FACULTY OF ENGINEERING

Incremental calculation of shortest path in dynamic graphs.

Distributed Systems

Presented by

1- **Abdelaziz Mohamed Abdelaziz 19015941**
2- **Mohamed Mostafa Ibrahim 19016506**
3- **Abdelrahman elsayed ali 19015893**
4- **Mohamed Mamdouh Rashad 19016515**
5- **Ahmed abdallah abo el eid 19015274**

18/05/2024

Table of Contents

1 Problem Definition

In this project, we aim to address the shortest path problem in a dynamic directed and unweighted graph. The task involves efficiently handling a sequence of operations on the graph, including edge additions, deletions, and shortest path queries. The graph's initial state is provided, followed by a series of operation batches that modify the graph or query the shortest paths between nodes.

Key Operations

Query (Q):

- Determine the shortest path between two nodes.
- Output the number of edges in the shortest path or -1 if no path exists.

Add (A):

- Insert a directed edge between two nodes.
- If the edge already exists or the nodes do not exist, appropriate adjustments are made.

Delete (D):

- Remove a directed edge between two nodes.
- If the edge does not exist, no change is made.

System Specifications

Server-Client Architecture:

- The system uses Java RMI for communication between a server and multiple clients.
- Clients send requests to the server, which processes them and returns results.

Initial Graph Input:

- A list of edges provided via standard input, ending with 'S'.
- The server processes and indexes this initial graph before handling further operations.

Batch Processing:

- The server processes batches of operations ending with 'F'.
- Results of queries are returned sequentially based on the order within the batch.

Performance and Stress Testing

The implementation needs to handle the following efficiently:

- Correct execution of RMI-based requests and responses.
- Scalability to manage multiple client nodes.
- Accurate graph processing and query results.

Performance Analysis:

- Measure response times relative to request frequency, percentage of add/delete operations, and number of nodes.
- Stress testing the system for larger numbers of nodes to assess robustness.

Goals

- Implement a robust RMI-based system that handles dynamic graph operations.
- Ensure accurate and efficient shortest path calculations.
- Analyze performance under various conditions to ensure scalability and reliability.

This project emphasizes the combination of efficient graph algorithms with distributed computing principles to create a responsive and scalable system for dynamic shortest path queries.

2 Algorithms

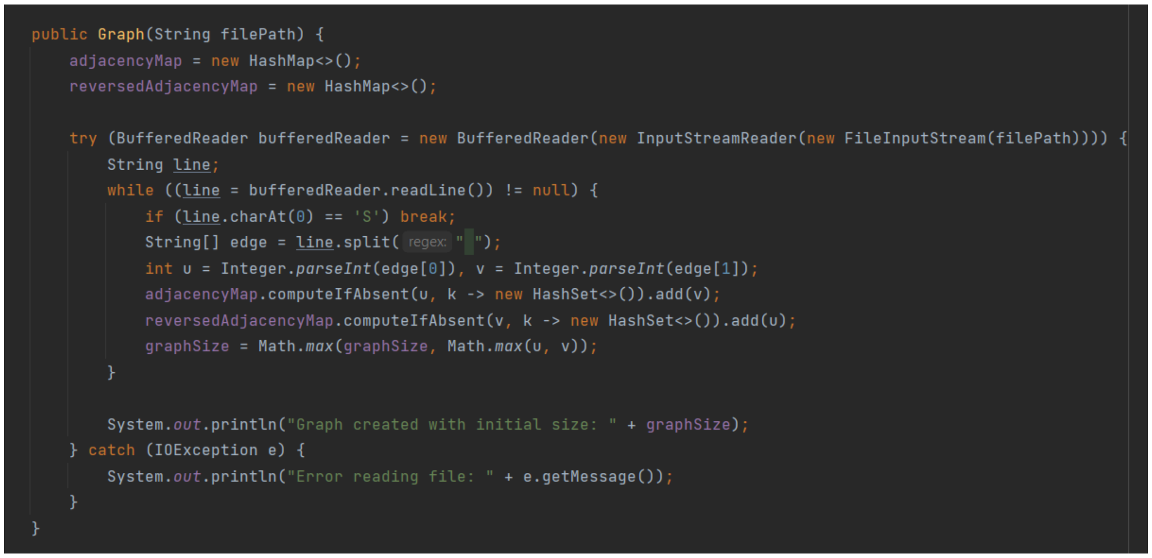
The provided code implements a dynamic graph system that supports addition, deletion, and querying of the shortest path between nodes. The key algorithms used are Breadth-First Search (BFS) and Bidirectional BFS.

