CS 6385 Project 1 Report

Introduction

During my studies in the CS6385 course, I came across the intriguing concept of k-core in graphs. This concept, as introduced in our lecture notes, particularly Lecture Note 13 on Clusters in Graphs, piqued my interest due to its potential applications in network analysis. This project is my attempt to delve deeper into this concept, understand its intricacies, and experiment with its practical applications.

Background

The k-core of a graph is essentially a subgraph where each node has at least 'k' neighbors. It's a way to identify dense subgraphs within a larger graph, which can be pivotal in understanding the structural properties and resilience of networks. For instance, in social network analysis, a high k-core value could indicate tightly-knit communities.

Task Breakdown

Task 1: Algorithm Implementation

For this task, I decided to implement an algorithm that would generate a random undirected graph with 25 nodes. The edges between these nodes are determined by a probability value `p`, which lies between 0 and 1. The core of this algorithm is based on the Erdos-Renyi model, a well-known model in random graph generation.

Once the graph is generated, the next step is to compute its core value. This is done by iteratively checking for the existence of k-cores and increasing the value of k until no k-core is found.

Task 2: Visualization

Visualization plays a crucial role in understanding and interpreting data. For this task, I utilized the NetworkX library to visualize the generated graphs. This not only helped me understand the structure of the graphs but also provided a visual representation of how the core varies with different edge probabilities.

Task 3: Analyzing the Relationship

This task was particularly interesting. I ran the algorithm for different values of `p`, ranging from 0.05 to 0.95. The objective was to understand how the core value changes with varying edge probabilities. The results were then plotted to provide a visual representation of this relationship.

