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

REPORT

Laboratory work no.4
DFS and BFS algorithms

Elaborated:
st. gr. FAF-213 Gutu Dinu

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

ALGORITHM ANALYSIS

Objective

1. Analysis and study of the algorithms.
2. Empirical analysis of the aforementioned 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 is an alternative to mathematical analysis in evaluating algorithm complexity. It can be used to gain insight into the complexity class of an algorithm, compare the efficiency of different algorithms for solving similar problems, evaluate the performance of different implementations of the same algorithm, or examine the efficiency of an algorithm on a specific computer. The typical steps in empirical analysis include defining the purpose of the analysis, choosing a metric for efficiency such as number of operations or execution time, determining the properties of the input data, implementing the algorithm in a programming language, generating test data, running the program on the test data, and analyzing the results. The choice of efficiency metric depends on the purpose of the analysis, with number of operations being appropriate for complexity class evaluation and execution time being more relevant for implementation performance. After the program is run, the results are recorded and synthesized through calculations of statistics or by plotting a graph of problem size against efficiency measure.

Introduction

DFS (Depth-First Search) and BFS (Breadth-First Search) are two popular algorithms used for traversing or searching a graph or a tree data structure. They both explore the nodes of a graph in different orders, which can be useful for solving different types of problems.

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm begins at the root node (choosing an arbitrary node to serve as the root node in the case of a graph) and proceeds to investigate each branch as far as it can go before turning around. To aid in graph backtracking, additional memory, typically a stack, is required to keep track of the nodes found thus far along a particular branch. Depending on the application area, DFS uses a different time and space analysis. DFS is frequently used to traverse an entire graph in theoretical computer science, and it does so in O (| V | + | E |), where |V| is the number of vertices and |E| is the number of edges. The graph's size is linear in this

case. The stack of vertices on the current search path as well as the collection of previously visited vertices are stored in these applications using space O(|V|) in the worst case.

Breadth-first search (BFS) on the other hand is an algorithm for searching a tree data structure for a node that satisfies a given property. Before moving on to the nodes at the next depth level, it begins at the root of the tree and investigates every node there. To maintain track of the child nodes that were discovered but haven't been fully investigated, more memory, typically a queue, is required.

In contrast, the (simple) depth-first search method may get stuck in an endless branch and never reach the solution node since it explores the node branch as far as it can before going back and extending other nodes. The latter disadvantage is avoided by iteratively deepening depth-first search, albeit at the cost of repeatedly examining the top of the tree. Yet, neither depth-first technique requires additional RAM to function.

Comparison Metric

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

Input Format

As input, the algorithms will receive two binary trees which they will have to search for 5 nodes resulting in a total of 10 nodes, one tree will be balanced while the other unbalanced both having a total of 15 nodes ranging from 1 to 150.

