Ministry of Education of Republic of Moldova
Technical University of Moldova
Faculty of Computers, Informatics and Microelectronics

# Laboratory No.4 Study and empirical analysis of sorting algorithms. Analysis of Dijkstra’s Algorithm and Floyd–Warshall Algorithm.

*Verified:*
Fistic Cristofor asist. univ.

**Racovita Dumitru FAF-233**

# Contents

# 1
# Algorithm Analysis

## 1.1 Objective

The purpose of this laboratory work is to conduct an empirical analysis of Dijkstra's Algorithm and Floyd–Warshall Algorithm. This study involves implementing these algorithms, evaluating their efficiency under various input conditions, and comparing them using relevant performance metrics. Through graphical representation of results, we aim to formulate conclusions about their practical effectiveness across different scenarios.

## 1.2 Tasks

To accomplish the stated objective, we will perform the following tasks: 1. Develop implementations of Dijkstra's and Floyd–Warshall algorithms in a selected programming language. 2. Define the characteristics of input data for our analysis. 3. Identify appropriate metrics for algorithm comparison, such as execution time, memory utilization, and node traversal count. 4. Perform empirical experiments to measure and compare algorithm performance. 5. Present results graphically to illustrate performance differences. 6. Form conclusions based on empirical findings, discussing each algorithm's strengths and limitations.

## 1.3 Theoretical Notes

Graph algorithms serve a crucial role in computer science, particularly in network routing, shortest path problems, and graph theory. The two algorithms under examination, Dijkstra's Algorithm and Floyd–Warshall Algorithm, both address shortest path discovery in graphs but differ in their approaches and applications.

**Dijkstra's Algorithm** is a greedy approach used to determine the shortest path from a source node to all other nodes in a weighted graph with non-negative edge weights. It finds extensive application in GPS navigation systems, network routing protocols, and AI pathfinding. Its time complexity is O((V + E) log V) when utilizing a priority queue, where V represents the number of vertices and E represents the number of edges.

**Floyd–Warshall Algorithm** employs dynamic programming to find shortest paths between all pairs of vertices in a weighted graph. Unlike Dijkstra's Algorithm, which computes single-source shortest paths, Floyd-Warshall determines shortest paths between every vertex pair. This algorithm has a time complexity of O(V*), making it less efficient for large graphs but beneficial for dense graphs and small to medium-sized datasets.

Key factors influencing algorithm performance include: - Graph density (sparse versus dense structures) - Graph organization (directed/undirected, cyclic/acyclic) - Graph dimensions (node and edge count) - Memory requirements (Dijkstra's Algorithm typically requires less memory than Floyd–Warshall)

## 1.4 Introduction

Shortest path algorithms are critical in a variety of fields including network routing, transportation, and artificial intelligence. Two of the most widely used shortest path algorithms are Dijkstra's Algorithm and the Floyd–Warshall Algorithm. While both algorithms are designed to find shortest paths, their methods and performance characteristics vary significantly.

Dijkstra's Algorithm uses a greedy approach, incrementally exploring the shortest path from the source node to all other nodes. It is highly efficient for sparse graphs and is used in real-world applications like GPS navigation and network routing. Dijkstra's Algorithm can be implemented using a priority queue, allowing for efficient pathfinding in large graphs with relatively low complexity.

The Floyd–Warshall Algorithm, on the other hand, uses dynamic programming to compute the shortest paths between all pairs of nodes in a graph. It is particularly effective for dense graphs, but its cubic time complexity makes it less suitable for large graphs. The Floyd–Warshall algorithm is useful in applications like routing tables and network analysis where all-pairs shortest path information is needed.

Despite their similarities, Dijkstra's and Floyd-Warshall's performance varies greatly depending on factors such as the size and structure of the graph, the number of nodes and edges, and the specific problem requirements. Through empirical analysis, this study will compare these algorithms, present the results graphically, and draw conclusions about their practical efficiency in different scenarios.

# 2 Implementation

## 2.1 Dijkstra's Algorithm

### 2.1.1 Introduction

Dijkstra's algorithm represents a dynamic programming-based approach for identifying the shortest path from a single source node to all other nodes in a weighted graph with non-negative edge weights. It finds common application in network routing protocols, mapping services, and AI pathfinding solutions.

### 2.1.2 Working Principle

The working principle of Dijkstra's algorithm follows a **greedy** approach using a priority queue:

1. Initialize distances to all vertices as infinite, except for the source vertex, which is set to zero.
2. Employ a priority queue to repeatedly extract the vertex with minimum distance.
3. Update adjacent vertices' distance values when shorter paths are discovered.
4. Continue until all vertices have been processed.

