Assignment Two

Shortest Paths Algorithms

Team members

نور الدين طه احمد	22011310
محمد السيد محمد	22011102
مينا نبيل يوسف منصور	22011290
مايكل مجدي نجيب	22011082

Introduction

This report presents the theoretical and empirical analysis of three classic shortest path algorithms:

- **Dijkstra's Algorithm**
- **Bellman-Ford Algorithm**
- **Floyd-Warshall Algorithm**

We analyze each algorithm in terms of **time complexity**, **space complexity**, and compare their **performance across various graph sizes and densities** based on two metrics:

- Time to compute the shortest path between **two specific nodes**.
- Time to compute shortest paths between **all pairs of nodes**.

Algorithms Overview

- **Dijkstra's Algorithm**
  - **Description**: A greedy algorithm that finds the shortest paths from a single source to all vertices with non-negative edge weights using a priority queue
  - **Time Complexity**:
    - Best Case (Sparse Graph with Min-Heap): $O((V + E) \log V)$
    - Worst Case (Dense Graph): $O(V^2)$
    - Average Case: $O((V + E) \log V)$
  - **Space Complexity**:
    - $O(V + E)$ (for adjacency list and distance tracking)
  - **Efficiency**:
    - Very efficient on sparse graphs with non-negative weights.
    - Cannot handle negative edge weights.

- **Bellman-Ford Algorithm**
  - **Description**: A dynamic programming algorithm that relaxes all edges up to V-1 times to find the shortest path, allowing for negative weights
  - **Time Complexity**:
    - Best Case: $O(V)$ (early termination possible)
    - Worst Case: $O(V \cdot E)$
    - Average Case: $O(V \cdot E)$
  - **Space Complexity**:
    - $O(V)$ (for distance tracking)
  - **Efficiency**:
    - Handles graphs with negative weights.
    - Slower than Dijkstra on dense graphs or graphs without negative weights.

- **Floyd-Warshall Algorithm**
    - **Description**: A dynamic programming algorithm that computes shortest paths between all pairs of vertices.
    - **Time Complexity**:
        - Best, Worst, and Average Case: $O(V^3)$
    - **Space Complexity**:
        - $O(V^2)$ (for distance matrix)
    - **Efficiency**:
        - Excellent for small, dense graphs where all-pairs shortest paths are needed.
        - Not efficient for large graphs.

Discussion and Analysis

- **Time Complexity Comparison**
    - **Dijkstra's Algorithm** performs efficiently on sparse graphs using a min-heap with time complexity around **$O((V + E) \log V)$**, but fails with negative weights.
    - **Bellman-Ford Algorithm** is slower due to **$O(V \cdot E)$** time but supports graphs with **negative weights** and detects **negative cycles**.
    - **Floyd-Warshall Algorithm** has **$O(V^3)$** time complexity, making it suitable only for **small dense graphs** where **all-pairs shortest paths** are needed.
- **Space Complexity Comparison**
    - **Dijkstra's Algorithm** uses **$O(V + E)$** space, depending on the graph representation.
    - **Bellman-Ford Algorithm** is more memory-efficient with **$O(V)$** space for distance tracking.
    - **Floyd-Warshall Algorithm** requires a **$V \times V$ matrix**, making it the least space-efficient with **$O(V^2)$** space.

Comparison

Time to get shortest path between 2 node in ns

size	Dijkstra	Bellman-Ford	Floyd-Warshall
100	4,894,900	3,276,600	5,620,400
10000	85,560,800	224,170,600	985,257,001,300

Time to get shortest path between all pairs in ns

size	Dijkstra	Bellman-Ford	Floyd-Warshall
100	6,787,000	16,923,600	5,263,300
10000	429,655,739,300	1,646,718,894,000	1,885,857,001,000

Space complexity in KB

size	Dijkstra	Bellman-Ford	Floyd-Warshall
100	120	140	900
10000	22,000	32,000	95,000

Conclusion

- For **single-source shortest path on sparse graphs**: **Dijkstra's Algorithm** is preferred due to its speed and efficiency.
- For **graphs with negative weights**: **Bellman-Ford** is the reliable choice.
- For **dense graphs where all-pairs shortest paths are required**: **Floyd-Warshall** is the most suitable, despite its cubic time and high memory usage.

