Ministry of Education, Culture and Research of the Republic of Moldova

Technical University of Moldova

Department of Software and Automation Engineering

**REPORT**

Laboratory work No. 4

Discipline: Algorithms’ Analysis

Topic: Empirical analysis of algorithms: Depth First Search (DFS), Breadth First Search(BFS)

Elaborated: Țărnă Cristina,

student group FAF-211

Verified: Fiștic Cristofor,

university assistant

Chișinău – 2023

Algorithm analysis

**Objective:**

Implement DFS and BFS algorithms to traverse trees**.**

**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:**

**Depth-first search** (**DFS**) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for traversing or searching [tree](https://en.wikipedia.org/wiki/Tree_data_structure) or [graph](https://en.wikipedia.org/wiki/Graph_(data_structure)) data structures. The algorithm starts at the [root node](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Extra memory, usually a [stack](https://en.wikipedia.org/wiki/Stack_(abstract_data_type)), is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

**Breadth-first search** (**BFS**) is an [algorithm](https://en.wikipedia.org/wiki/Algorithm) for searching a [tree](https://en.wikipedia.org/wiki/Tree_(data_structure)) data structure for a node that satisfies a given property. It starts at the [tree root](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) and explores all nodes at the present [depth](https://en.wikipedia.org/wiki/Tree_(data_structure)#Terminology) prior to moving on to the nodes at the next depth level. Extra memory, usually a [queue](https://en.wikipedia.org/wiki/Queue_(data_structure)), is needed to keep track of the child nodes that were encountered but not yet explored.

