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

Laboratory No.2 Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, one of choice.

Verified:
Fistic Cristofor asist. univ.

Titerez Vladislav FAF-233

Moldova, April 2025

Contents

1 Algorithm Analysis

1.1 Objective

The objective of this work is to conduct an empirical analysis of the Depth First Search (DFS) and Breadth First Search (BFS) algorithms. The study aims to implement these algorithms, analyze their efficiency based on various input conditions, and compare them using relevant performance metrics. Through graphical representations of the results, we seek to draw conclusions about their practical efficiency in different scenarios.

1.2 Tasks

To achieve the outlined objective, the following tasks will be performed: 1. Implement the DFS and BFS algorithms in a chosen programming language. 2. Define the characteristics of the input data to be used in the analysis. 3. Select appropriate metrics for comparing the algorithms, such as execution time, memory usage, and node traversal count. 4. Conduct empirical experiments to measure and compare the performance of the algorithms. 5. Present the obtained results in graphical form to highlight performance differences. 6. Draw conclusions based on the empirical findings, discussing the strengths and weaknesses of each algorithm.

1.3 Theoretical Notes

Graph traversal algorithms are fundamental in computer science, particularly in artificial intelligence, network analysis, and pathfinding problems. The two primary graph traversal strategies, Depth First Search (DFS) and Breadth First Search (BFS), serve different purposes and exhibit distinct performance characteristics.

Depth First Search (DFS) explores as far as possible along each branch before backtracking. This approach is implemented using recursion or an explicit stack. DFS

is particularly useful for solving problems such as maze navigation, topological sorting, and strongly connected components identification in directed graphs. The time complexity of DFS is $O(V + E)$, where V is the number of vertices and E is the number of edges.

Breadth First Search (BFS) explores all neighbors of a node before proceeding to the next level. It is implemented using a queue and is especially useful in shortest path algorithms (e.g., unweighted graph shortest path problems) and network broadcasting. Like DFS, BFS has a time complexity of $O(V + E)$.

Key properties influencing algorithm performance include: - Graph density (sparse vs. dense graphs) - Graph structure (directed vs. undirected, cyclic vs. acyclic) - Graph size (number of nodes and edges) - Memory usage (DFS uses less memory in most cases, while BFS requires more due to queue storage)

Empirical analysis helps in evaluating these properties by measuring execution time, memory consumption, and other relevant factors under different input conditions.

1.4 Introduction

Graph traversal is a fundamental problem in computer science with applications in artificial intelligence, social network analysis, route planning, and network topology. Two of the most widely used graph traversal algorithms are Depth First Search (DFS) and Breadth First Search (BFS). While both algorithms systematically visit nodes in a graph, their traversal strategies and performance characteristics differ significantly.

DFS follows a deep exploration strategy, venturing as far as possible along a branch before backtracking. This approach makes it efficient for solving problems that require exhaustive searches, such as puzzle solving, pathfinding in mazes, and cycle detection in graphs. DFS can be implemented either recursively or using an explicit stack.

BFS, on the other hand, explores all neighbors at the present depth before moving on to the next level. This approach is particularly effective in finding the shortest path in an unweighted graph and is widely used in network broadcasting, peer-to-peer applications, and web crawling.

Despite their theoretical similarities, DFS and BFS perform differently based on input conditions such as graph structure, density, and size. Through empirical analysis, we aim to compare these algorithms using selected performance metrics, present the results graphically, and draw conclusions about their practical efficiency. The study will provide insights into when one algorithm may be preferred over the other in real-world applications.

1.5 Comparison Metric

The primary comparison metric in this study is the execution time of each sorting algorithm, denoted as $T(n)$. This metric provides insights into the practical efficiency

of algorithms under varying input sizes and conditions. While theoretical complexity provides an upper-bound estimate, execution time accounts for hardware optimizations, cache behavior, and implementation details that influence performance.

1.6 Input Format

Each sorting algorithm will be tested on multiple datasets of different sizes to evaluate its efficiency across various conditions. The datasets used for testing are:

- **Small Dataset:** Designed for analyzing the performance of algorithms on smaller inputs, this dataset consists of randomly generated arrays with sizes ranging from:

$$10, 50, 100, 200, 500$$

- **Large Dataset:** Used for testing scalability and efficiency on larger inputs, this dataset contains arrays of increasing sizes:

$$1000, 5000, 10000, 50000, 100000$$

This structured input format ensures a comprehensive evaluation of sorting algorithms, providing a clear comparison between their theoretical and empirical performance.

