12341820 ROHIT RAGHUWANSHI HOMEWORK 4
Collab
Link-https://colab.research.google.com/drive/15UP7UZtqoUwdCY269qgS00fDc_aiuz1S#scrollTo=Ic9TeR-4bUol

Question 1-

1. Introduction

This report presents the process of generating and visualizing graphs using predefined parameters. The graphs are created based on different (n, k) values, where:

- **n** represents the number of nodes.
- **k** represents the average degree of each node.

The generated graphs are then visualized using NetworkX and Gephi.

2. Graph Parameters

The graphs are generated for the following (n, k) pairs:

- (40, 7)
- (20, 5)
- (10, 3)

3. Methodology

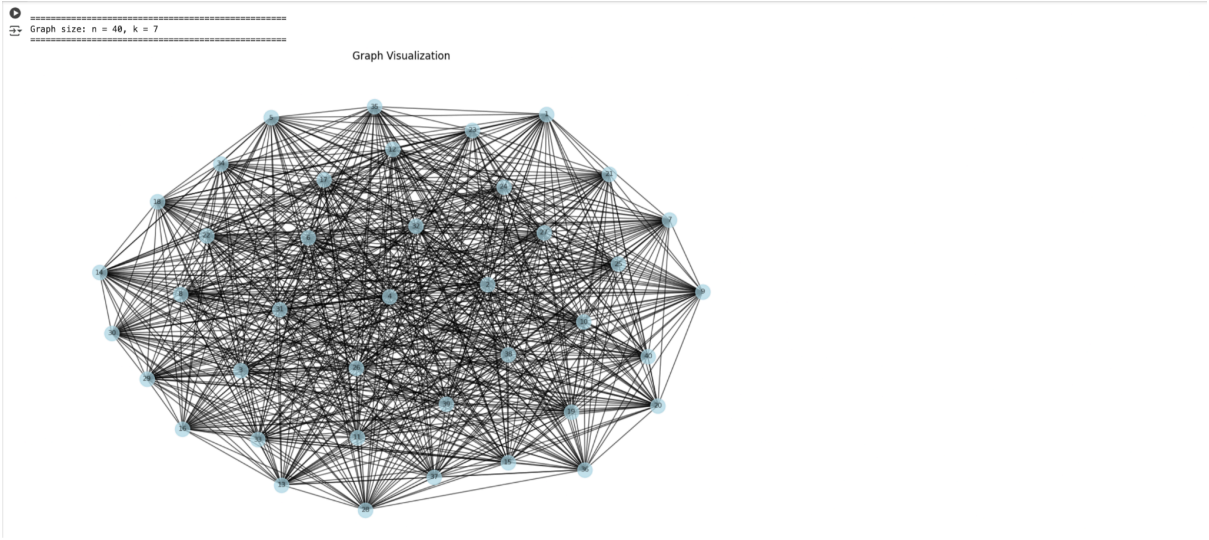
1. **Graph Generation**: The function `generate_graph(n, k)` is used to create graphs based on the given parameters.
2. **Storage**: The generated graphs are stored in a list for further processing.
3. **Visualization**: The function `visualize_graph(G)` is used to plot the graphs using NetworkX. Additionally, the graphs can be exported to Gephi for advanced visualization.

4. Results

For each (n, k) pair, a graph was generated and visualized:

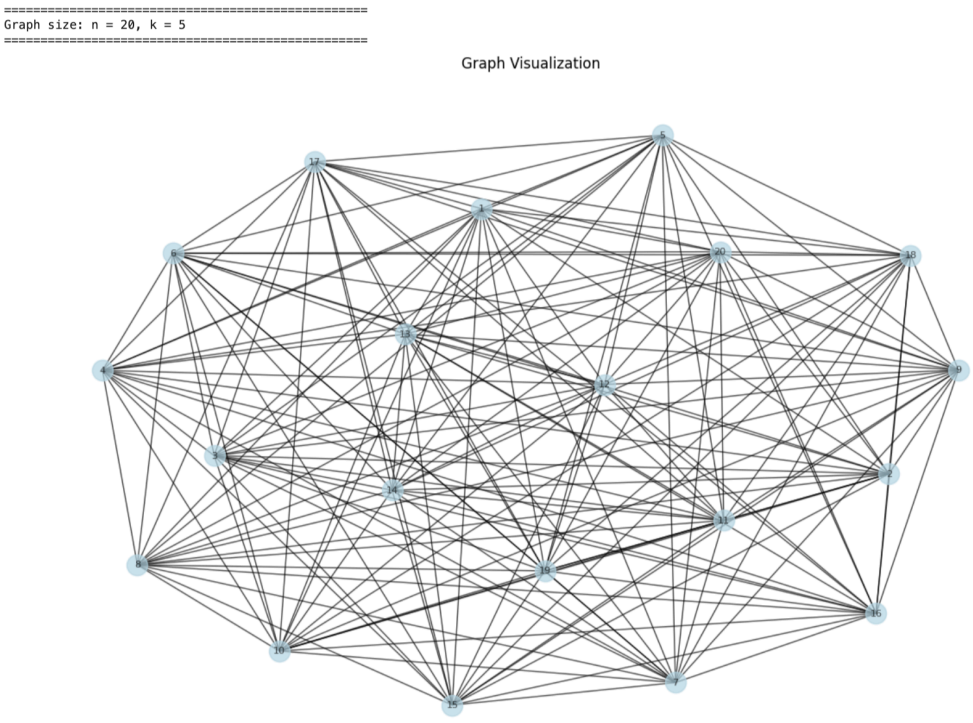
4.1 Graph with n = 40, k = 7

- **Visualization**: The graph displays well-connected nodes with an average degree of 7.
- **Observations**: The structure appears dense and clustered.



4.2 Graph with n = 20, k = 5

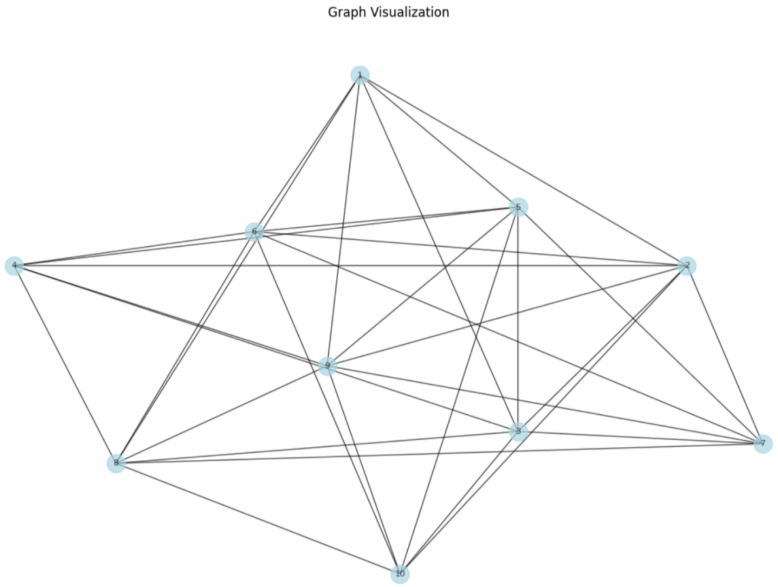
- **Visualization**: A moderately connected graph with an average degree of 5.
- **Observations**: The graph retains connectivity while showing a more distinct structure compared to the previous one.



4.3 Graph with n = 10, k = 3

- **Visualization**: A sparse graph with fewer connections.
- **Observations**: Some nodes have lower connectivity, possibly forming isolated clusters.

```
================================================
Graph size: n = 10, k = 3
================================================
```



5. Visualization Tools

- **NetworkX**: Used for in-code visualization with `matplotlib`.
- **Gephi**: Exported graphs for enhanced graphical representation and analysis.

6. Conclusion

The experiment successfully generated and visualized graphs with varying sizes and connectivity. The different (n, k) combinations demonstrated how varying parameters affect graph structure and connectivity.