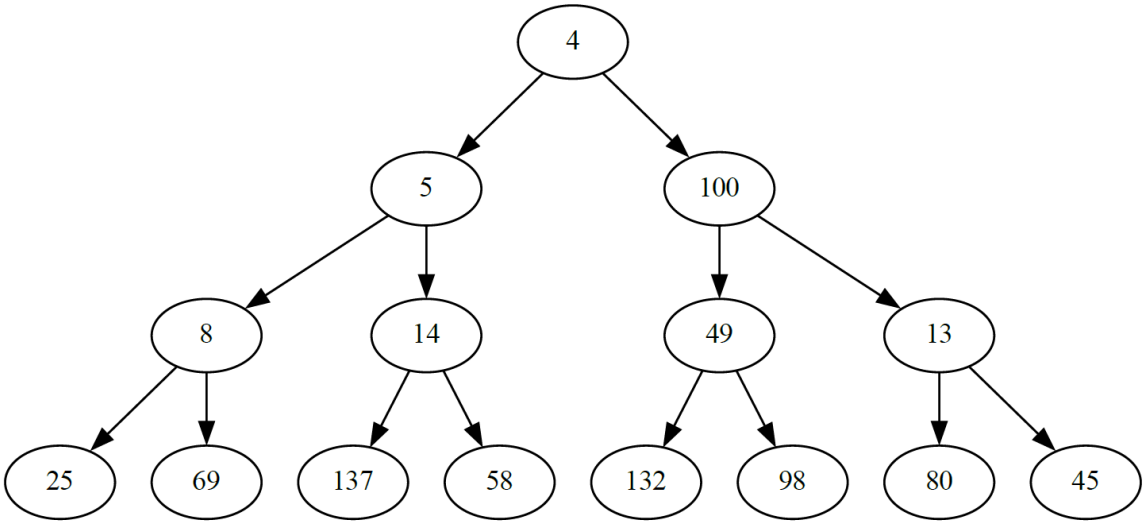

Figure 1. Balanced tree

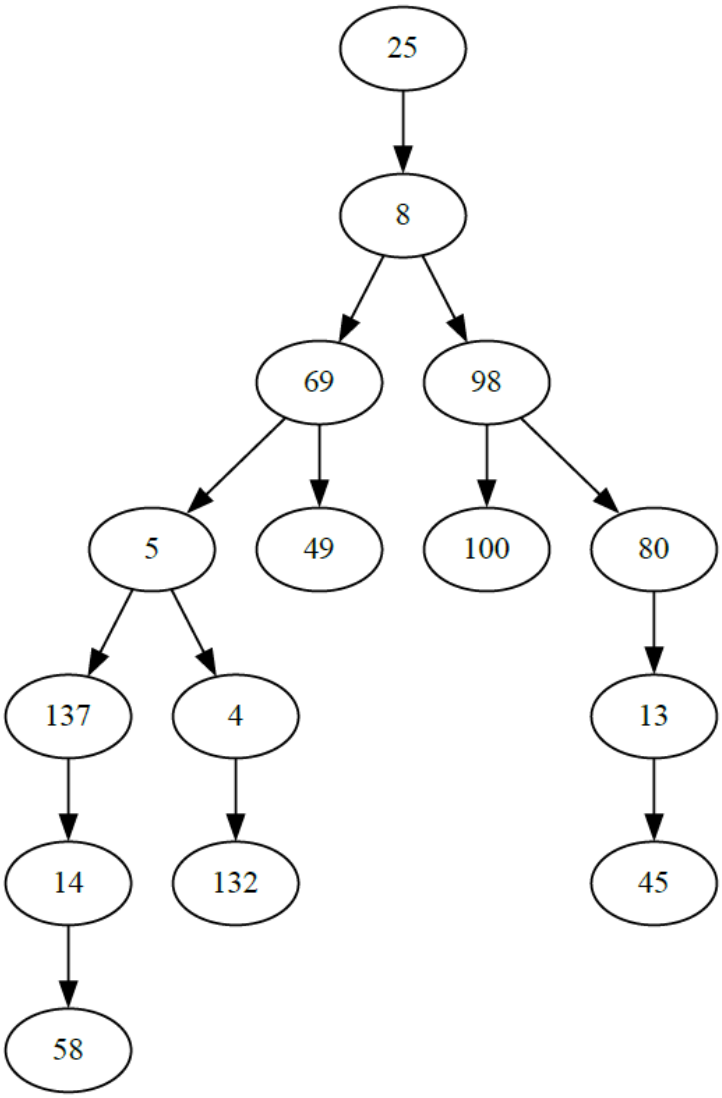

Figure 2. Unbalanced tree

IMPLEMENTATION

Depth-first search

Depth First Search (DFS) is a popular algorithm used for traversing or searching a graph or a tree data structure. It works by exploring as far as possible along each branch before backtracking. It begins at the root node (or any arbitrary node) and then visits the adjacent node of the current node. It repeats this process until it reaches the leaf node. When it reaches the leaf node, it backtracks to the previous node and starts exploring the next path.

Algorithm Description:

Pseudocode for one of the implementations of DFS:

```
procedure DFS_iterative(G, v) is
    let S be a stack
    S.push(v)
    while S is not empty do
        v = S.pop()
        if v is not labeled as discovered then
            label v as discovered
            for all edges from v to w in G.adjacentEdges(v) do
                S.push(w)
```

Implementation

```
def dfs(root, value):
    if root is None:
        return []
    stack = [root]
    visited = []
    while stack:
        node = stack.pop()
        visited.append(node.data)
        if node.data == value:
            return visited
        if node.right:
            stack.append(node.right)
        if node.left:
            stack.append(node.left)
    return None
```