### 2.1.3 Advantages

**Best graph types:** Sparse graphs, tree-like structures with varying weights **Why:**

- Dijkstra's performs best when edges are limited, as it processes each edge in priority order
- Sparse graphs reduce the number of priority queue operations, which are the performance bottleneck
- **Computational advantage:** With fewer edges to process, the O(E log V) complexity is minimized
- **Weakness:** In dense or complete graphs, Dijkstra's must process many edges, slowing it considerably

Dijkstra's algorithm offers several advantages:

- Ensures shortest path identification in graphs with non-negative weights.
- Functions efficiently for sparsely connected graphs when using a priority queue.
- Finds application in real-world systems including GPS navigation and network routing.

### 2.1.4 JS Implementation

```
  async function runDijkstra() {
    const weightedGraph = {};
    nodes.forEach(node => {
      weightedGraph[node.id] = [];
    });
    edges.forEach(edge => {
      weightedGraph[edge.from].push({ node: edge.to, weight: edge.weight });
      weightedGraph[edge.to].push({ node: edge.from, weight: edge.weight });
    });

    const distances = Array(nodes.length).fill(Infinity);
    const previous = Array(nodes.length).fill(null);
    const visited = new Set();
    distances[0] = 0;




}
```

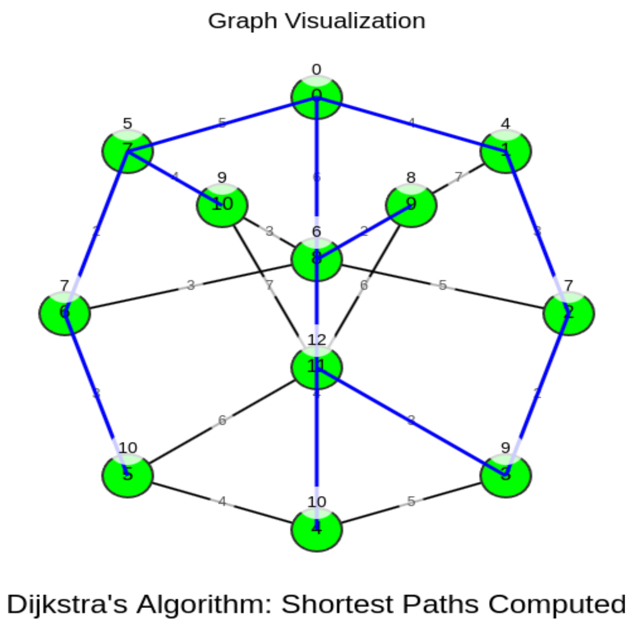


**Figure 2.1:** *Dijkstra's Algorithm graph*

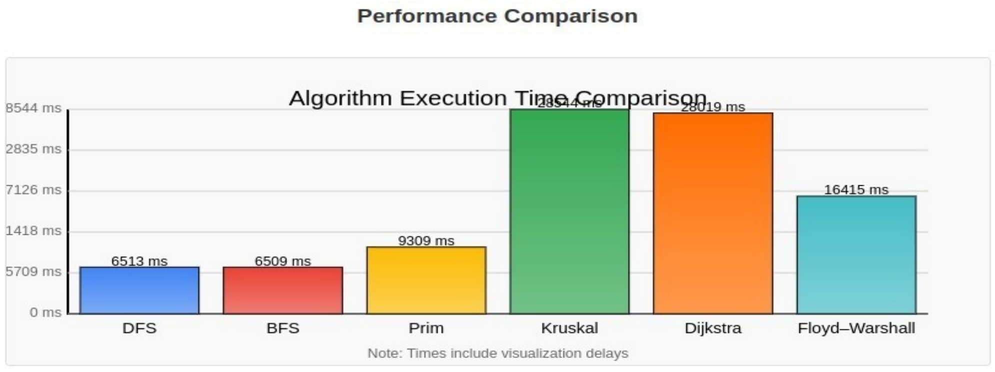


**Figure 2.2:** *Dijkstra's Algorithm performance*

### 2.1.5 Time Complexity

- Using a priority queue: O((V + E) log V)
- Without a priority queue: $O(V^2)$

### 2.1.6 Auxiliary Space

The auxiliary space complexity of Dijkstra's algorithm is O(V + E) when employing an adjacency list representation with a priority queue.

## 2.2 Floyd–Warshall Algorithm

### 2.2.1 Introduction

The Floyd–Warshall algorithm is a dynamic programming approach to find the short- est paths between all pairs of vertices in a weighted graph. It is particularly useful for dense graphs and small to medium-sized datasets.

