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

Laboratory No.3 Study and empirical analysis of sorting algorithms. Analysis of DFS, and BFS.

Verified:
Fistic Cristofor asist. univ.

Racovita Dumitru FAF-233

Moldova, April 2025

Contents

1 Algorithm Analysis

1.1 Objective

The aim of this laboratory work is to perform empirical analysis on Depth First Search (DFS) and Breadth First Search (BFS) algorithms. This study involves implementing these algorithms, evaluating their performance across different input scenarios, and comparing them using appropriate performance metrics. By visualizing the results graphically, we intend to draw meaningful conclusions about their practical effectiveness in various situations.

1.2 Tasks

To fulfill the stated objective, we will undertake the following tasks: 1. Create implementations of DFS and BFS algorithms in our selected programming language. 2. Specify the properties of input data for our analysis. 3. Identify suitable metrics for algorithm comparison, including execution time, memory consumption, and node traversal efficiency. 4. Execute empirical tests to evaluate and compare algorithm performance. 5. Display the results graphically to illustrate performance variations. 6. Formulate conclusions based on empirical evidence, highlighting the strengths and limitations of each algorithm.

1.3 Theoretical Notes

Graph traversal algorithms are essential in computer science, with applications in artificial intelligence, network analysis, and pathfinding challenges. The two main traversal strategies, Depth First Search (DFS) and Breadth First Search (BFS), fulfill different purposes and exhibit unique performance profiles.

Depth First Search (DFS) progresses as deeply as possible along each path before backtracking. This method utilizes recursion or an explicit stack. DFS is particularly valuable for maze navigation, topological ordering, and identifying strongly connected components in directed graphs. The time complexity of DFS is O(V + E), where V represents vertices and E represents edges.

Breadth First Search (BFS) explores all adjacent nodes before advancing to the next depth level. It utilizes a queue data structure and excels in shortest path algorithms (e.g., for unweighted graphs) and network broadcasting scenarios. Similar to DFS, BFS has a time complexity of O(V + E).

Key factors affecting algorithm performance include: - Graph density (sparse versus dense structures) - Graph organization (directed/undirected, cyclic/acyclic) - Graph dimensions (node and edge count) - Memory utilization (DFS typically requires less memory, while BFS demands more due to queue requirements)

Empirical analysis helps evaluate these characteristics by measuring execution speed, memory usage, and other relevant factors under various input conditions.

1.4 Introduction

Graph traversal represents a fundamental challenge in computer science with wide-ranging applications in artificial intelligence, social network analysis, route planning, and network topology. Depth First Search (DFS) and Breadth First Search (BFS) are two of the most widely implemented graph traversal algorithms. While both systematically visit graph nodes, they employ different traversal approaches and demonstrate distinct performance characteristics.

DFS employs a strategy of deep exploration, advancing as far as possible along a path before backtracking. This approach proves efficient for problems requiring exhaustive searches, such as puzzle resolution, maze navigation, and cycle identification in graphs. DFS can be implemented through recursion or with an explicit stack.

BFS, conversely, examines all neighboring nodes at the current depth before proceeding to the next level. This method is particularly effective for identifying shortest paths in unweighted graphs and is commonly applied in network broadcasting, peer-to-peer systems, and web crawling applications.

1.5 Comparison Metric

In this study, the primary comparison metric is the execution time of each algorithm, represented as T(n). This measurement offers insights into the practical efficiency of algorithms under varying input sizes and conditions. While theoretical complexity provides an upper-bound estimate, actual execution time accounts for hardware optimizations, cache behavior, and implementation details that influence real-world 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 mea- surement.

2.1 Depth First Search (DFS)

2.1.1 Introduction

Depth First Search (DFS) represents a core graph traversal algorithm used to systematically explore graph nodes and edges. DFS begins at a designated source node and explores as deeply as possible along each branch before backtracking. It is commonly employed in maze solving, cycle detection, topological sorting, and graph pathfinding. DFS implementations may use either recursion or an explicit stack. The algorithm is applicable to both directed and undirected graphs, functioning effectively with 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 at a designated node (root)
2. Mark the current node as visited
3. For each unvisited neighbor of the current node:
4. Recursively apply DFS to that neighbor
5. Backtrack when all neighbors have been 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 Applications

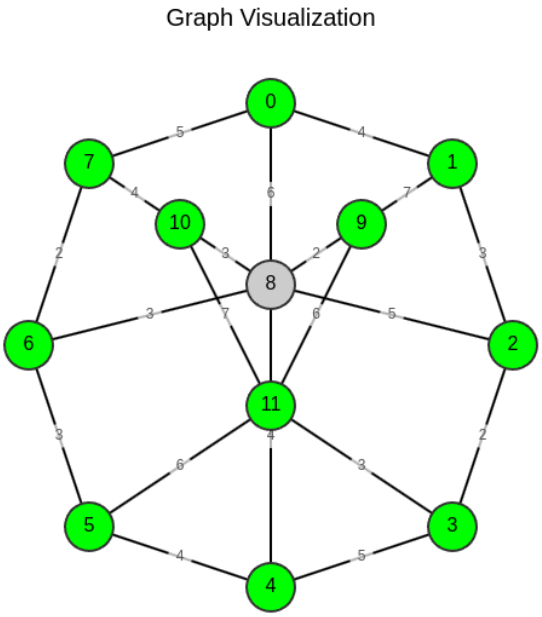
Topological sorting
Finding connected components
Maze solving
Cycle detection