Sample Run 1

```
7 12
0 1 2
0 2 7
0 4 12
4 0 -4
1 3 2
2 4 2
2 1 3
2 3 -1
4 6 -7
3 5 2
5 6 2
6 3 1
```

```
Welcome to the Graph CLI tool
Please enter the path to the graph file: src/main/java/org/example/graph.txt


Graph
0: [[1, 2], [2, 7], [4, 12]]
1: [[3, 2]]
2: [[4, 2], [1, 3], [3, -1]]
3: [[5, 2]]
4: [[0, -4], [6, -7]]
5: [[6, 2]]
6: [[3, 1]]
```

```
--- MAIN MENU ---
1. Find shortest paths from source node to all other nodes
2. Find shortest paths between all pairs of nodes
3. Check if the graph contains a negative cycle
4. Exit
Enter your choice: 1
Enter the source node: 0

Choose an algorithm:
1. Dijkstra's Algorithm (no negative weights)
2. Bellman-Ford Algorithm
3. Floyd-Warshall Algorithm
Enter your choice: 1
Dijkstra's algorithm does not support negative weights!
```

```
Choose an algorithm:
1. Dijkstra's Algorithm (no negative weights)
2. Bellman-Ford Algorithm
3. Floyd-Warshall Algorithm
Enter your choice: 2
Path from 0 to 0: [0], cost = 0
Path from 0 to 1: [0, 1], cost = 2
Path from 0 to 2: [0, 2], cost = 7
Path from 0 to 3: [0, 2, 4, 6, 3], cost = 3
Path from 0 to 4: [0, 2, 4], cost = 9
Path from 0 to 5: [0, 2, 4, 6, 3, 5], cost = 5
Path from 0 to 6: [0, 2, 4, 6], cost = 2
```

```
--- MAIN MENU ---
1. Find shortest paths from source node to all other nodes
2. Find shortest paths between all pairs of nodes
3. Check if the graph contains a negative cycle
4. Exit
Enter your choice: 2

Choose an algorithm:
1. Dijkstra's Algorithm
2. Bellman-Ford Algorithm
3. Floyd-Warshall Algorithm
Enter your choice: 2
Enter source node: 3
Enter destination node: 5
Path from 3 to 5: [3, 5], cost = 2
```

```
--- MAIN MENU ---
1. Find shortest paths from source node to all other nodes
2. Find shortest paths between all pairs of nodes
3. Check if the graph contains a negative cycle
4. Exit
Enter your choice: 3

Choose an algorithm to check for negative cycles:
1. Using Bellman-Ford Algorithm
2. Using Floyd-Warshall Algorithm
Enter your choice: 2
The graph does not contain any negative cycles.
```

Sample Run 2

```
3 3
0 1 1
1 2 -1
2 0 -1
```

```
Welcome to the Graph CLI tool
Please enter the path to the graph file: src/main/java/org/example/graph-with-negative-cycle.txt


Graph
0: [[1, 1]]
1: [[2, -1]]
2: [[0, -1]]
```

```
--- MAIN MENU ---
1. Find shortest paths from source node to all other nodes
2. Find shortest paths between all pairs of nodes
3. Check if the graph contains a negative cycle
4. Exit
Enter your choice: 1
Enter the source node: 0

Choose an algorithm:
1. Dijkstra's Algorithm (no negative weights)
2. Bellman-Ford Algorithm
3. Floyd-Warshall Algorithm
Enter your choice: 2
Negative cycle detected when starting from node 0
```

```
--- MAIN MENU ---
1. Find shortest paths from source node to all other nodes
2. Find shortest paths between all pairs of nodes
3. Check if the graph contains a negative cycle
4. Exit
Enter your choice: 3

Choose an algorithm to check for negative cycles:
1. Using Bellman-Ford Algorithm
2. Using Floyd-Warshall Algorithm
Enter your choice: 2
The graph contains at least one negative cycle.
```