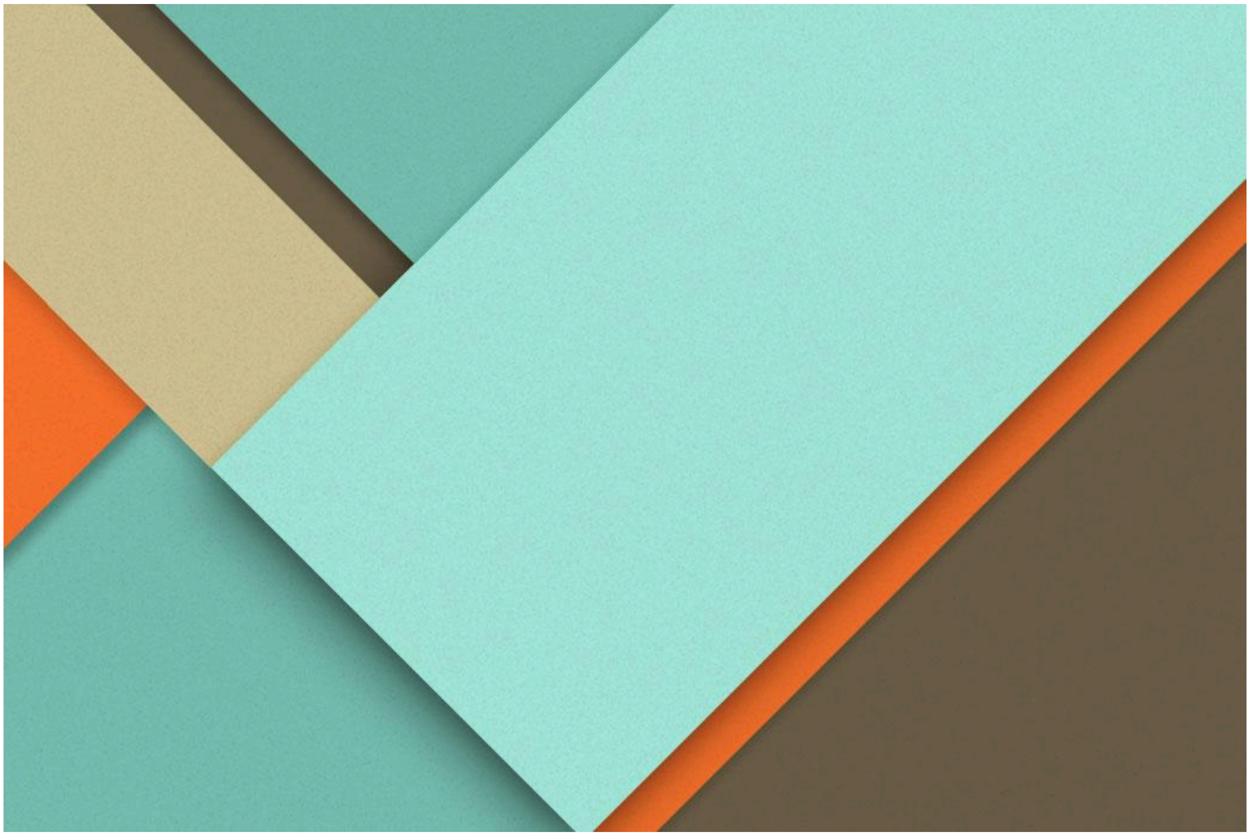


Graph DataBase

15.04.2024

TEAM - SHMAD

Anirudh M - 21CS10004

Marla Mayukha - 21CS10041

Meduri Harshith Chowdary - 21CS10042

P Datta Ksheeraj - 21CS30037

Sanskar Mittal - 21CS10057

Abstract

Graph databases have become increasingly essential for processing large-scale interconnected data efficiently. In this project, our goal is to explore graph exploration and process-related queries using some graph processing systems, such as ApacheGraph, Pregel (GoldenOrb), Giraph, Neo4j, and Stanford GPS.

We aim to demonstrate their capabilities by loading a substantial graph dataset from the Stanford SNAP large graph repository and developing an intuitive interface to execute basic graph queries. Additionally, we aspire to implement advanced functionalities like PageRank computation to enhance the comprehensiveness of our project.

Furthermore, we plan to meticulously profile the performance of the graph processing system to gain insights into its efficiency and scalability. This project aims to deepen our understanding of graph databases and their practical implications in handling vast interconnected datasets.

This project proposes the utilization of Neo4j, a leading graph database management system, to fulfill the objectives of loading a large graph from the Stanford SNAP repository, providing an interface for running simple graph queries, computing PageRank, and profiling performance. Neo4j offers the following features beneficial for the project:

- Efficient handling of large-scale graph data.
- Intuitive query language (Cypher) for easy graph data interaction.
- Support for implementing custom graph algorithms, including PageRank.
- Built-in tools for performance profiling and optimization.