2.1.5 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)*

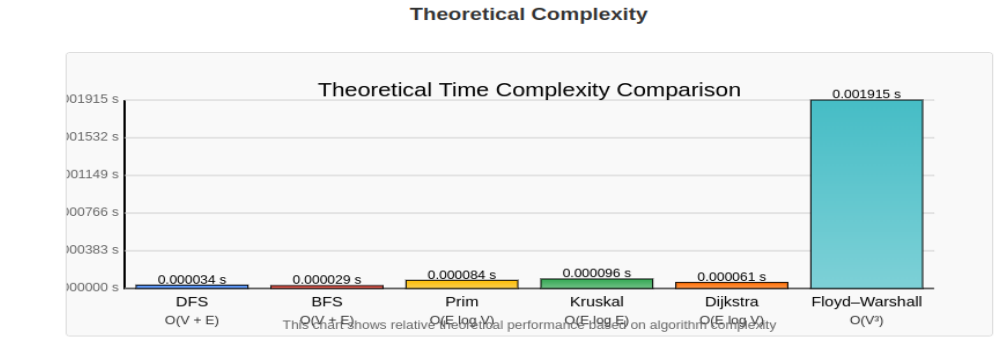


Figure 2.2: *Depth First Search (DFS) performance*

2.1.6 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.

2.1.7 Auxiliary Space

Auxiliary Space:

- Best and Average Case: 0(V)
- Worst Case: 0(V)

DFS requires memory for storing the recursion stack (in recursive implementations) or an explicit stack (in iterative implementations). In the worst case (for a linear graph), recursion depth may 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 neighboring nodes before advancing to the next depth level. It finds widespread use in shortest path discovery, network analysis, and AI problem-solving.

2.2.2 Working Principle

The BFS algorithm functions as follows:

- Begin at a source node and mark it as visited.
- Add the source node to a queue.
- While the queue contains elements:
- Dequeue a node and process it.
- Enqueue all its unvisited adjacent nodes and mark them as visited.
- Continue until all reachable nodes have been 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 algo- rithms).

- Can be implemented iteratively using a queue, making it less prone to stack over- flow errors.

2.2.4 JS Implementation

```
1 async function runBFS() {
2   drawGraph();
3   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)*

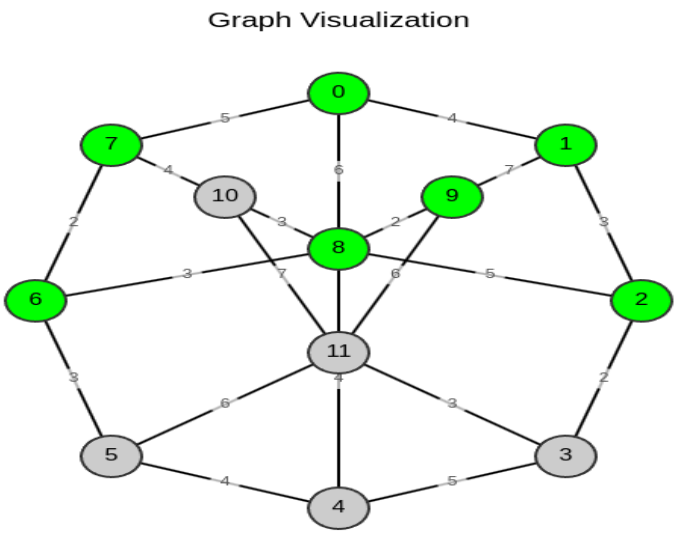


Figure 2.3: *Breadth First Search (BFS) graph*

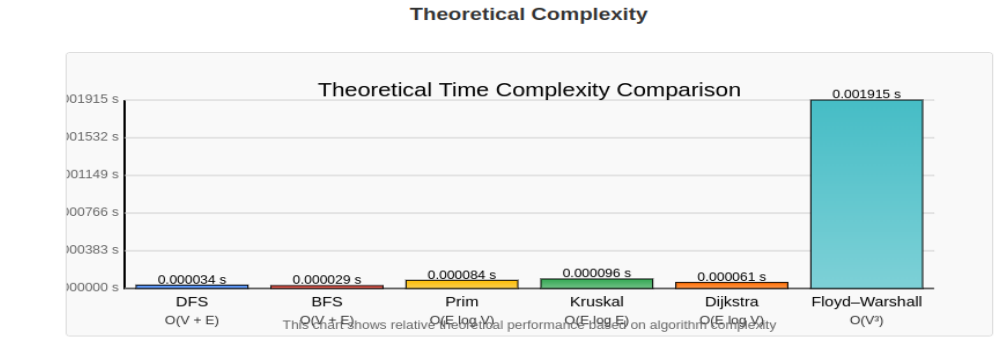


Figure 2.4: *Breadth First Search (BFS) performance*

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 examined two fundamental graph traversal algorithms—Depth FirstSearch (DFS) and Breadth First Search (BFS)—evaluating their efficiency, time complexity, auxiliary space requirements, and optimal usage scenarios. Our objectivewas to highlight their strengths, weaknesses, and applicability to different problem domains.

Depth First Search (DFS) DFS is a graph traversal technique that explores as deeply 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 depth in worst-case scenarios. DFS proves particularly useful for applications such as cycle detection, topological sorting, and maze navigation. However, its recursive nature may lead to stack overflow in deeply nested graphs, making iterative implementations preferable in certain situations.

Breadth First Search (BFS) BFS investigates all neighboring nodes at the current depth level before proceeding to nodes at the next level. Like DFS, it maintains a time complexity of O(V + E) but requires O(V) auxiliary space due to the queue needed for level-order traversal. BFS excels at finding shortest paths in unweighted graphs, network broadcasting, and web crawling. Despite its predictable performance, BFS may prove inefficient in memory-restricted environments due to large queue sizes when traversing graphs with high branching factors.

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

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

Future research could concentrate on optimizing these algorithms for large-scale graphs, including hybrid approaches balancing depth and breadth traversal strategies. Improvements in memory-efficient implementations and parallel processing techniques could further expand their applicability in real-world contexts.