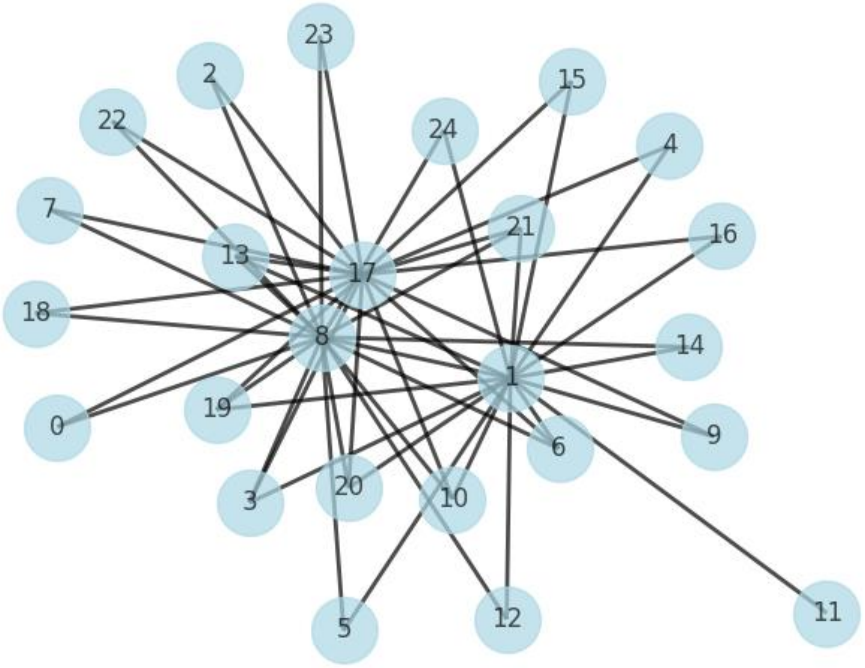
Insights and Observations

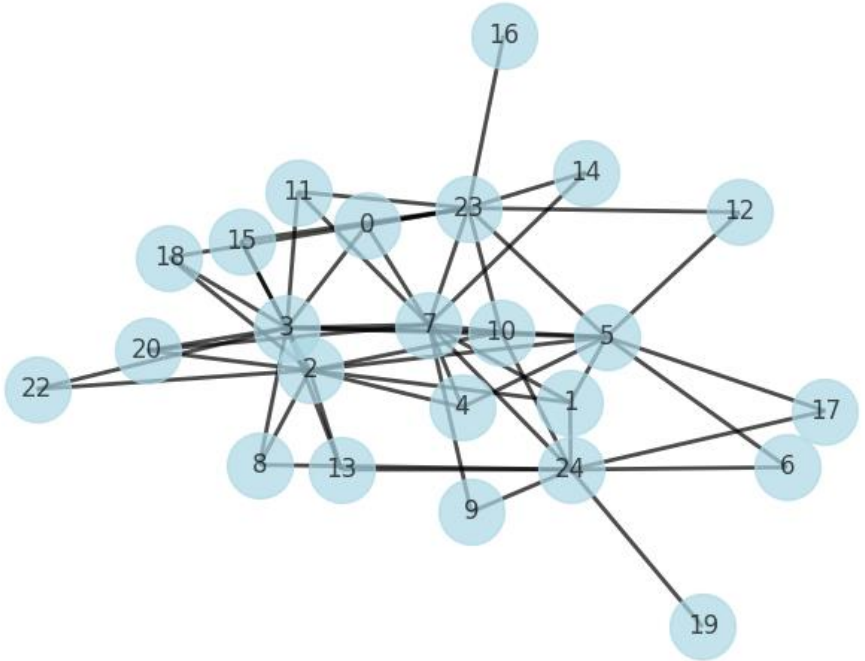
From my experiments, I observed that as the edge probability increases, the core value of the graph also tends to increase. This is intuitive since a higher edge probability means more edges are likely to be formed, leading to denser subgraphs.

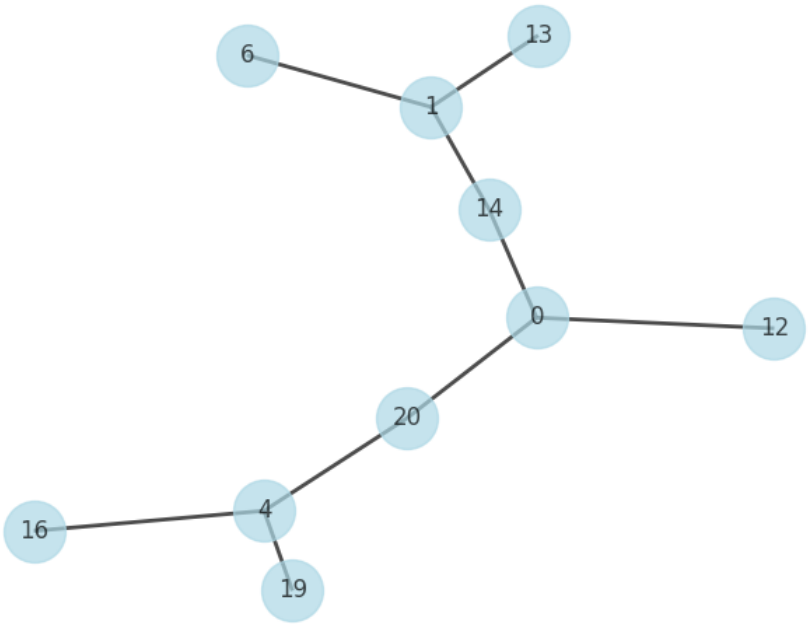
The visualizations were particularly enlightening. For lower values of `p`, the graphs were sparse with few connections. However, as `p` increased, the graphs became denser, and the core structure became more evident.