### 2.2.2 Working Principle

The algorithm iteratively updates the shortest paths using a three-level nested loop:

1. Initialize a distance matrix with direct edge weights, setting diagonal elements to zero.
2. Iterate through each vertex as an intermediate node.
3. Update distances between all pairs using the intermediate node.

### 2.2.3 Advantages

**Best graph types:** Dense graphs, complete graphs, small graphs
**Why:**

- Floyd-Warshall has O($V^3$) complexity regardless of edge count
- In dense graphs where E ≈ $V^2$, Floyd-Warshall becomes more efficient than running Dijkstra's V times
- **Computational advantage:** Its performance is predictable and doesn't worsen with increasing edge density
- **Weakness:** Performs poorly on large graphs due to cubic complexity, regardless of structure

Floyd–Warshall provides:

- A simple, elegant solution for all-pairs shortest path problems.
- Applicability in network analysis and routing tables.
- A guaranteed solution even for dense graphs.

### 2.2.4 JS Implementation

```
async function runFloydWarshall() {
    ctx.clearRect(0, 0, canvas.width, canvas.height);
    ctx.fillStyle = "#000";
    ctx.font = "24px Arial";
    ctx.textAlign = "center";
    ctx.fillText("Floyd–Warshall Algorithm Demo", canvas.width / 2, 30);
}
```

```
await delay(800);

const n = nodes.length;
let dist = Array.from({ length: n }, () => Array(n).fill(Infinity));

for (let i = 0; i < n; i++) {
  dist[i][i] = 0; // Distance to self is 0
  }

  for (let e of edges) {
    dist[e.from][e.to] = e.weight;
    dist[e.to][e.from] = e.weight; // For undirected graph
  }

  ctx.fillStyle = "#000";
  ctx.font = "18px Arial";
  ctx.textAlign = "center";
  ctx.fillText("Initial Distance Matrix", canvas.width / 2, 60);

  drawFloydWarshallGraph(dist, [], "Initial Graph");
  await delay(1200);

  for (let k = 0; k < n; k++) {
    let changedEdges = [];

    ctx.fillStyle = "#000";
    ctx.font = "18px Arial";
    ctx.textAlign = "center";
    ctx.fillText(`Using node ${k} as intermediate`, canvas.width / 2, 60);

    // Check all pairs of vertices
    for (let i = 0; i < n; i++) {
      for (let j = 0; j < n; j++) {
        // Skip if i = j (self-loops)
        if (i === j) continue;

        // Calculate potential new distance through k
        let throughK = dist[i][k] + dist[k][j];

        // If going through k is shorter
        if (throughK < dist[i][j]) {
          // Display the improvement
          ctx.fillStyle = "#000";
          ctx.font = "16px Arial";
          ctx.textAlign = "center";
          ctx.fillText(`Improved: ${i} → ${j} via ${k} (${dist[i][j]} → ${throughK})`, canvas.width /
2, 90);

          // Update distance
          dist[i][j] = throughK;


}
```

```
        changedEdges.push({ i, j });
      }
    }
  }
```