Figure 3. DFS algorithm

Results

Node elements	4	58	13	45	49
DFS Balanced	4.00000000012e-07	2.99999999953e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07

Node elements	4	58	13	45	49
DFS Unbalanced	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07

Figure 4. DFS results cases

The time complexity of for our DFS implementation is O(N), where N is the number of nodes in the tree. The algorithm visits each node in the tree once, and for each node, it performs a constant amount of work (i.e., pushing and popping from the stack, appending to the visited list, and checking if the node matches the target value). Therefore, the total amount of work performed by the algorithm is proportional to the number of nodes in the tree. In the worst-case scenario, where the target value is not present in the tree, the algorithm will visit every node, resulting in a time complexity of O(N). However, in the best-case scenario, where the target value is found in the root node, the algorithm will only visit one node, resulting in a time complexity of O(1).

Nr of nodes	1	2	3	4	5
DFS Balanced	4.00000000012e-07	2.00000000006e-07	2.3333333332166667e-07	2.2499999999275e-07	2.199999999954e-07

Nr of nodes	1	2	3	4	5
DFS Unbalanced	2.00000000006e-07	1.00000000003e-07	1.3333333333733334e-07	1.5000000000450001e-07	1.600000000048e-07

Figure 5. DFS results per number of nodes

In this figure the numbers from the previous results were taken and averaged over the number of nodes searched. Thus, creating a better estimate for the algorithm's performance. Below are the graphs that represent these results one for each type of tree.