**Introduction:**

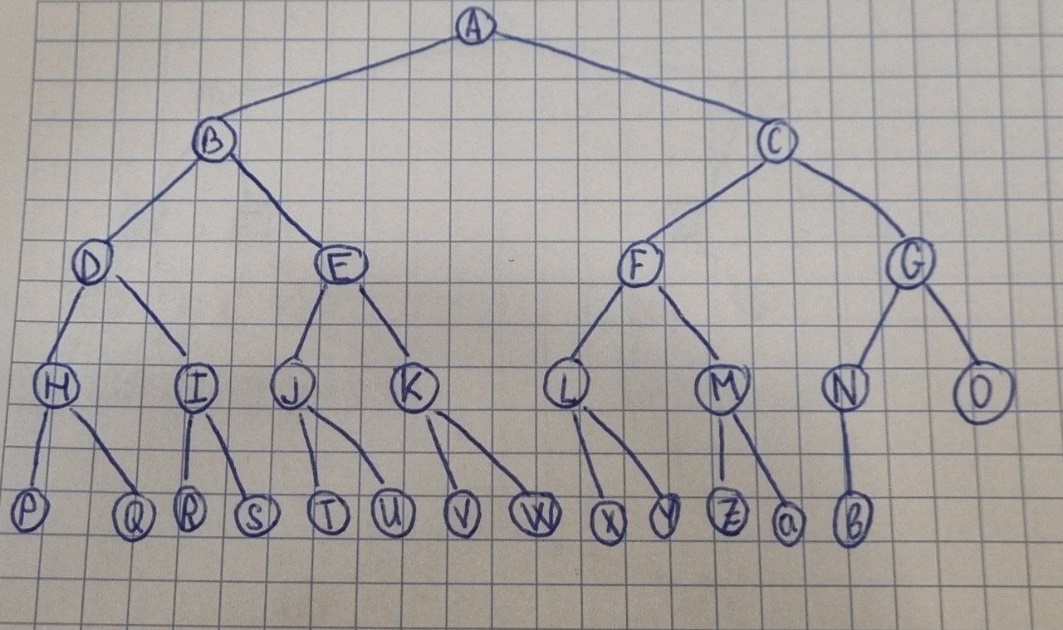
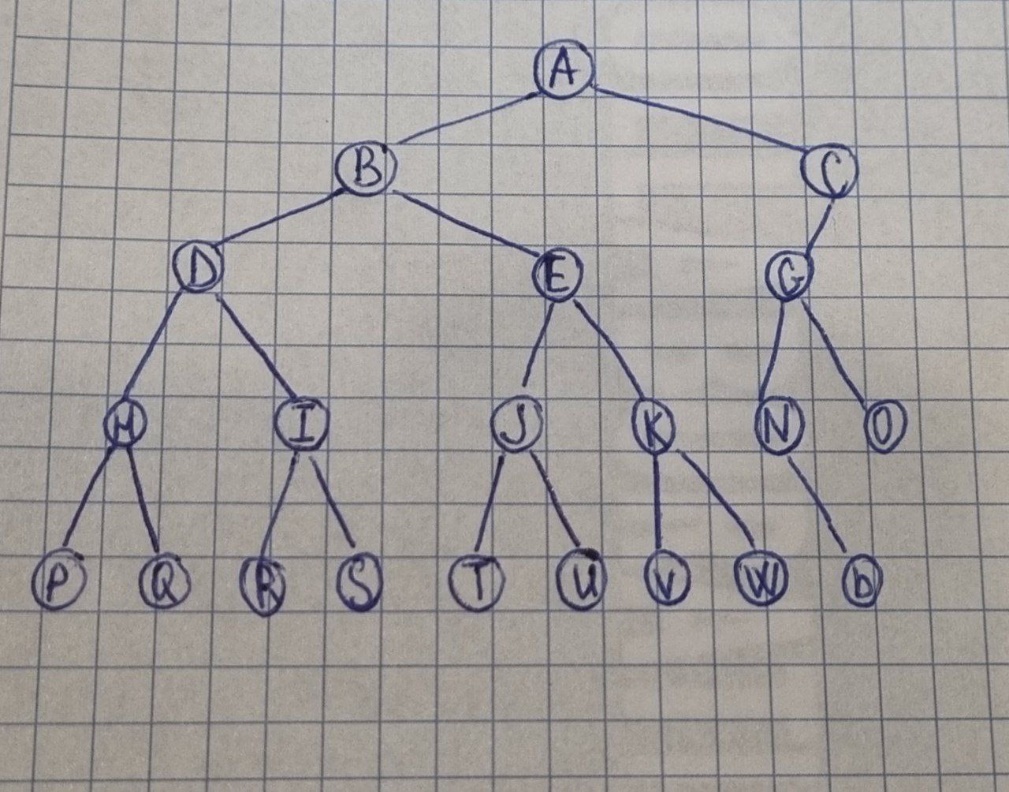
In this laboratory work I have to implement DFS and BFS algorithms and apply them on a balanced and an unbalanced tree. Also I need to analyse the outputs.

**Comparison metric:**

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

**Input format:**

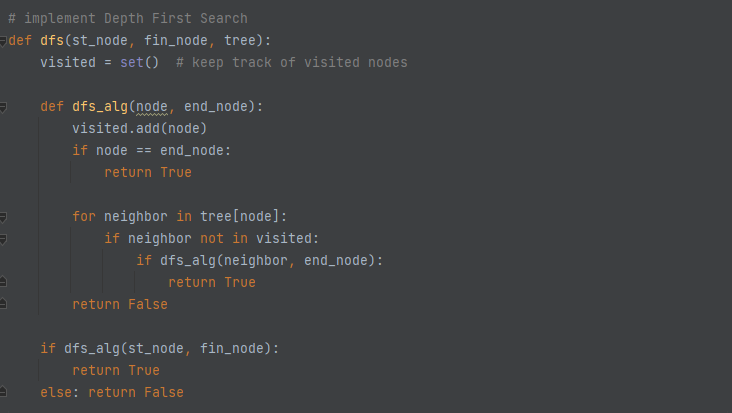
The input for this laboratory work are two trees, one balanced and the other unbalanced.

