**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 3:

Study and Empirical Analysis of Algorithms: Depth First Search, Breadth First Search

Elaborated:

st. gr. FAF-233Iamandii Ion

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

# ALGORITHM ANALYSIS

## Objective

## Analysis of Depth First Search (DFS), Breadth First Search(BFS)

## Tasks:

## 1 Implement the algorithms listed above in a programming language

## 2 Establish the properties of the input data against which the analysis is performed

## 3 Choose metrics for comparing algorithms

## 4 Perform empirical analysis of the proposed algorithms

## 5 Make a graphical presentation of the data obtained

## 6 Make a conclusion on the work done.

## Theoretical Notes:

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

## Introduction:

Graph traversal is a core concept in computer science, particularly in the study of data structures and algorithms. Two fundamental strategies for exploring graphs are Depth First Search (DFS) and Breadth First Search (BFS). These algorithms serve as the backbone for numerous applications, such as detecting cycles in graphs, finding connected components, solving puzzles and mazes, parsing expressions, and even aiding in web crawling and social network analysis.

DFS explores as far as possible along each branch before backtracking, effectively diving deep into the graph structure. It is typically implemented using recursion or an explicit stack. This approach is particularly useful for tasks that require exploring all possible paths, such as topological sorting or solving constraint satisfaction problems.

In contrast, BFS visits all the neighbors of a node before moving to the next level, expanding outward in layers. It uses a queue to keep track of nodes to visit next, making it ideal for finding the shortest path in unweighted graphs and for level-order traversal of trees.

Within this laboratory, we will be analyzing the 2 algorithms empirically.

## Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n))

## Input Format:

The input for the graph traversal algorithms consists of directed graphs represented as adjacency lists, where each graph contains a varying number of nodes (e.g., 10, 50, 100, up to 500). Edges are generated randomly with assigned weights (although weights are not used in traversal). Two types of graphs are created for each node count: sparse graphs, where edges are less likely (20% probability), and dense graphs, where edges are highly likely (80% probability). Each graph is assumed to be directed, with unique node identifiers ranging from 0 to n−1n−1. The traversal always starts from node 0, which may or may not reach all nodes if the graph is disconnected. This variation in input structure—node count and graph density—provides a comprehensive basis for analyzing traversal performance across different conditions.

# IMPLEMENTATION

Both Depth First Search (DFS) and Breadth First Search (BFS) algorithms are implemented in their naïve iterative forms in Python and analyzed empirically based on the time taken to traverse graphs of varying sizes and densities. While general trends align with theoretical expectations, the exact performance varies depending on system-specific memory and runtime conditions. The program uses a consistent experimental setup, repeating each traversal on both sparse and dense graphs per input size to record aggregate performance. The measured times are plotted for visual comparison, and an average is computed to draw conclusions.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

Github repo: <https://github.com/ion190/aa-labs/tree/main/lab3>

## Depth First Search Algorithm:

In Depth First Search (or DFS) for a graph, we traverse all adjacent vertices one by one. When we traverse an adjacent vertex, we completely finish the traversal of all vertices reachable through that adjacent vertex. This is similar to a tree, where we first completely traverse the left subtree and then move to the right subtree. The key difference is that, unlike trees, graphs may contain cycles (a node may be visited more than once). To avoid processing a node multiple times, we use a boolean visited array.