2 Implementation

These two algorithms will be implemented in their native form in JavaScript and analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on the memory of the device used. The error margin determined will constitute 0.0005 seconds as per experimental measurement.

2.1 Depth First Search (DFS)

2.1.1 Introduction

Depth First Search (DFS) is a fundamental graph traversal algorithm used to explore nodes and edges of a graph systematically. DFS starts at a chosen source node and explores as far as possible along each branch before backtracking. It is commonly used in solving maze problems, cycle detection, topological sorting, and pathfinding in graphs. DFS can be implemented using recursion or an explicit stack. It can be applied to both directed and undirected graphs and works for both cyclic and acyclic structures.

2.1.2 Working Principle

The working principle of DFS follows a **recursive** or **iterative** approach using a stack:

1. Start from a source node and mark it as visited.
2. Explore its adjacent (neighbor) nodes one by one.
3. If an adjacent node is unvisited, recursively visit that node and continue deeper.
4. Backtrack when no further adjacent unvisited nodes exist.
5. Repeat until all reachable nodes are visited.

DFS ensures all connected components in the graph are explored, and in cases of disconnected graphs, DFS must be executed for each unvisited component.

2.1.3 Advantages

DFS offers several advantages:

- Uses less memory compared to Breadth-First Search (BFS) in sparse graphs.
- Provides an efficient way to check connectivity in graphs.
- Helps in finding strongly connected components in directed graphs.
- Useful for topological sorting in Directed Acyclic Graphs (DAGs).
- Helps in cycle detection in graphs.

2.1.4 JS Implementation

```
async function runDFS() {
  drawGraph();
  await delay(500);
  const visited = {};
  const order = [];

  async function dfs(node) {
    visited[node] = true;
    order.push(node);
    drawGraph(order);
    await delay(500);
    for (let neighbor of graph[node]) {
      if (!visited[neighbor]) {
        await dfs(neighbor);
      }
    }
  }

  await dfs(0);
}
```