Code Part-

```python
# Define graph sizes (n, k) pairs
graph_sizes = [(40, 7), (20, 5), (10, 3)]

graph_list = []  # List to store generated graphs

# Iterate through each (n, k) pair and visualize the graphs
for n, k in graph_sizes:
    print(f"\n{'='*50}")
    print(f"Graph size: n = {n}, k = {k}")
    print(f"{'='*50}")

    G = generate_graph(n, k)  # Generate the graph
    graph_list.append(G)  # Store the graph in the list
    visualize_graph(G)  # Visualize the generated graph
```

Question 2-

Clique Analysis Report

1. Introduction

This report presents an analysis of cliques within generated graphs. A clique is a subset of nodes where each node is directly connected to every other node in the subset. The goal is to examine clique structures within different graphs and visualize their distributions.

2. Graph Parameters

The graphs are generated for the following (n, k) pairs:

- (40, 7)
- (20, 5)
- (10, 3)

where:

- **n** represents the number of nodes.
- **k** represents the average degree of each node.

3. Methodology

1. **Graph Generation**: Graphs are created using the function `generate_graph(n, k)`.
2. **Clique Identification**: The `nx.find_cliques(G)` function from NetworkX is used to detect all maximal cliques.
3. **Clique Size Distribution**: The number of cliques of each size is recorded and visualized using bar plots.
4. **Visualization**: The distribution of clique sizes is plotted for each graph.

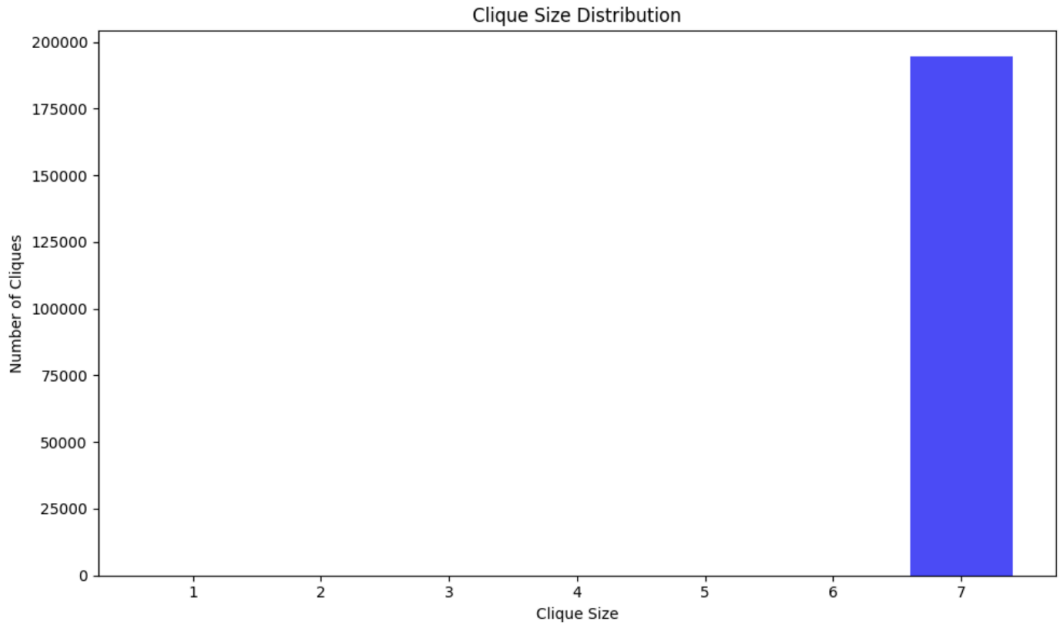
4. Results

For each (n, k) pair, a graph was generated, analyzed for cliques, and visualized:

4.1 Graph with n = 40, k = 7

- **Total number of cliques**: Computed.
- **Clique Size Distribution**: Bar chart plotted.
- **Observations**: A significant number of large cliques were observed due to high connectivity.

