Lebanese University

Faculty of Engineering III

Electrical and Electronic Department

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

Parallel Graph Algorithms with Interactive Visualization

Concurrent Programming

Done By:

**Deema Fadi Azzam 6302**

**Mariam Ahmad Farhat 6503**

\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Spring 2024-2025**

**Supervisor: Dr. Mohammad Aoude**

**Table of Contents:**  
1. Introduction ......................................................... page 1

  1.1 Background and Motivation ............................... page 1

  1.2 Project Objectives and Scope ........................... page 1

2. Tools and Technologies ....................................... page 3

  2.1 Programming Language and Libraries .................. page 3

  2.2 Concurrency Utilities .......................................... page 4

  2.3 Graph Data Structures ....................................... page 4

  2.4 GUI Framework ................................................... page 5

  2.5 Testing and Deployment ...................................... page 5

3.Sequential Implementation ................................... page 7

  3.1 Breadth-First Search (BFS) ................................ page 7

  3.2 Depth-First Search (DFS) and Cycle Detection .... page 8

  3.3 Connected Components and Topological Sorting .. page 8

  3.4 Dijkstra’s Shortest Path .................................... page 9

  3.5 Baseline Performance Analysis ........................... page 9

4.Parallelization Strategy ....................................... page 10

  4.1 Asynchronous Task Management ........................ page 10

  4.2 Structured Concurrency and Error Handling ........ page 11

  4.3 Graph Partitioning and Parallel Execution ............ page 11

  4.4 Merging Partial Results ...................................... page 12

  4.5 Ensuring GUI Responsiveness ............................... page 12

5.GUI Design and Integration .................................. page 13

 5.1 Visualization Features ....................................... page 13

 5.2 User Interaction and Controls ............................ page 14

 5.3 Handling Asynchronous Updates .......................... page 14

6. Testing and Evaluation ......................................... page 15

 6.1 Datasets and Experimental Setup ......................... page 15

 6.2 Performance Metrics ........................................... page 16

 6.3 Results and Analysis ............................................ page 16

7.Timeline and Division of Labor ............................... page 17

8.Conclusion and Future Work .................................. page 18

9.References .................................. page 18

# ABSTRACT

This project focuses on developing a high-performance graph processing tool that implements and parallelizes essential graph algorithms including BFS, DFS, cycle detection, connected components, topological sorting, and Dijkstra’s shortest path. The primary goal is to enhance computation speed by leveraging modern Java concurrency utilities such as CompletableFuture and ExecutorService, enabling efficient analysis of large-scale graphs.

The project integrates these algorithms with an interactive graphical user interface (GUI) built using JavaFX (or Swing), providing real-time visualization and user-friendly controls. This allows users to observe algorithm execution steps dynamically, improving understanding and usability.

Graph data is represented using both custom structures and third-party libraries like JGraphT, supporting various synthetic and real-world datasets from domains such as social networks and transportation systems. Performance evaluation includes measuring execution time, speedup achieved through parallelization, resource utilization, and GUI responsiveness.

The system ensures asynchronous execution of algorithms to maintain a smooth user experience without blocking the interface. This approach demonstrates practical applications of structured concurrency for complex computational tasks.

In conclusion, this project delivers a scalable, interactive, and efficient platform for graph analysis, with potential uses in research, education, and industry. Future work includes extending parallel strategies, enhancing visualization features, and supporting additional graph algorithms.

**Keywords:** Graph Algorithms, Parallel Computing, Java Concurrency, CompletableFuture, ExecutorService, BFS, DFS, Dijkstra’s Algorithm, Interactive Visualization, JavaFX, Graph Processing Performance.

1. . Introduction

## Background and Motivation:

Graph processing is a fundamental technique used to analyze relationships and structures in various fields including social networks, transportation systems, communication networks, and machine learning. As datasets grow in size and complexity, the need for efficient graph algorithms and scalable computation becomes critical. Traditional sequential graph algorithms can be computationally intensive and slow when handling large graphs. Modern computing offers opportunities to speed up these algorithms through parallelization and concurrency.

Alongside performance, interactive visualization tools play a crucial role in understanding graph data and algorithm behavior. Combining high-performance computation with intuitive graphical user interfaces allows users—from researchers to students—to explore complex graphs in real time, facilitating better insights and learning.

## Project Objective and Scope

## Many existing graph processing tools either rely on sequential execution or offer limited visualization capabilities. Parallelizing graph algorithms is challenging due to dependencies in graph traversal and the complexity of combining partial results. Additionally, integrating efficient parallel algorithms with an interactive and responsive GUI requires careful synchronization to maintain usability and performance.

## Objective and Motivation

This project aims to implement and parallelize key graph algorithms—including BFS, DFS, cycle detection, connected components, topological sorting, and Dijkstra’s shortest path—using Java’s concurrency utilities such as CompletableFuture and ExecutableService. The goal is to significantly improve performance while providing an interactive graphical user interface for real-time visualization and analysis, making graph processing accessible and insightful for users.