**Figure 1. The input unbalanced tree**

**Figure 2. The input balanced tree**

**IMPLEMENTATION**

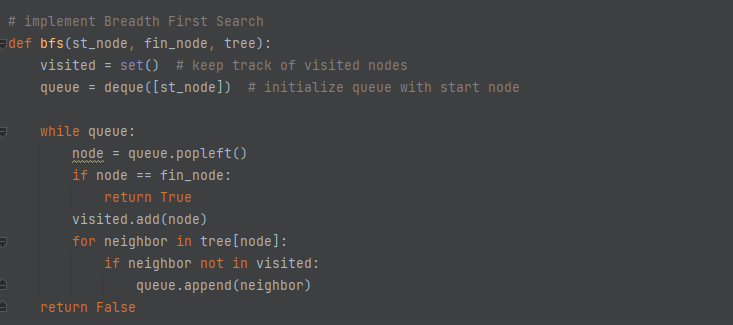
**Depth-First Search**

The function initializes a set **visited** to keep track of the visited nodes. It then defines a nested function **dfs\_alg** that takes a node and the finishing node as arguments and recursively searches for a path between the two nodes using DFS. The function marks the current node as visited, and if it reaches the finishing node, it returns **True**. If the function cannot find a path between the start node and the finishing node, it returns **False**. The function is called with the starting and finishing nodes and the tree as arguments. If a path is found, the function returns **True**; otherwise, it returns **False**.

**Figure 3. Implementation DFS**

**Breadth-First Search**

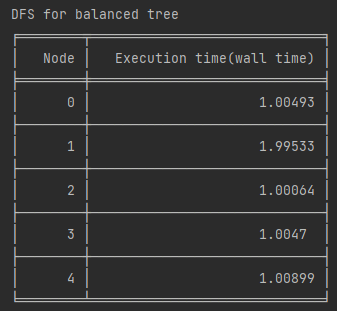
The function initializes a set **visited** to keep track of the visited nodes and a deque **queue** to store the nodes to be visited. The function initializes the queue with the starting node.

The function then enters a while loop that continues until the queue is empty. In each iteration of the loop, the function dequeues the node at the front of the queue and checks if it is the finishing node. If it is, the function returns **True**. If the node is not the finishing node, the function adds it to the **visited** set and iterates over its neighboring nodes. For each neighboring node that has not been visited, the function adds it to the back of the queue. If the function cannot find a path between the start node and the finishing node, it returns **False**. The function is called with the starting and finishing nodes and the tree as arguments. If a path is found, the function returns **True**; otherwise, it returns **False**.

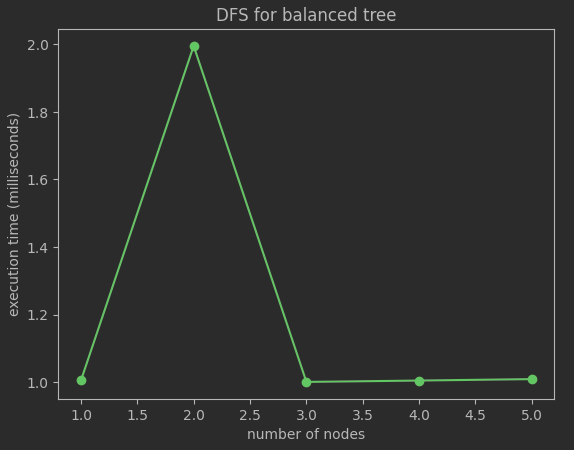
**Figure 4. Implementation BFS**

**Results**

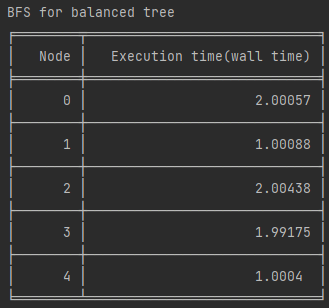
**5 nodes to find:** “D”, “K”, “L”, “T”, “a”