Graphs

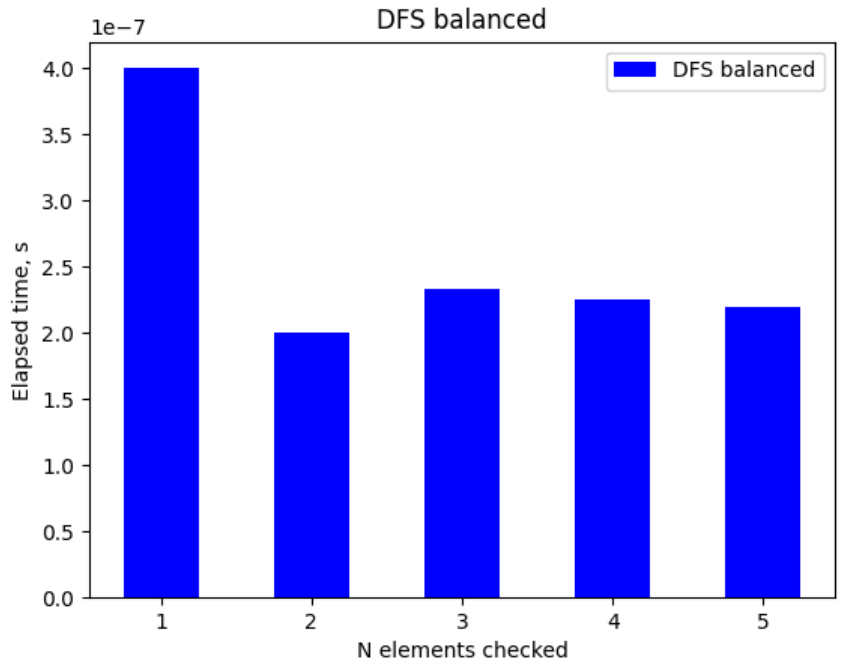

Figure 6. DFS balanced graph

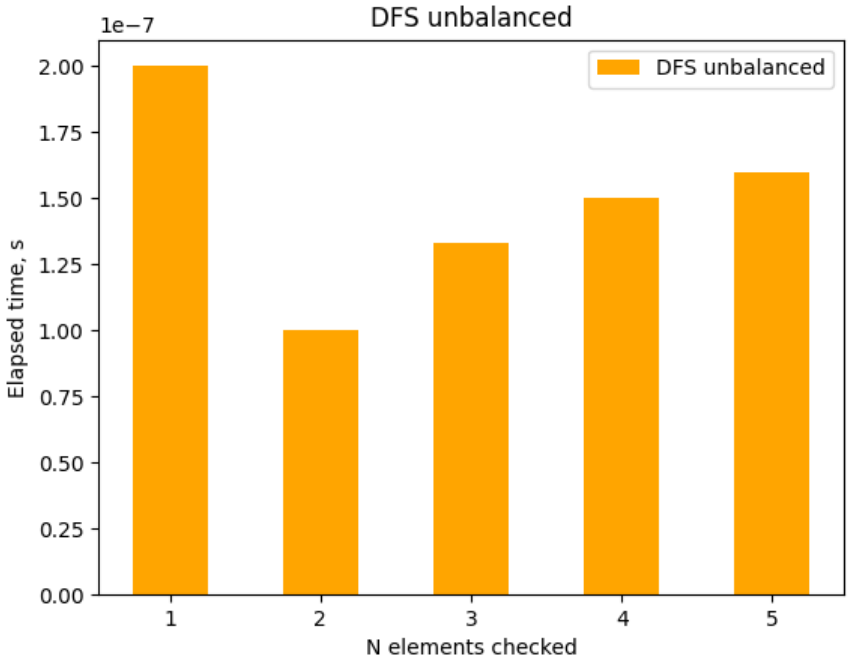

Figure 7. DFS unbalanced graph

Breadth-first search

BFS (Breadth-First Search) is another popular algorithm used for traversing or searching a graph or a tree data structure. It explores all the nodes of a graph or tree at a given depth (level) before moving on to the next level. It uses a queue data structure to keep track of the nodes that need to be explored. It enqueues the adjacent nodes of the current node and dequeues the next node from the front of the queue. This ensures that nodes are

visited in the order they were added to the queue, which guarantees that the shortest path is found when searching for a path between two nodes.

Algorithm Description:

Pseudocode for one of the implementations of BFS:

```
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