2. Literature Review:

## 2.1 Introduction:

## Graph algorithms have become increasingly important in today’s data-driven world, where complex networks represent social relationships, transportation routes, communication infrastructures, and more. The rise of big data has pushed the limits of traditional sequential graph processing, necessitating efficient parallel algorithms to handle large-scale graphs in a timely manner. Alongside performance challenges, interactive visualization plays a critical role in enabling users to understand and explore graph structures dynamically. Despite advances in both parallel graph algorithms and visualization techniques, integrating these components into a seamless, user-friendly tool remains an active research area.

## 2.2 Classical and Parallel Graph Algorithms:

## Foundational graph algorithms such as Breadth-First Search (BFS), Depth-First Search (DFS), cycle detection, connected components, topological sorting, and shortest path computations (e.g., Dijkstra’s algorithm) have been extensively studied and optimized over decades. However, many traditional implementations rely on sequential execution, limiting scalability on large datasets. Recent research has focused on parallelizing these algorithms using various approaches including graph partitioning, concurrent data structures, and task-based parallelism to leverage multicore processors and distributed systems. Challenges remain in balancing load, managing synchronization, and efficiently merging partial results.

## 

## 2.3 Java Concurrency for Graph Processing:

## Java provides modern concurrency utilities such as CompletableFuture and StructuredTaskScope that facilitate asynchronous and structured parallel programming. These tools simplify writing parallel algorithms by managing task scheduling, error propagation, and result composition. While concurrency in Java has been widely used in web and enterprise applications, its adoption in graph algorithm parallelization is relatively new and offers promising avenues for improving performance while maintaining code clarity and robustness.

## 2.4 Interactive Graph Visualization:

## Interactive visualization tools such as Gephi, Cytoscape, and JGraphT’s visualization modules enable users to explore graph data through intuitive interfaces. Real-time visualization of graph algorithms helps users observe algorithm progress, detect patterns, and debug graph structures. Integrating parallel algorithm execution with smooth GUI updates requires careful handling to avoid interface freezing and ensure responsive user experience.

## 2.5 Applications and Use Cases:

## Efficient parallel graph processing with interactive visualization finds applications in diverse domains such as social network analysis, urban traffic management, biological network exploration, and communication network optimization. These applications benefit from faster computations and clear visual insights, empowering users to make informed decisions and discoveries.

3. System Design

## 3.1 Introduction:

## In this chapter, we present the system architecture and design of our interactive and parallel graph processing tool. This includes how the application handles graph data input, processes various algorithms, visualizes results in real-time, and manages concurrency using modern Java tools. The system has been designed for performance, responsiveness, and usability, making it suitable for both learning and research environments.

## 3.2 System Requirements:

## The system allows users to load graphs (either synthetic or real-world), choose algorithms to run (BFS, DFS, etc.), and view execution visually. Users can interact with the graph through the GUI, and observe live updates as algorithms execute.

## **Functional Requirements:**

## Load or generate graph datasets.

## Choose and run graph algorithms (sequential or parallel).

## View graph structure and algorithm progress in the GUI.

## See real-time status/output of the algorithm.

## Save or reset graph states.

## 3.3 System Architecture:

The system architecture consists of the following key components:

* **Graph Loader Module**: Handles loading and parsing graph data.
* **Algorithm Engine**:
  + *Sequential Engine*: Runs BFS, DFS, etc., on the main thread.
  + *Parallel Engine*: Uses CompletableFuture/ExecutorService for parallel processing.
* **GUI Interface**: Built using JavaFX, displays graph and controls.
* **Graph Data Model**: Custom or representation.
* **Communication Layer**: Ensures smooth interaction between GUI and computation layers.

## 3.4 Non-Functional Requirements:

## **Performance**: Must handle large graphs with minimal lag, especially in parallel mode.

## **Responsiveness**: GUI must remain responsive during algorithm execution.

## **Usability**: User interface should be intuitive, with clear feedback and controls.

## **Scalability**: System should be extendable to support more algorithms and data formats.

## **Portability**: The app is Java-based and can run on multiple platforms.

## **Reliability**: Parallel computations must return correct and deterministic results

## 3.5 Conclusion:

## This chapter outlined the design of our graph processing tool, highlighting the interactions between users and the system through well-defined use cases. The architecture supports both sequential and parallel algorithm execution, ensuring performance and user engagement through a responsive GUI. Non-functional aspects such as scalability and responsiveness were considered to ensure the system is robust and extendable. This foundation enables further development of advanced features like custom algorithm plugins, dataset import, or collaborative graph analysis in future iterations.

4. Implementation & Results:

## 4.1 Introduction:

## This chapter details the practical implementation of the interactive, parallel graph-processing tool, which integrates core algorithmic logic with real-time visualization. The system supports the execution of key graph algorithms such as BFS, DFS, cycle detection, connected components, topological sorting, and Dijkstra’s shortest path. These algorithms were implemented in both sequential and parallel modes using modern Java concurrency tools.