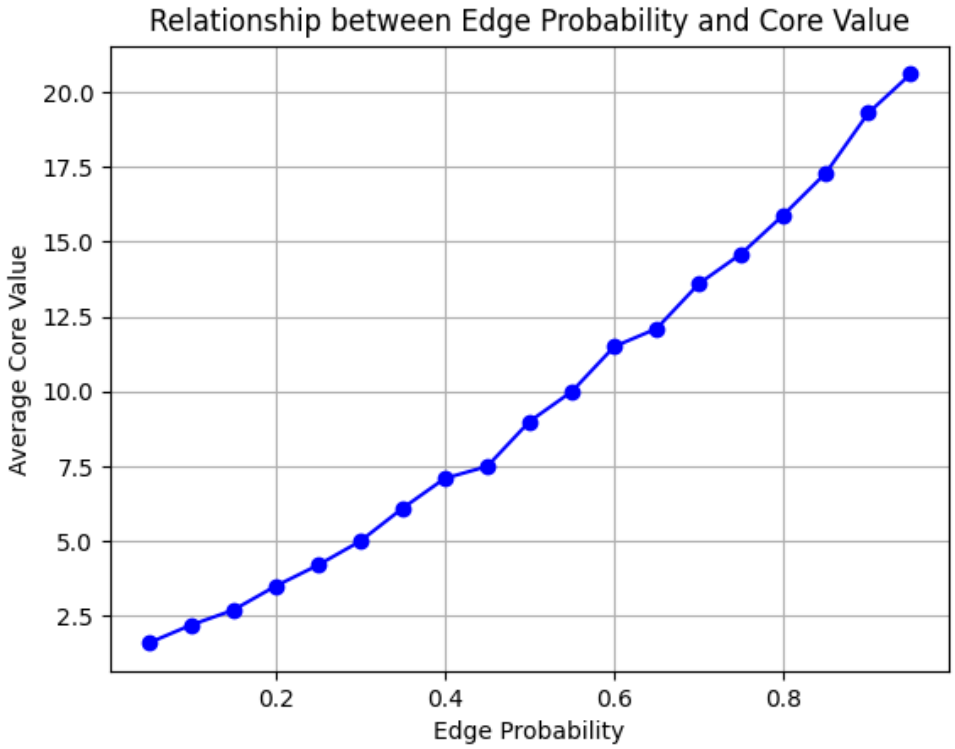
Results:

```
PS C:\Users\revan> & C:/Python312/python.exe c:/Users/revan/OneDrive/Documents/CS6385_ATN/cs6385_project1_test.py
Sample: Average Core for p=0.5 is 9.3
PS C:\Users\revan>
```









Conclusion

This project has been an enlightening journey into the world of graph theory and network analysis. The k-core concept, though simple in its definition, offers deep insights into the structure and properties of graphs. Through this project, I've not only deepened my understanding of the k-core concept but also honed my skills in algorithm implementation and data visualization.