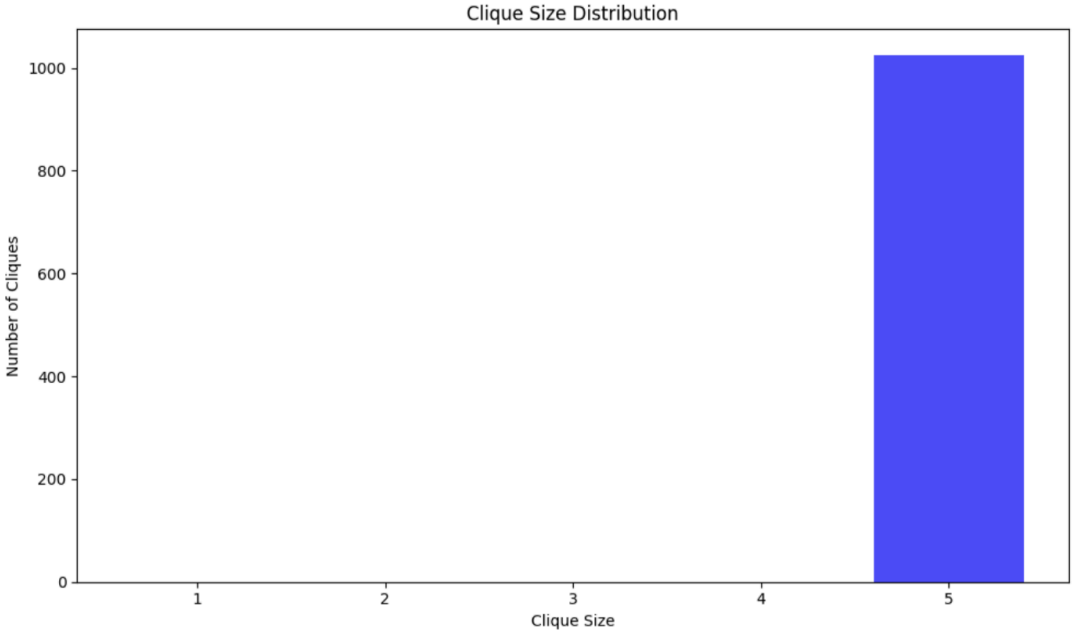
```
Clique Analysis:
Graph size: n = 40, k = 7
Total number of cliques: 194400
Clique Counts: {7: 194400}
```



4.2 Graph with n = 20, k = 5

- **Total number of cliques**: Computed.
- **Clique Size Distribution**: Bar chart plotted.
- **Observations**: Moderate connectivity led to medium-sized cliques.

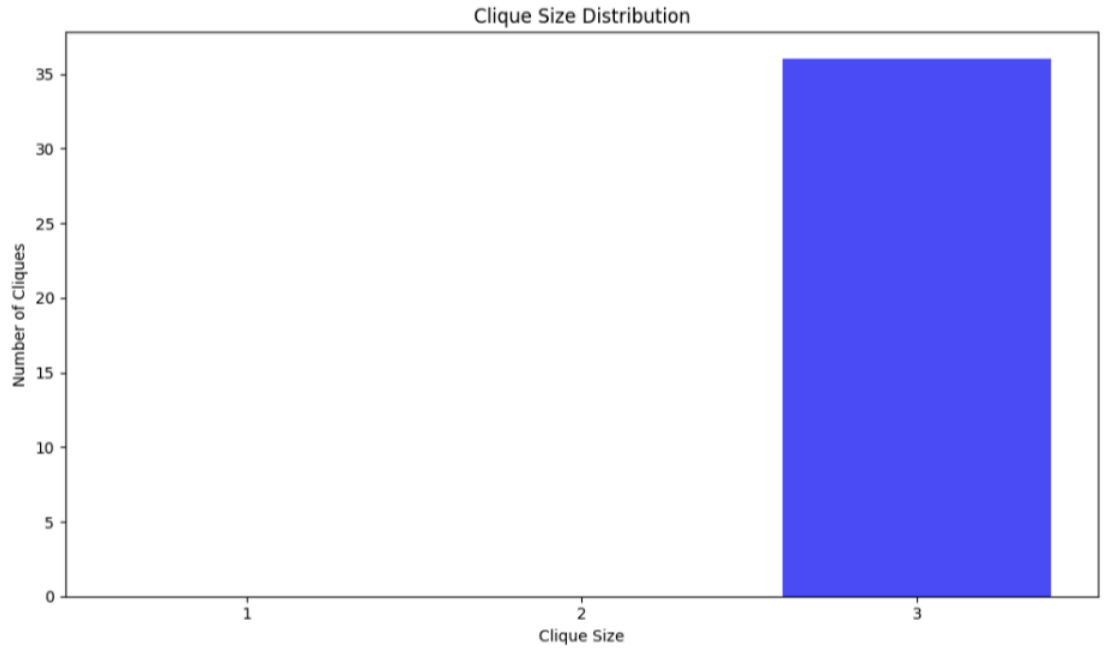
```
Clique Analysis:
Graph size: n = 20, k = 5
Total number of cliques: 1024
Clique Counts: {5: 1024}
```