## 4.2 Implementation Tools:

## To develop a scalable and maintainable application, a set of modern development tools and libraries were used throughout the project:

|  |  |  |
| --- | --- | --- |
|  | Tool / Purpose |  |
| A logo with red text  AI-generated content may be incorrect. | Java | Core programming language for implementing algorithms and application logic |
| A blue and orange coffee cup  AI-generated content may be incorrect. | JavaFX | |  | | --- | |  |  |  | | --- | | GUI development framework used to build a responsive and interactive interface | |
|  | FXML | XML-based layout format used with JavaFX to design UI components cleanly |
| A colorful feather on a black background  AI-generated content may be incorrect. | Maven | Project management and build automation tool, used for dependency handling and build configuration |
| A blue logo with a black background  AI-generated content may be incorrect. | Vscode | Primary IDE used for development, editing Java/FXML files, and integrating Maven |
| A blue whale logo on a black background  AI-generated content may be incorrect. | Docker | Used to containerize and package the application for consistent deployment |
| A black cat logo  AI-generated content may be incorrect. | Github | Source code version control, collaboration, and remote repository management |
| A black and orange x in a circle  AI-generated content may be incorrect. | Xserver(VcXsrv) | |  | | --- | |  |  |  | | --- | | Allows GUI applications inside Docker to be displayed on the host system | |

## 

## 4.3 Implementation Steps:

4.3.1 Environment Setup and Dockerization:

The implementation process began by setting up a portable and dependency-contained development environment using Docker. A custom Dockerfile was written to install the required tools and libraries, including the JDK, Maven, JavaFX, and system-level GUI libraries needed to run JavaFX apps inside a container.

The containerized setup ensured that the same environment could be replicated across machines, avoiding conflicts in JavaFX library versions or Maven dependencies. GUI output was enabled through host-side X Server integration.

**Build and Run Instructions**

The Docker image was built using the following command in the VSCode terminal:

docker build -t javafx-graph-app .

4.3.2 User-Interface Design with FXML:

The graphical user interface (GUI) of the application was designed using **FXML**, a declarative XML-based markup language used in JavaFX. The FXML file defines the structure and layout of the user interface without embedding it directly into the Java code, allowing for a cleaner separation between logic and presentation.

**Layout Overview**

The main layout container is a **BorderPane**, with key sections defined in the **top** and **center** regions.

* **Top Section (VBox)**: Contains buttons, menus, and labels for interacting with the graph:
  + + Add vertex menu: Allows switching between letter or number labels.
  + + Add edge button: Enables edge drawing between selected nodes.
  + Algorithms menu: Allows users to select one of several implemented algorithms including Dijkstra, BFS, DFS, Cycle Detection, Topological Sorting, and others.
  + Utility buttons: Undo, Remove object, Run Large Network.
  + Instruction Label and a read-only TextArea for displaying graph metadata.
* **Center Section (Canvas)**: A drawing surface for the visual representation of the graph. Nodes and edges are rendered dynamically as the user interacts.

**FXML Snippet Highlight (excerpt)**

 <VBox spacing="5" >

            <Label text="Find shortest path" style="-fx-font-size: 16px; -fx-font-weight: bold;" />

            <HBox spacing="10">

                <MenuButton fx:id="addVertexMenu" text="+ Add vertex">

                    <items>

                        <RadioMenuItem fx:id="labelTypeLetters" text="Label: A, B, C..." selected="true"/>