By leveraging Neo4j's capabilities, we aim to efficiently achieve our project objectives, including loading, querying, analyzing, and profiling large graph datasets from the Stanford SNAP repository. Stanford GPS is equally capable as Neo4j in the areas of our interest for this project, so that'll be an alternative for now.

Objectives

Our project aims to delve into the realm of graph databases and explore their utility in processing large-scale interconnected data efficiently. Specifically, we aim to achieve the following objectives:

- **System Exploration:** Evaluate and compare various graph processing systems, including Neo4j and Stanford GPS, to understand their features, capabilities, and suitability for our project requirements.

- **Dataset Loading:** Load a substantial graph dataset from the Stanford SNAP large graph repository into the chosen graph processing system. This dataset selection will be based on factors such as size, complexity, and relevance to our project objectives.

- **Interface Development:** Develop an intuitive interface that allows users to execute basic graph queries seamlessly. The interface should provide functionalities for querying nodes, edges, and properties within the graph dataset.

- **Advanced Functionality Implementation:** Implement advanced functionalities such as PageRank computation (listed below under **Bonuses**) within the selected graph processing system. PageRank is a critical algorithm for ranking web pages in search engine results and will enhance the comprehensiveness of our project.

- **Performance Profiling:** Profile the performance of the graph processing system to gain insights into its efficiency, scalability, and resource utilization. This involves measuring metrics such as query execution time, memory usage, and throughput.

- **Comparison and Evaluation:** Compare the performance and usability of different graph processing systems, including Neo4j and Stanford GPS, based on our project requirements. Evaluate their strengths and weaknesses to determine the most suitable solution.

Bonuses : **PageRank, Clustering, Components, Communities, GraphSage**.

Methodology

The Neo4j dataset analysis script employs various methodologies and techniques to interact with the Neo4j graph database, extract useful insights, and perform graph-based computations. Here's a detailed overview of the methodologies used:

Establishing Connection to Neo4j Database:

- The script starts by establishing a connection to the **Neo4j database** using the **GraphDatabase module**.
- Connection parameters such as **URI, username, and password** are provided to authenticate and access the database.

Loading Dataset into Neo4j:

- The script allows loading a dataset into the Neo4j database by reading from a file (***dataset.txt***) and executing **Cypher queries** to create nodes and relationships.
- It provides options (load) to either load the dataset, delete existing data, or use previously loaded data.

Query Execution and Profiling:

- The script defines various functions to execute Cypher queries against the Neo4j database.
- Each query is associated with a **specific functionality** such as counting nodes, retrieving node properties, finding neighbors, calculating centrality measures, etc.
- **Profiling information** is extracted from query execution results to analyze query **performance and resource utilization**.

Data Analysis Queries:

The script includes a set of predefined queries to perform different types of graph analyses, including:

- **Counting nodes and relationships** in the graph.
- Retrieving **all nodes or specific node** properties.
- Finding **connected nodes, common neighbors, and shortest paths** between nodes.
- Computing centrality measures like **node centrality, clustering coefficient, PageRank, and community detection**.
- Performing **triangle counting** and identifying **triangles containing a specific node**.
- Exploring **connected components** and **graph structure**.

Graph Algorithm Integration:

- The script integrates graph algorithms provided by the Neo4j Graph Data Science (GDS) library.
- Algorithms such as **Louvain community detection**, **PageRank**, and **GraphSAGE** are applied to analyze the graph structure and identify patterns.
- These algorithms provide additional insights into the **graph's topology, community structure, and node centrality**.

Error Handling and Exception Management:

- The script incorporates **error handling** mechanisms to catch and handle exceptions that may occur during query execution or database interaction.
- It prints meaningful error messages to help users identify and resolve issues effectively.

Cleanup and Resource Management:

- At the end of the script execution, resources such as database connections are properly closed to prevent resource leaks.
- Optionally, the script allows dropping the graph from memory to **release memory resources** used by graph algorithms.

Overall, the **script** provides a **comprehensive toolkit** for **analyzing and exploring graph** data stored in a Neo4j database. It combines **Cypher queries** with **graph algorithms** to **extract valuable insights** and facilitate decision-making based on the underlying graph structure and properties. Additionally, the script offers flexibility by allowing users to customize queries and adapt the analysis workflow to specific use cases and requirements.

There are **2** versions of scripts, **a command-line script** and also a **gui script** - for better user experience.