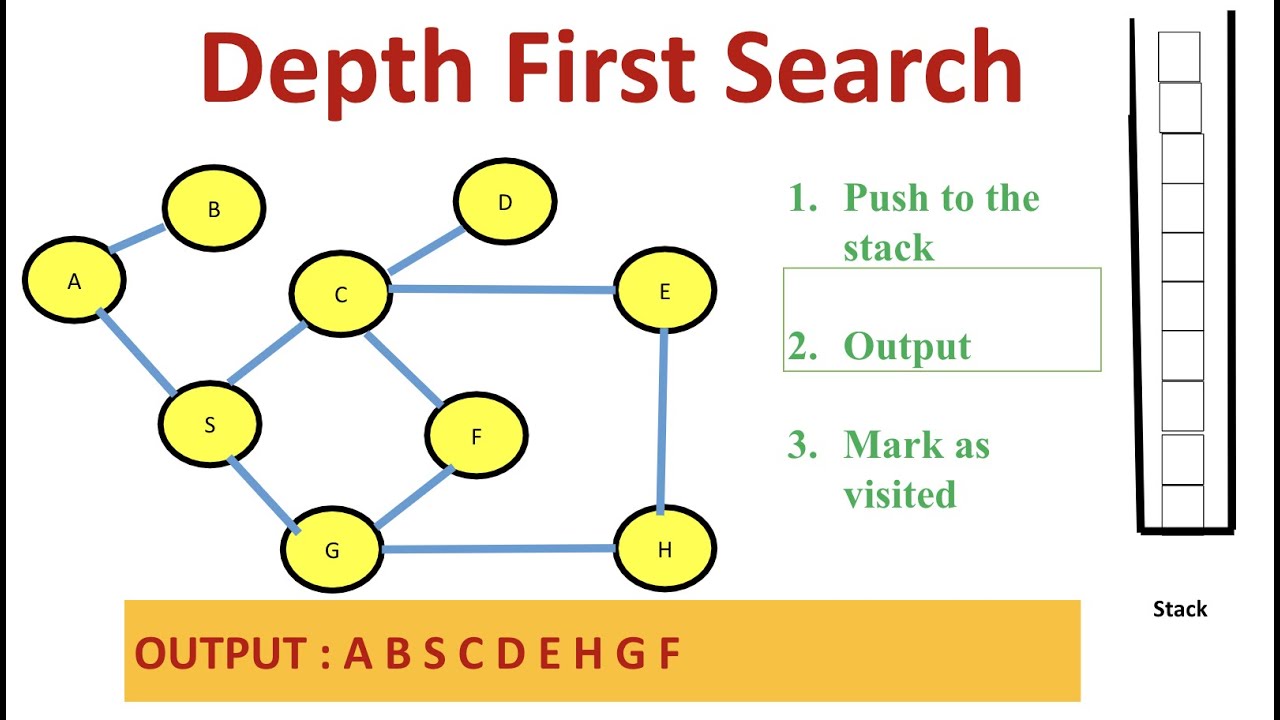


Figure 1. Depth First Search example

*Algorithm Description:*

The Depth First Search algorithm follows the algorithm as shown in the next pseudocode:

procedure DFS(G, v) is

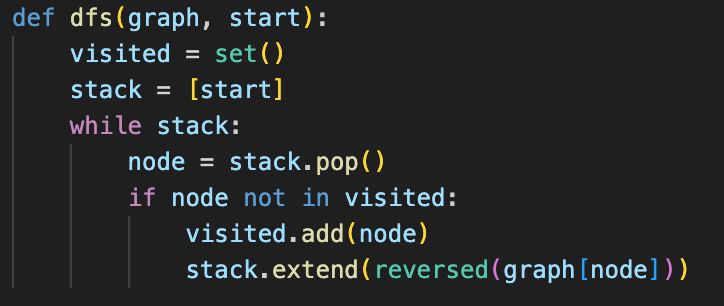
label v as discovered

for all directed edges from v to w that are in G.adjacentEdges(v) do

if vertex w is not labeled as discovered then

recursively call DFS(G, w)

*Implementation:*



*Figure 2 Depth First Search algorithm in Python*

## Breadth First Search Algorithm:

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

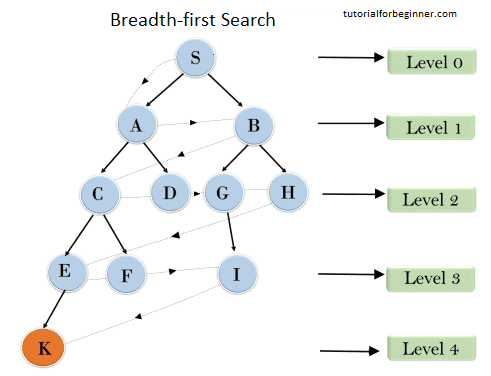


Figure 3. Breadth First Search example

*Algorithm Description:*

The Breadth First Search algorithm follows the algorithm as shown in the next pseudocode:

procedure BFS(G, root) is

let Q be a queue

label root as explored

Q.enqueue(root)

while Q is not empty do

v := Q.dequeue()

if v is the goal then

return v

for all edges from v to w in G.adjacentEdges(v) do

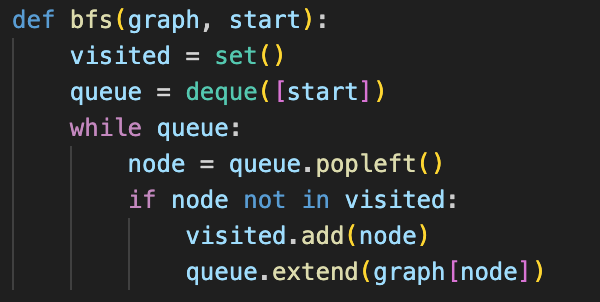
if w is not labeled as explored then

label w as explored

w.parent := v

Q.enqueue(w)

