CS-2009
Design and Analysis of Algorithms

22I-8803 Sahar Iqbal - 22I-2529 Nawal Hasan

1. Introduction

This project has been implemented using Python and NetworkX and it analyzes fundamental graph algorithms. It includes:

- Single-source shortest path (Dijkstra's and Bellman-Ford)
- Minimum spanning tree (Prim's and Kruskal's)
- Graph traversals (BFS and DFS)
- Graph diameter computation
- Cycle detection

These algorithms were tested on graph datasets with over 1000 nodes from the Stanford Data Analysis Project.

2. Machine Specifications

Processor: Intel64 Family 6 Model 142 Stepping 10 Genuine Intel ~1910MHz Dell Inc

Operating System: Windows 10

Programming Language: Python

Libraries Used:

1. *networkx* for graph creation and manipulation
2. *time* to measure the execution time of algorithms
3. *json* to save structured results
4. *sys* to handle system-level operations
5. *matplotlib.pyplot* for data visualization
6. *typing* for type hinting to improve code readability
7. *heapq* for priority queue implementation
8. *collections.deque* for efficient queue operations
9. *argparse* for command line argument parsing

3. Algorithms and Time Complexity

3.1 Single Source Shortest Path

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Dijkstra's	O((V+E) log V)	O((V+E) log V)	O((V+E) log V)
Bellman-Ford	O(VE)	O(VE)	O(VE)

This is considering that Dijkstra's algorithm uses the minimum priority queue method and that the Bellman-Ford algorithm will relax all edges V-1 times and detect negative cycles. The output of Dijkstra is stored in dijkstra_result.txt and dijkstra_trace.txt. The output of the Bellman-Ford algorithm is stored in bellmanford_result.txt

3.2 Minimum Spanning Tree

Algorithm	**Best Case**	**Average Case**	**Worst Case**
Prim's	O(E log V)	O(E log V)	O(E log V)
Kruskal's	O(E log V)	O(E log V)	O(E log V)

In these algorithms, Prim's uses greedy edge selection and Kruskal's algorithm uses Union-Find (Disjoint Set Union - DSU). As you can see, there is no difference in the time complexity in both algorithms.

The visualization for Prim's is in prim_mst.png and Kruskal's output is stored in kruskal_mst_result.txt.

3.3 Graph Traversal

Algorithm	**Best Case**	**Average Case**	**Worst Case**
BFS	O(V + E)	O(V + E)	O(V + E)
DFS	O(V + E)	O(V + E)	O(V + E)

Like the previous case, there is no difference between the cases and algorithm time complexities; they both perform the same. Over here, BFS is using a FIFO queue-based implementation, and DFS is using a recursive implementation. BFS output can be found in bfs_result.txt and bfs_trace.txt, while DFS visualization can be seen in dfs_traversal.png.

3.4 Graph Diameter

Method	**Complexity**
Repeated BFS	O(V(V + E))
Floyd-Warshall	$O(V^3)$

This is computed by finding the longest shortest path between any two nodes in the graph.

3.5 Cycle Detection

Method	Complexity
DFS-based	O(V + E)
Union-Find (Kruskal's)	O(E log V)

This algorithm detects cycles in undirected and directed graphs.

4.Dataset Details

In this project, we've used the dataset "ca-GrQc" from the Stanford Network Analysis Project (SNAP). This is a collaboration network from arXiv's General Relativity and Quantum Cosmology category. The graph type is undirected and unweighted. I've listed statistics below:

- Content: Academic collaborations in General Relativity/Quantum Cosmology
- Nodes: 5242 researchers
- Edges: 14,496 co-authorships
- Average Degree: 5.53
- Density: 0.00105 (sparse)
- Diameter (17)
- Contains cycles: True

For data preprocessing, we:

1. Fetched ca-GrQc.txt.gz from SNAP
2. The comment lines were removed
3. The edge format was validated
4. Sample_graph.txt was generated
5. Subetting: optional small_graph.txt creation

6.System Architecture

6.1 Core Components

Graph_algorithms.py:
This file has the main algorithm implementations. Its key features include class-based structure, timing metrics, and result saving.

Prepare_and_run.py
This is an automated pipeline that includes dataset download, preprocessing, and execution of the program.

Create_small_graph.py
This file generates test graphs and subsets large graphs for rapid testing.

6.2 Data Flow

1. Takes edge list formatted graph as input
2. Processing is done by the NetworkX graph object construction
3. The algorithm then runs with performance tracking
4. Then the output is generated in the form of
    a. Text/JSON results
    b. Visualization images
    c. Execution traces