```python
def bfs(root, value):
    if root is None:
        return []
    queue = [root]
    visited = []
    while queue:
        node = queue.pop(0)
        visited.append(node.data)
        if node.data == value:
            return visited
        if node.left:
            queue.append(node.left)
        if node.right:
            queue.append(node.right)
    return None
```

Figure 8. BFS algorithm

Results

Node elements	4	58	13	45	49
BFS Balanced	1.000000000029e-06	6.00000000017e-07	3.00000000064e-07	2.00000000006e-07	2.99999999953e-07

Node elements	4	58	13	45	49
BFS Unbalanced	3.00000000064e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07

Figure 9. BFS results cases

The time complexity of this BFS implementation, like in the previous case is also O(N), where N is the number of nodes in the tree. The algorithm performs a constant amount of work for each node, visiting each node in the tree only once (i.e., dequeuing from the queue, appending to the visited list, and checking if the node matches the target value). As a result, the algorithm's overall workload is proportional to the number of nodes in the tree. In the worst-case scenario, where the target value is not present in the tree, the algorithm will visit every node, resulting in a time complexity of O(N). However, in the best-case scenario, where the target value is found in the root node, the algorithm will only visit one node, resulting in a time complexity of O(1). It should also be noted that the space complexity of this BFS algorithm is also proportional to the maximum number of nodes at a given level of the tree, which can be up to N/2 in the worst case scenario (i.e., a perfectly balanced tree). Therefore, the space complexity of this algorithm is also O(N).

Nr of nodes	1	2	3	4	5
BFS Balanced	1.000000000029e-06	5.000000000145e-07	5.333333333486668e-07	4.7500000002750004e-07	4.200000000232e-07

