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

Laboratory work 3: Empirical analysis of algorithms: Depth First Search (DFS), Breadth First Search(BFS)

Elaborated:
st. gr. FAF-221 Gavriliuc Tudor

Verified:
asist. univ. Fiștic Cristofor
prof. univ. Andrievschi-Bagrin Veronica

Chişinău - 2024

TABLE OF CONTENTS

Contents

ALGORITHM ANALYSIS

Objective

Study and analyze different graph traversing 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:

Empirical analysis provides an alternative approach to understanding the efficiency of algorithms when mathematical complexity analysis is impractical or insufficient. This method proves beneficial in various scenarios:

1. Initial Insights: It offers preliminary insights into an algorithm's complexity class, aiding in the understanding of its efficiency characteristics.
2. Comparative Analysis: It facilitates the comparison of multiple algorithms tackling the same problem, allowing for informed decisions regarding efficiency.
3. Implementation Comparison: Empirical analysis enables the comparison of different implementations of the same algorithm, providing insights into which may perform better in practice.
4. Hardware-specific Evaluation: It helps in assessing an algorithm's efficiency on a particular computing platform, taking into account hardware constraints and capabilities.

The empirical analysis of an algorithm typically involves the following steps:
Establishing Analysis Goals: Clearly define the objectives and scope of the analysis.

1. Choosing Efficiency Metrics: Select appropriate metrics, such as the number of operations executed or the execution time, based on the analysis goals.
2. Defining Input Data Properties: Determine the characteristics of the input data relevant to the analysis, including data size or specific attributes.
3. Implementation: Develop the algorithm in a programming language, ensuring it accurately reflects the intended logic.
4. Generating Input Data Sets: Create multiple sets of input data to cover a range of scenarios and edge cases.
5. Execution and Data Collection: Execute the program for each input data set,

recording relevant performance metrics.

6. Data Analysis: Analyze the collected data, either by computing synthetic quantities like mean and standard deviation or by plotting graphs to visualize the relationship between problem size and efficiency metrics.
7. The choice of efficiency measure depends on the analysis's objectives. For instance, if assessing complexity class or verifying theoretical estimates, counting the number of operations may be suitable. Conversely, if evaluating algorithm implementation behavior, measuring execution time becomes more relevant.

8. Post-execution, recorded results undergo analysis. This involves computing statistical measures or plotting graphs to visualize the algorithm's performance characteristics in terms of problem size and efficiency metrics. Such analyses aid in making informed decisions regarding algorithm selection and optimization strategies.

Introduction:

DFS:

Depth First Traversal (DFS) is an algorithm used to explore or search through tree or graph data structures. When applied to graphs, DFS starts at a chosen node (often referred to as the root node in tree structures) and proceeds by exploring as deeply as possible along each branch before backtracking.

One key difference between DFS for trees and DFS for graphs is that graphs may contain cycles, which means a node could be visited multiple times. To prevent processing a node more than once, DFS for graphs typically uses a boolean array called "visited" to keep track of visited nodes.

The time complexity of DFS for graphs is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. The auxiliary space complexity is also $O(V + E)$, as it requires an additional visited array of size V and stack space for recursive calls or iterative DFS implementation.

BFS:

Breadth First Search (BFS) is a fundamental graph traversal algorithm that systematically explores all the connected nodes of a graph in a level-by-level manner. It starts at a specified vertex and visits all its neighboring vertices at the current depth level before moving on to the vertices at the next depth level.

BFS is characterized by the following steps:

1. Initialization: Enqueue the starting node into a queue and mark it as visited.
2. Exploration: While the queue is not empty:
   - Dequeue a node from the queue and visit it.
   - For each unvisited neighbor of the dequeued node:
     - Enqueue the neighbor into the queue.
     - Mark the neighbor as visited.
3. Termination: Repeat step 2 until the queue is empty.

The time complexity of the BFS algorithm is $O(V + E)$, where V is the number of vertices and E is the number of edges in the graph. This is because BFS explores all the vertices and edges in the graph. In the worst case, it visits every vertex and edge once.