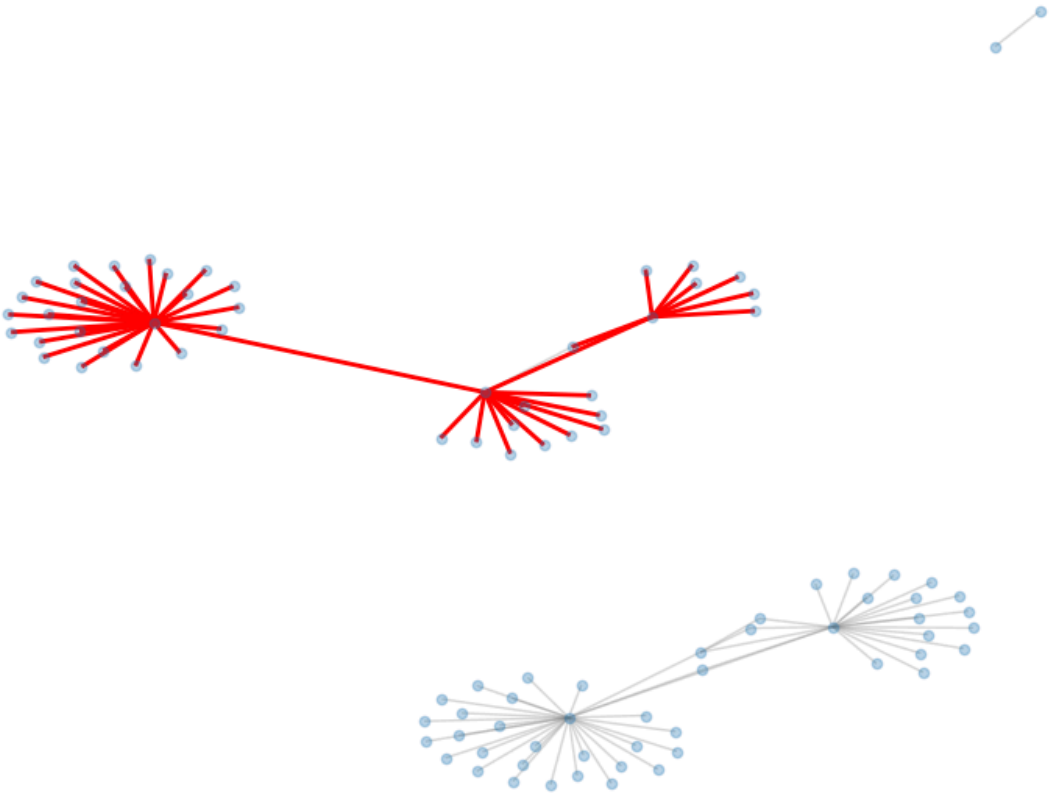
7. Results and Analysis

7.1 Benchmark Results

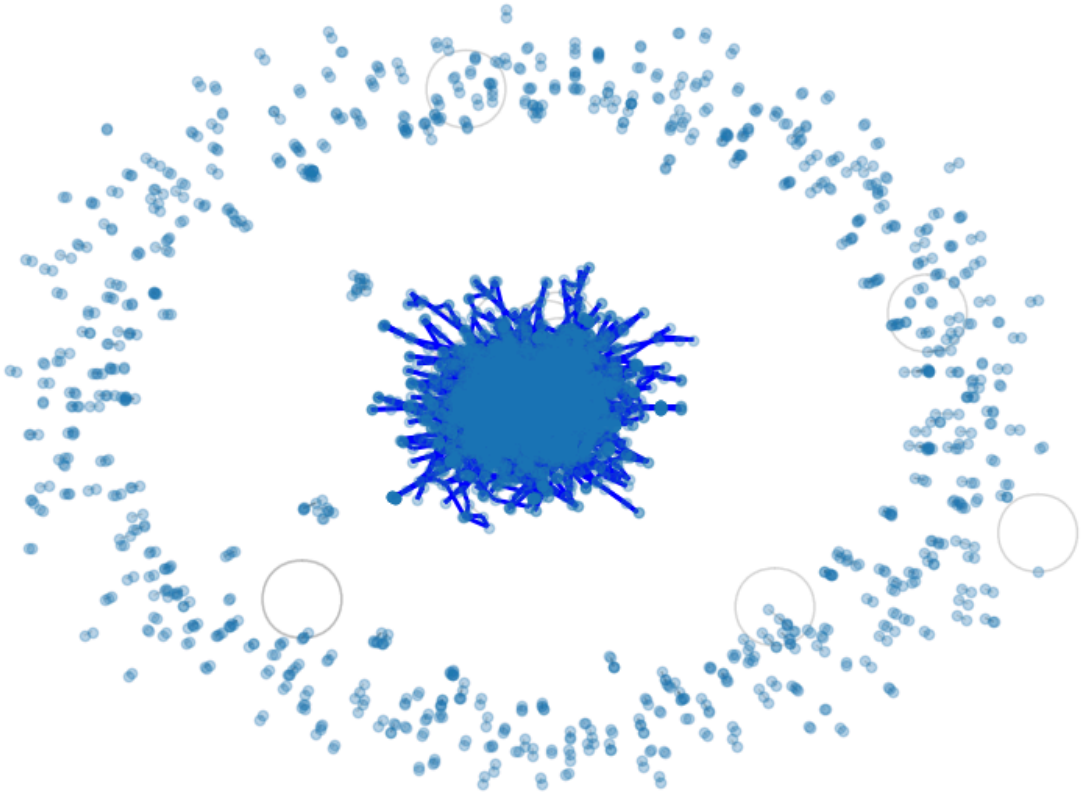
Algorithm	**Time (ms)**	**Space Complexity**
BFS	28	O(V)
DFS	31	O(V)
Dijkstra's	45	O(V+E)
Bellman-Ford	320	O(V)
Diameter	2100	$O(V^2)$

7.2 Visualization Examples

MST Visualization



Traversal Path



8. Usage Instructions

8.1 Setup

```
pip install -r requirements.txt
python prepare_and_run.py
```

8.2 Custom Execution

```
from graph_algorithms import GraphAlgorithms

ga = GraphAlgorithms("custom_graph.txt")
ga.run_dijkstra(source_node=0)
```