Nr of nodes	1	2	3	4	5
BFS Unbalanced	3.00000000064e-07	1.50000000032e-07	1.6666666668999997e-07	1.7500000001899997e-07	1.8000000001639996e-07

Figure 10. BFS results per number of nodes

In this figure the numbers from the previous results were taken and averaged over the number of nodes searched. In turn creating a better estimate for this algorithm’s performance. Below are the graphs that represent these results one for each type of tree.

Graphs

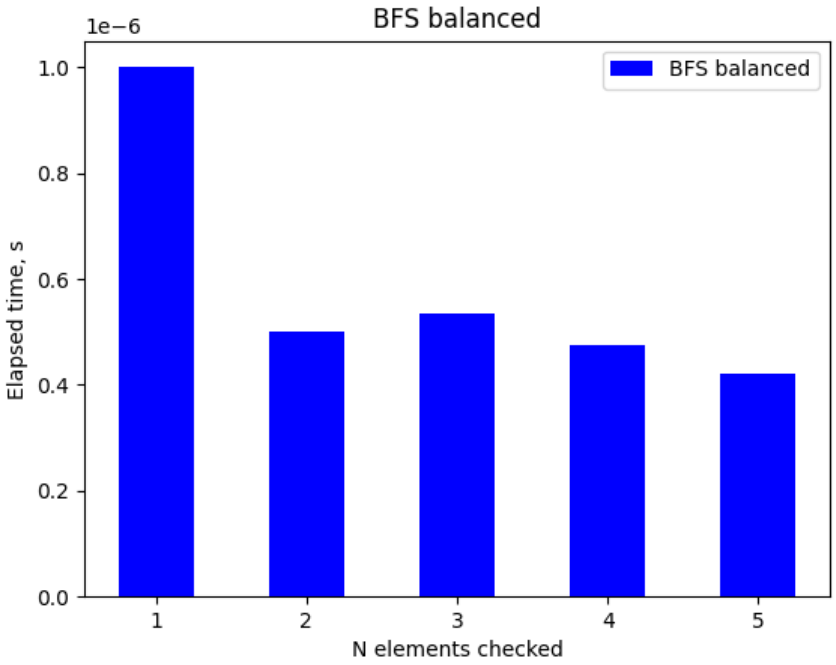

Figure 11. BFS balanced graph

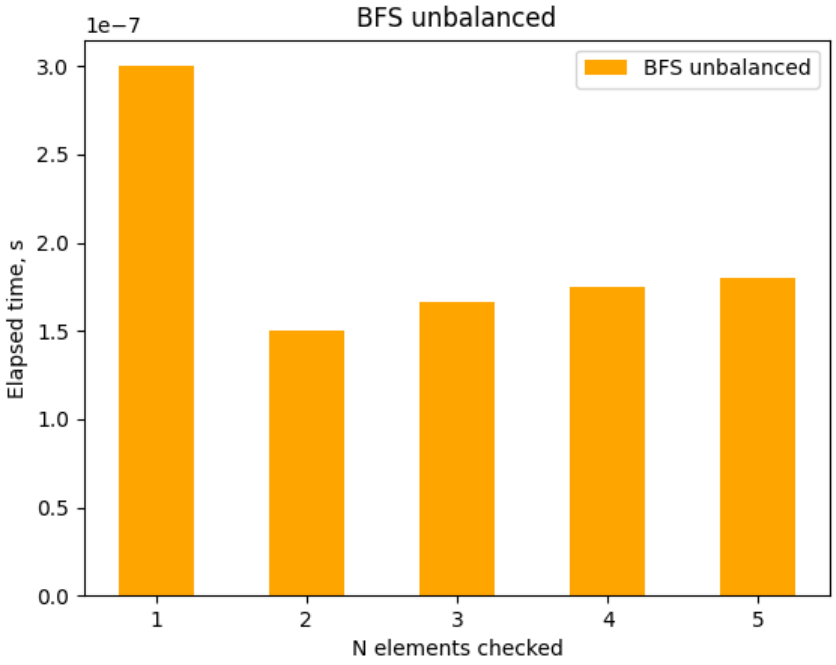

Figure 12. BFS unbalanced graph

All Algorithms

All results

Node elements	4	58	13	45	49
BFS Balanced	1.000000000029e-06	6.00000000017e-07	3.00000000064e-07	2.00000000006e-07	2.99999999953e-07
BFS Unbalanced	3.00000000064e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07
DFS Balanced	4.00000000012e-07	2.99999999953e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07
DFS Unbalanced	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07	2.00000000006e-07

Figure 13. All algortihms' results cases

Nr of nodes	1	2	3	4	5
BFS Balanced	1.000000000029e-06	5.000000000145e-07	5.333333333486668e-07	4.7500000002750004e-07	4.200000000232e-07
BFS Unbalanced	3.00000000064e-07	1.50000000032e-07	1.6666666668999997e-07	1.7500000001899997e-07	1.8000000001639996e-07
DFS Balanced	4.00000000012e-07	2.00000000006e-07	2.3333333332166667e-07	2.2499999999275e-07	2.199999999954e-07
DFS Unbalanced	2.00000000006e-07	1.00000000003e-07	1.3333333333733334e-07	1.5000000000450001e-07	1.600000000048e-07

Figure 14. All algortihms' results per number of nodes

All graphs

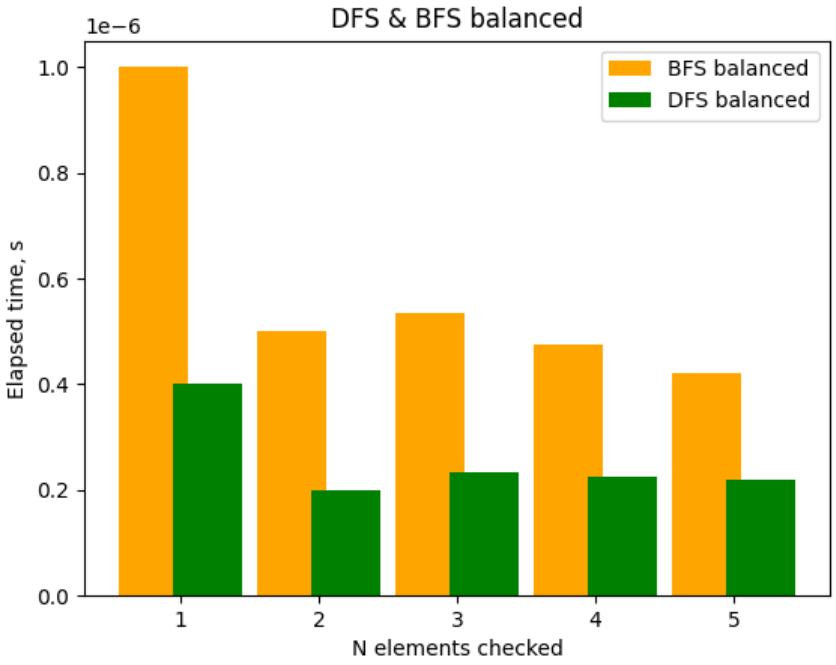

Figure 15. DFS and BFS search on a balanced tree

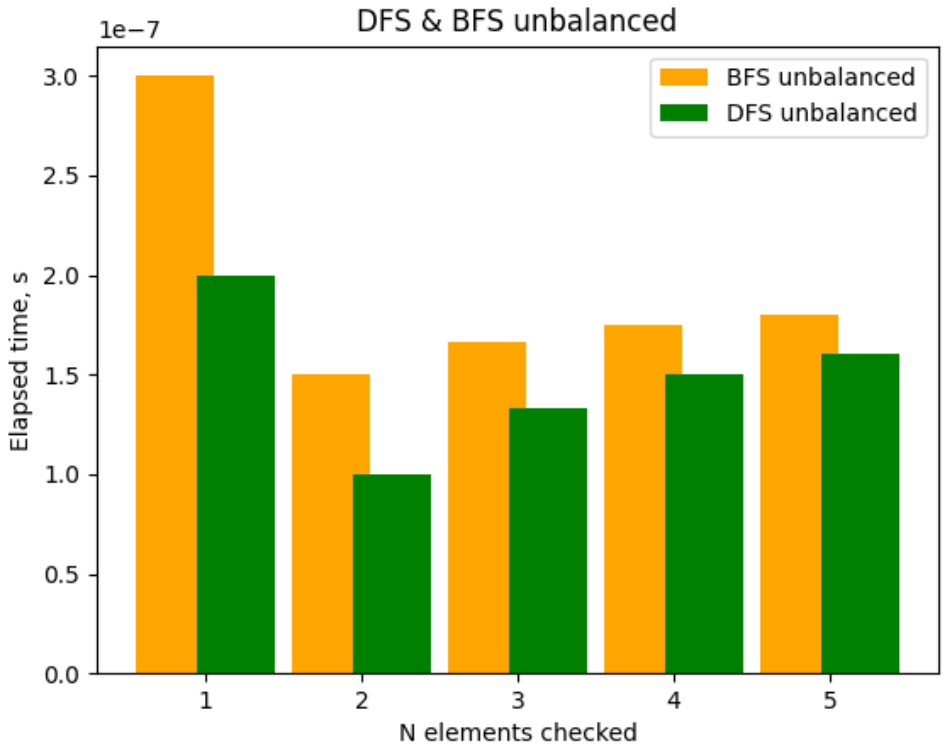

Figure 16. DFS and BFS search on an unbalanced tree

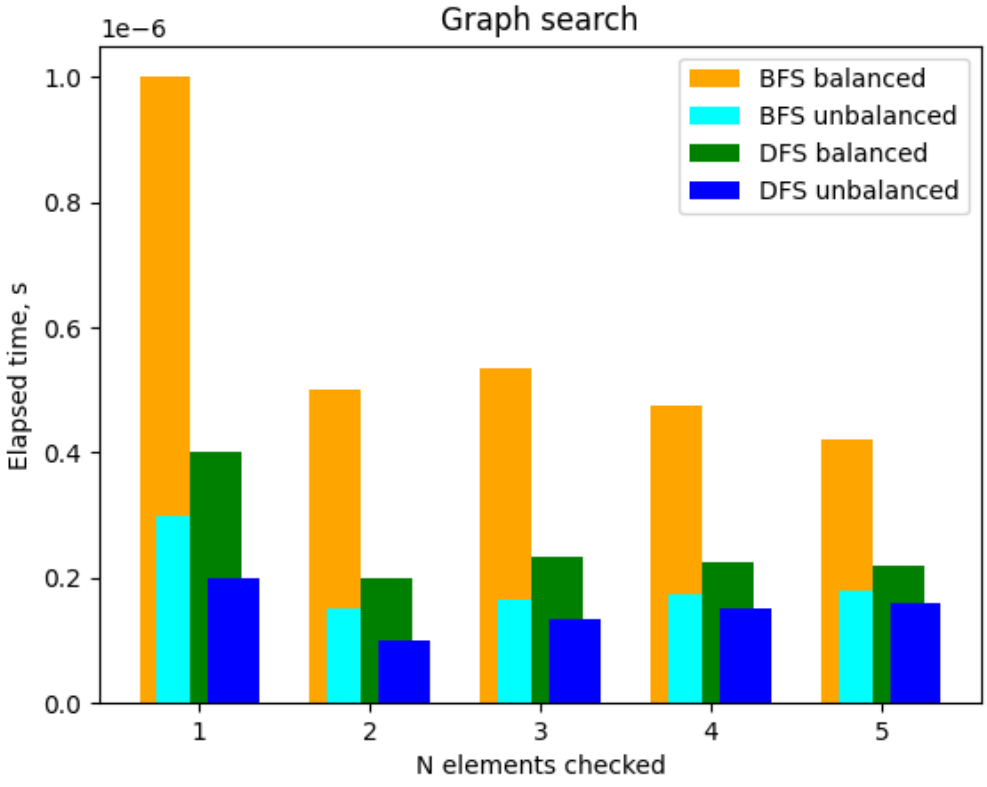

Figure 17. All algortihms' graphs