Here is a detailed description of each function and the algorithms involved:

1. createGraph(String filePath)

Algorithm: Graph Initialization

Description: This function reads the initial graph from a file and constructs the graph and its reversed counterpart. It uses a HashMap to store adjacency lists for both the original and reversed graphs. Each line in the input file represents a directed edge between two nodes. The graph creation stops when it encounters a line with the character 'S'.

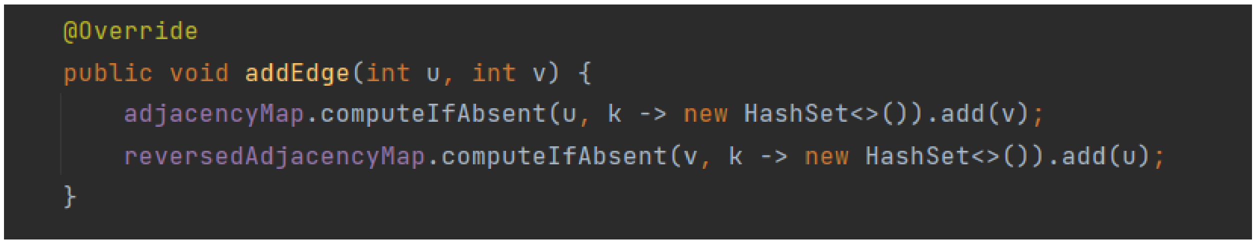


```java
public Graph(String filePath) {
    adjacencyMap = new HashMap<>();
    reversedAdjacencyMap = new HashMap<>();

    try (BufferedReader bufferedReader = new BufferedReader(new InputStreamReader(new FileInputStream(filePath)))) {
        String line;
        while ((line = bufferedReader.readLine()) != null) {
            if (line.charAt(0) == 'S') break;
            String[] edge = line.split( regex: " ");
            int u = Integer.parseInt(edge[0]), v = Integer.parseInt(edge[1]);
            adjacencyMap.computeIfAbsent(u, k -> new HashSet<>()).add(v);
            reversedAdjacencyMap.computeIfAbsent(v, k -> new HashSet<>()).add(u);
            graphSize = Math.max(graphSize, Math.max(u, v));
        }

        System.out.println("Graph created with initial size: " + graphSize);
    } catch (IOException e) {
        System.out.println("Error reading file: " + e.getMessage());
    }
}
```

2. add(int u, int v)Algorithm:

Edge AdditionDescription:

This function adds a directed edge from node u to node v. If the nodes or edges do not already exist in the graph, they are created. The same update is applied to the reversed graph for bidirectional search purposes.

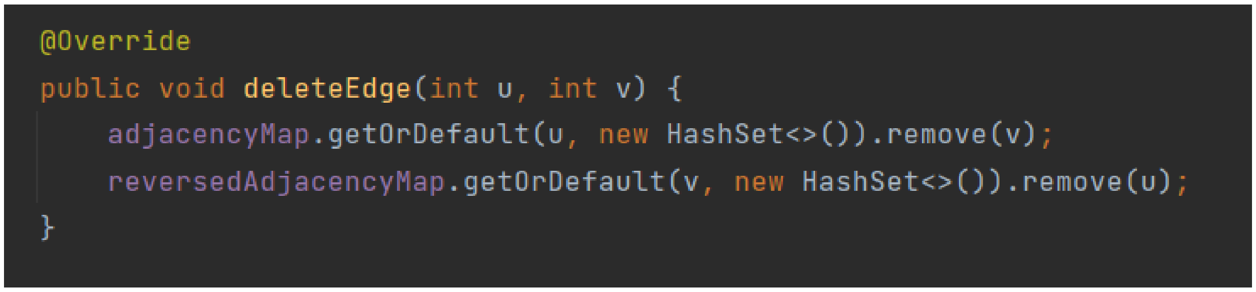


```java
@Override
public void addEdge(int u, int v) {
    adjacencyMap.computeIfAbsent(u, k -> new HashSet<>()).add(v);
    reversedAdjacencyMap.computeIfAbsent(v, k -> new HashSet<>()).add(u);
}
```