Code Link in References

Queries (cypher)

- **gds -** Graph Data Science Library available in Neo4j
- **PROFILE -** To get Profile Performance results along with the **query** output
- **gds.graph.project -** used to create a **GDS graph** from **Neo4j database**
- **gds.<algorithm>.stream** - used to run the **<algorithm>** on created **GDS graph**

1. **Creating Projection using gds.graph.project**
   - CALL gds.graph.project.cypher('myGraph', MATCH (n) RETURN id(n) AS id', MATCH (n)-[:EDGE_TO]->(m) RETURN id(n) AS source, id(m) AS target') YIELD graphName, nodeCount, relationshipCount RETURN graphName, nodeCount, relationshipCount
   - creates a **projection** of the entire graph that would be required later for **gds.stream** calls
2. **All Nodes**
   - PROFILE MATCH (n) RETURN n
   - Returns all nodes from the graph
3. **Connected Nodes**
   - PROFILE MATCH (n)-[:EDGE_TO]->(m) WHERE n.id = {node_id} RETURN m
   - Returns all nodes that have **EDGE_TO** from the node **node_id**
4. **Common Neighbors**
   - PROFILE MATCH (n1)-[:EDGE_TO]->(m)<-[:EDGE_TO]->(n2) WHERE n1.id = {node_id1} AND n2.id = {node_id2} RETURN m
   - Returns all nodes that have **EDGE_TO** from both the nodes **node_id1** and **node_id2**
5. **Shortest Path**
   - PROFILE MATCH path = shortestPath((n1)-[:EDGE_TO*]->(n2)) WHERE n1.id = {node_id1} AND n2.id = {node_id2} RETURN nodes(path)
   - Returns the **shortest path** from the node **node_id1** to the node **node_id2.**
   - Uses the **Bellman-Ford Algorithm** for computing the shortest path.
6. **All Shortest Paths**
   - PROFILE MATCH path = AllShortestPath((n1)-[:EDGE_TO*]->(n2)) WHERE n1.id = {node_id1} AND n2.id = {node_id2} RETURN nodes(path)
   - Returns **all** the **shortest paths** from the node **node_id1** to the node **node_id2.**
   - Uses a repeated application of the **Bellman-Ford Algorithm** to compute all shortest length paths.
7. **K - length Paths**

- PROFILE MATCH path = (n1)-[:EDGE_TO*{k}]-(n2) WHERE n1.id = {node_id1} AND n2.id = {node_id2} RETURN nodes(path)
- Returns all the **k length** paths from the node **node_id1** to the node **node_id.**

8. **Triangle Count**
   - PROFILE MATCH (a)-[:EDGE_TO]->(b)-[:EDGE_TO]->(c)-[:EDGE_TO]->(a) RETURN count(DISTINCT [a, b, c]) AS triangle_count
   - Returns all the **number** of **triangles** in the entire graph
9. **Triangles containing a Node**
   - PROFILE MATCH (a)-[:EDGE_TO]->(b)-[:EDGE_TO]->(c)-[:EDGE_TO]->(a) WHERE a.id = {node_id} OR b.id = {node_id} OR c.id = {node_id} RETURN DISTINCT a.id, b.id, c.id
   - Returns all the **triangles** containing the node **node_id** as a **vertex** in the entire graph
10. **Clustering Coefficient**
    - PROFILE MATCH (a)-[:EDGE_TO]->(b)-[:EDGE_TO]->(c)-[:EDGE_TO]->(a) WHERE a.id = {node_id} OR b.id = {node_id} OR c.id = {node_id} RETURN count(DISTINCT) as triangles
    - Returns the **number of triangles** containing the node **node_id** as a **vertex** in the entire graph
    - PROFILE MATCH (n)-[:EDGE_TO]->(m) WHERE n.id = {node_id} RETURN count(m) AS degree - returns all the **degree** of the node **node_id**
    - *Clustering_coefficient = triangles / ((degree * (degree - 1) / 2) - triangles)*
11. **Community Detection using gds.louvain.stream**
    - Used **Louvian Algorithm (gds.louvian)** for Community Detection
    - CALL gds.louvain.stream('myGraph') YIELD nodeId, communityId RETURN gds.util.asNode(nodeId).id AS id, communityId ORDER BY [communityId, id] ASC
    - Returns the **node_id** and its **communityId** in **ASC** order
    - Used **dictionary** to store **all nodes** in a **community** i.e., same **communityId**
    - Used a **heapq** heap object to get the **top 10 communities** by **population**
12. **Page Rank using gds.pageRank.stream**
    - Used **PageRank Algorithm (gds.pageRank)** for page ranking nodes
    - CALL gds.pageRank.stream('myGraph', {{scaler: 'L1Norm'}}) YIELD nodeId, score RETURN gds.util.asNode(nodeId).id AS node, score ORDER BY score DESC
    - Returns the **node_id** and its **page rank** score **normalized** over **1**
    - Used a **heapq** heap object to get the **top 10 nodes** by **page rank** value
13. **Centrality using gds.alpha.eigenvector**
    - Used **Eigen Vector Algorithm (gds.alpha.eigenvector)** for centrality calcs.
    - CALL gds.alpha.eigenvector.write('myGraph', { writeProperty: 'eigenvector_centrality', maxIterations: 20, // Number of iterations, adjust as

needed tolerance: 0.0001 }) MATCH (n) RETURN n.id AS node, n.eigenvector_centrality AS eigenvectorCentrality ORDER BY eigenvectorCentrality DESC