The space complexity of BFS is $O(V)$, as it uses a queue to keep track of the vertices that need to be visited. In the worst case, the queue can contain all the vertices in the graph. Therefore, the space complexity of BFS is proportional to the number of vertices V in the graph.

Comparison Metric:

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

Input Format:

As input, each algorithm will receive 8 series of numbers of nodes 4, 8, 16, 32, 64, 128, 256, ,512.

Next, using this numbers of nodes, it will be generated randomly graphs with that amount of nodes.

IMPLEMENTATION

All algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar toother experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used.

Depth First Search

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal

```
class GraphDFS:

    def __init__(self):
        self.graph = defaultdict(list)

    1 usage
    def addEdge(self, u, v):
        self.graph[u].append(v)

    2 usages
    def DFSUtil(self, v, visited):

        visited.add(v)

        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    1 usage
    def DFS(self, v):

        visited = set()
        self.DFSUtil(v, visited)
```

	Nr of vertices	DFS	BFS
0	4	0.000003	0.000015
1	8	0.000003	0.000006
2	16	0.000003	0.000008
3	32	0.000009	0.000018
4	64	0.000029	0.000063
5	128	0.000096	0.000211
6	256	0.000342	0.000483
7	512	0.001445	0.003463

Figure 1. Time table for algorithms

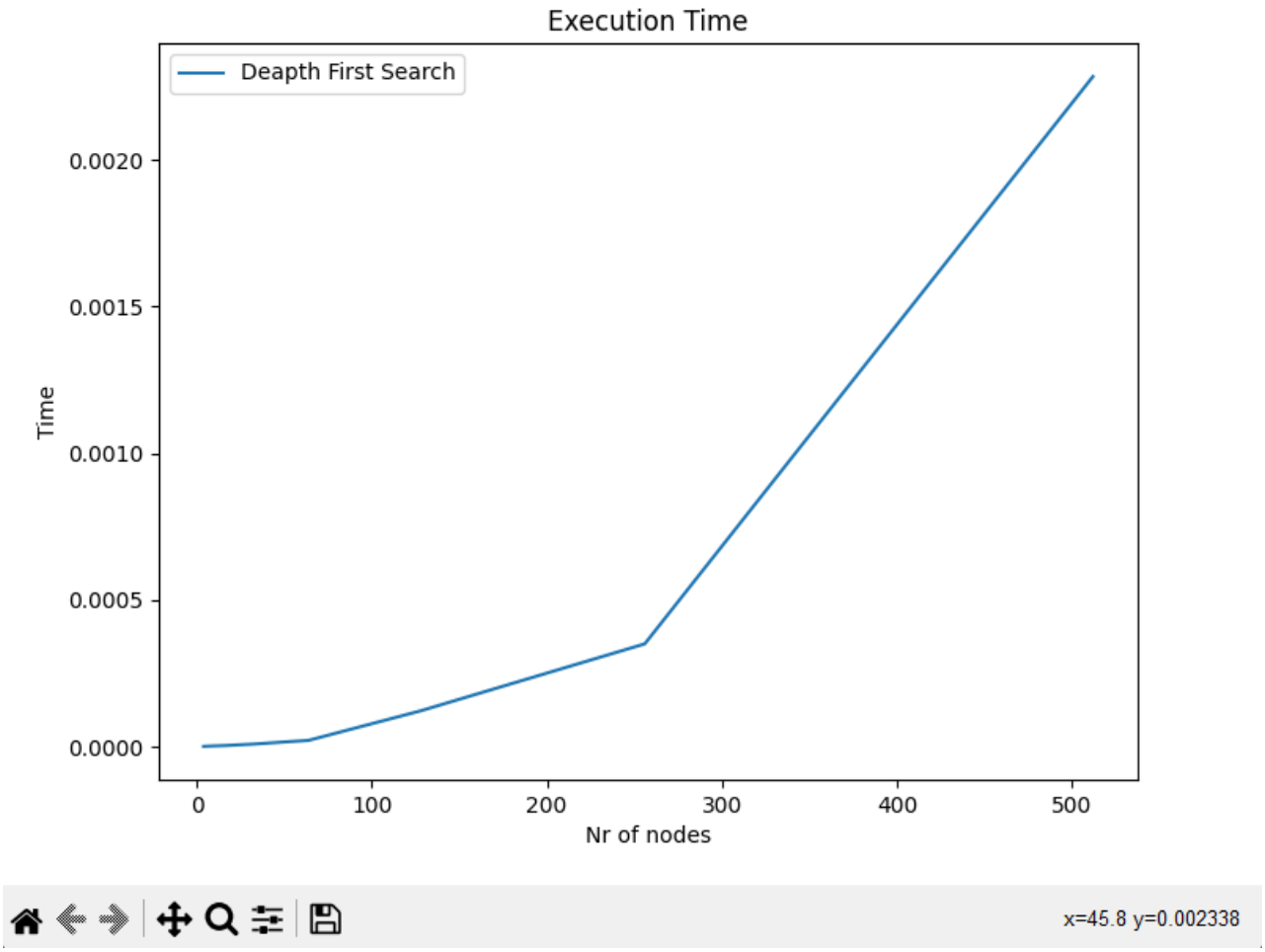


Figure 2. Graph for DFS

The graph illustrates a direct relationship between the number of nodes and the time taken for execution, indicating that depth-first search (DFS) requires more time to traverse graphs with a higher number of nodes. This phenomenon arises due to the inherent nature of DFS, which involves exploring all edges within the graph. Consequently, as the number of nodes increases, so does the number of edges to be traversed, leading to longer execution times for DFS operations.