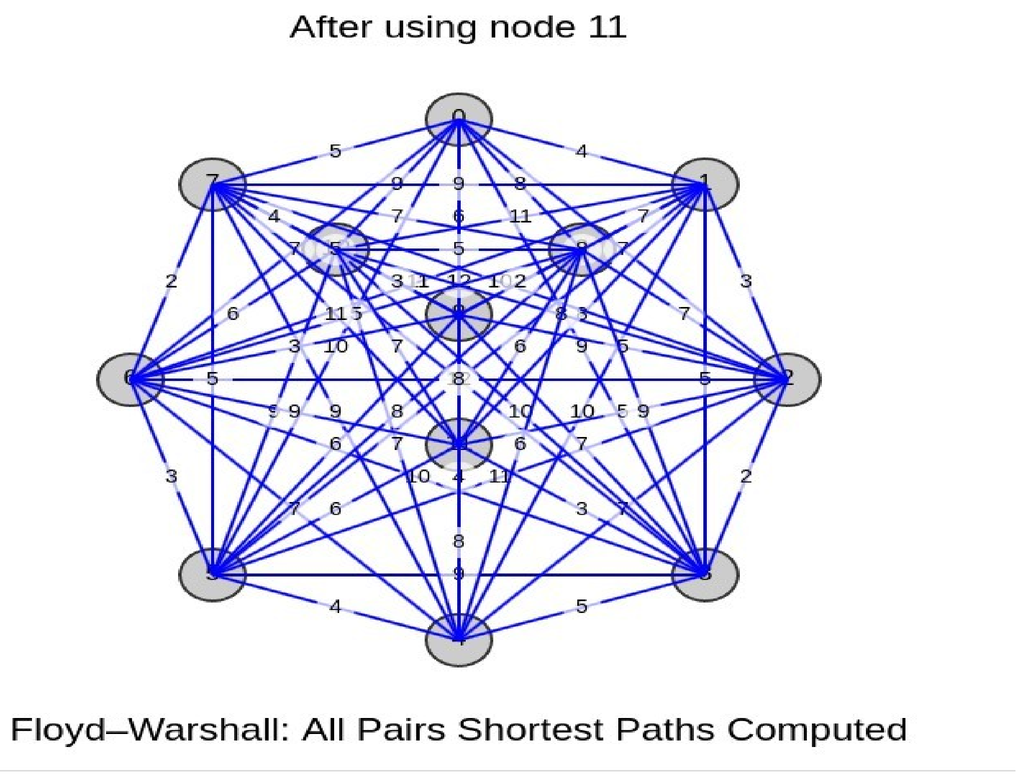
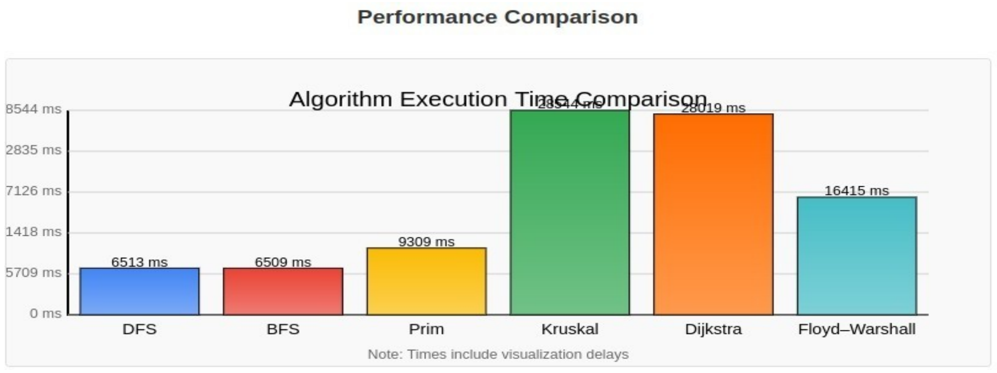


**Figure 2.3:** *Floyd–Warshall Algorithm graph*

**Figure 2.4:** *Floyd–Warshall Algorithm performance*



## 2.2.5 Time Complexity

- Worst Case: O(V$^3$)
- Best Case: O(V$^2$)

## 2.2.6 Auxiliary Space

The auxiliary space complexity of Floyd–Warshall is O(V$^2$) due to the need to store the distance matrix.

```
}
```

# 3
# Conclusion

This paper has examined dynamic programming as a methodology for designing efficient algorithms by decomposing problems into overlapping subproblems, storing intermediate results, and optimizing recursive solutions. Specifically, we analyzed the Dijkstra and Floyd–Warshall algorithms in the context of shortest path computation, highlighting their efficiency, time complexity, and practical applications.

**Dijkstra's Algorithm** Dijkstra's algorithm exemplifies dynamic programming applied to finding the shortest path from a single source vertex to all other vertices in a weighted graph. With time complexity of $O(V^2)$ for adjacency matrix implementation and $O((V+E) \log V)$ when utilizing a priority queue, it demonstrates high efficiency for graphs with non-negative weights. Dijkstra's algorithm finds widespread application in network routing, GPS navigation, and AI pathfinding. However, its dependency on priority queue structures makes it less effective for graphs containing negative weights.

**Floyd–Warshall Algorithm** The Floyd–Warshall algorithm functions as an all-pairs shortest path solution that applies dynamic programming principles by progressively refining path costs between all vertex pairs. Its $O(V^*)$ time complexity makes it more appropriate for dense graphs requiring comprehensive path computations. While highly effective for smaller graphs, its cubic complexity presents limitations in large-scale applications. Floyd–Warshall commonly serves in network analysis, transitive closure calculations, and routing optimization.

**Summary** Each algorithm addresses distinct use cases with specific trade-offs:

- **Dijkstra's Algorithm**: Ideal for single-source shortest path problems in graphs with non-negative weights.
- **Floyd–Warshall Algorithm**: Optimal for all-pairs shortest path problems in dense graph structures.

Future research might explore hybrid approaches combining elements from both algorithms, such as adapting Dijkstra's algorithm for negative-weight graphs or enhancing Floyd–Warshall's scalability through parallel computing techniques. Additionally, advances in memory-efficient implementations could further expand their real-world applicability.

https://github.com/dmracovit/AA/lab3_4_5