5 - Applying community detection on network with {G.number_of_nodes()}
nodes...")
```

```
# Import community detection algorithms
from networkx.algorithms import community

# 1. Girvan-Newman (Betweenness-based)
comp = community.girvan_newman(G)
communities_gn = next(comp)
communities_gn = [list(c) for c in communities_gn]

# 2. Louvain (Modularity-based) - using greedy modularity
communities_louvain = list(community.greedy_modularity_communities(G))
communities_louvain = [list(c) for c in communities_louvain]

# 3. Label Propagation
communities_lp = list(community.label_propagation_communities(G))
communities_lp = [list(c) for c in communities_lp]

# 4. Asynchronous Label Propagation
communities_async = list(community.asyn_lpa_communities(G))
communities_async = [list(c) for c in communities_async]

algorithms = {
    'Girvan-Newman': communities_gn,
    'Greedy Modularity': communities_louvain,
    'Label Propagation': communities_lp,
    'Async Label Prop': communities_async
}

# Print community structure
print("\nCommunity Structure:")
print("="*80)
for name, comms in algorithms.items():
    print(f"\n{name}: {len(comms)} communities")
    for i, comm in enumerate(comms, 1):
        print(f"  Community {i}: {len(comm)} nodes - {sorted(comm)[:10]}{'...' if len(comm) > 10
else ''}")

# Compute evaluation metrics
print("\n" + "="*80)
print("EVALUATION METRICS")
print("="*80)

# Modularity
modularities = {}
for name, comms in algorithms.items():
    mod = community.modularity(G, comms)
    modularities[name] = mod
    print(f"{name} - Modularity: {mod:.4f}")
```

```
# Coverage (fraction of edges within communities)
print("\n")
coverages = {}
for name, comms in algorithms.items():
    cov = community.coverage(G, comms)
    coverages[name] = cov
    print(f"{name} - Coverage: {cov:.4f}")

# Visualization: Plot communities
fig, axes = plt.subplots(2, 2, figsize=(16, 14))
axes = axes.ravel()

pos = nx.spring_layout(G, k=0.5, iterations=50, seed=42)

for idx, (name, comms) in enumerate(algorithms.items()):
    ax = axes[idx]

    # Assign colors to communities
    color_map = {}
    colors = plt.cm.Set3(np.linspace(0, 1, len(comms)))

    for comm_idx, comm in enumerate(comms):
        for node in comm:
            color_map[node] = colors[comm_idx]

    node_colors = [color_map[node] for node in G.nodes()]

    nx.draw_networkx_nodes(G, pos, node_color=node_colors,
                           node_size=300, ax=ax, edgecolors='black', linewidths=1.5)
    nx.draw_networkx_edges(G, pos, alpha=0.3, ax=ax)
    nx.draw_networkx_labels(G, pos, font_size=8, ax=ax)

    ax.set_title(f'{name}\n{len(comms)} communities | Modularity: {modularities[name]:.3f}',
                 fontsize=12, fontweight='bold')
    ax.axis('off')

plt.suptitle('Community Detection Comparison', fontsize=15, fontweight='bold', y=0.995)
plt.tight_layout()
plt.savefig('q5_community_detection.png', dpi=300, bbox_inches='tight')
plt.show()

# Bar plots for comparison
fig, axes = plt.subplots(1, 2, figsize=(14, 5))

alg_names = list(algorithms.keys())

# Modularity comparison
```

```
mod_vals = [modularities[name] for name in alg_names]
axes[0].bar(alg_names, mod_vals, color='steelblue', alpha=0.7, edgecolor='black',
linewidth=1.5)
axes[0].set_ylabel('Modularity', fontsize=12)
axes[0].set_title('Modularity Comparison', fontsize=13, fontweight='bold')
axes[0].tick_params(axis='x', rotation=15)
axes[0].grid(axis='y', alpha=0.3)

# Coverage comparison
cov_vals = [coverages[name] for name in alg_names]
axes[1].bar(alg_names, cov_vals, color='coral', alpha=0.7, edgecolor='black', linewidth=1.5)
axes[1].set_ylabel('Coverage', fontsize=12)
axes[1].set_title('Coverage Comparison', fontsize=13, fontweight='bold')
axes[1].tick_params(axis='x', rotation=15)
axes[1].grid(axis='y', alpha=0.3)

plt.tight_layout()
plt.savefig('q5_metrics_comparison.png', dpi=300, bbox_inches='tight')
plt.show()


# QUESTION 6: Epidemic diffusion models

# Create network
# ===== OPTION 1: Using Random Values =====
n = 500
G = nx.watts_strogatz_graph(n, k=6, p=0.1, seed=42)

# ===== OPTION 2: Using Downloaded Dataset =====
# UNCOMMENT THE FOLLOWING TO USE YOUR DATASET
"""
G = nx.read_edgelist('your_network.txt', nodetype=int)
# If network is too large, sample it:
# if G.number_of_nodes() > 1000:
#     nodes_sample = list(G.nodes())[:500]
#     G = G.subgraph(nodes_sample)
n = G.number_of_nodes()
"""

print(f"\nQ6 - Simulating epidemic models on network with {n} nodes...")

# Simulation parameters
num_steps = 100
initial_infected = 5

# Model 1: SI Model (Susceptible-Infected)
def simulate_SI(G, beta, num_steps, initial_infected):
    """SI Model: S -> I"""
```

```
    n = G.number_of_nodes()
    nodes = list(G.nodes())

    # Initialize states: 0=Susceptible, 1=Infected
    state = {node: 0 for node in nodes}

    # Initial infections (random)
    initial_nodes = np.random.choice(nodes, initial_infected, replace=False)
    for node in initial_nodes:
        state[node] = 1

    # Track over time
    S_count = [n - initial_infected]
    I_count = [initial_infected]

    for step in range(num_steps):
        new_infections = []

        for node in nodes:
            if state[node] == 0:  # Susceptible
                # Check infected neighbors
                infected_neighbors = [nb for nb in G.neighbors(node) if state[nb] == 1]

                # Infection probability
                prob_infection = 1 - (1 - beta) ** len(infected_neighbors)

                if np.random.random() < prob_infection:
                    new_infections.append(node)

        # Update states
        for node in new_infections:
            state[node] = 1

        # Count states
        susceptible = sum(1 for s in state.values() if s == 0)
        infected = sum(1 for s in state.values() if s == 1)

        S_count.append(susceptible)
        I_count.append(infected)

    return S_count, I_count

# Model 2: SIS Model (Susceptible-Infected-Susceptible)
def simulate_SIS(G, beta, gamma, num_steps, initial_infected):
    """SIS Model: S -> I -> S"""
    n = G.number_of_nodes()
    nodes = list(G.nodes())
```

```
state = {node: 0 for node in nodes}
initial_nodes = np.random.choice(nodes, initial_infected, replace=False)
for node in initial_nodes:
    state[node] = 1

S_count = [n - initial_infected]
I_count = [initial_infected]

for step in range(num_steps):
    new_infections = []
    new_recoveries = []

    for node in nodes:
        if state[node] == 0:  # Susceptible
            infected_neighbors = [nb for nb in G.neighbors(node) if state[nb] == 1]
            prob_infection = 1 - (1 - beta) ** len(infected_neighbors)

            if np.random.random() < prob_infection:
                new_infections.append(node)

        elif state[node] == 1:  # Infected
            if np.random
```