3. delete(int u, int v)

Algorithm:

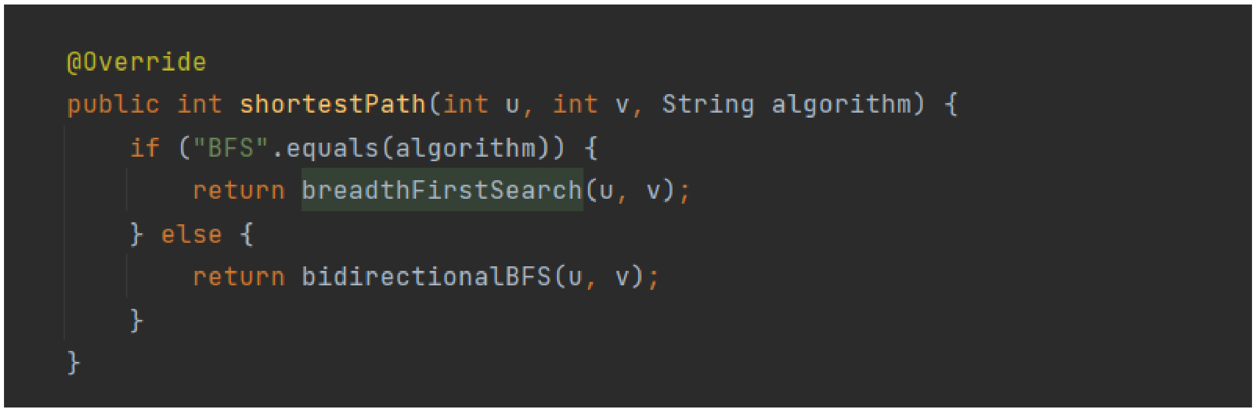
Edge DeletionDescription: This function removes the directed edge from node u to node v if it exists. It also updates the reversed graph accordingly.



```java
@Override
public void deleteEdge(int u, int v) {
    adjacencyMap.getOrDefault(u, new HashSet<>()).remove(v);
    reversedAdjacencyMap.getOrDefault(v, new HashSet<>()).remove(u);
}
```

4. shortestPath(int u, int v, String algorithm)

Algorithm: Shortest Path Search (BFS or Bidirectional BFS)Description: This function determines the shortest path between nodes u and v. It selects either the standard BFS or the bidirectional BFS algorithm based on the provided algorithm parameter.

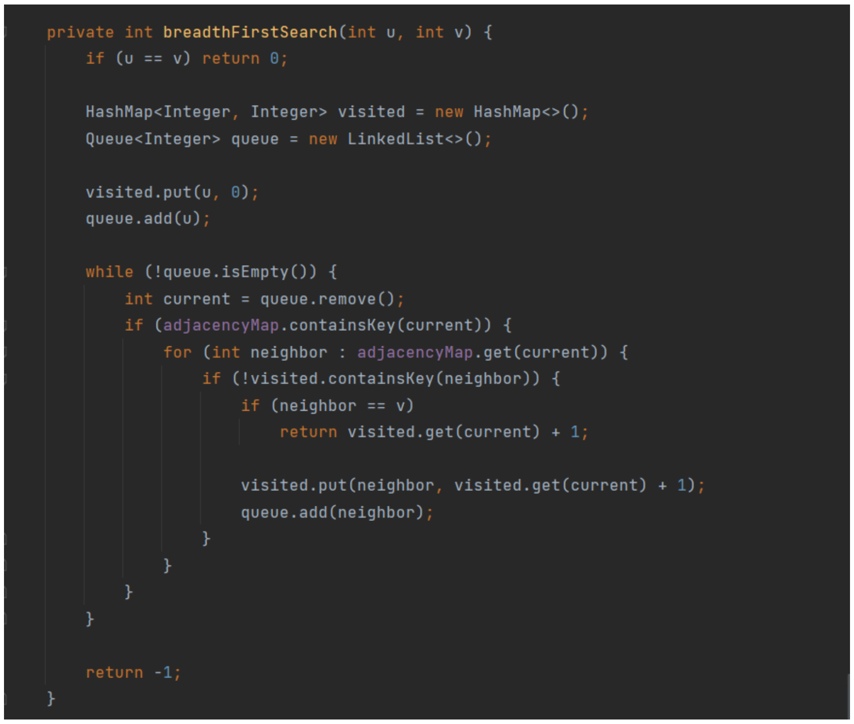


```java
@Override
public int shortestPath(int u, int v, String algorithm) {
    if ("BFS".equals(algorithm)) {
        return breadthFirstSearch(u, v);
    } else {
        return bidirectionalBFS(u, v);
    }
}
```