- Returns the **node_id** and its **centrality** score based on **eigen vectors**
- Used a **heapq** heap object to get the **top 10 nodes** by **page rank** value

14. Connected Components using gds.wcc.stream

- Used **Weakly Connected Components Algorithm** (**gds.wcc**) for finding components in the entire graph
- CALL gds.wcc.stream('myGraph') YIELD nodeId, componentId RETURN gds.util.asNode(nodeId).id AS node, componentId ORDER BY [componentId, node] ASC
- Returns the **node_id** and its **componentId** in **ASC** order
- Used **dictionary** to store **all nodes** in a **component** i.e., same **componentId**
- Used a **heapq** heap object to get the **top 10 components** by **population**

15. GraphSage using gds.beta.graphSage

- Used **Graph Sage Algorithm** (**gds.beta.graphSage**) for **training** on the **entire** graph
- CALL gds.beta.graphSage.train('myGraph', { modelName: 'graphsage-mean', nodeLabels: ['Node'], featureProperties: ['id'], aggregator: 'mean', activationFunction: 'sigmoid', sampleSizes: [25, 10], degreeAsProperty: true, epochs: 5, searchDepth: 5, batchSize: 1000, learningRate: 0.01, embeddingSize: 16, negativeSampleWeight: 5.0, includeProperties: true })
- CALL gds.beta.graphSage.stream('myGraph', {modelName: 'graphsage-mean'}) YIELD nodeId, embedding RETURN gds.util.asNode(nodeId).id AS node, embedding ORDER BY node
- Returns the **node_id** and its **embedding**

16. Dropping Projection using gds.graph.drop

- CALL gds.graph.drop(‘myGraph’)
- Drops the **projection - GDS Graph** with label **‘myGraph’** from the **database**

References: (some - look at code for all)

Code link: https://github.com/harshith-chowdary/DBMS_Lab_Spr24_Term_Project

Video Demonstration: Dbms_Demo.mp4

1. https://en.wikipedia.org/wiki/PageRank
2. https://github.com/neo4j/neo4j
3. https://en.wikipedia.org/wiki/Clustering_coefficient
4. https://en.wikipedia.org/wiki/Louvain_method
5. https://snap.stanford.edu/data/

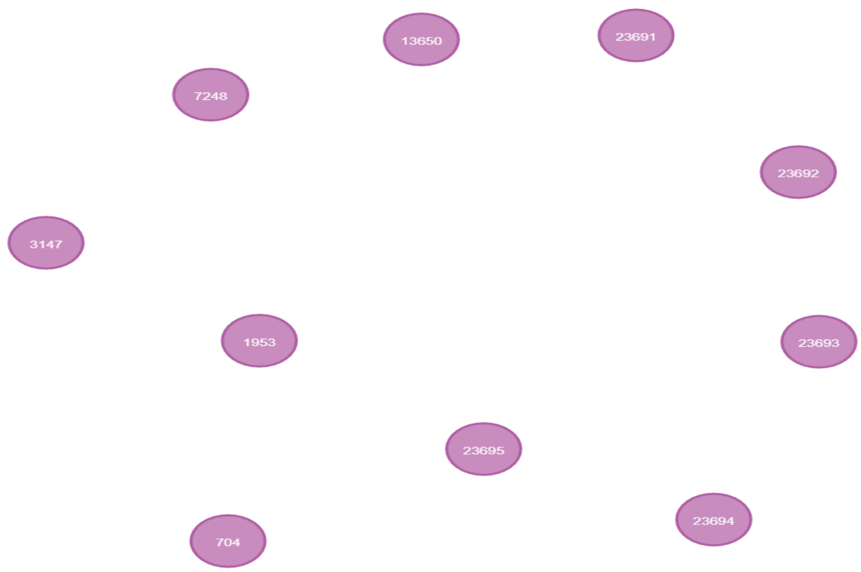
Profile Performance Computation:

- Used **PROFILE** field in front of each **Cypher** query, which returns **Profile Performace** results along with the query results.
- **Created** a **REGEX Parser** to process the **Profile Performance** results from the entire **result** string.
- Produced **performance** metrics like **time, memory, database accesses, database page cache hits/misses, pipeline information,** etc.