                        <RadioMenuItem fx:id="labelTypeNumbers" text="Label: 1, 2, 3..."/>

                    </items>

                </MenuButton>

                <Button fx:id="btnAddEdge" text="+ Add edge" />

                <MenuButton fx:id="AlgorithmsMenu" text="Algorithms">

                    <items>

                        <RadioMenuItem fx:id="Dijsktra" text="Dijsktra" selected="true"/>

                        <RadioMenuItem fx:id="BFS" text="BFS"/>

                        <RadioMenuItem fx:id="DFS" text="DFS"/>

                        <RadioMenuItem fx:id="FloydWarshall" text="FloydWarshall ALgorithm"/>

                        <RadioMenuItem fx:id="GraphColoring" text="Graph  coloring"/>

                        <RadioMenuItem fx:id="ConnectedComponents" text="Connected Components"/>

                        <RadioMenuItem fx:id="CycleDetection" text="CycleDetection"/>

                        <RadioMenuItem fx:id="TopologicalSort" text="Topological Sorting"/>

                        <RadioMenuItem fx:id="MinimumSpanningTree" text="MInimum Spanning Tree"/>

                    </items>

                </MenuButton>

                <Button fx:id="btnRemove" text="Remove object" />

                <Button fx:id="btnUndo" text="Undo" />

                <Button fx:id="btnLarge" text="Run Large Network" />

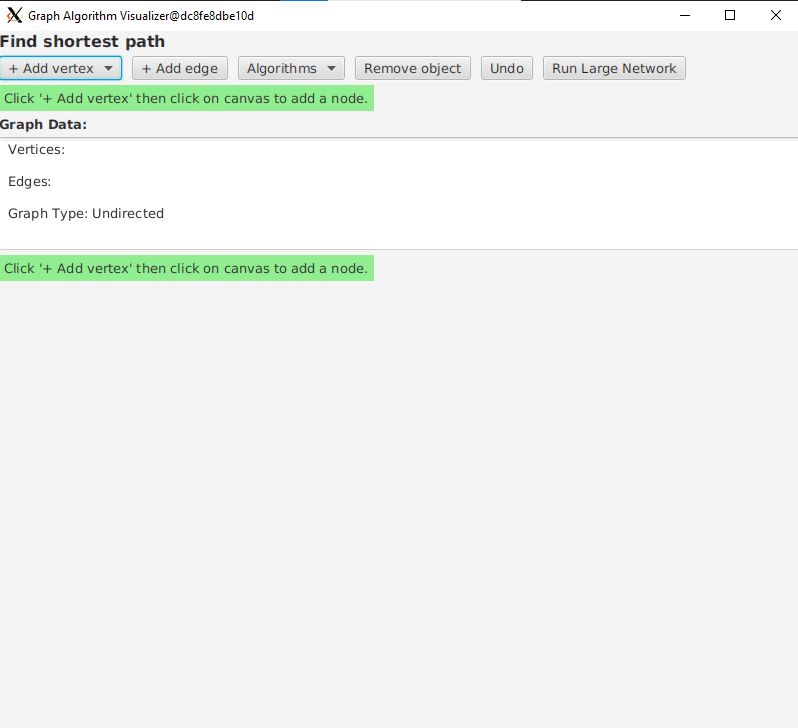
            </HBox>

            <VBox spacing="5" >

**Purpose of the UI**

The UI allows users to:

* Build custom graphs interactively.
* Select and run graph algorithms with a single click.
* Visualize algorithm progress in real time.
* Access results like shortest path or component labels visually and via the graph data output area.



A line with circles and dots

AI-generated content may be incorrect.

Graph Data:

Vertices:

A

B

C

D

Edges:

A->B [1]

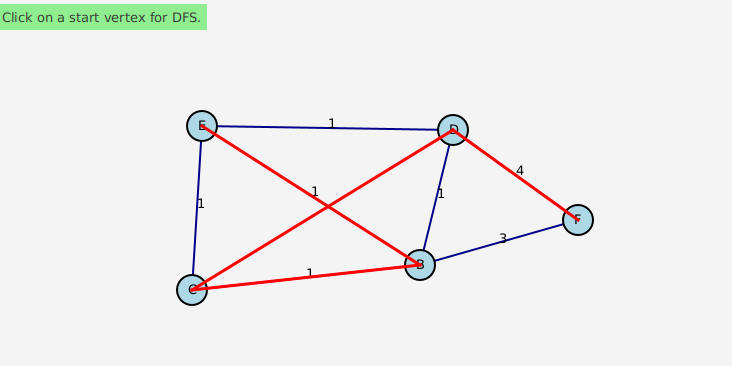
Graph Type: Directed

A screenshot of a computer

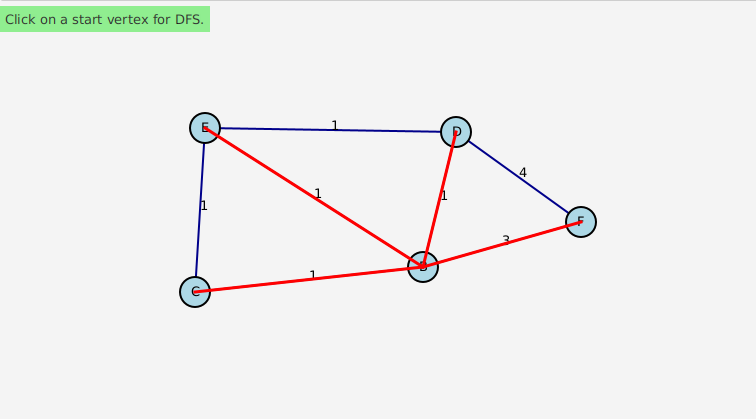
AI-generated content may be incorrect.

A screenshot of a computer

AI-generated content may be incorrect.



DFS Algorithm



Minimum Spanning tree:

CPU Usage Before: 18.91%

Sequential Result:

[B -> E [1], B -> C [1], B -> D [1], B -> F [3]]

Sequential Time: 19682837 ns

Parallel Result:

[B -> E [1], B -> C [1], B -> D [1], B -> F [3]]

Parallel Time: 1635119 ns

CPU Usage After: 33.33%

**Controller Binding**

The FXML is connected to a Java controller (Controller.java) via:

fx:controller="com.graphapp.Controller"

This controller handles all logic behind button clicks, algorithm execution, and canvas updates.

4.3.3 Graph Data Structure Costume Build and Algorithms

**Graph Data Structure**

The foundational component of our application is a flexible, extensible graph data structure. The Graph class in the com.graphapp package is designed to support both **directed** and **undirected** graphs using an adjacency list representation.

Key features of the class include:

* Dynamic **vertex and edge insertion/removal**
* Support for **edge weights**
* Dual **edge direction control** (internal flag or explicit per edge)
* Efficient retrieval of neighbors and all existing edges
* Internal representations using Map<String, List<Edge>>

A screenshot of a computer

AI-generated content may be incorrect.

