Dijkstra's Algorithm Implementation

Dijkstra's Algorithm finds the shortest path from a single source node to all other nodes in a graph with non-negative edge weights. It works by iteratively selecting the node with the smallest known distance, updating its neighbors' distances, and repeating this until all nodes are processed.

Key Idea: Gradually explore the graph, ensuring the shortest path to each node is found step by step.

Input: A graph (can be represented as an adjacency list/matrix) we use adjacency list in our implementation with nodes, edges, and non-negative weights.

Output: The shortest distance from the source to all other nodes.

Type of Graph: Works on both directed and undirected graphs.

```java
import java.util.*;

class DijkstraAlgorithm {  3 usages

    static class Edge {  10 usages
        int target, weight;  5 usages

        Edge(int target, int weight) {  3 usages
            this.target = target;
            this.weight = weight;
        }
    }

    // Dijkstra's Algorithm implementation
    public static void dijkstra(List<List<Edge>> graph, int source) {  1 usage
        int n = graph.size();
        int[] distances = new int[n];
        Arrays.fill(distances, Integer.MAX_VALUE);
        distances[source] = 0;

        PriorityQueue<Edge> pq = new PriorityQueue<>(Comparator.comparingInt(edge -> edge.weight));
        pq.add(new Edge(source,  weight: 0));

        while (!pq.isEmpty()) {
            Edge current = pq.poll();
            int currentNode = current.target;
            int currentDistance = current.weight;

            if (currentDistance > distances[currentNode]) continue;

            for (Edge neighbor : graph.get(currentNode)) {
                int newDist = currentDistance + neighbor.weight;
                if (newDist < distances[neighbor.target]) {
                    distances[neighbor.target] = newDist;
                    pq.add(new Edge(neighbor.target, newDist));
                }
            }
        }
    }
```

Class Edge:
Input: target (the destination node), weight (the edge weight)
- Set target to the given target node
- Set weight to the given weight

Function dijkstra(graph, source):
1. Initialize:
- distances: an array of size n (number of nodes), filled with infinity
- Set distances[source] = 0 (distance to the source itself is zero)
- Create a priority queue named pq to handle nodes
- Add the source node to pq with distance 0

2. While the priority queue is not empty:
- Remove the node with the smallest distance (current)
- If current's distance is outdated, skip to the next iteration

3. For each neighbor of the current node:
- Calculate new distance: currentDistance + neighbor's weight
- If the new distance is smaller than the recorded distance:
- Update distances array with the new distance
- Add the neighbor to the priority queue with the updated distance

4. At the End of the function: distances array contains the shortest path from the source to all nodes

```java
public static List<List<Edge>> generateRandomGraph(int n, int edges) {  1 usage
    Random rand = new Random();
    List<List<Edge>> graph = new ArrayList<>();
    for (int i = 0; i < n; i++) {
        graph.add(new ArrayList<>());
    }

    for (int i = 0; i < edges; i++) {
        int source = rand.nextInt(n);
        int target = rand.nextInt(n);
        int weight = rand.nextInt( bound: 10) + 1;

        graph.get(source).add(new Edge(target, weight));
    }

    return graph;
}
```