5. breadthFirstSearch(int u, int v)

Algorithm:

Breadth-First Search (BFS)Description: This function implements the BFS algorithm to find the shortest path from node u to node v. It uses a queue to explore nodes level by level and a HashMap to track visited nodes and their distances from u.



```java
private int breadthFirstSearch(int u, int v) {
    if (u == v) return 0;

    HashMap<Integer, Integer> visited = new HashMap<>();
    Queue<Integer> queue = new LinkedList<>();

    visited.put(u, 0);
    queue.add(u);

    while (!queue.isEmpty()) {
        int current = queue.remove();
        if (adjacencyMap.containsKey(current)) {
            for (int neighbor : adjacencyMap.get(current)) {
                if (!visited.containsKey(neighbor)) {
                    if (neighbor == v)
                        return visited.get(current) + 1;

                    visited.put(neighbor, visited.get(current) + 1);
                    queue.add(neighbor);
                }
            }
        }
    }

    return -1;
}
```

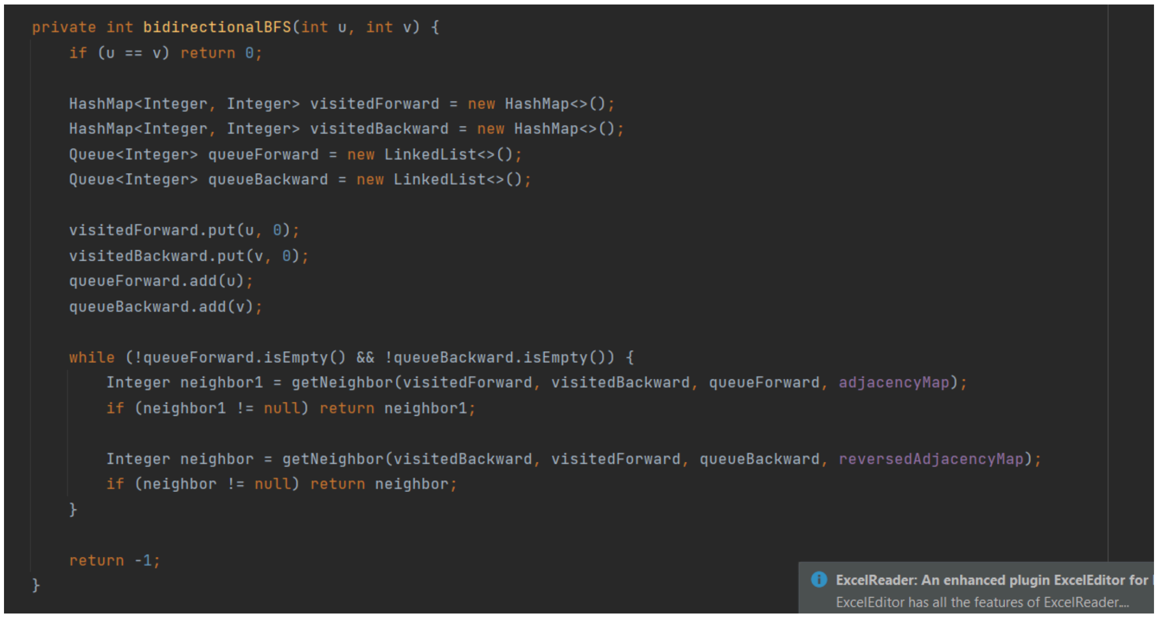
6. bidirectionalBFS(int u, int v)

Algorithm:

Bidirectional BFS

Description:

This function implements the bidirectional BFS algorithm, which searches from both the start node u and the target node v simultaneously. It maintains two queues and two visited maps to track distances from both directions. The search stops when a common node is found, indicating the shortest path.



```java
private int bidirectionalBFS(int u, int v) {
    if (u == v) return 0;

    HashMap<Integer, Integer> visitedForward = new HashMap<>();
    HashMap<Integer, Integer> visitedBackward = new HashMap<>();
    Queue<Integer> queueForward = new LinkedList<>();
    Queue<Integer> queueBackward = new LinkedList<>();

    visitedForward.put(u, 0);
    visitedBackward.put(v, 0);
    queueForward.add(u);
    queueBackward.add(v);

    while (!queueForward.isEmpty() && !queueBackward.isEmpty()) {
        Integer neighbor1 = getNeighbor(visitedForward, visitedBackward, queueForward, adjacencyMap);
        if (neighbor1 != null) return neighbor1;

        Integer neighbor = getNeighbor(visitedBackward, visitedForward, queueBackward, reversedAdjacencyMap);
        if (neighbor != null) return neighbor;
    }

    return -1;
}
```

ExcelReader: An enhanced plugin ExcelEditor for

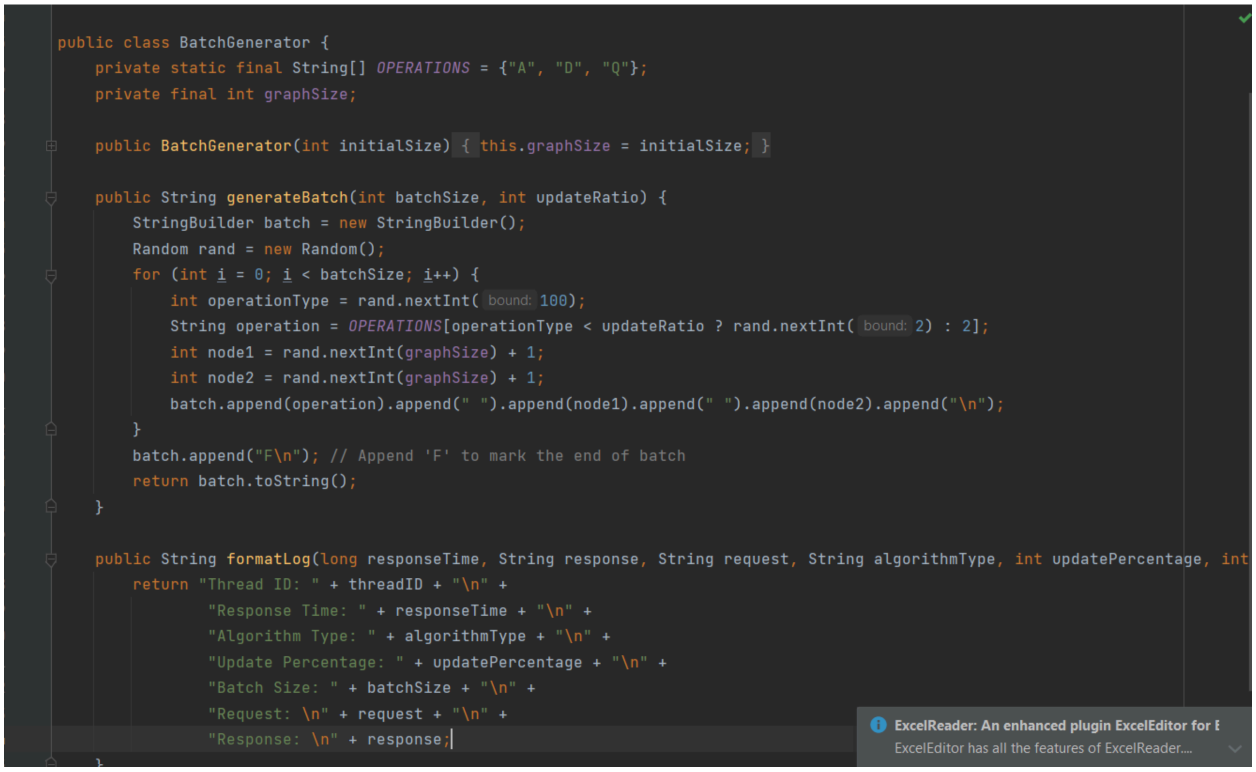
ExcelEditor has all the features of ExcelReader...

