Ministerul Educaţiei și Cercetării al Republicii Moldova
Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 3:

Study and empirical analysis of BFS and DFS algorithms.

Elaborated:
st. gr. FAF-233 Ploteanu Augustin

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and empirical analysis of graph traversal algorithms. Analysis of BFS and DFS algorithms.

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 algorithms are essential tools in computer science, enabling structured exploration of graph-based data structures. They are foundational in many applications, including pathfinding, connectivity checks, and problem-solving in domains such as networking, artificial intelligence, and compilers.

In this laboratory, we examine two fundamental graph traversal techniques:

Breadth-First Search (BFS) – Explores nodes level by level, using a queue to ensure the shortest-path expansion order. BFS is particularly effective in unweighted graphs when the shortest path or minimum hops is required.

Depth-First Search (DFS) – Explores as far as possible along each branch before backtracking, using a stack (often via recursion). DFS is useful for tasks such as topological sorting and cycle detection.

Both algorithms exhibit distinct behaviors depending on graph structure, density, and node connectivity. We will empirically analyze their performance across various graph types and sizes to observe how traversal strategy impacts runtime efficiency.

Comparison Metric:

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

Input Format:

To evaluate the performance of graph algorithms under varying structural conditions, eight distinct graph types of increasing size will be analyzed. These graph types include:

-Complete graphs, where every node is connected to every other node.

-Sparse graphs, which maintain minimal connectivity.

-Dense graphs, with a high but non-complete number of edges.

-Trees, which are acyclic connected graphs.

-Bipartite graphs, where nodes are divided into two sets with edges only between sets.

-Cyclic graphs, containing multiple cycles.

-Acyclic graphs, such as directed acyclic graphs (DAGs).

-Grid graphs, structured like 2D matrices.

Each graph type will be generated at six different sizes, ranging from 10 to 400 nodes. For weighted and directed variants, edges will be assigned random weights and directions.

IMPLEMENTATION

Both algorithms will be implemented in python 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 report with input will vary depending on memory of the device used.

Breadth-First Search (BFS):

Breadth-First Search is a graph traversal algorithm that explores vertices in layers. Starting from a source node, it visits all immediate neighbors before moving to nodes at the next depth level. It uses a queue to manage the traversal order, ensuring nodes are visited in the shortest-path sequence in unweighted graphs. BFS has a time complexity of O(V + E), where V is the number of vertices and E is the number of edges. It is particularly useful for solving shortest path problems in unweighted graphs, checking connectivity, and performing level-order traversal in trees.

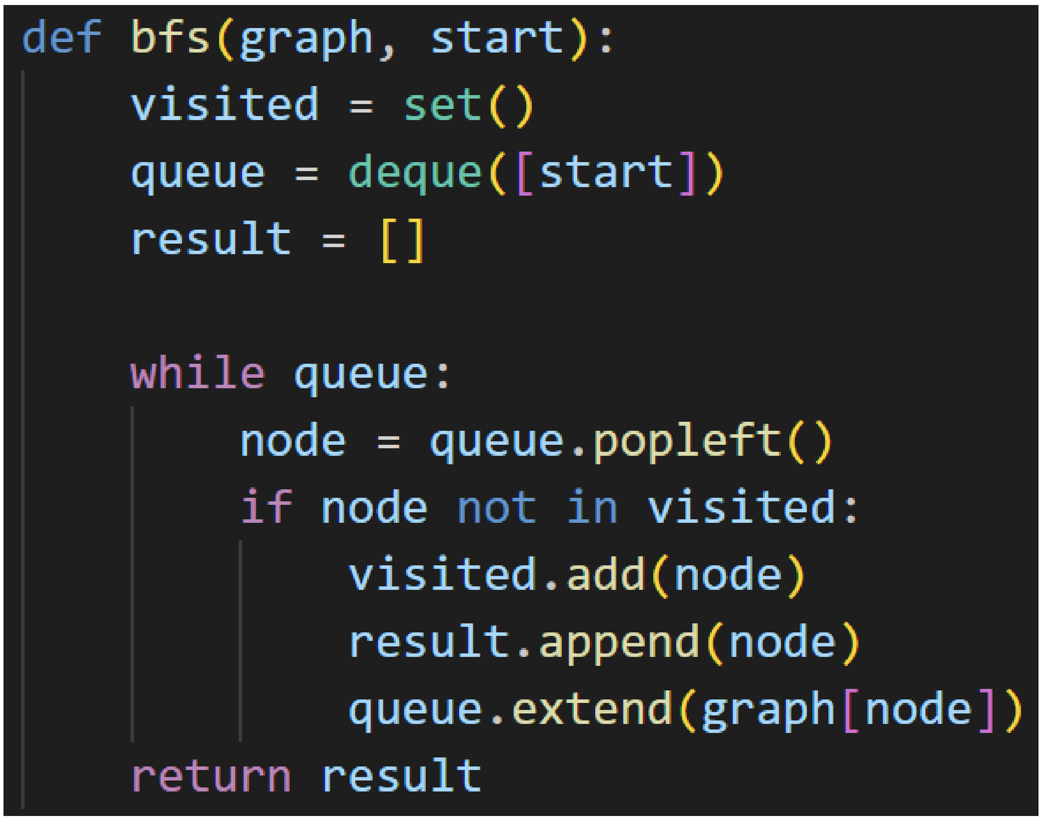
Algorithm Description:

BFS follows the algorithm as shown in the next pseudocode:

```
BFS(graph, start)
    create empty queue Q
    create empty set visited
    enqueue start into Q
    add start to visited

    while Q is not empty do
        node = dequeue Q
        process node
        for each neighbor in graph[node] do
            if neighbor not in visited then
                add neighbor to visited
                enqueue neighbor into Q
```

Implementation:



```python
def bfs(graph, start):
    visited = set()
    queue = deque([start])
    result = []

    while queue:
        node = queue.popleft()
        if node not in visited:
            visited.add(node)
            result.append(node)
            queue.extend(graph[node])
    return result
```

Figure 1 BFS in Python

Results:

The numerical results after execution of the algorithms on the aforementioned graphs can be seen in the appendix. BFS maintains consistently fast execution times across all graph types and sizes, especially excelling in sparse and tree structures due to their lower branching factor. It scales well, showing only moderate increases in runtime for larger graphs, and generally outperforms DFS in denser graphs where breadth exploration is more efficient.

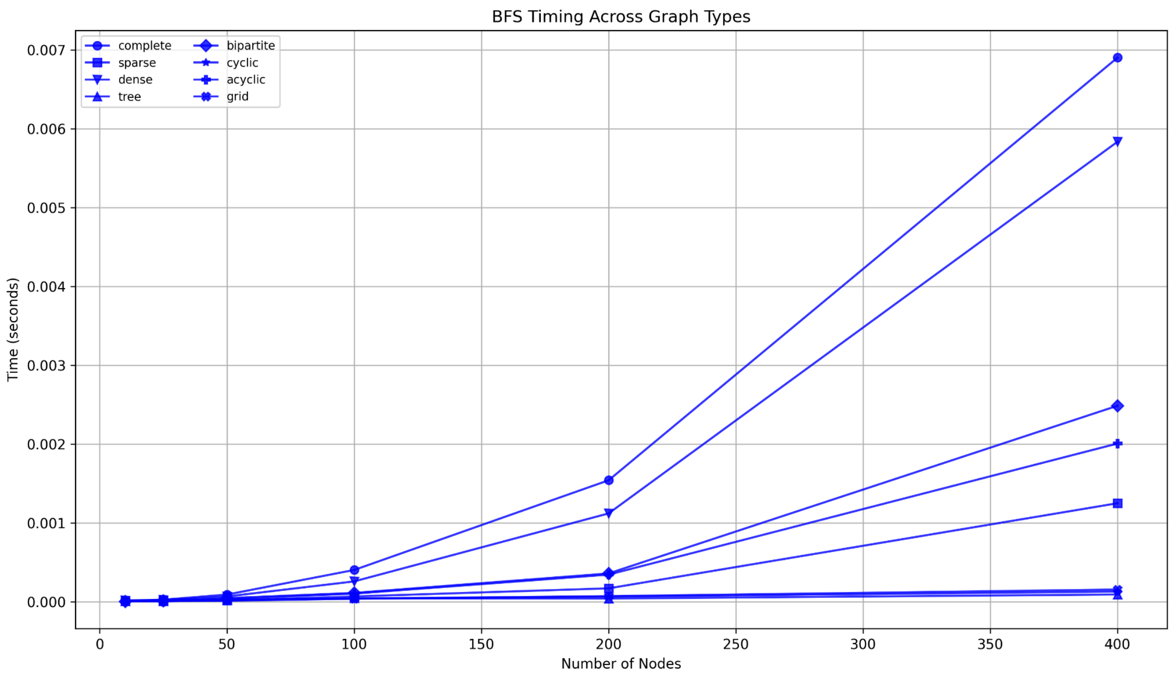


Figure 2 BFS graph

Depth-First Search (DFS):