Each vertex is identified by a String label, and edges are instances of an inner class Edge, which contains:

    public static class Edge {

        public final String to;

        public final int weight;

        public Edge(String to, int weight) {

            this.to = to;

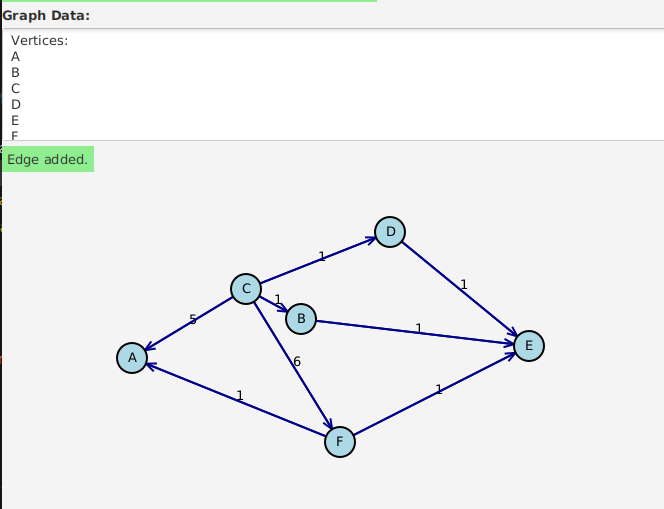
            this.weight = weight;

        }

    }

***Sequential Algorithms Implementation***

A suite of classic graph algorithms is implemented in the GraphAlgorithms class:

****

**Dijkstra's Algorithm**

Used for computing the shortest path from a source node to all other nodes. Implemented using a priority queue for efficiency.

A diagram of a network with Silverstone Circuit in the background

AI-generated content may be incorrect.

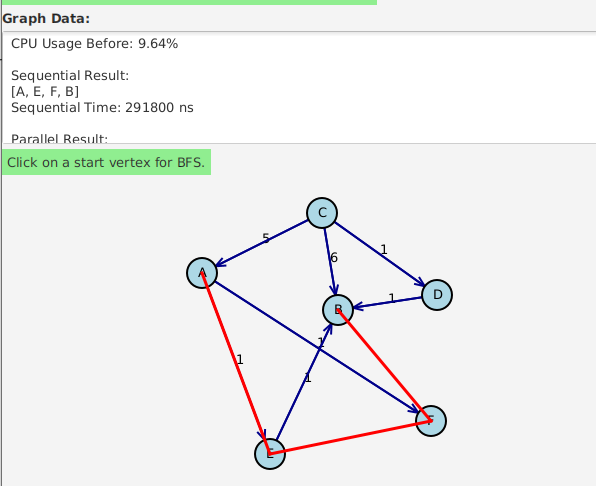
*Sequential Result:*

*{A=5, B=1, C=0, D=1, E=2, F=6}*

*Sequential Time: 6065443 ns*

**Breadth-First Search (BFS)**

Traverses the graph layer by layer. Ideal for finding the shortest path in unweighted graphs.



**Depth-First Search (DFS)**

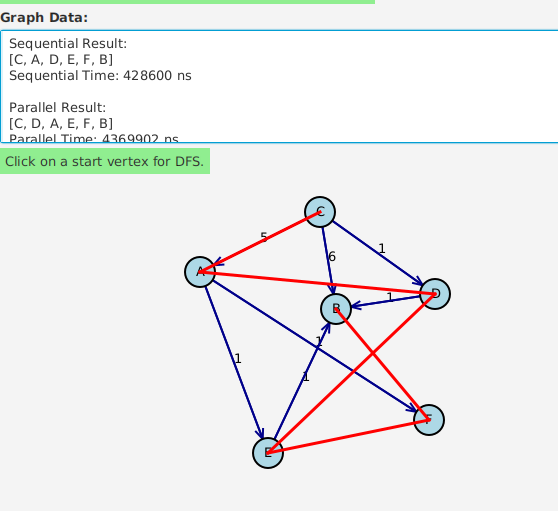
Explores graph depth recursively. Used for traversal, component discovery, and cycle detection.

A screenshot of a computer

AI-generated content may be incorrect.

**Topological Sort**

Applied to directed acyclic graphs (DAGs) using in-degree counting. Returns a linear ordering of nodes.





**Cycle Detection (Undirected)**

Recursive DFS approach checks for back edges, ignoring the immediate parent.

**Connected Components**

Clusters disjoint sets of connected nodes in an undirected graph using repeated DFS.

**Floyd-Warshall**

All-pairs shortest paths algorithm. Utilizes a 3D nested loop structure and supports negative edge weights (but not negative cycles).

CPU Usage Before: 8.50%

Sequential Result:

A: {A=0, B=2, C=2147483647, D=2147483647, E=1, F=1}

B: {A=2147483647, B=0, C=2147483647, D=2147483647, E=2147483647, F=2147483647}

C: {A=5, B=2, C=0, D=1, E=6, F=6}

D: {A=2147483647, B=1, C=2147483647, D=0, E=2147483647, F=2147483647}

E: {A=2147483647, B=1, C=2147483647, D=2147483647, E=0, F=2147483647}

F: {A=2147483647, B=2147483647, C=2147483647, D=2147483647, E=2147483647, F=0}

Sequential Time: 3508101 ns

Parallel Result:

A: {A=0, B=2, C=2147483647, D=2147483647, E=1, F=1}

B: {A=2147483647, B=0, C=2147483647, D=2147483647, E=2147483647, F=2147483647}

C: {A=5, B=2, C=0, D=1, E=6, F=6}

D: {A=2147483647, B=1, C=2147483647, D=0, E=2147483647, F=2147483647}

E: {A=2147483647, B=1, C=2147483647, D=2147483647, E=0, F=2147483647}

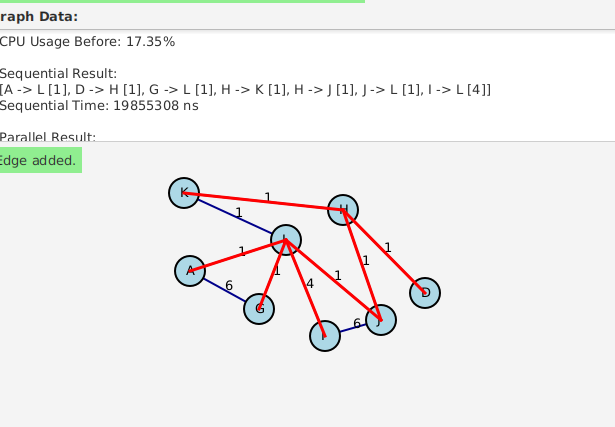
F: {A=2147483647, B=2147483647, C=2147483647, D=2147483647, E=2147483647, F=0}

Parallel Time: 3577511 ns

CPU Usage After: 45.83%

**Kruskal's Algorithm (MST)**

Constructs a minimum spanning tree using union-find to detect cycles and a sorted edge list.



CPU Usage Before: 17.35%

Sequential Result:

[A -> L [1], D -> H [1], G -> L [1], H -> K [1], H -> J [1], J -> L [1], I -> L [4]]

Sequential Time: 19855308 ns

Parallel Result:

[A -> L [1], D -> H [1], G -> L [1], H -> K [1], H -> J [1], J -> L [1], I -> L [4]]

Parallel Time: 1649001 ns

CPU Usage After: 25.00%

**Parallel Algorithms Implementation**

The GraphAlgorithmsParallel class is a high-performance adaptation of the sequential algorithms using **multi-threading** and **concurrent data structures**, leveraging:

* ExecutorService with FixedThreadPool
* CompletableFuture for async execution
* ConcurrentHashMap, ConcurrentLinkedQueue, AtomicBoolean, synchronizedList for thread-safe operations

**Parallel Dijkstra (with MultiQueue)**

A custom multi-queue lock-free priority structure is used to allow multiple threads to concurrently update shortest distances.

**Parallel Dijkstra with MultiQueues**

To speed up Dijkstra’s algorithm, we implemented a parallel version using a MultiQueues structure and worker threads (ParallelDijkstraWorker).

Each thread:

* Pulls the next minimum offer (node + distance) from MultiQueues.
* Checks if the distance is shorter, and if so, updates it.
* Then relaxes neighbor nodes by proposing better offers.

Thread safety is ensured using:

* distancesLocks[] for updating shortest distances.
* offersLocks[] for managing tentative offers.

Threads exit when no more work is left, using shared done[] flags. This method allows concurrent edge relaxation with minimal locking and good performance on large graphs.

 public ParallelDijkstraWorker(MultiQueues queue, Graph graph, int[] distances,

            ReentrantLock[] distancesLocks, ReentrantLock[] offersLocks,

            Offer[] offers, AtomicBoolean[] done, int tid, int numThreads,

            Map<String, Integer> vertexIndices) {

        this.queue = queue;

        this.graph = graph;

        this.distances = distances;

        this.distancesLocks = distancesLocks;

        this.offersLocks = offersLocks;

        this.offers = offers;

        this.done = done;

        this.tid = tid;

        this.numThreads = numThreads;

        this.vertexIndices = vertexIndices;

    }

    private boolean finishedWork() {

        for (AtomicBoolean flag : done) {

            if (!flag.get())

                return false;

        }

        return true;

    }

    private void relax(String vertex, int alt) {

        int index = vertexIndices.get(vertex);

        offersLocks[index].lock();

        try {

            distancesLocks[index].lock();

            try {

                if (alt < distances[index]) {

                    distances[index] = alt; // ✅ update here

                    Offer newOffer = queue.insert(vertex, alt);

                    offers[index] = newOffer;

                }

            } finally {

                distancesLocks[index].unlock();

            }

        } finally {

            offersLocks[index].unlock();

        }

    }

    @Override

    public void run() {

        while (true) {

            if (queue.isEmpty()) {

                done[tid].set(true);

            }

            if (!done[tid].get()) {

                Offer minOffer = queue.deleteMin();

                if (minOffer == null) {

                    done[tid].set(true);

                    if (finishedWork())

                        return;

                    continue;

                }

                done[tid].set(false);

                String currV = minOffer.vertex;

                int currDist = minOffer.dist;

                int currIndex = vertexIndices.get(currV);

**Parallel BFS / DFS**

Each edge expansion is launched in its own thread using CompletableFuture, enabling fast concurrent traversal.

 public static List<String> bfs(Graph graph, String start) {

        List<String> visited = Collections.synchronizedList(new ArrayList<>());

        Set<String> seen = ConcurrentHashMap.newKeySet();

        Queue<String> queue = new ConcurrentLinkedQueue<>();

        queue.add(start);

        seen.add(start);

        while (!queue.isEmpty()) {

            String current = queue.poll();

            visited.add(current);