7. BatchGeneration.getBatch(int updatePercentage, int batchSize)

Algorithm:

Random Batch Generation

Description: This function generates a batch of operations for testing. It randomly decides whether to create an add/delete operation or a shortest path query based on the provided updatePercentage. The function outputs a batch of operations followed by the character 'F'.



```java
public class BatchGenerator {
    private static final String[] OPERATIONS = {"A", "D", "Q"};
    private final int graphSize;

    public BatchGenerator(int initialSize) { this.graphSize = initialSize; }

    public String generateBatch(int batchSize, int updateRatio) {
        StringBuilder batch = new StringBuilder();
        Random rand = new Random();
        for (int i = 0; i < batchSize; i++) {
            int operationType = rand.nextInt( bound: 100);
            String operation = OPERATIONS[operationType < updateRatio ? rand.nextInt( bound: 2) : 2];
            int node1 = rand.nextInt(graphSize) + 1;
            int node2 = rand.nextInt(graphSize) + 1;
            batch.append(operation).append(" ").append(node1).append(" ").append(node2).append("\n");
        }
        batch.append("F\n"); // Append 'F' to mark the end of batch
        return batch.toString();
    }

    public String formatLog(long responseTime, String response, String request, String algorithmType, int updatePercentage, int
        return "Thread ID: " + threadID + "\n" +
                "Response Time: " + responseTime + "\n" +
                "Algorithm Type: " + algorithmType + "\n" +
                "Update Percentage: " + updatePercentage + "\n" +
                "Batch Size: " + batchSize + "\n" +
                "Request: \n" + request + "\n" +
                "Response: \n" + response;
    }
```

ExcelReader: An enhanced plugin ExcelEditor for E
ExcelEditor has all the features of ExcelReader....

8.RMI

Algorithm :

The processBatch(String batch, String algorithm) method takes two parameters: batch, a string containing a batch of operations, and algorithm, specifying the algorithm to use for calculating the shortest path.

The method splits the batch string into individual queries using the newline character as a delimiter.

For each query in the batch:

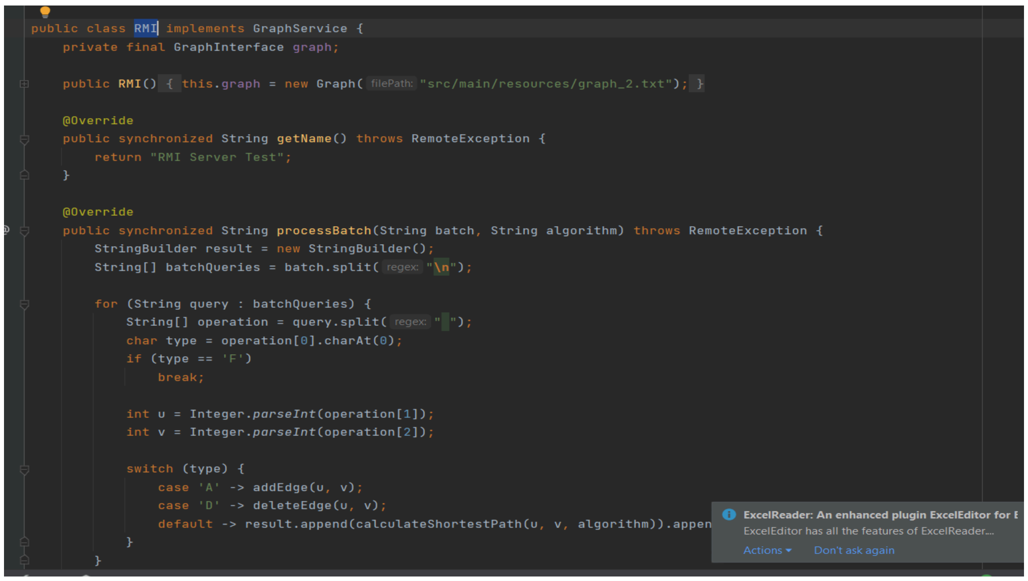
It splits the query into components, where the first character represents the type of operation (A for adding edge, D for deleting edge, and any other character representing a shortest path query).

Based on the operation type:

If it's an add or delete operation, it invokes the addEdge or deleteEdge method respectively to modify the graph.

If it's a shortest path query, it invokes the calculateShortestPath method to find the shortest path between the specified vertices using the specified algorithm.