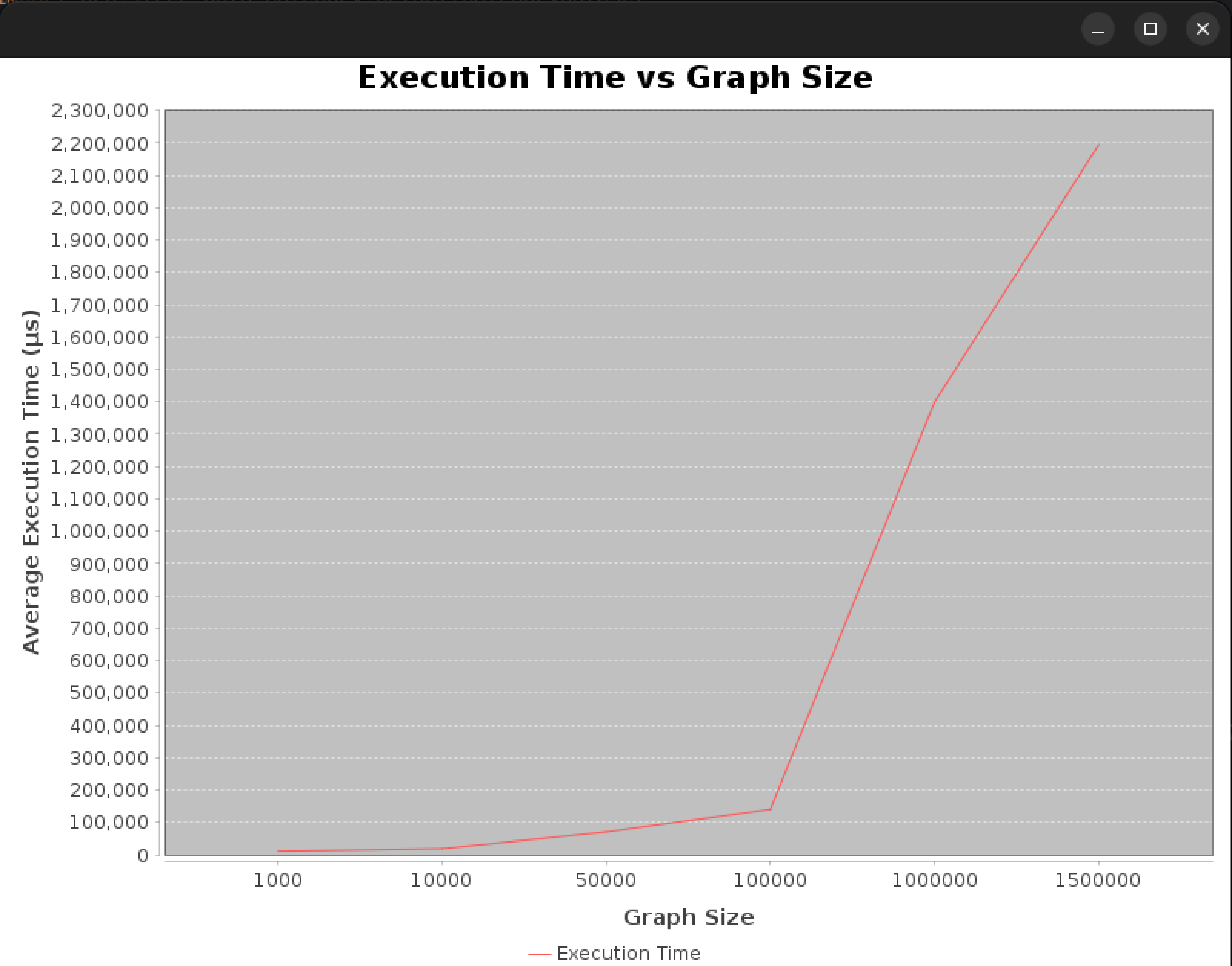
```
Function generateRandomGraph(n, edges):
    Input:
        - n: Number of nodes
        - edges: Number of edges to create
    Output:
        - A graph represented as an adjacency list

    1. Initialize an empty graph:
        - Create a list graph with n empty lists (one for each node)

    2. For each edge (up to the specified number of edges):
        - Randomly pick a source node (0 to n-1)
        - Randomly pick a target node (0 to n-1)
        - Randomly generate a weight (1 to 10)

        - Add an edge from source to target with the generated weight

    3. Return the graph as an adjacency list
```

```java
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        int[] sizes = {1_000, 10_000, 50_000,100_000, 1_000_000, 1_500_000};
        String fileName = "execution_times.csv";
        int runs = 10;

        try (FileWriter writer = new FileWriter(fileName)) {
            writer.write("GraphSize,AverageExecutionTime\n");

            for (int n : sizes) {
                long totalExecutionTime = 0;

                for (int i = 0; i < runs; i++) {
                    List<List<DijkstraAlgorithm.Edge>> graph = DijkstraAlgorithm.generateRandomGraph(n, n*2);

                    long startTime = System.nanoTime();
                    DijkstraAlgorithm.dijkstra(graph, 0);
                    long endTime = System.nanoTime();

                    long executionTime = (endTime - startTime) / 1_000;
                    totalExecutionTime += executionTime;
                }
                long averageExecutionTime = totalExecutionTime / runs;
                writer.write(n + "," + averageExecutionTime + " µs\n");
                System.out.println("Graph size: " + n + " | Average execution time: " + averageExecutionTime + " µs");
                System.out.println("remaining heap size (bytes): " + (Runtime.getRuntime().maxMemory()-Runtime.getRuntime().freeMemory()));

            }

            System.out.println("Execution times saved to " + fileName);
        } catch (IOException e) {
            System.err.println("Error writing to file: " + e.getMessage());
        }
    }
}
```

Function Main():

```
1. Define input parameters:
    - sizes: Array of graph sizes to test
    - fileName: Name of the CSV file to save results
    - runs: Number of runs per graph size

2. Open a FileWriter to write results to csv file
    - make CSV header: "GraphSize,AverageExecutionTime"

3. For each graph size n in sizes:
    - Initialize totalExecutionTime to 0

    - Repeat for `runs` times:
        a. Generate a random graph with n nodes and 2*n until to m * n edges as u like
        b. Record the start time
        c. Run Dijkstra's Algorithm on the graph starting from node 0
        d. calculate the end time
        e. Calculate execution time in microseconds (endTime - startTime) / 1_000
        f. Add execution time to totalExecutionTime

    - Calculate the average execution time (totalExecutionTime / runs)
    - Write the graph size and average execution time to the CSV file
    - Print the results to the console

4. Close the FileWriter
```

```
GraphSize,AverageExecutionTime
1000,11047 µs
10000,18717 µs
50000,70611 µs
100000,139897 µs
1000000,1398116 µs
1500000,2193247 µs
```