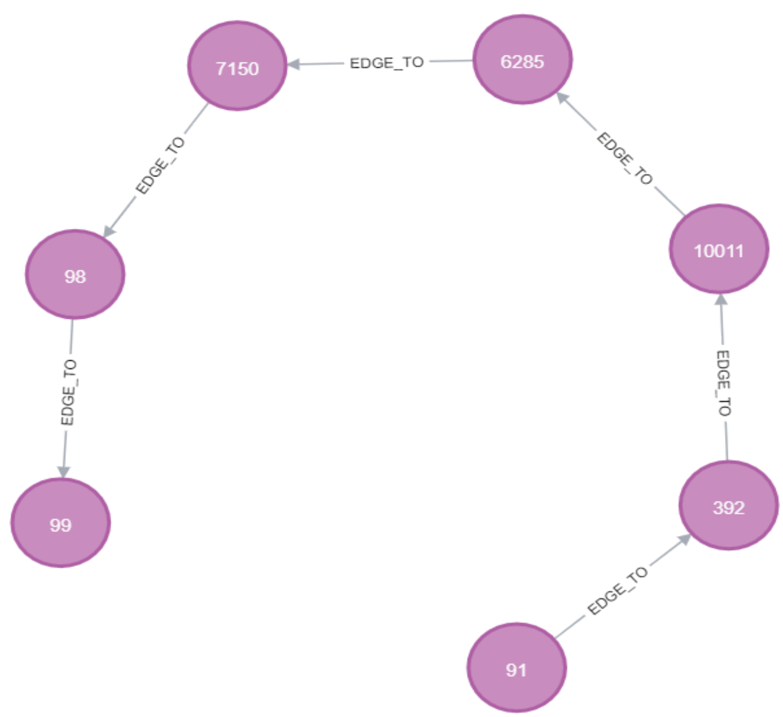
```python
def parse_profile(profile_string):
    profile_info = {}

    # Extracting operator details using regex
    operator_regex = r'\| (.+?) \| (.+?) \| (.+?) \| (.+?) \| (.+?) \| (.+?) \| (.+?) \| (.+?) \| (.+?) \| (.+?) \|'
    operator_matches = re.findall(operator_regex, profile_string, re.MULTILINE)

    # If matches found
    if operator_matches:
        operators = []
        for match in operator_matches:
            operator = {
                "Operator": match[0].strip(),
                "Id": match[1].strip(),
                "Details": match[2].strip(),
                "Estimated Rows": match[3].strip(),
                "Rows": match[4].strip(),
                "DB Hits": match[5].strip(),
                "Memory (Bytes)": match[6].strip(),
                "Page Cache Hits/Misses": match[7].strip(),
                "Time (ms)": match[8].strip(),
                "Pipeline": match[9].strip()
            }
            operators.append(operator)
        profile_info["operators"] = operators

    # Extracting database accesses and allocated memory using regex
    database_regex = r'Total database accesses: (\d+), total allocated memory: (\d+)'
    database_matches = re.search(database_regex, profile_string)
    if database_matches:
        profile_info["database_accesses"] = int(database_matches.group(1))
        profile_info["allocated_memory"] = int(database_matches.group(2))

    return profile_info
```

Results: (snaps of query results)