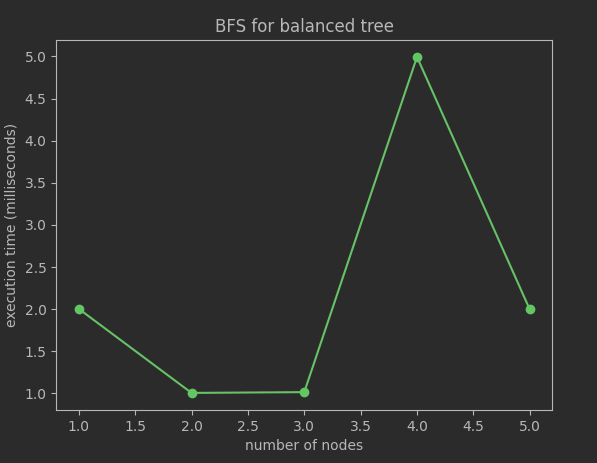
**Figure 4. Result DFS for balanced tree**



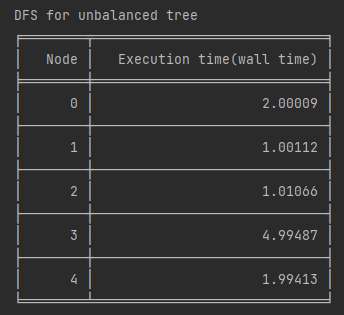
**Figure 5. Plot with results for DFS on a balanced tree**



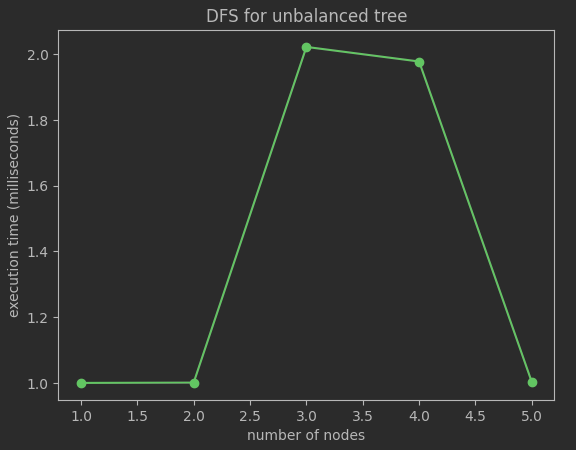
**Figure 6. Result BFS for balanced tree**



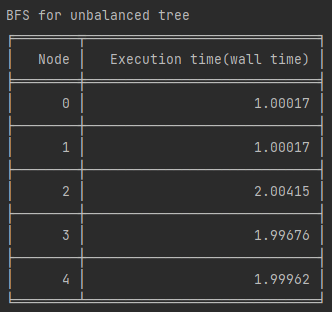
**Figure 7. Plot with results for BFS on a balanced tree**