Breadth First Search:

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors.

```
class GraphBFS:
    def __init__(self):
        # Initialize an empty adjacency list using defaultdict
        self.adjList = defaultdict(list)

    4 usages
    def addEdge(self, u, v):
        # Add an undirected edge between vertices u and v
        self.adjList[u].append(v)
        self.adjList[v].append(u)

    1 usage
    def show(self, adjList):
        # Display the adjacency list (not used in BFS algorithm)
        for key, value in adjList.items():
            print(key, "    ", value, "\n")

    1 usage
    def bfs(self, startNode):
        # Initialize a queue for BFS traversal
        queue = deque()
        # Find the maximum node in the graph
        max_node = max(self.adjList.keys(), default=-1)
        # Initialize a boolean array to keep track of visited nodes
        visited = [False] * (max_node + 1)

        # Mark the startNode as visited and enqueue it
        visited[startNode] = True
        queue.append(startNode)

        # Perform BFS traversal
        while queue:
            # Dequeue a node from the queue
            currentNode = queue.popleft()

            # Iterate through all neighbors of the current node
            for neighbor in self.adjList[currentNode]:
                # Check if the neighbor has not been visited
                if not visited[neighbor]:
                    # Mark the neighbor as visited and enqueue it
                    visited[neighbor] = True
                    queue.append(neighbor)
```