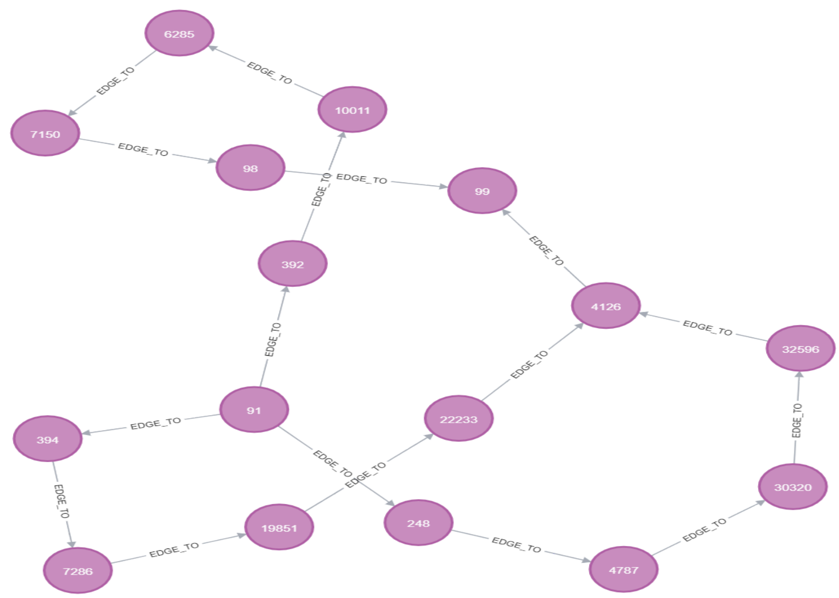
1. **Neighbors to *91***



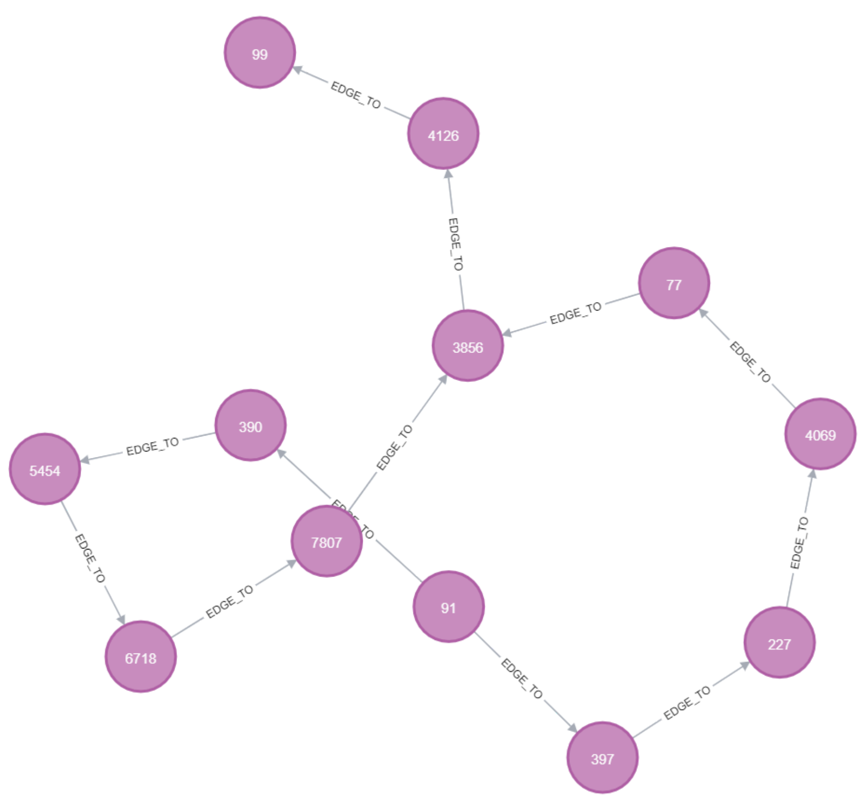
2. **Shortest Path from *91 to 99* - BEST SINGLE**