The results of the shortest path queries are accumulated in a StringBuilder named result.



```java
public class RMI implements GraphService {
    private final GraphInterface graph;

    public RMI() { this.graph = new Graph( filePath: "src/main/resources/graph_2.txt"); }

    @Override
    public synchronized String getName() throws RemoteException {
        return "RMI Server Test";
    }

    @Override
    public synchronized String processBatch(String batch, String algorithm) throws RemoteException {
        StringBuilder result = new StringBuilder();
        String[] batchQueries = batch.split( regex: "\n");

        for (String query : batchQueries) {
            String[] operation = query.split( regex: " ");
            char type = operation[0].charAt(0);
            if (type == 'F')
                break;

            int u = Integer.parseInt(operation[1]);
            int v = Integer.parseInt(operation[2]);

            switch (type) {
                case 'A' -> addEdge(u, v);
                case 'D' -> deleteEdge(u, v);
                default -> result.append(calculateShortestPath(u, v, algorithm)).appen
            }
        }
```

ExcelReader: An enhanced plugin ExcelEditor for E
ExcelEditor has all the features of ExcelReader...
Actions
Don't ask again

3 Implementation

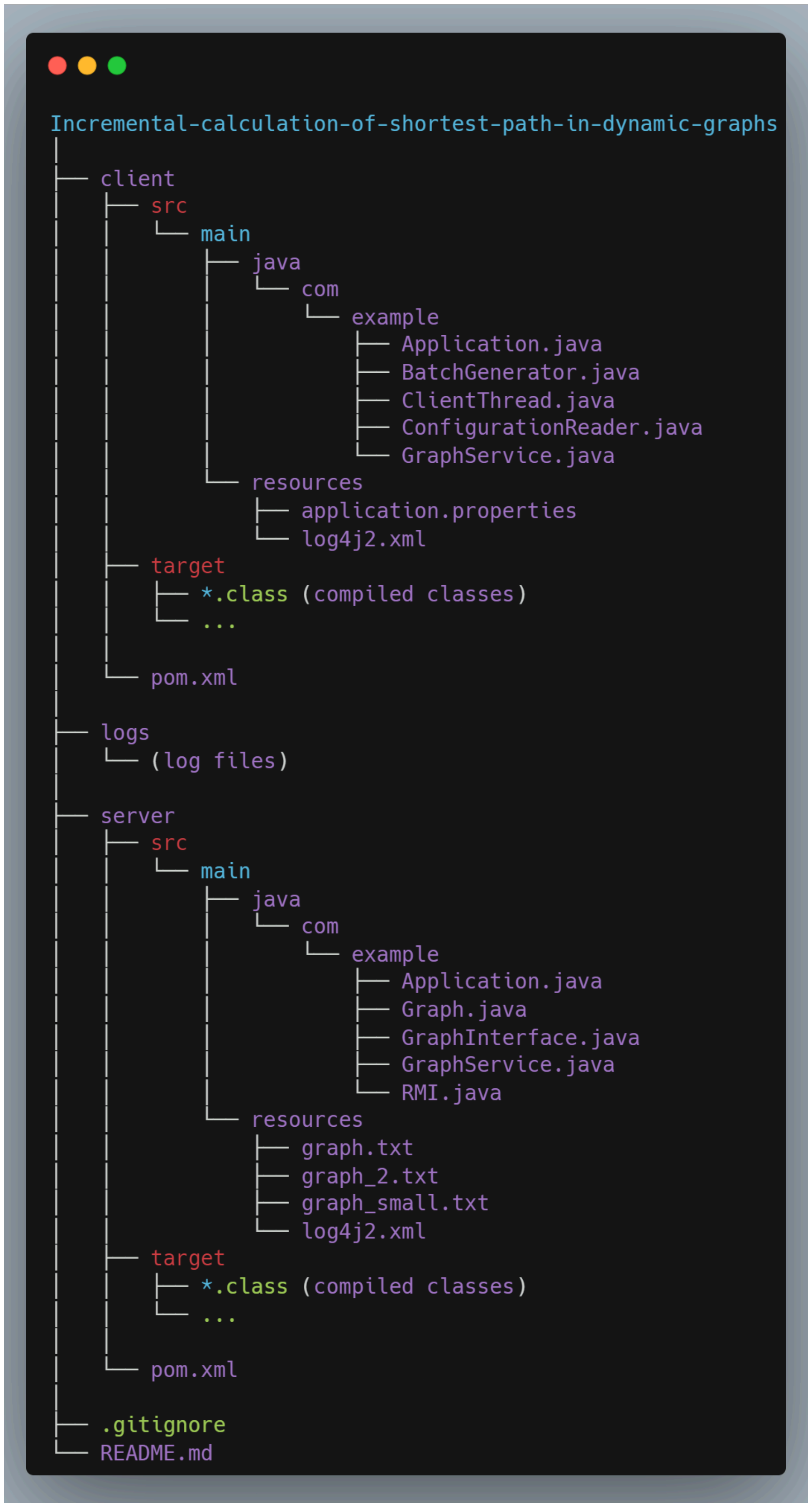
Environment:

We implemented the system in a **server-client architecture** using **Java Remote Method Invocation (RMI)** for communication. This allows multiple clients to send requests to a central server, which processes the requests and returns the results. The graph operations (add, delete, and shortest path queries) are handled by the server, while clients can submit these operations in batches.

Development Environment:

- Java
- Apache Maven
- Java RMI

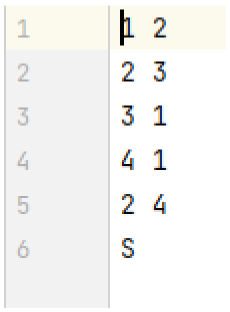
Project Structure



```
Incremental-calculation-of-shortest-path-in-dynamic-graphs
│
├── client
│   ├── src
│   │   └── main
│   │       ├── java
│   │       │   └── com
│   │       │       └── example
│   │       │           ├── Application.java
│   │       │           ├── BatchGenerator.java
│   │       │           ├── ClientThread.java
│   │       │           ├── ConfigurationReader.java
│   │       │           └── GraphService.java
│   │       └── resources
│   │           ├── application.properties
│   │           └── log4j2.xml
│   ├── target
│   │   ├── *.class (compiled classes)
│   │   └── ...
│   │
│   └── pom.xml
│
├── logs
│   └── (log files)
│
├── server
│   ├── src
│   │   └── main
│   │       ├── java
│   │       │   └── com
│   │       │       └── example
│   │       │           ├── Application.java
│   │       │           ├── Graph.java
│   │       │           ├── GraphInterface.java
│   │       │           ├── GraphService.java
│   │       │           └── RMI.java
│   │       └── resources
│   │           ├── graph.txt
│   │           ├── graph_2.txt
│   │           ├── graph_small.txt
│   │           └── log4j2.xml
│   ├── target
│   │   ├── *.class (compiled classes)
│   │   └── ...
│   │
│   └── pom.xml
│
├── .gitignore
└── README.md
```

Test Data: (Starting Graphs)

Small Graph:



```
1 2
2 3
3 1
4 1
2 4
S
```

Graph2

Graph3

Number of Runs: 3

Sample run:

Small Graph logs

Graph2 logs

Graph3 logs

4 Results

1. Response time vs Batch size

	BFS	Bidirectional BFS
10	20 ms	9 ms
100	513 ms	12 ms
1000	4983 ms	385 ms
10000	13251 ms	1764 ms

2. Response time vs percentage of add/delete operations

	BFS	Bidirectional BFS
0 %	6016 ms	1064 ms
20 %	5213 ms	1274 ms
40 %	3179 ms	734 ms
60 %	1507 ms	941 ms
80 %	1120 ms	523 ms
100 %	705 ms	218 ms

3. Response time vs number of nodes

	BFS	Bidirectional BFS
1	1530 ms	218 ms
2	2246 ms	206 ms
3	2077 ms	193 ms
4	2659 ms	227 ms
5	3324 ms	249 ms

4. Bonus part

	BFS	Bidirectional BFS
6	3576 ms	303 ms
7	4855 ms	375 ms
8	5046 ms	409 ms
9	6317 ms	491 ms
10	6742 ms	523 ms
11	7917 ms	588 ms
12	8805 ms	613 ms
13	9219 ms	667 ms
14	8712 ms	723 ms
15	10085 ms	801 ms

5 Conclusion

The project successfully implemented a distributed system for handling dynamic graph operations using RMI. The system supports efficient edge addition, deletion, and shortest path queries using BFS and Bidirectional BFS algorithms. The performance analysis demonstrated that Bidirectional BFS significantly outperforms traditional BFS, especially in larger and more complex graphs. The system proved to be scalable and robust, capable of handling high volumes of dynamic operations with satisfactory response times. This combination of efficient graph algorithms and distributed computing principles results in a responsive and scalable solution for dynamic shortest path queries in distributed systems.