Execution Time vs Graph Size
Average Execution Time (µs)
2,300,000
2,200,000
2,100,000
2,000,000
1,900,000
1,800,000
1,700,000
1,600,000
1,500,000
1,400,000
1,300,000
1,200,000
1,100,000
1,000,000
900,000
800,000
700,000
600,000
500,000
400,000
300,000
200,000
100,000
0
1000
10000
50000
100000
1000000
1500000
Graph Size
Execution Time

Summary of Time Complexity:

1. Before Treating Edges as ($2n$):
   - Graph Generation: ($O(\text{edges})$)
   - Dijkstra's Algorithm: ($O((n + \text{edges}) \log n)$)

2. After Treating Edges as ($2n$):
   - Graph Generation: ($O(n)$), because the number of edges is set to $2n$.
   - Dijkstra's Algorithm: ($O(n \log n)$)

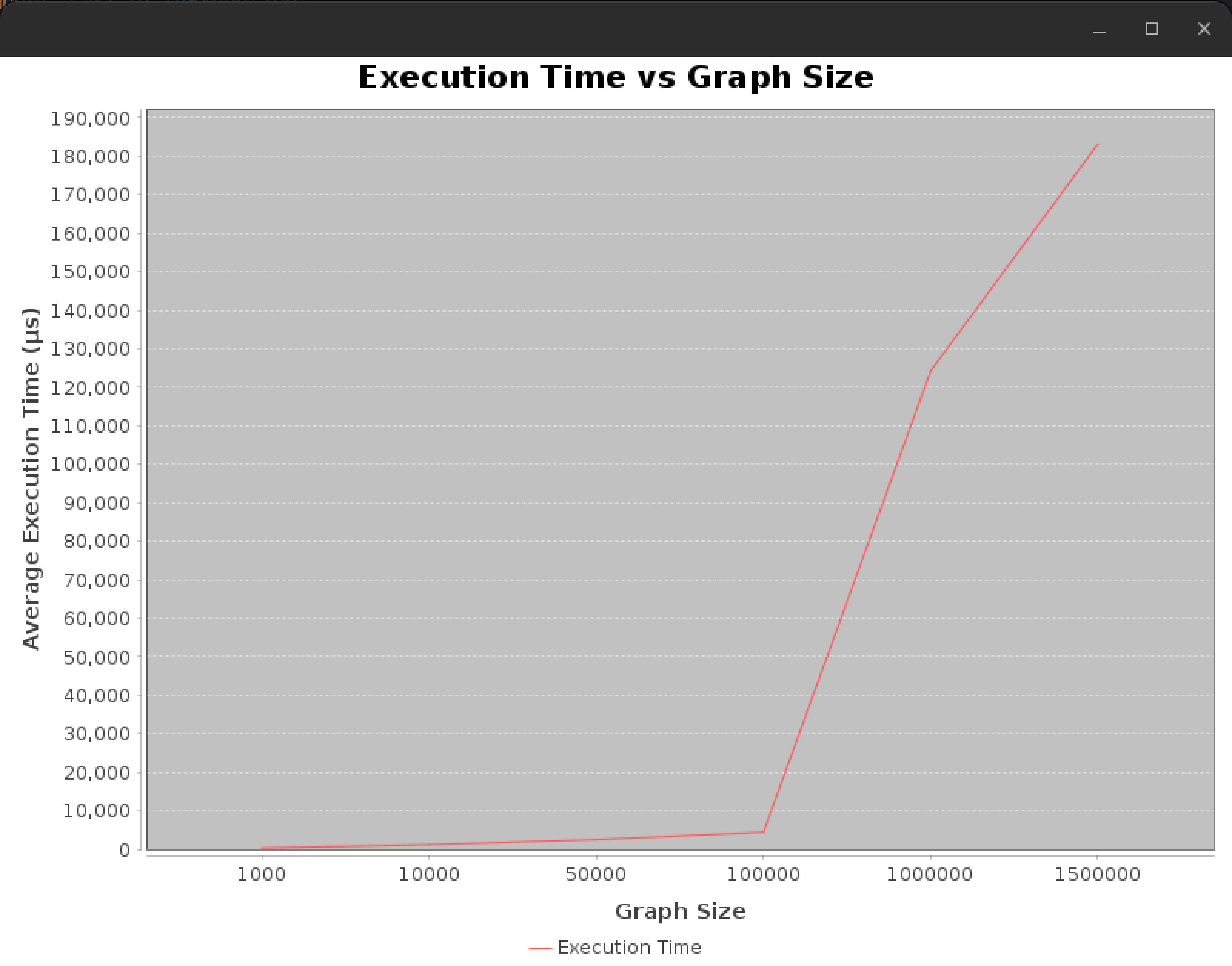
Overall Time Complexity:
- Before treating edges as ($2n$): ($O(e) + O((n + e) \log n)$.
- After treating edges as ($2n$): ($O(n \log n)$).

```
public static void dijkstra(List<List<Edge>> graph, int source) {  no usages
    int n = graph.size();
    FibonacciHeap fibonacciHeap = new FibonacciHeap();
    FibonacciHeap.Node[] nodes = new FibonacciHeap.Node[n];


    for (int i = 0; i < n; i++) {
        nodes[i] = fibonacciHeap.insert(i, Integer.MAX_VALUE);
    }


    fibonacciHeap.decreaseKey(nodes[source],  newDistance: 0);

    while (!fibonacciHeap.isEmpty()) {
        FibonacciHeap.Node currentNode = fibonacciHeap.extractMin();
        int current = currentNode.value;
        int currentDistance = currentNode.distance;

        for (Edge neighbor : graph.get(current)) {
            int newDist = currentDistance + neighbor.weight;
            if (newDist < currentNode.distance) {
                fibonacciHeap.decreaseKey(nodes[neighbor.target], newDist);
            }
        }
    }
}
```

```
function dijkstra(graph, source):
    n = size of graph
    heap = new FibonacciHeap()
    nodes = array of size n

    for i = 0 to n-1:
        nodes[i] = heap.insert(∞)

    heap.decreaseKey(nodes[source], 0)

    while heap is not empty:
        currentNode = heap.extractMin()
        if currentNode is null:
            break

        current = currentNode.value
        currentDistance = currentNode.key

        for each neighbor in graph[current]:
            newDist = currentDistance + neighbor.weight
            if newDist < currentDistance:
                heap.decreaseKey(nodes[neighbor.target], newDist)
```

```java
import java.io.FileWriter;
import java.io.IOException;
import java.util.List;