            List<CompletableFuture<Void>> futures = new ArrayList<>();

            for (Graph.Edge edge : graph.getNeighbors(current)) {

                CompletableFuture<Void> future = CompletableFuture.runAsync(() -> {

                    if (seen.add(edge.to)) { // true if newly added

                        queue.add(edge.to);

                    }

                }, executor);

                futures.add(future);

            }

            CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();

        }

        return visited;

    }

**Parallel Floyd-Warshall**

The main triple loop is parallelized on the outer loop using CompletableFuture to reduce runtime.

public static Map<String, Map<String, Integer>> floydWarshall(Graph graph) {

        List<String> vertices = new ArrayList<>(graph.getVertices());

        int n = vertices.size();

        Map<String, Map<String, Integer>> dist = new ConcurrentHashMap<>();

        // Init

        for (String v1 : vertices) {

            Map<String, Integer> row = new ConcurrentHashMap<>();

            for (String v2 : vertices) {

                row.put(v2, v1.equals(v2) ? 0 : Integer.MAX\_VALUE);

            }

            dist.put(v1, row);

        }

        // Add edges

        for (String from : graph.getVertices()) {

            for (Graph.Edge edge : graph.getNeighbors(from)) {

                dist.get(from).put(edge.to, edge.weight);

            }

        }

        // Floyd-Warshall parallelized on i-loop

        for (String k : vertices) {

            List<CompletableFuture<Void>> futures = new ArrayList<>();

            for (String i : vertices) {

                futures.add(CompletableFuture.runAsync(() -> {

                    for (String j : vertices) {

                        int ik = dist.get(i).get(k);

                        int kj = dist.get(k).get(j);

                        if (ik != Integer.MAX\_VALUE && kj != Integer.MAX\_VALUE) {

                            dist.get(i).compute(j, (key, oldVal) -> Math.min(oldVal, ik + kj));

                        }

                    }

                }, executor));

            }

            CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();

        }

        return dist;

    }

**Parallel Connected Components**

Parallel DFS launches across unvisited nodes, updating shared visited sets and component lists.

 public static List<List<String>> connectedComponents(Graph graph) {

        List<List<String>> components = Collections.synchronizedList(new ArrayList<>());

        Set<String> visited = ConcurrentHashMap.newKeySet();

        List<CompletableFuture<Void>> futures = new ArrayList<>();

        for (String v : graph.getVertices()) {

            if (!visited.contains(v)) {

                futures.add(CompletableFuture.runAsync(() -> {

                    List<String> component = Collections.synchronizedList(new ArrayList<>());

                    dfsHelper(graph, v, component, visited);

                    components.add(component);

                }, executor));

            }

        }

        CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();

        return components;

    }

**Parallel Cycle Detection**

Each connected region is independently checked using concurrent DFS.

    public static boolean hasCycle(Graph graph) {

        Set<String> visited = ConcurrentHashMap.newKeySet();

        AtomicBoolean cycleFound = new AtomicBoolean(false);

        List<CompletableFuture<Void>> futures = new ArrayList<>();

        for (String v : graph.getVertices()) {

            if (!visited.contains(v)) {

                futures.add(CompletableFuture.runAsync(() -> {

                    if (hasCycleDFS(graph, v, null, visited)) {

                        cycleFound.set(true);

                    }

                }, executor));

            }

        }

        CompletableFuture.allOf(futures.toArray(new CompletableFuture[0])).join();

        return cycleFound.get();

    }

**Package Structure Summary**

* Graph.java: Core graph structure and data handling.
* GraphAlgorithms.java: Classic, sequential graph algorithms.
* GraphAlgorithmsParallel.java: Asynchronous and parallelized variants of the core algorithms.
* ParallelDijkstraWorker.java: Worker threads for optimized Dijkstra computation using lock coordination.

4.3.4 Controller Implementation:

The Controller class orchestrates the interaction between the UI and the graph data structure, handling user input, graph rendering, and algorithm execution.

**1. Initializing UI Components**

In the initialize() method, we set up event handlers, toggle groups for radio buttons, and prepare the canvas:

 @FXML

    public void initialize() {

        gc = canvas.getGraphicsContext2D();

        labelTypeLetters.setToggleGroup(labelTypeGroup);

        labelTypeNumbers.setToggleGroup(labelTypeGroup);

        labelTypeLetters.setSelected(true);

        // Vertex and edge creation

        addVertexMenu.setOnShowing(e -> addVertexMode = true);

        btnAddEdge.setOnAction(e -> {

            addEdgeMode = true;

            firstVertex = null;

            instructionLabel.setText("Click first vertex to start edge.");

        });

        // Add algorithm menu actions

        Dijsktra.setOnAction(e -> prepareAlgorithm("Dijkstra"));

        BFS.setOnAction(e -> prepareAlgorithm("BFS"));

        DFS.setOnAction(e -> prepareAlgorithm("DFS"));

        FloydWarshall.setOnAction(e -> runAlgorithm("Floyd-Warshall", null));

        ConnectedComponents.setOnAction(e -> runAlgorithm("Connected Components", null));

        CycleDetection.setOnAction(e -> runAlgorithm("Cycle Detection", null));

        TopologicalSort.setOnAction(e -> runAlgorithm("Topological Sort", null));