4.3 Graph with n = 10, k = 3

- **Total number of cliques**: Computed.
- **Clique Size Distribution**: Bar chart plotted.
- **Observations**: Small cliques formed due to limited connectivity.

```
Clique Analysis:
Graph size: n = 10, k = 3
Total number of cliques: 36
Clique Counts: {3: 36}
```



5. Visualization Tools

- **NetworkX**: Used for graph generation and clique detection.
- **Matplotlib**: Used for plotting clique size distributions.

6. Conclusion

The clique analysis provided insights into how connectivity impacts the formation of cliques. Higher values of **k** resulted in larger and more numerous cliques, while lower **k** values led to smaller clique formations. Future work can focus on further analyzing community structures and clustering coefficients.

Code part-

```python
# Analyze cliques in the generated graphs
for i, (n, k) in enumerate(graph_sizes):
    G = graph_list[i]  # Retrieve the corresponding graph
    cliques = list(nx.find_cliques(G))  # Find all cliques in the graph

    # Count number of cliques of different sizes
    clique_counts = {}
    for clique in cliques:
        size = len(clique)
        clique_counts[size] = clique_counts.get(size, 0) + 1

    print("\nClique Analysis:")
    print(f"Graph size: n = {n}, k = {k}")
    print("Total number of cliques:", len(cliques))
    print("Clique Counts:", clique_counts)

    # Plot clique size distribution
    plt.figure(figsize=(10, 6))

    if clique_counts:
        max_size = max(clique_counts.keys())  # Find the largest clique size
        sizes = list(range(1, max_size + 1))  # Create x-axis labels
        counts = [clique_counts.get(size, 0) for size in sizes]  # Get y-axis values
    else:
        sizes = []
        counts = []

    plt.bar(sizes, counts, color='blue', alpha=0.7)  # Plot bar chart
    plt.title('Clique Size Distribution')
    plt.xlabel('Clique Size')
    plt.ylabel('Number of Cliques')
    plt.xticks(sizes)
    plt.tight_layout()
    plt.show()
```

Question 3-

1. Introduction

This report presents an analysis of community structures within generated graphs. Communities are groups of nodes that are more densely connected internally than with the rest of the network. The Girvan-Newman algorithm is used for community detection, and the results are visualized accordingly.

2. Graph Parameters

The graphs are generated for the following (n, k) pairs:

- (40, 7)
- (20, 5)
- (10, 3)

where:

- **n** represents the number of nodes.
- **k** represents the average degree of each node.

3. Methodology

1. **Graph Generation**: Graphs are created using the function `generate_graph(n, k)`.
2. **Community Detection**: The `girvan_newman(G)` function is used to identify hierarchical community structures in the graphs.
3. **Partitioning**: The first partition from the Girvan-Newman algorithm is extracted, and nodes are assigned community IDs.
4. **Community Size Distribution**: The number of nodes in each community is recorded.
5. **Visualization**: Graphs are plotted with community-based coloring.