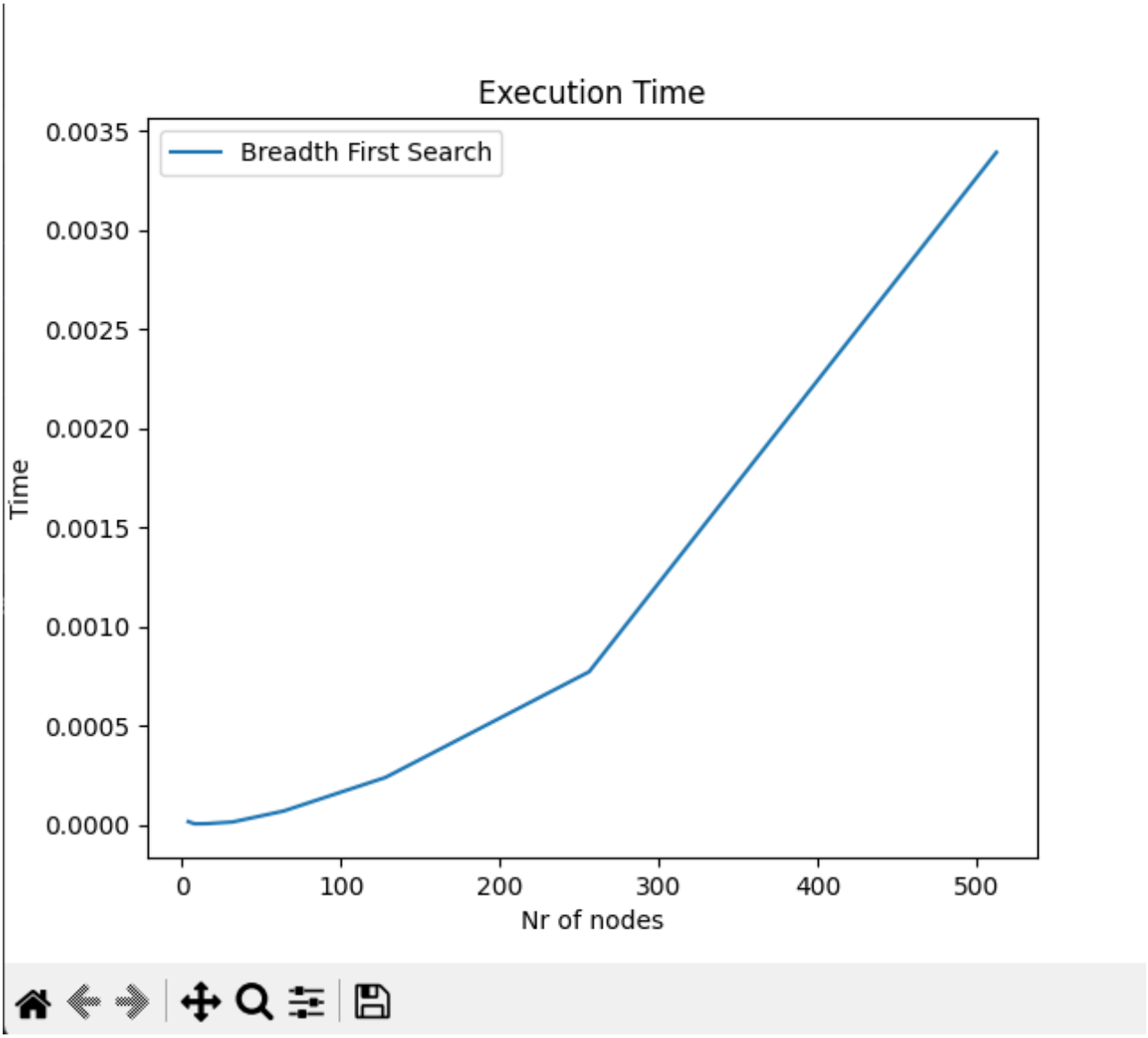
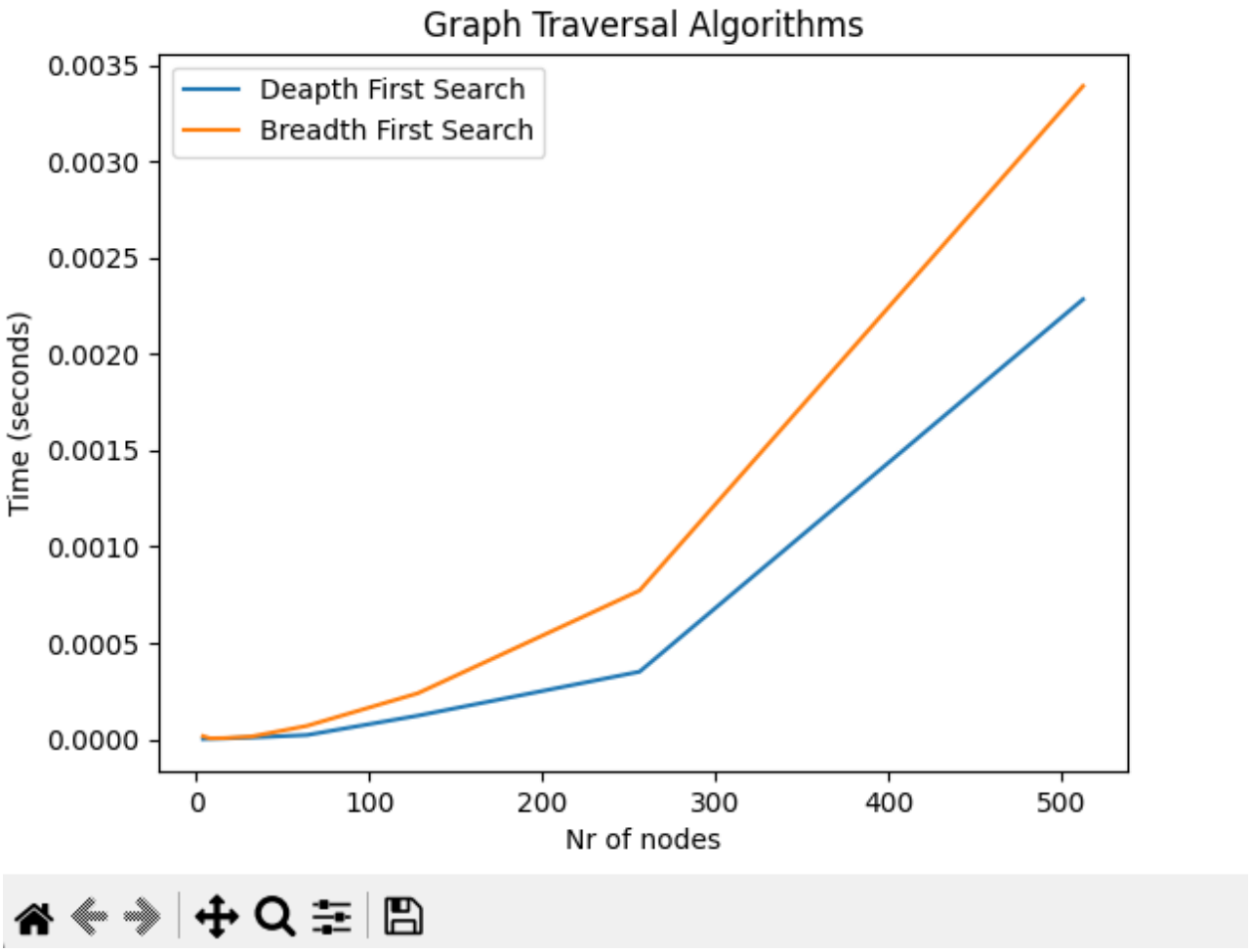


Figure 3. Graph for BFS

As time progresses, we observe a gradual expansion in the number of visited nodes. This pattern aligns with the nature of the Breadth First Search (BFS) algorithm, which systematically explores outward from an initial node, sequentially visiting all its immediate neighbors before proceeding to the neighbors of those neighbors. Hence, the increasing trend in the number of executed nodes over time reflects the iterative exploration of the graph by the BFS algorithm.

CONCLUSION



This examination illuminates the distinct characteristics and performance attributes of Depth-First Search (DFS) and Breadth-First Search (BFS) traversal algorithms on graphs. While both strategies efficiently navigate through graph structures, they exhibit differing time complexities depending on the scenario.

DFS exhibits a time complexity that increases proportionally with the number of nodes in the graph due to its deep exploration of single paths before backtracking. This characteristic makes DFS well-suited for tasks such as node detection and cycle identification, but may result in inefficiencies in graphs with expansive branching structures.

On the other hand, BFS demonstrates a steady increase in the number of explored nodes over time, making it ideal for tasks such as finding the shortest path or identifying connected components within a graph. However, in sparse graphs with few edges, BFS might explore unnecessary nodes, potentially leading to higher execution times compared to DFS.

References:

https://github.com/Tudor-Gavriliuc/AA