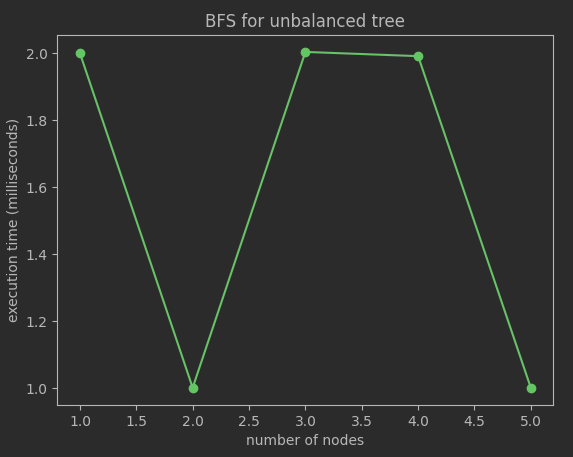


**Figure 8. Result DFS for unbalanced tree**

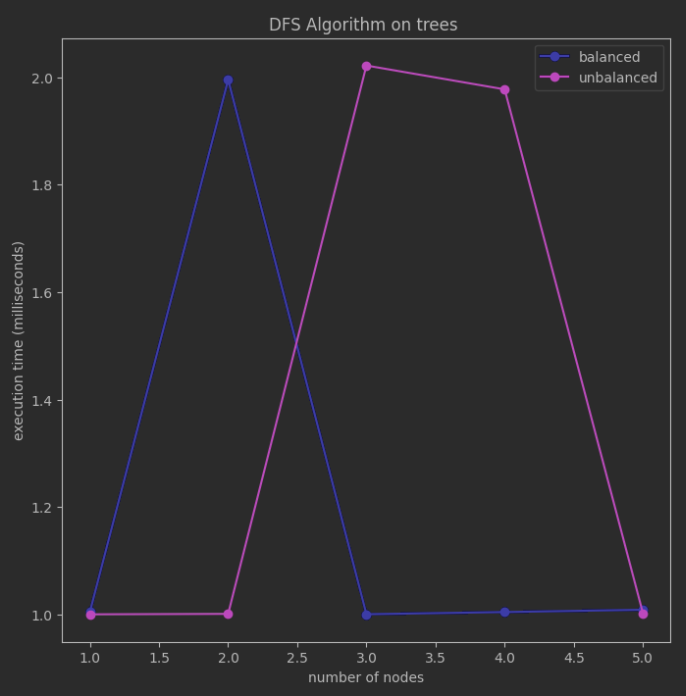


**Figure 9. Plot with results for DFS on an unbalanced tree**

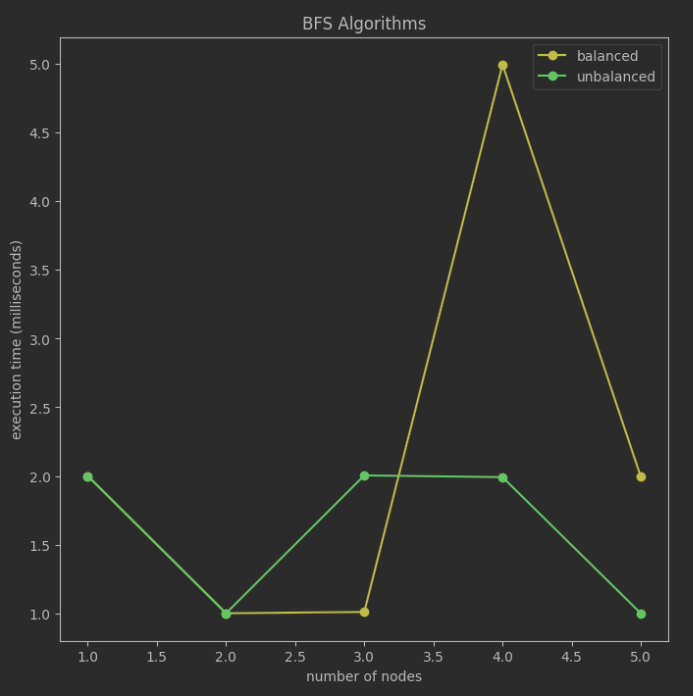
 **Figure 10. Result BFS for unbalanced tree**



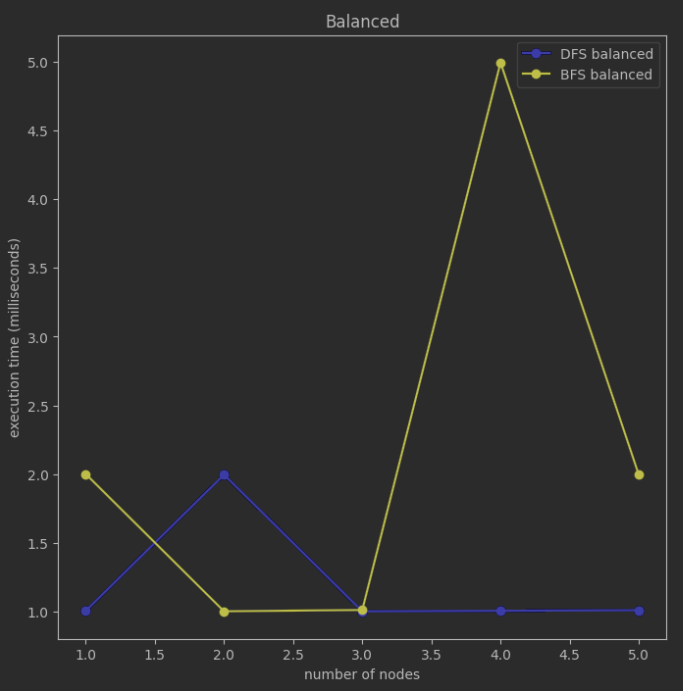
**Figure 11. Plot with results for BFS on an unbalanced tree**