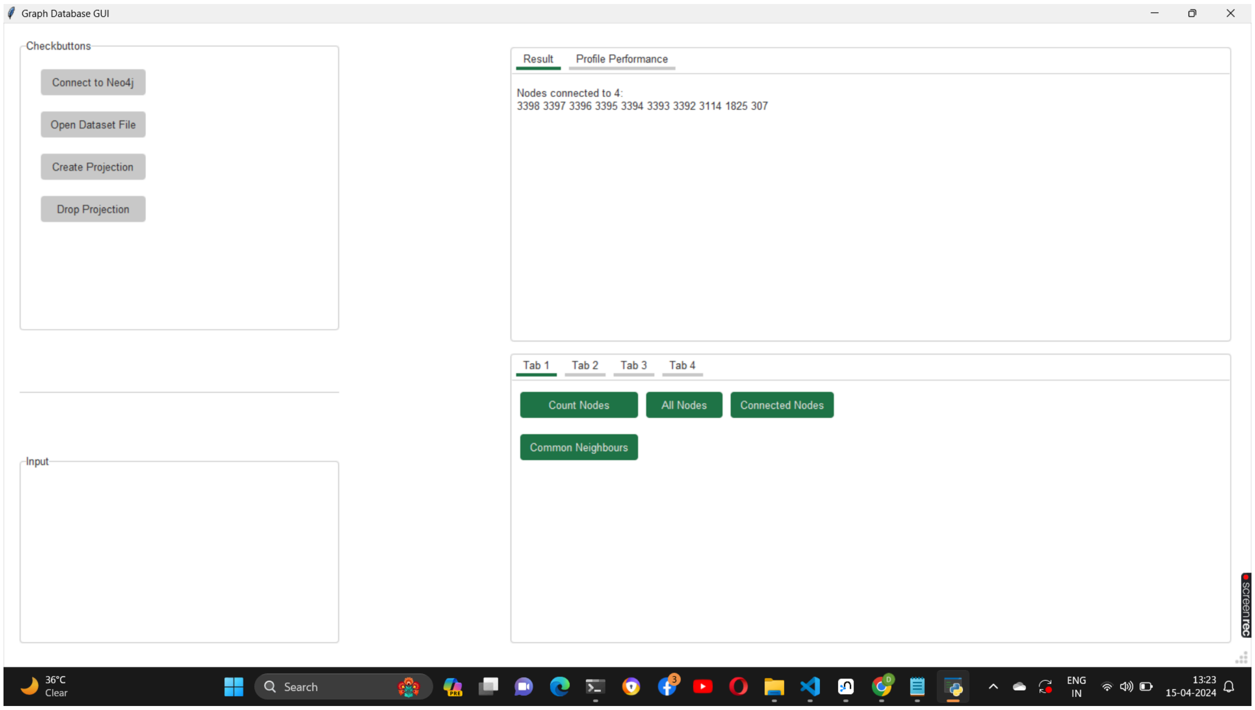
3. **Shortest Paths from *91 to 99***

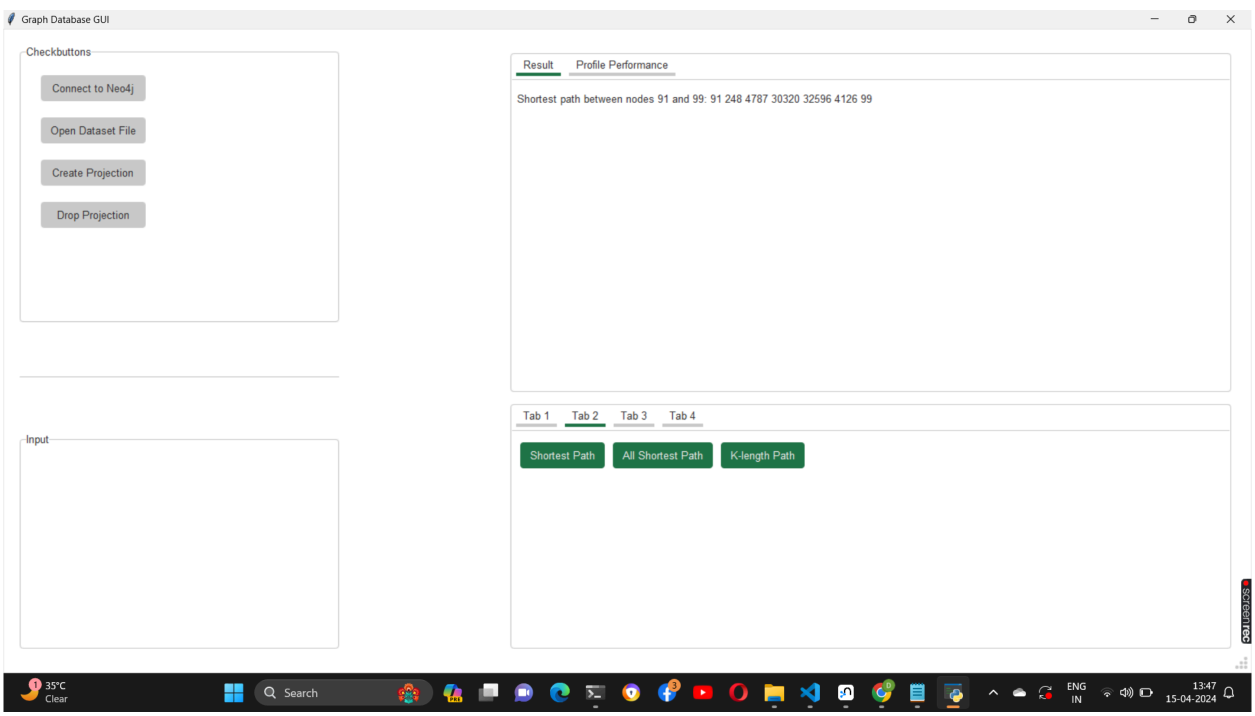


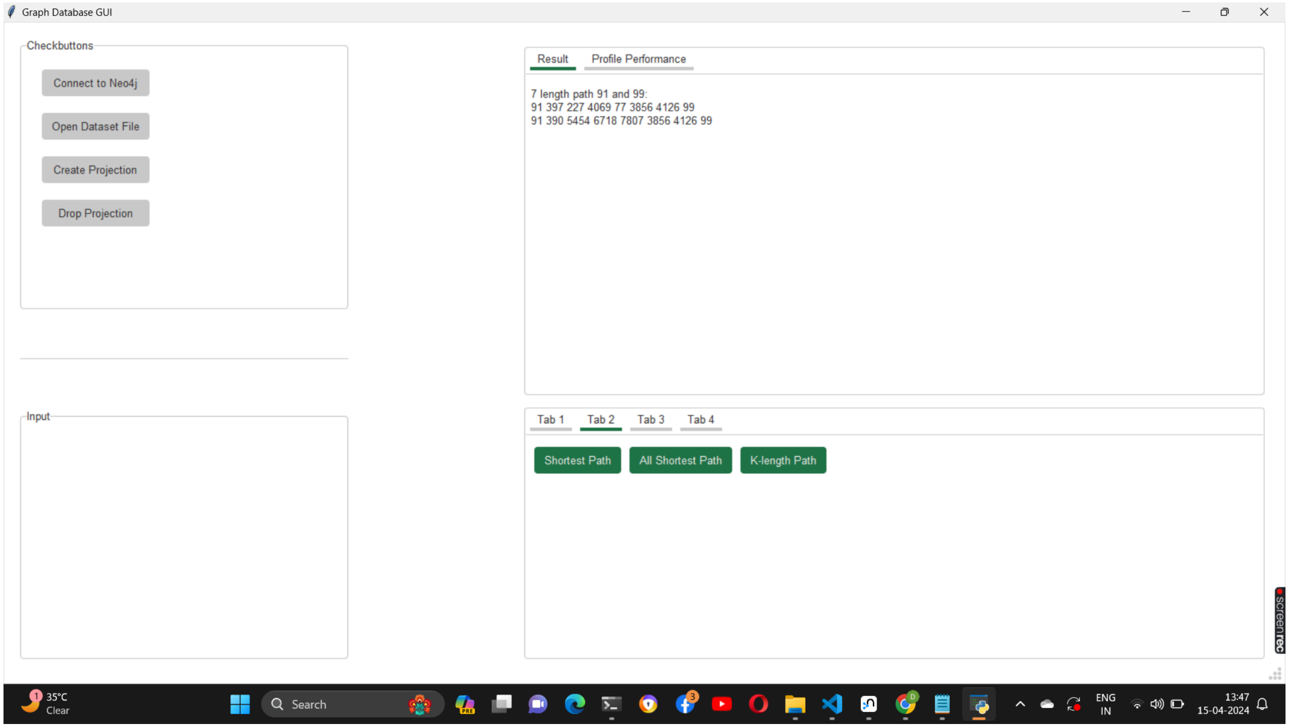