Appendix: Source Code

```
import random
import matplotlib.pyplot as plt
import networkx as nx

# --- UTILITY FUNCTIONS ---

def generate_graph(p):
    """
    Constructs a random undirected graph with 25 nodes.

    The graph is generated based on the Erdos-Renyi model, where each edge is
    included in the graph with probability p independent from every other edge.

    Args:
    - p (float): Probability of forming an edge between any two nodes.

    Returns:
    - dict: Graph represented as an adjacency list.
    """
    graph = {i: set() for i in range(25)}
    for i in range(25):
        for j in range(i+1, 25):
            if random.random() < p:
                graph[i].add(j)
                graph[j].add(i)
    return graph

def extract_k_core(graph, k):
    """
    Extracts the k-core from a graph.

    The k-core of a graph is a maximal subgraph in which each vertex has at least
degree k.
    This function prunes nodes with degree less than k until all nodes in the
graph satisfy this property.

    Args:
    - graph (dict): Input graph.
    - k (int): Desired core number.

    Returns:
    - dict: k-core of the graph.
```

```
    """
    while True:
        nodes_to_prune = [node for node, neighbors in graph.items() if
len(neighbors) < k]
        if not nodes_to_prune:
            break
        for node in nodes_to_prune:
            for neighbor in list(graph[node]):
                if neighbor in graph:
                    graph[neighbor].discard(node)
            del graph[node]
    return graph

def determine_core_value(graph):
    """
    Computes the core value of a graph.

    The core value is the highest k for which a non-empty k-core exists in the
graph.
    This function finds the largest k-core by iteratively checking and increasing
k.

    Args:
    - graph (dict): Input graph.

    Returns:
    - int: Core value.
    """
    k = 1
    while extract_k_core(graph.copy(), k):
        k += 1
    return k - 1

def average_core(p):
    """
    Calculates the average core value over multiple iterations.

    For a given edge probability p, this function generates multiple random
graphs
    and computes their average core value to provide a more stable estimate.

    Args:
    - p (float): Edge probability.

    Returns:
```

```
    - float: Average core value.
    """
    iterations = 10
    total_core = sum(determine_core_value(generate_graph(p)) for _ in
range(iterations))
    return total_core / iterations

def visualize(p):
    """
    Visualizes a graph using NetworkX for a given edge probability.

    This function provides a visual representation of the graph structure,
    highlighting the nodes and their connections. It's useful for understanding
    the graph's topology and core structure.

    Args:
    - p (float): Edge probability.
    """
    graph = generate_graph(p)
    k = determine_core_value(graph)
    core_graph = extract_k_core(graph, k)
    G = nx.Graph(core_graph)
    layout = nx.spring_layout(G)
    nx.draw(G, layout, with_labels=True, node_color='lightblue', node_size=1000,
width=2.0, alpha=0.7)
    plt.title(f'Graph Visualization (p={p}, Core={k})')
    plt.show()

def plot_relation():
    """
    Plots the relationship between edge probability and core value.

    This function visualizes how the core value of a graph changes as the edge
probability varies.
    It provides insights into the stability and structure of the graph as the
connectivity changes.
    """
    probabilities = [i/100 for i in range(5, 100, 5)]
    core_values = [average_core(p) for p in probabilities]
    plt.plot(probabilities, core_values, marker='o', linestyle='-', color='blue')
    plt.xlabel('Edge Probability')
    plt.ylabel('Average Core Value')
    plt.title('Relationship between Edge Probability and Core Value')
    plt.grid(True)
    plt.show()
```

```python
# --- TASK FUNCTIONS ---

def task1(p):
    """
    Task 1: Calculate average core value for a given edge probability.

    This task focuses on understanding the core structure of a graph for a specific edge probability.
    """
    return average_core(p)

def task2(p_values):
    """
    Task 2: Visualize the core numbers for a set of edge probabilities.

    Visualization helps in understanding the graph's structure and how the core varies with different probabilities.
    """
    for p in p_values:
        visualize(p)

def task3():
    """
    Task 3: Plot the relationship between edge probability and core value.

    This task provides a comprehensive view of how the core value changes across different edge probabilities.
    """
    plot_relation()

# --- MAIN EXECUTION ---

def main():
    """
    Main execution function.

    This function orchestrates the execution of all tasks, demonstrating the core concepts and visualizations.
    """
    sample_prob = 0.5
    print(f"Sample: Average Core for p={sample_prob} is {task1(sample_prob)}")
    task2([0.15, 0.5, 0.85])
    task3()
```

```
if __name__ == "__main__":
    main()
```

Readme:

Requirements:

- Python 3.x
- matplotlib
- network

Installation:

Before running the code, you need to ensure you have the required libraries installed. You can install them using **pip**:

pip install matplotlib networkx

Usage:

To execute the project and visualize the results:

1. Navigate to the directory containing the project.
2. Run the script using Python:

   python project1.py

References:

1. Lecture Note 13: Clusters in Graphs. CS6385 Course Material.
2. "Python Random Module". GeeksforGeeks. Link
3. "Python Matplotlib (Plotting Library)". W3Schools. Link
4. "NetworkX Basics". NetworkX Official Documentation. Link