Listing 2.1: *Depth First Search (DFS)*

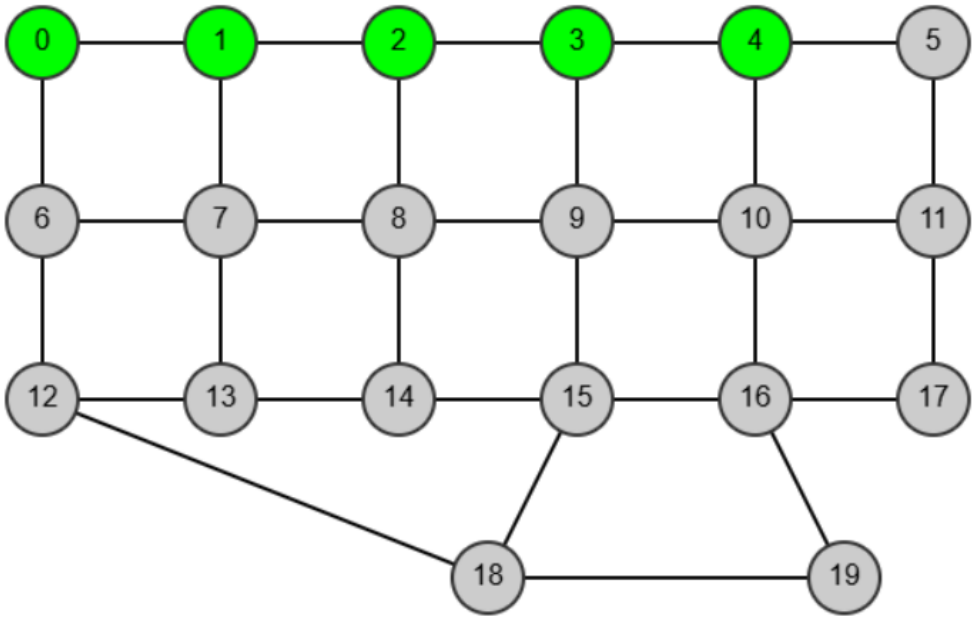

Figure 2.1: *Depth First Search (DFS) graph*

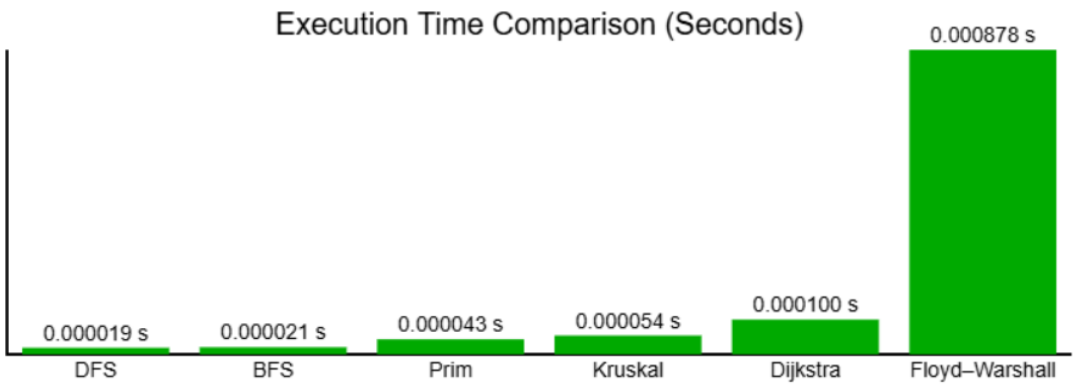

Figure 2.2: *Depth First Search (DFS) performance*

2.1.5 Time Complexity

Time Complexity:

- Best and Average Case: $O(V + E)$
- Worst Case: $O(V + E)$

where:

- V is the number of vertices (nodes).
- E is the number of edges.

Each vertex and edge is visited once, leading to a linear time complexity.

2.1.6 Auxiliary Space

Auxiliary Space:

- Best and Average Case: $O(V)$
- Worst Case: $O(V)$

DFS requires memory to store the recursion stack (for recursive DFS) or an explicit stack (for iterative DFS). In the worst case (for a linear graph), the depth of recursion can reach $O(V)$.

2.2 Breadth First Search (BFS)

2.2.1 Introduction

Breadth First Search (BFS) is a fundamental graph traversal algorithm that explores all neighbors of a node before moving to the next level. It is widely used in shortest path finding, network analysis, and AI problem-solving.

2.2.2 Working Principle

The BFS algorithm works as follows:

1. Start from a source node and mark it as visited.
2. Add the source node to a queue.
3. While the queue is not empty:
    - Dequeue a node and process it.
    - Enqueue all its unvisited adjacent nodes and mark them as visited.
4. Repeat until all reachable nodes are visited.

2.2.3 Advantages

BFS provides several benefits:

- Finds the shortest path in an unweighted graph.
- Useful in web crawling, social network analysis, and AI (e.g., maze-solving algorithms).
- Ensures that all reachable nodes are discovered in a systematic manner.
- Can be implemented iteratively using a queue, making it less prone to stack overflow errors.

2.2.4 JS Implementation

```
async function runBFS() {
  drawGraph();
  await delay(500);
```

```
  const visited = {};
  const order = [];
  const queue = [0];
  visited[0] = true;

  while (queue.length) {
    const node = queue.shift();
    order.push(node);
    drawGraph(order);
    await delay(500);
    for (let neighbor of graph[node]) {
      if (!visited[neighbor]) {
        visited[neighbor] = true;
        queue.push(neighbor);
      }
    }
  }
}
```