CONCLUSION

DFS (Depth-First Search) and BFS (Breadth-First Search) are two most common algorithms used for traversing or searching a graph or a tree data structure. Although BFS traverses all the nodes at a certain depth (level) before continuing on to the next level, DFS explores a graph or tree by traveling as far as possible down each branch before turning around.

When it comes to performance, DFS has a space complexity of O(h), where h is the height of the tree, and a time complexity of O(n), where n is the number of nodes in the tree. BFS, on the other hand, has a space complexity of O(w), where w is the maximum width of the tree, and a time complexity of O(n).

In terms of binary trees, both DFS and BFS can be used to traverse them. DFS can be implemented using either the pre-order, in-order, or post-order traversal techniques. Pre-order traversal visits the root node first, then the left subtree, and finally the right subtree. In-order traversal visits the left subtree first, then the root node, and finally the right subtree. Post-order traversal visits the left subtree first, then the right subtree, and finally the root node.

BFS, on the other hand, goes level by level through a binary tree. The process visits each node one at a time, beginning with the root node and continuing with the left and right child nodes at the following level.

Although in my case I achieved the same outcome of traversing a tree using both DFS and BFS the methods used were the basic implementation ones extrapolated a bit to suit my specific problem.

In conclusion, DFS and BFS are both robust algorithms that can be used to address a variety of graph and tree-related issues. The specifics of the situation at hand and the characteristics of the graph or tree being traversed determine which algorithm is most suitable.

BIBLIOGRAPHY

Github: https://github.com/Grena30/APA_Labs/tree/main/Lab4