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. 5

Discipline: Algorithms’ Analysis

Topic: Empirical analysis of algorithms: Dijkstra and Floyd–Warshall using dynamic programming

Elaborated: Tudor Sclifos,

student group FAF-211

Verified: Fiștic Cristofor,

university assistant

Chișinău – 2023

Algorithm analysis

**Objective:**

Implement algorithms Dijkstra and Floyd–Warshall using dynamic programming**.**

**Tasks:**

1 To study the dynamic programming method of designing algorithms.

2 To implement in a programming language algorithms Dijkstra and Floyd–Warshall using dynamic programming.

3 Do empirical analysis of these algorithms for a sparse graph and for a dense graph.

4 Increase the number of nodes in graphs and analyze how this influences the algorithms. Make a graphical presentation of the data obtained

5 To make a report.

**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:**

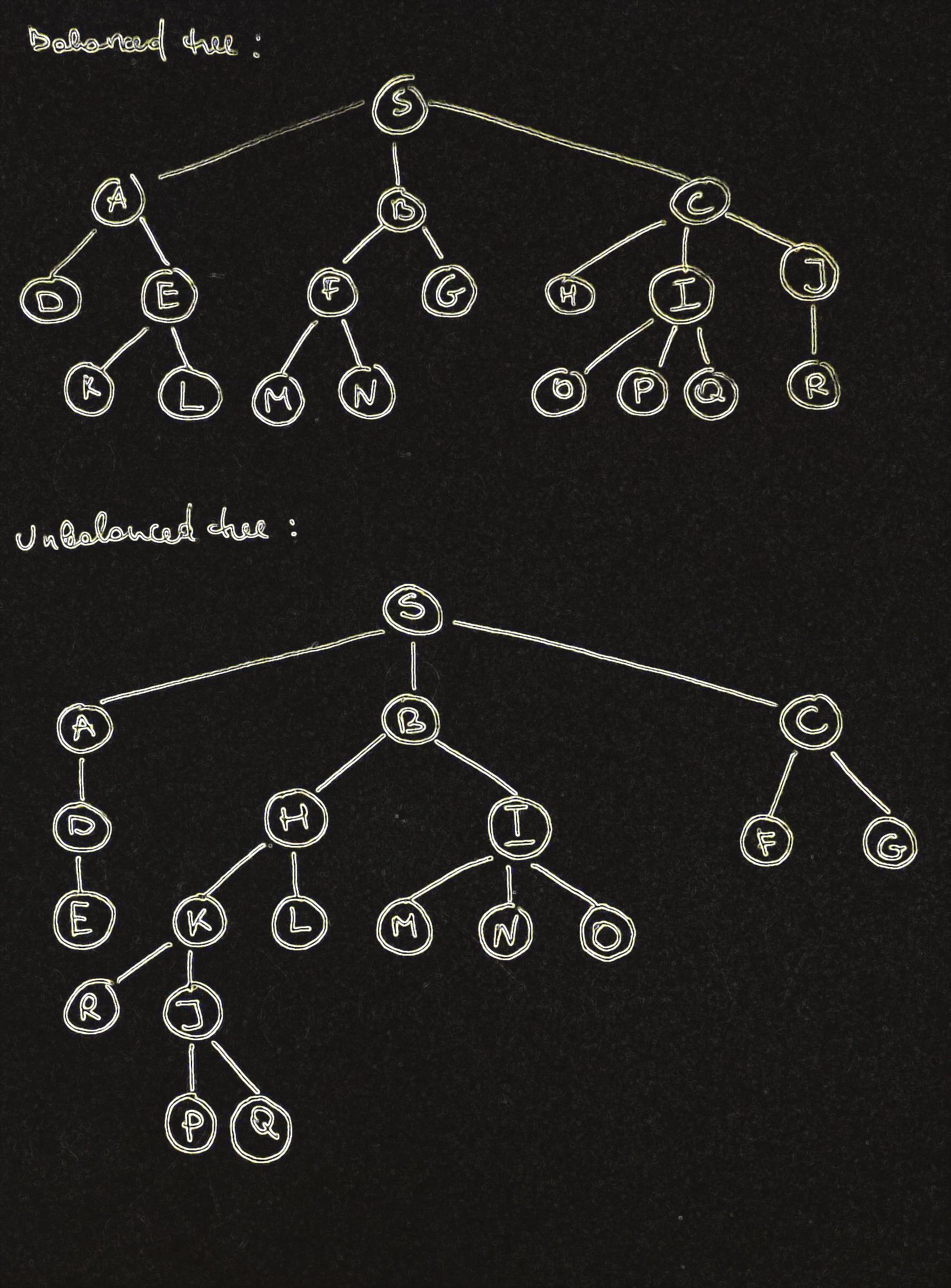
In this laboratory work I have to implement dfs and bfs algorithms and apply them on a balanced and an unbalanced tree. Also I need to analyse the outputs.