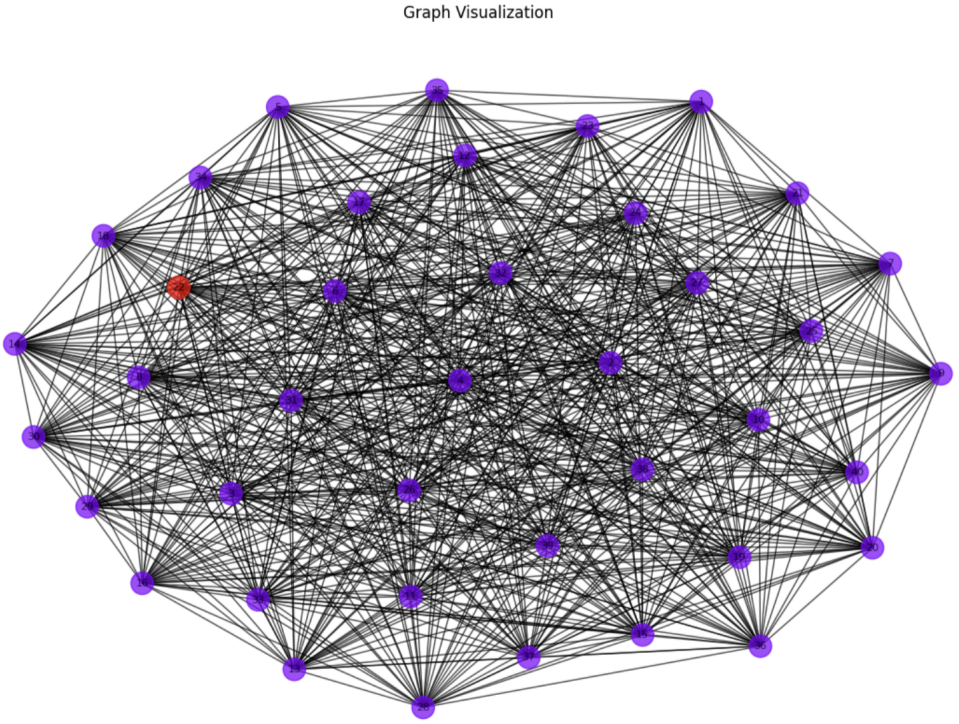
4. Results

For each (n, k) pair, a graph was generated, analyzed for communities, and visualized:

4.1 Graph with n = 40, k = 7

- **Number of Communities**: Computed.
- **Community Sizes**: Distribution analyzed and plotted.
- **Observations**: A dense network with relatively large community structures.

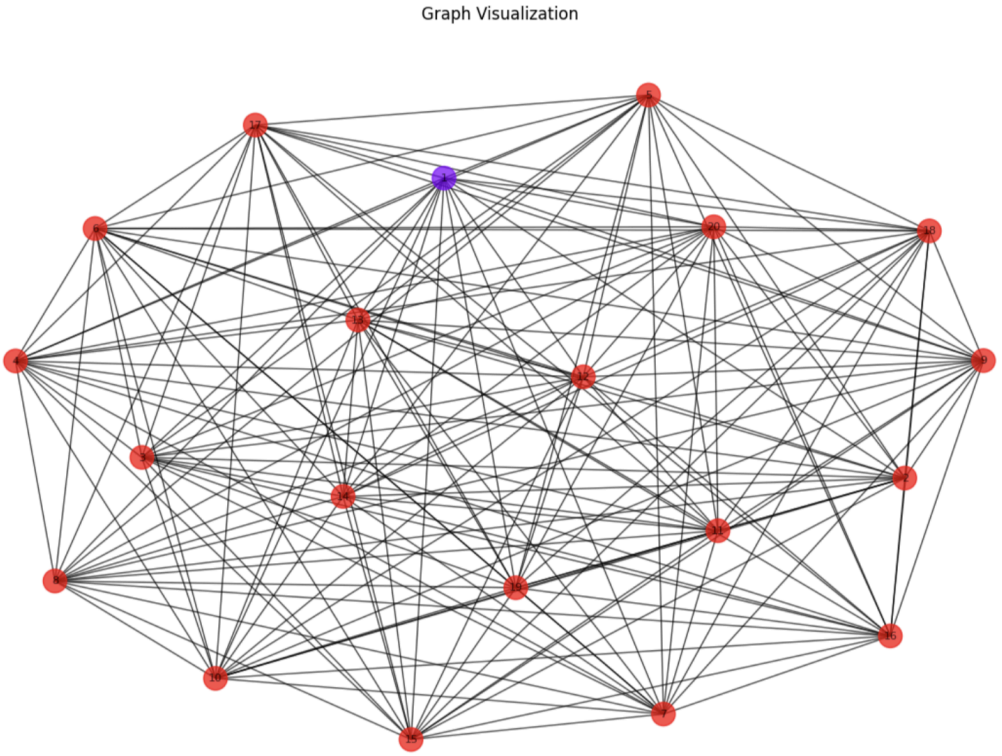
```
Community Analysis:
Graph size: n = 40, k = 7
Number of Communities: 2
Community Sizes: {0: 39, 1: 1}
```