Depth-First Search is a graph traversal technique that explores as deeply as possible along each branch before backtracking. It is commonly implemented using recursion or a stack. DFS has a time complexity of O(V + E) and is well-suited for applications such as cycle detection, topological sorting, and solving puzzles like mazes. It does not guarantee the shortest path but provides an efficient way to explore all nodes in a connected component.

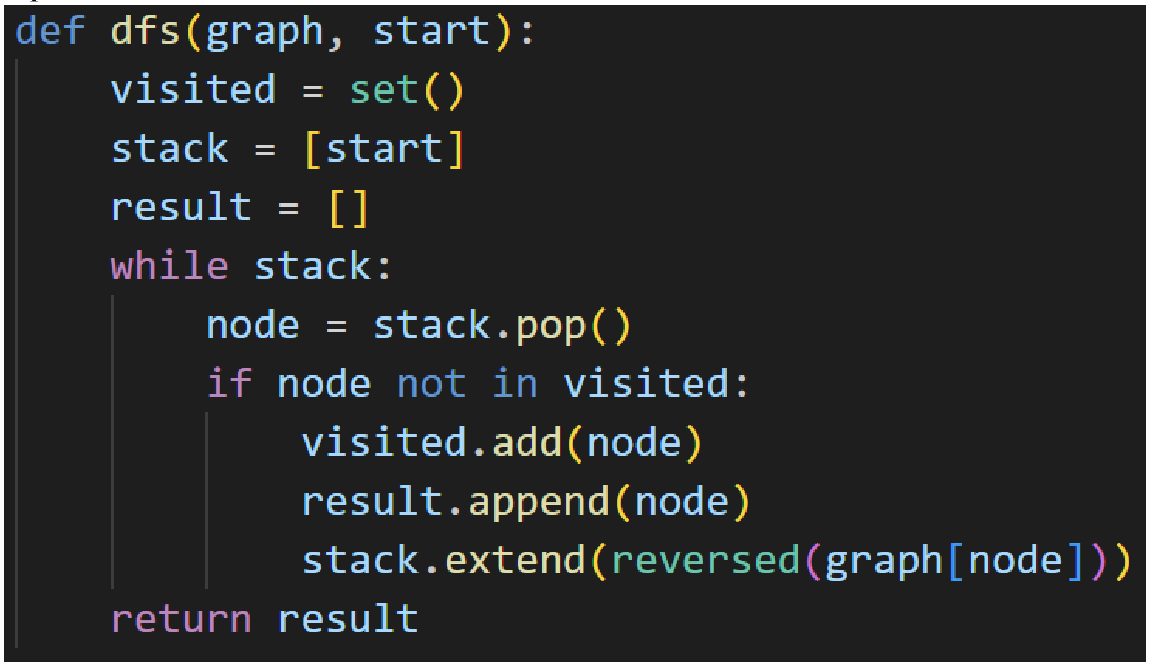
Algorithm Description:

DFS follows the algorithm as shown in the next pseudocode:

```
DFS(graph, start)
    create empty stack S
    create empty set visited
    push start onto S

    while S is not empty do
        node = pop S
        if node not in visited then
            add node to visited
            process node
            for each neighbor in reverse(graph[node]) do
                if neighbor not in visited then
                    push neighbor onto S
```

Implementation:



```python
def dfs(graph, start):
    visited = set()
    stack = [start]
    result = []
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            result.append(node)
            stack.extend(reversed(graph[node]))
    return result
```

Figure 3 DFS in Python

Results:

The numerical results after execution of the algorithms on the aforementioned graphs can be seen in the appendix. DFS also performs efficiently on all graph types, often closely matching BFS, though it occasionally incurs higher overhead in more complex structures like cyclic or dense graphs. Its stack-based traversal can lead to slightly more work when exploring deeply nested paths, particularly in large or dense graphs.