**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. The trees are shown on the next page.

 **Figure 1. The input trees**

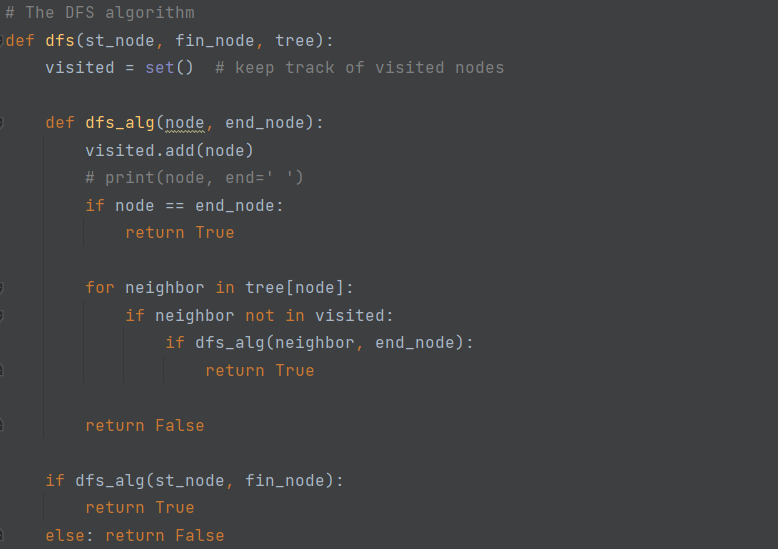
**IMPLEMENTATION**

**Depth-First Search**

**Implementation:**

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 2. Implementation DFS**

**Breadth-First Search**

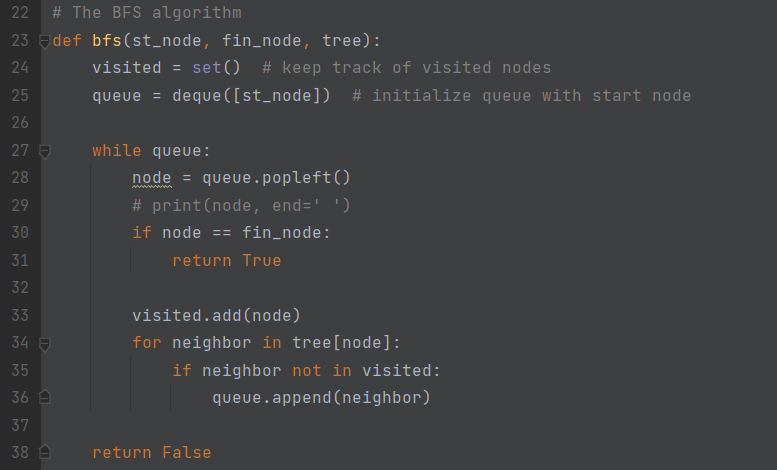
**Implementation:**

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**.