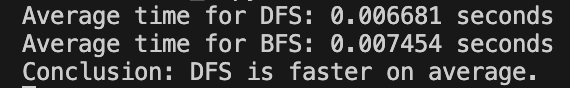
*Implementation:*



*Figure 4 Breadth First Search algorithm in Python*

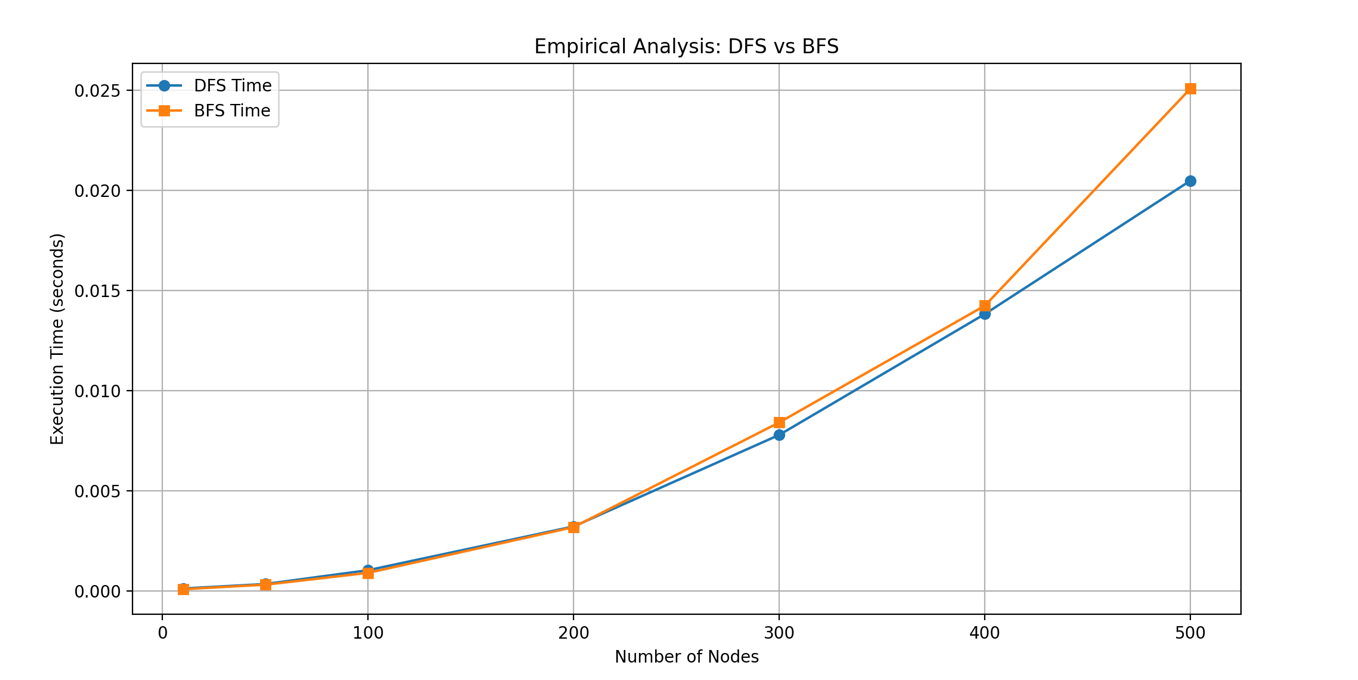
***Results:***

After running the function for different sizes of graphs and saving the time for each, we obtained the following results:



*Figure 5 Results for various set of inputs*

The graph showing the time needed to sort different sizes of graphs:



*Figure 6 Graph of DFS and BFS algorithms*

## Depth First Search (DFS) and Breadth First Search (BFS) both have a time complexity of O(V + E), where V is the number of vertices (nodes) and E is the number of edges in the graph. This reflects the fact that each algorithm visits every vertex once and explores every edge at most once. In sparse graphs (where E is much less than V²), this results in faster traversal times. In dense graphs (where E approaches V²), both algorithms still maintain linear complexity with respect to the total input size. Although they share the same theoretical time complexity, actual performance may differ slightly due to implementation details such as stack vs. queue usage, memory access patterns, and system-level factors like cache locality and branching behavior.

# CONCLUSION

Through empirical analysis, this study evaluates the performance of two fundamental graph traversal algorithms—Depth First Search (DFS) and Breadth First Search (BFS)—in terms of execution time and adaptability across graphs of varying sizes and densities, with the goal of identifying optimal use cases and performance characteristics.

Depth First Search (DFS) explores a graph by going as deep as possible along each branch before backtracking, using a stack (explicitly or via recursion). It has a time complexity of O(V + E), making it efficient for both sparse and dense graphs. In practice, DFS often performs slightly faster than BFS on dense graphs due to reduced memory overhead and a more linear access pattern. However, it does not guarantee the shortest path in unweighted graphs and may fail to reach all nodes if the graph is disconnected from the start node.

Breadth First Search (BFS) traverses level by level using a queue, also achieving O(V + E) time complexity. It is the algorithm of choice when shortest-path guarantees are required in unweighted graphs. BFS can be slower in practice than DFS on dense graphs due to queue operations and broader search frontiers, which may increase memory usage. Nonetheless, its systematic nature makes it more suitable for complete traversals and pathfinding tasks.

Based on this empirical analysis, DFS is often faster for full graph exploration, particularly in denser structures where stack-based traversal benefits from better memory locality. BFS, on the other hand, is preferred in applications where shortest paths or layer-based exploration are needed, despite its higher memory footprint. Both algorithms are foundational in graph theory and serve distinct purposes depending on the structure of the graph and the requirements of the task.