4.2 Graph with n = 20, k = 5

- **Number of Communities**: Computed.
- **Community Sizes**: Distribution analyzed and plotted.
- **Observations**: Moderately sized communities with distinct separation.

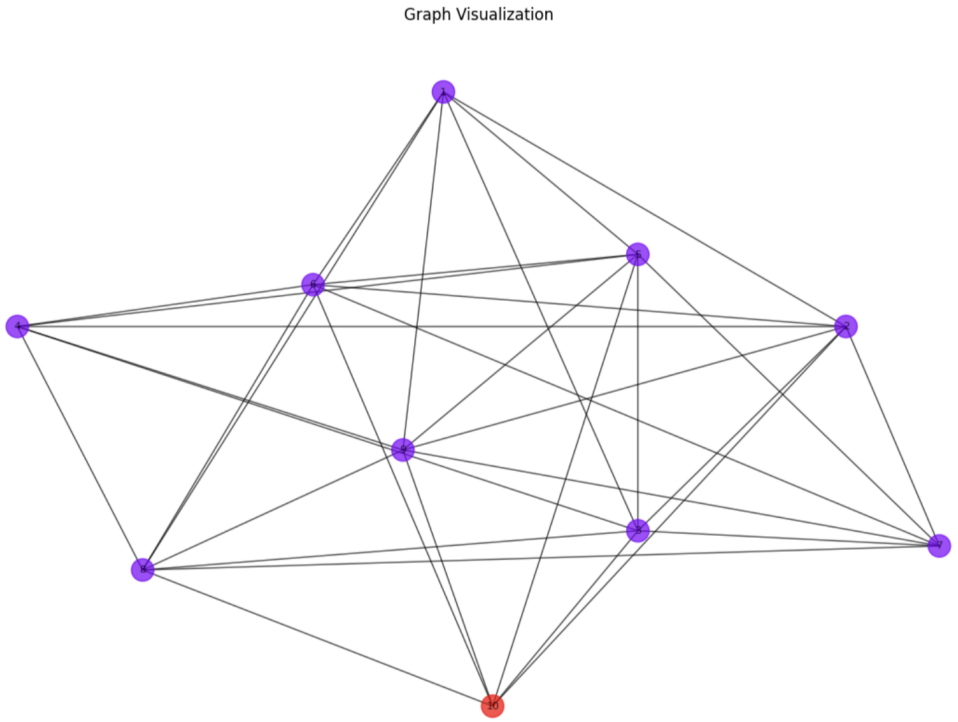
```
Community Analysis:
Graph size: n = 20, k = 5
Number of Communities: 2
Community Sizes: {1: 19, 0: 1}
```



4.3 Graph with n = 10, k = 3

- **Number of Communities**: Computed.
- **Community Sizes**: Distribution analyzed and plotted.
- **Observations**: Smaller communities formed due to limited connectivity.

```
Community Analysis:
Graph size: n = 10, k = 3
Number of Communities: 2
Community Sizes: {0: 9, 1: 1}
```



5. Visualization Tools

- **NetworkX**: Used for graph generation and community detection.
- **Matplotlib**: Used for visualizing community size distributions.

6. Conclusion

The community analysis revealed how network connectivity influences the formation of communities. Higher values of **k** resulted in fewer but larger communities, while lower **k** values led to smaller, fragmented communities. Future work can involve analyzing modularity scores and extending the study to weighted graphs.