public class Main {
    public static void main(String[] args) {
        int[] sizes = {1_000, 10_000, 50_000,100_000, 1_000_000, 1_500_000};
        String fileName = "execution_times.csv";
        int runs = 10;

        try (FileWriter writer = new FileWriter(fileName)) {
            writer.write("GraphSize,AverageExecutionTime\n");

            for (int n : sizes) {
                long totalExecutionTime = 0;

                for (int i = 0; i < runs; i++) {
                    List<List<DijkstraWithFibonacciHeap.Edge>> graph = DijkstraWithFibonacciHeap.generateRandomGraph(n, n*2);

                    long startTime = System.nanoTime();
                    DijkstraWithFibonacciHeap.dijkstra(graph, 0);
                    long endTime = System.nanoTime();

                    long executionTime = (endTime - startTime) / 1_000;
                    totalExecutionTime += executionTime;
                }
                long averageExecutionTime = totalExecutionTime / runs;
                writer.write(n + "," + averageExecutionTime + " µs\n");
                System.out.println("Graph size: " + n + " | Average execution time: " + averageExecutionTime + " µs");
                System.out.println("remaining heap size (bytes): " + (Runtime.getRuntime().maxMemory()-Runtime.getRuntime().freeMemory()));

            }

            System.out.println("Execution times saved to " + fileName);
        } catch (IOException e) {
            System.err.println("Error writing to file: " + e.getMessage());
        }
    }
}
```

```
GraphSize,AverageExecutionTime
1000,313 µs
10000,1168 µs
50000,2502 µs
100000,4420 µs
1000000,124236 µs
1500000,183157 µs
```


Execution Time vs Graph Size
Average Execution Time (µs)
190,000
180,000
170,000
160,000
150,000
140,000
130,000
120,000
110,000
100,000
90,000
80,000
70,000
60,000
50,000
40,000
30,000
20,000
10,000
0
1000
10000
50000
100000
1000000
1500000
Graph Size
Execution Time

Summary of Time Complexity:

Before Treating Edges as (2n):
1. Graph Generation:
  - Time Complexity: (O(edges))
2. Dijkstra's Algorithm:
  - Extract Min: (O(n log n))
  - Decrease Key: (O(edges))
  - Overall Time Complexity: (O(n log n + edges)) so it is less than binary heap one.

After Treating Edges as (2n):
1. Graph Generation:
  -Time Complexity: (O(n))
2. Dijkstra's Algorithm:
  - Extract Min: (O(n log n))
  - Decrease Key: (O(2n))
  - Overall Time Complexity: (O(n log n + 2n)) ==> O(n log n).