Listing 2.2: *Breadth First Search (BFS)*

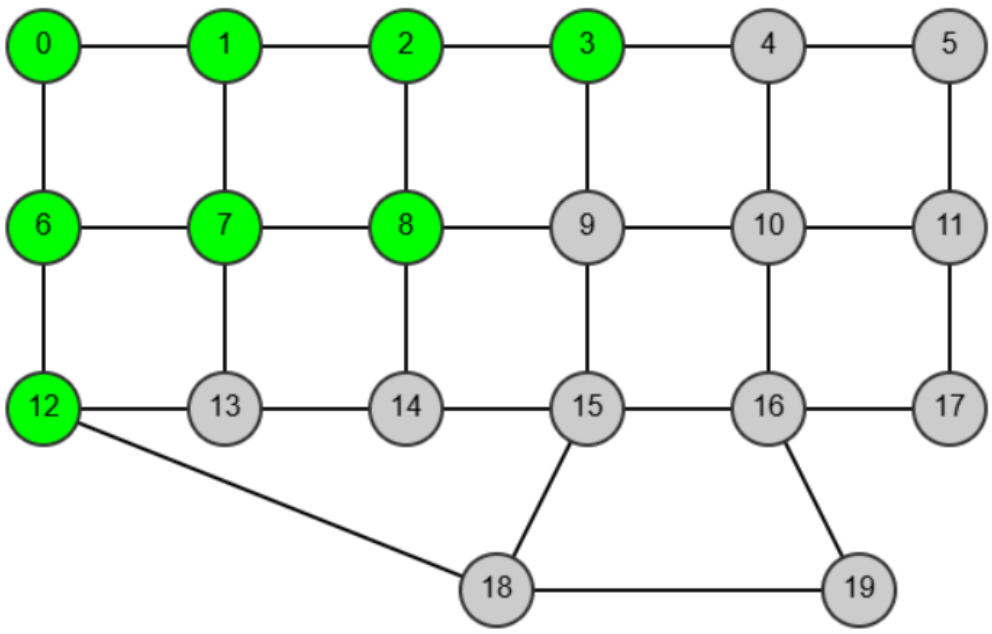

Figure 2.3: *Breadth First Search (BFS) graph*

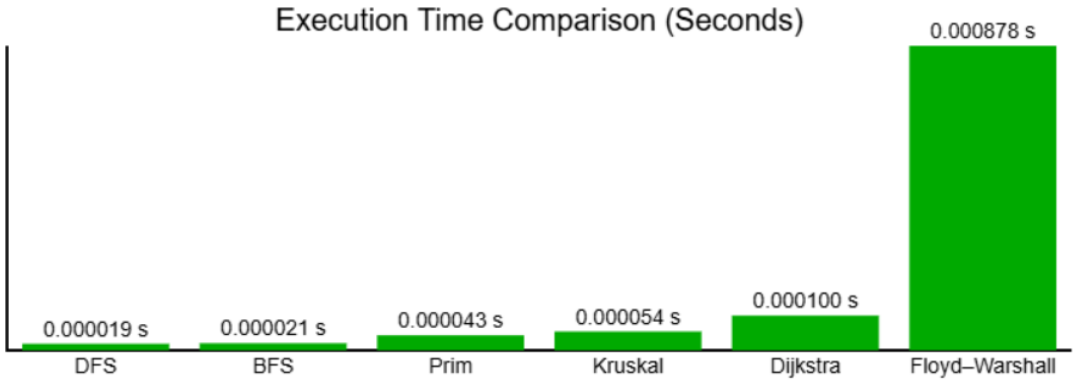

Figure 2.4: *Breadth First Search (BFS) performance*

2.2.5 Time Complexity

Time Complexity:

- Best and Average Case: $O(V + E)$
- Worst Case: $O(V + E)$

2.2.6 Auxiliary Space

Auxiliary Space:

- Best and Average Case: $O(V)$
- Worst Case: $O(V)$

3 Conclusion

This paper has analyzed two fundamental graph traversal algorithms—Depth First Search (DFS) and Breadth First Search (BFS)—evaluating their efficiency, time complexity, auxiliary space requirements, and best-use scenarios. The objective was to highlight their strengths, weaknesses, and applicability in different problems.

Depth First Search (DFS)

DFS is a graph traversal technique that explores as deep as possible along a branch before backtracking. With a time complexity of $O(V + E)$, it efficiently processes all vertices and edges in a graph. Its space complexity of $O(V)$ is determined by the recursion stack in the worst case. DFS is particularly useful for applications such as cycle detection, topological sorting, and maze-solving algorithms. However, its recursive nature can lead to stack overflow in deep graphs, making iterative implementations preferable in some cases.

Breadth First Search (BFS)

BFS explores all neighboring nodes at the present depth level before proceeding to the next level. Like DFS, it has a time complexity of $O(V+E)$ but uses $O(V)$ auxiliary space due to the queue required for level-order traversal. BFS is ideal for finding the shortest path in unweighted graphs, network broadcasting, and web crawling. Despite its predictable performance, BFS may be inefficient in memory-constrained environments due to the large queue size when traversing broad graphs.

Summary

Each traversal algorithm has unique advantages, making them suitable for different applications:

- **DFS**: Best suited for deep graph exploration, cycle detection, and topological sorting.
- **BFS**: Optimal for shortest path discovery, level-order processing, and network traversal.

Future research could focus on optimizing DFS and BFS for large-scale graphs, including hybrid approaches that balance depth and breadth traversal strategies. Enhancements in memory-efficient implementations and parallel processing techniques could further expand their applicability in real-world scenarios. https://github.com/vvtttvv/AA/tree/main/lab3_4_5