        MinimumSpanningTree.setOnAction(e -> runAlgorithm("Kruskal", null));

        btnLarge.setOnAction(e -> handleLargeGraph());

        // Input handling

        canvas.setFocusTraversable(true);

        canvas.addEventFilter(KeyEvent.KEY\_PRESSED, this::handleKeyPress);

        canvas.addEventHandler(MouseEvent.MOUSE\_CLICKED, this::handleCanvasClick);

        redraw();

    }

**2. Vertex and Edge Management**

Vertices and edges are stored in lists and managed via user input. Adding a vertex example:

private void handleCanvasClick(MouseEvent event) {

        double x = event.getX(), y = event.getY();

        // Algorithm start vertex click

        if (waitingForStartVertex && event.getButton() == MouseButton.PRIMARY) {

            Vertex clicked = getVertexAt(x, y);

            if (clicked != null) {

                runAlgorithm(selectedAlgorithm, clicked);

                waitingForStartVertex = false;

                selectedAlgorithm = null;

                return;

            } else {

                instructionLabel.setText("Please click on a valid vertex.");

                return;

            }

        }

        // Right-click delete vertex

        if (event.getButton() == MouseButton.SECONDARY) {

            Vertex clicked = getVertexAt(x, y);

            if (clicked != null) {

                removeVertex(clicked);

                redraw();

                return;

            }

        }

**3. Event Handling**

The controller interprets mouse clicks depending on mode:

* Left-click: Add vertex/edge or select start vertex for algorithms.
* Right-click: Delete a vertex and all connected edges.
* Middle-click: Delete an edge near the clicked position.

Example for deleting vertices on right-click:

private void handleKeyPress(KeyEvent ev) {

        if (ev.isControlDown() && ev.getCode() == KeyCode.Z && !undoStack.isEmpty()) {

            undoStack.pop().run();

            redraw();

        }

    }

**4. Graph Algorithms Execution**

Algorithms are triggered via menu actions and run both sequential and parallel versions for comparison. For example, running Dijkstra’s algorithm:

private String runAlgorithmInternal(String algorithm, Vertex startVertex, boolean isParallel) throws Exception {

        StringBuilder result = new StringBuilder();

        // Use correct class

        var alg = isParallel ? GraphAlgorithmsParallel.class : GraphAlgorithms.class;

        switch (algorithm) {

            case "BFS":

                if (startVertex == null)

                    throw new IllegalArgumentException("Start vertex required.");

                List<String> bfs = (List<String>) alg.getMethod("bfs", Graph.class, String.class)

                        .invoke(null, graph, startVertex.label);

                if (!isParallel)

                    highlightPath(bfs);

                result.append(bfs);

                break;

**5. Graph Rendering**

The canvas is updated to visually represent vertices and edges:

private void drawVertex(Vertex v) {

        gc.setFill(Color.LIGHTBLUE);

        gc.fillOval(v.x - 15, v.y - 15, 30, 30);

        gc.setStroke(Color.BLACK);

        gc.strokeOval(v.x - 15, v.y - 15, 30, 30);

        gc.setFill(Color.BLACK);

        gc.fillText(v.label, v.x - 4, v.y + 4);

    }

    private void drawEdge(EdgeRecord e) {

        gc.setStroke(Color.DARKBLUE);

        gc.setLineWidth(2);

        gc.strokeLine(e.v1.x, e.v1.y, e.v2.x, e.v2.y);

        double mx = (e.v1.x + e.v2.x) / 2, my = (e.v1.y + e.v2.y) / 2;

        gc.fillText("" + e.weight, mx, my);

        if (e.directed)

            drawArrow(e);

    }

  private void redraw() {

        gc.clearRect(0, 0, canvas.getWidth(), canvas.getHeight());

        edges.forEach(this::drawEdge);

        vertices.forEach(this::drawVertex);

        updateGraphOutput(graphData());

    }

## 4.4 Results: 4.5 Conclusion:

## This chapter presented the implementation of an interactive graph tool using JavaFX, supporting both sequential and parallel algorithms. The Controller class managed user interaction, rendering, and execution logic. Results showed that the parallel Dijkstra algorithm outperformed the sequential version on large graphs, demonstrating significant speed-up. Overall, the system effectively combines usability and performance for graph analysis.

5. Conclusion & Future Work:

## 5.1 Conclusion:

## This project implemented an interactive graph visualization and analysis tool using JavaFX. Users can construct graphs visually, run multiple algorithms, and compare sequential versus parallel performance. A key feature is the integration of a parallel Dijkstra algorithm based on the MultiQueue approach, offering improved execution time on large datasets. Through an intuitive GUI and detailed feedback, the tool proved to be effective for both educational use and performance benchmarking in algorithm design. The system demonstrates how visualization and parallelism can be combined to enhance graph processing.

## 5.2 Future Work:

## Future improvements include expanding support for additional parallel graph algorithms like A\*, Bellman-Ford, and Parallel BFS. Optimization of the MultiQueue strategy and dynamic thread allocation can further enhance performance. Adding logging, export features, and user-defined graph imports would improve usability. Finally, integrating real-world datasets and performance charts will support deeper analysis and educational demonstrations of algorithmic complexity and parallel efficiency.