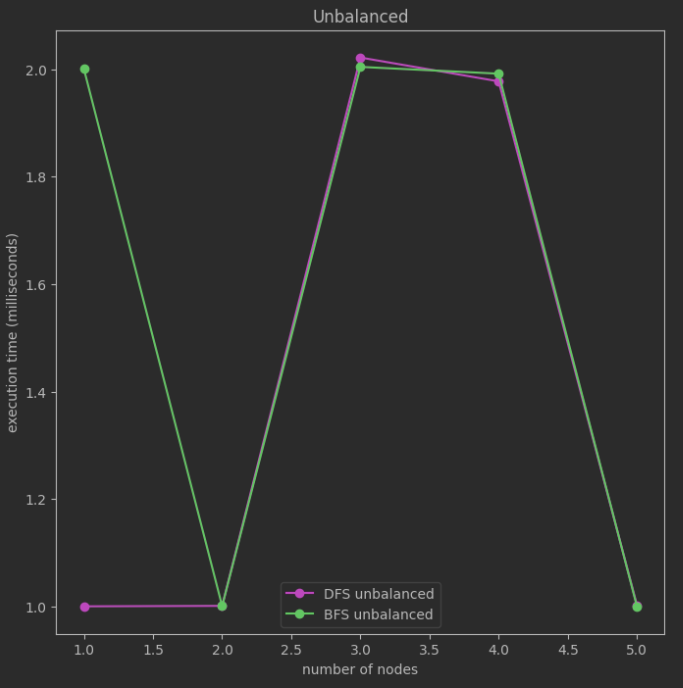
**Figure 12. Plot with results for DFS on a trees**



**Figure 12. Plot with results for BFS on a trees**



**Figure 13. Plot with results for BFS and DFS on a balanced tree**



**Figure 13. Plot with results for BFS and DFS on an unbalanced tree**

**Conclusion:**

For a balanced tree, both DFS and BFS have a time complexity of O(n) and a space complexity of O(h), where h is the height of the tree. However, since the tree is balanced, the height of the tree is log(n), where n is the number of nodes. Therefore, both algorithms have a time complexity of O(log(n)) and a space complexity of O(log(n)). In this case, either algorithm could be considered the best algorithm.

For an unbalanced tree, DFS has a worst-case time complexity of O(n) and a worst-case space complexity of O(h), where h is the height of the tree. In the worst case, where the tree is a straight line, the height of the tree is n, so the worst-case time complexity of DFS is O(n) and the worst-case space complexity is also O(n). On the other hand, BFS has a worst-case time complexity of O(n) and a worst-case space complexity of O(w), where w is the maximum width of the tree. In the worst case, where the tree is completely unbalanced and has a width of 1, the worst-case space complexity of BFS is O(n).

Therefore, for an unbalanced tree with a straight-line structure, DFS may be the best algorithm to use, as it has a lower worst-case space complexity than BFS. However, if the unbalanced tree has a wider structure, BFS may be the better algorithm to use, as it has a lower worst-case space complexity than DFS.

Overall, the choice of algorithm will depend on the specific properties of the tree and the goals of the analysis.

**Link to GitHub:** [**https://github.com/CristinaT21/APA\_LABS**](https://github.com/CristinaT21/APA_LABS)