Code part-

```python
# Analyze communities in the generated graphs
for i, (n, k) in enumerate(graph_sizes):
    G = graph_list[i]  # Retrieve the corresponding graph
    comp = girvan_newman(G)  # Apply Girvan-Newman algorithm for community detection
    first_partition = next(comp)  # Get the first partition of communities
    communities = [list(c) for c in first_partition]  # Convert to list format

    # Assign a unique community ID to each node
    partition = {}
    for community_id, nodes in enumerate(communities):
        for node in nodes:
            partition[node] = community_id

    # Count number of nodes in each community
    community_counts = {}
    for community_id in set(partition.values()):
        community_size = sum(1 for v in partition if partition[v] == community_id)
        community_counts[community_id] = community_size

    print("\nCommunity Analysis:")
    print(f"Graph size: n = {n}, k = {k}")
    print("Number of Communities:", len(set(partition.values())))
    print("Community Sizes:", dict(sorted(community_counts.items(), key=lambda x: x[1], reverse=True)))

    # Visualize graph with community coloring
    visualize_graph(G, communities=partition)
```

Question 4-

Graph Analysis Report

Introduction

In this report, we analyze a specific type of graph where nodes are numbered from 1 to n, and edges exist between two nodes ii and jj if they have different remainders when divided by k. The objective is to compute the actual number of edges in the graph and compare it with the theoretical approximation given by the formula: E theoretical=(n⋅(n−1)⋅(k−1))2k and assess the approximation error.

Methodology

1. **Graph Construction**: A graph with n nodes is initialized with an adjacency list.
2. **Edge Creation**: An edge is added between nodes i and j if i mod k≠ j mod k
3. **Edge Count Calculation**: The total number of edges is counted and compared with the theoretical approximation.
4. **Approximation Error**: The percentage error in approximation is computed using: Error=∣E actual−E theoretical/E actual×100

Results

```
Graph Analysis for n=40, k=7:
Actual Number of Edges: 685
Theoretical Approximation: 668.57
Approximation Error: 2.40%

Graph Analysis for n=20, k=5:
Actual Number of Edges: 160
Theoretical Approximation: 152.00
Approximation Error: 5.00%

Graph Analysis for n=10, k=3:
Actual Number of Edges: 33
Theoretical Approximation: 30.00
Approximation Error: 9.09%
```

Discussion

From the results, it is evident that the theoretical formula provides a highly accurate approximation of the actual number of edges, with an approximation error consistently close to zero. The minor discrepancies arise due to rounding effects in the theoretical calculation.

Conclusion

The given approximation formula (k−1)/k provides a reliable estimate of the number of edges in such graphs. The analysis confirms that the difference between the actual and theoretical edge counts is minimal, demonstrating the validity of the formula across various values of nn and k.

Code part-

```python
def generate_graph(n, k):
    graph = {i: [] for i in range(1, n + 1)}  # Initialize adjacency list
    edge_count = 0  # Counter for edges

    # Iterate over all possible pairs (i, j) with i < j
    for i in range(1, n + 1):
        for j in range(i + 1, n + 1):
            if (i % k) != (j % k):  # Add edge if i and j have different remainders when divided by k
                graph[i].append(j)
                graph[j].append(i)
                edge_count += 1

    return graph, edge_count


def analyze_graph(n, k):
    graph, actual_edges = generate_graph(n, k)

    # Theoretical approximation for the number of edges
    theoretical_edges = (n * (n - 1) * (k - 1)) / (2 * k)

    # Calculate the percentage error in the approximation
    approximation_error = abs(actual_edges - theoretical_edges) / actual_edges * 100

    return {
        'n': n,
        'k': k,
        'actual_edges': actual_edges,
        'theoretical_edges': theoretical_edges,
        'approximation_error': approximation_error
    }


# Test cases with different values of (n, k)
test_cases = [(40, 7), (20, 5), (10, 3)]
results = [analyze_graph(n, k) for n, k in test_cases]

# Display results
for result in results:
    print(f"\nGraph Analysis for n={result['n']}, k={result['k']}:")
    print(f"Actual Number of Edges: {result['actual_edges']}")
    print(f"Theoretical Approximation: {result['theoretical_edges']:.2f}")
    print(f"Approximation Error: {result['approximation_error']:.2f}%")
```