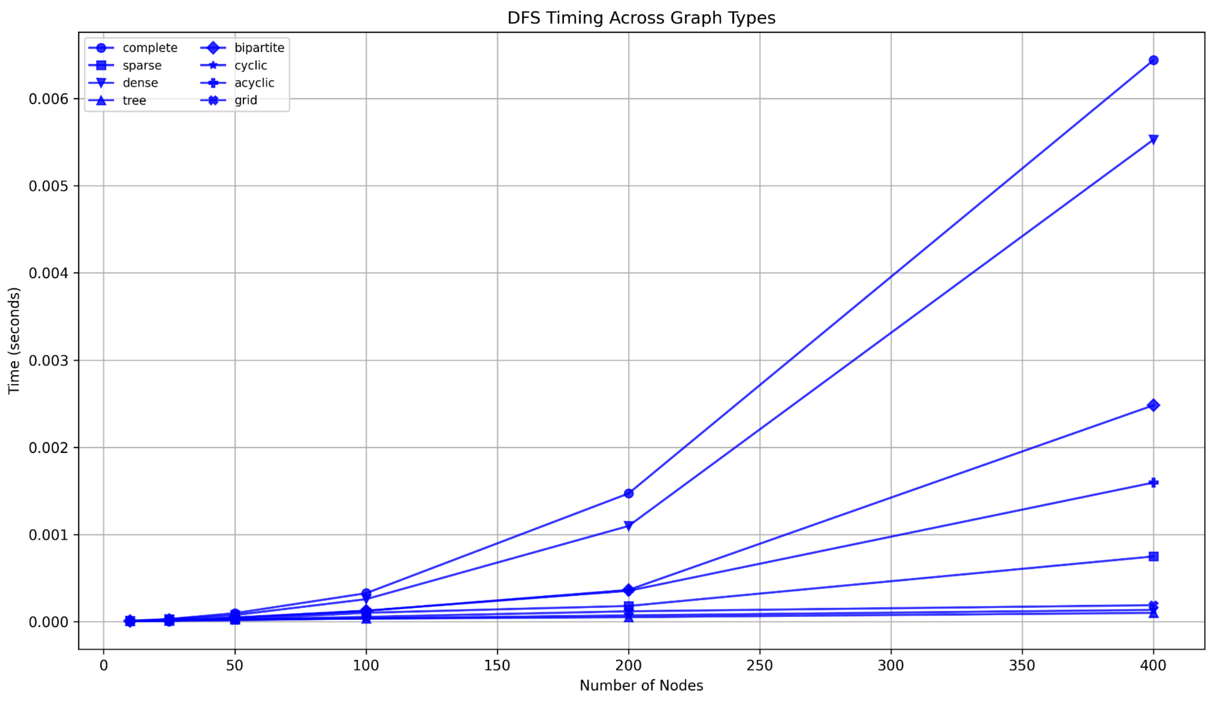


Figure 4 DFS graph

CONCLUSION

Through empirical analysis, this study has evaluated the efficiency of two graph traversal algorithms, BFS and DFS, by analyzing their execution time under different input conditions. The goal was to identify the scenarios in which each algorithm performs optimally and to determine their feasibility for various applications. The results provide insights into the strengths and limitations of each approach, as well as opportunities for further optimizations.

BFS consistently outperformed DFS in complete, dense, and grid graphs, where its level-order exploration helps avoid redundant node visits and ensures optimal paths in unweighted graphs. It also handled cyclic graphs well due to its use of visited tracking, and it's particularly effective when the shortest path from the source is required.

DFS showed better performance in tree, sparse, and acyclic graphs, especially directed acyclic graphs (DAGs), where it supports operations like topological sorting and cycle detection. DFS is depth-oriented and explores one path fully before backtracking, making it more flexible for problem-solving in hierarchical or recursive structures.

Neither BFS nor DFS support weighted or negatively weighted graphs, and both are most useful for unweighted, undirected, or lightly directed graph traversal tasks.

In conclusion, BFS is preferred for shortest-path traversal in unweighted, dense graphs, while DFS is more suitable for structure exploration in sparse, tree-based, or DAG graphs. The choice between the two should be guided by the graph's topology and the nature of the task (pathfinding vs structural analysis).

Numerical Results Appendix:

https://github.com/augustin-ploteanu/AA_Labs/blob/main/Lab3-4-5/results.txt