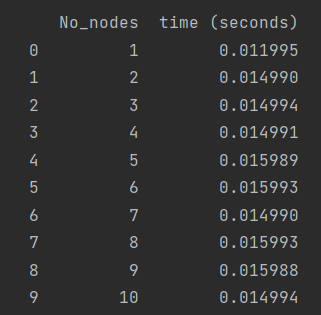


**Figure 3. Implementation BFS**

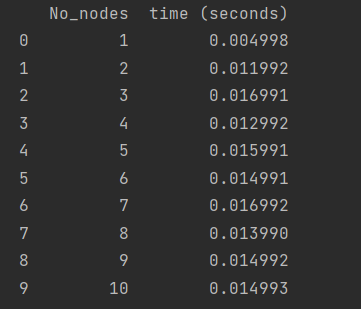
**Results for balanced tree (BT):**

**Nodes to find:**

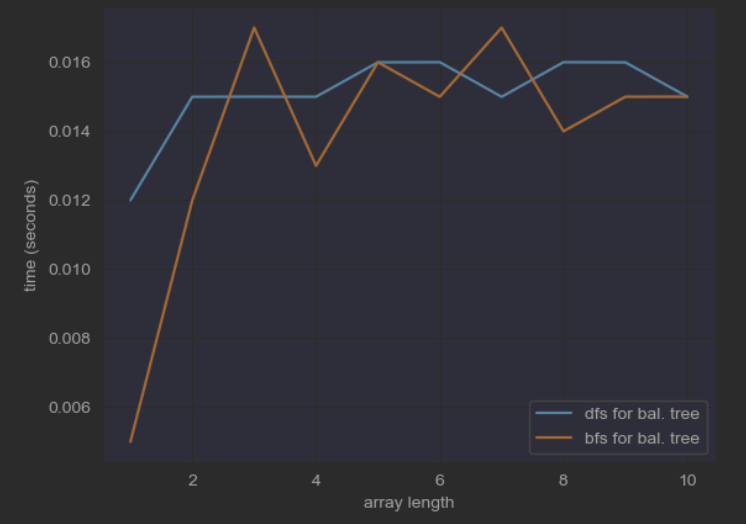




**Figure 4. Result DFS for BT**



**Figure 5. Result BFS for BT**

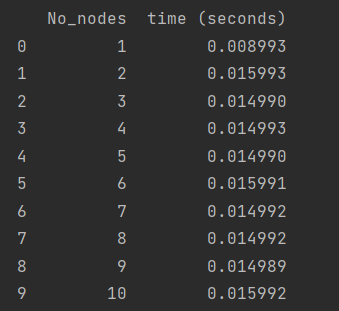


**Figure 6. Plot with results for BFS and DFS on a BT**

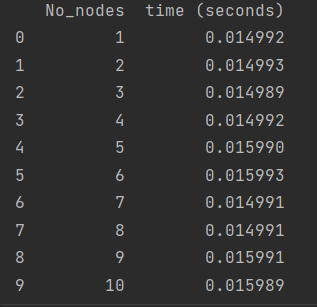
**Results for unbalanced tree (UT):**

**Nodes to find:**





**Figure 7. Result DFS for UT**



**Figure 8. Result BFS for UT**



**Figure 9. Plot with results for BFS and DFS on an UT**

**Conclusion:**

Both Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms have time complexity proportional to the number of nodes and edges in a tree. However, the time complexity of these algorithms can vary based on the structure of the tree.

In a balanced tree, the height of the tree is logarithmic in the number of nodes. In this case, both DFS and BFS have a time complexity of O(N log N), where N is the number of nodes in the tree. This is because each node will be visited only once, and there are log N levels in the tree.

In an unbalanced tree, the height of the tree can be linear in the number of nodes. In this case, DFS can have a time complexity of O(N) in the worst-case scenario, where the path to the finishing node is the longest possible path in the tree. In contrast, BFS can have a time complexity of O(N^2), where N is the number of nodes, in the worst-case scenario when the tree is a linked list. This is because in a linked list tree, each node has only one child, and every node is added to the queue in BFS, resulting in N iterations of the loop.

Therefore, in general, DFS is faster than BFS for unbalanced trees, while BFS is faster than DFS for balanced trees. However, the actual time complexity of these algorithms can vary depending on the specific structure of the tree and the location of the starting and finishing nodes.

**Link to GitHub:** <https://github.com/SexomQ/AlgorithmsAnalysis-labs>