4. **Length '7' paths from *91 to 99***

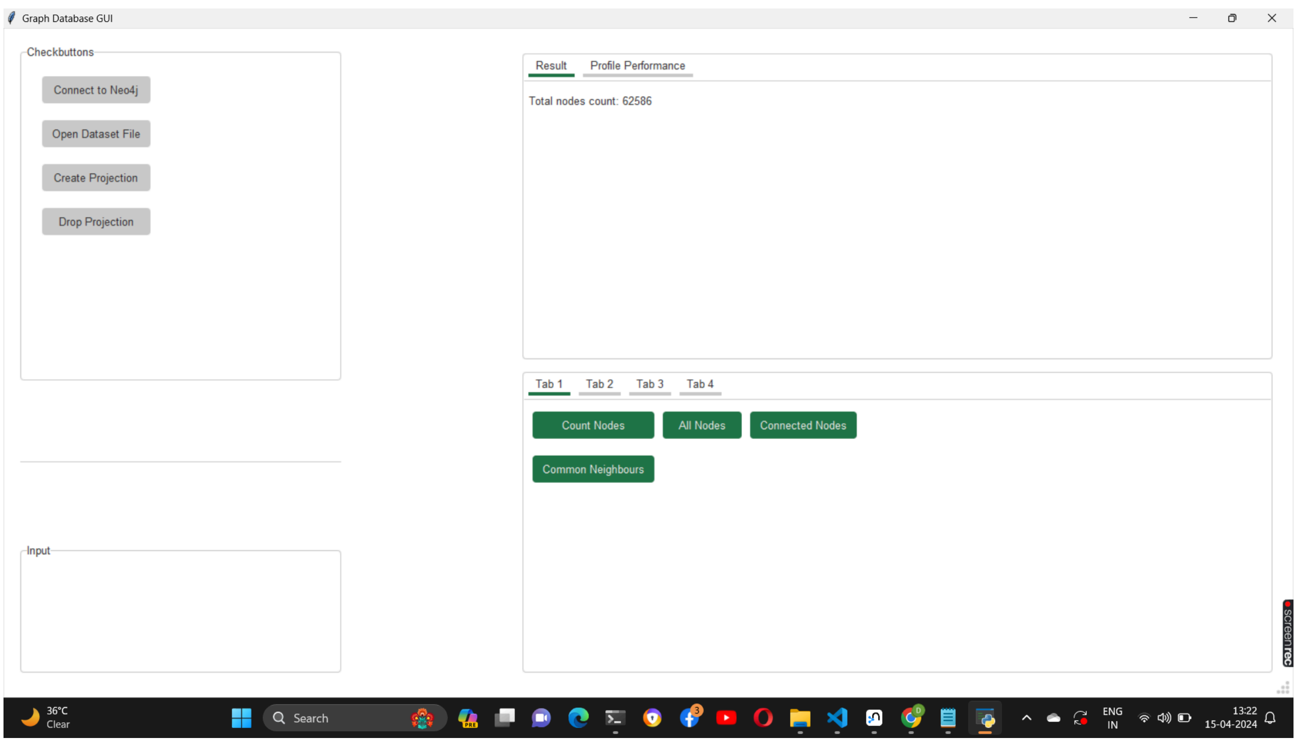


Results: (snaps of INTERFACE)









CODE & REFERENCES in ' Page 8 '