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 Iațco Sorin

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

ALGORITHM ANALYSIS

Objective

Empirical analysis of algorithms for specified 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 alternate approach to analyzing complexity is to use empirical analysis, which involves implementing the algorithm in a programming language and running it with multiple sets of input data to obtain data on its efficiency. This method is useful for various purposes, such as gaining preliminary information on an algorithm's complexity class, comparing the efficiency of different algorithms or implementations, or assessing the performance of an algorithm on a specific computer.

To perform empirical analysis, the purpose of the analysis is established, and an efficiency metric is chosen, such as the number of executions of an operation or execution time. The properties of the input data are also established, and the algorithm is implemented in a programming language. After generating multiple sets of input data, the program is run for each set, and the obtained data is analyzed.

The choice of efficiency measure depends on the purpose of the analysis. If the goal is to obtain information on the complexity class or check the accuracy of a theoretical estimate, then the number of operations performed is appropriate. However, if the aim is to assess the behavior of the implementation of an algorithm, then execution time is more suitable. The results are recorded and either synthetic quantities, such as mean and standard deviation, are calculated, or a graph with appropriate pairs of points is plotted to analyze the data.

Introduction

DFS (Depth First Search) and BFS (Breadth First Search) are two popular graph traversal algorithms used to explore and search through a graph data structure.

DFS explores the graph by visiting one of the neighbors of the current node, and then recursively exploring that neighbor's neighbors until it reaches a dead end. At that point, it backtracks to the previous node and explores a different neighbor. DFS can be implemented using a stack, which stores the nodes to be visited in a last-in-first-out (LIFO) order.

BFS, on the other hand, explores the graph by visiting all the neighbors of the current node before moving on to the neighbors of those neighbors. It uses a queue to store the nodes

to be visited in a first-in-first-out (FIFO) order.

DFS is often used to find paths or cycles in a graph, while BFS is often used to find the shortest path between two nodes in an unweighted graph. In general, BFS is more suitable for finding shortest paths and DFS is more suitable for finding paths in general.

Both DFS and BFS have a time complexity of O(V+E), where V is the number of vertices and E is the number of edges in the graph. However, the space complexity can be different, as DFS uses a stack and may require more space for recursion, while BFS uses a queue and may require more space for maintaining a list of visited nodes.

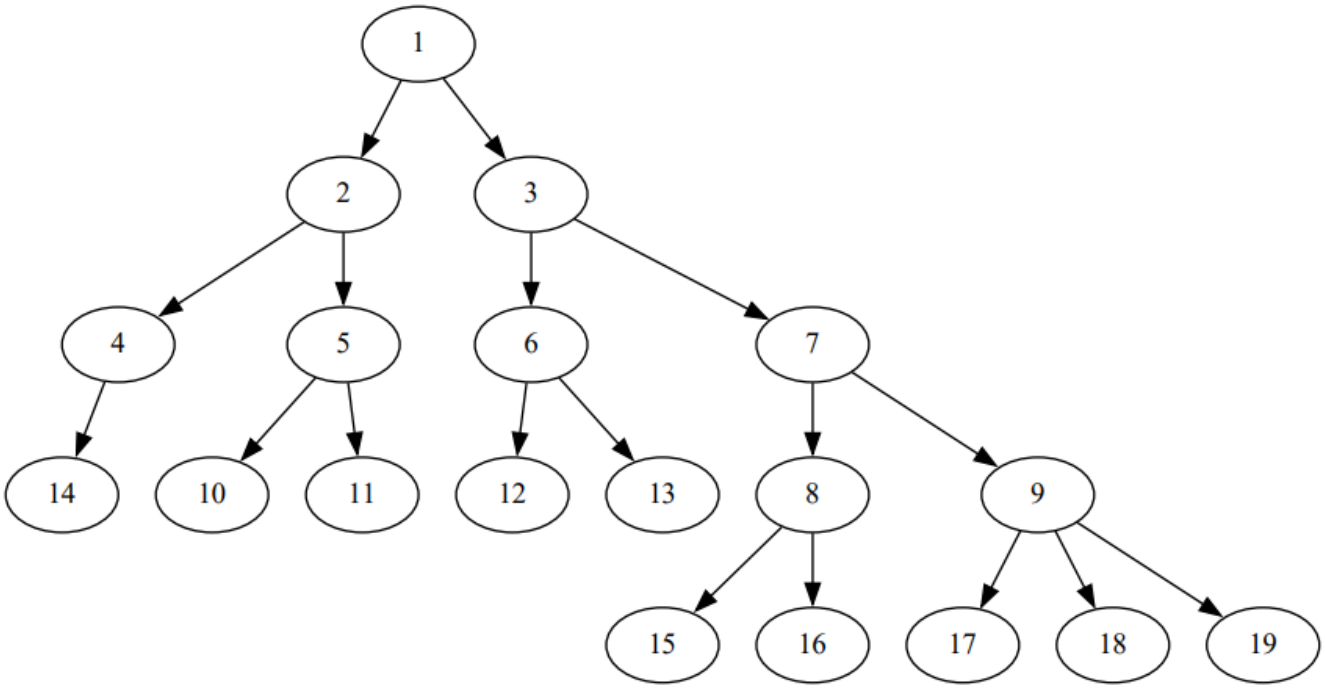
Overall, DFS and BFS are both important algorithms in graph theory and have many practical applications in fields such as computer networking, social networks, and recommendation systems.

Comparison Metric

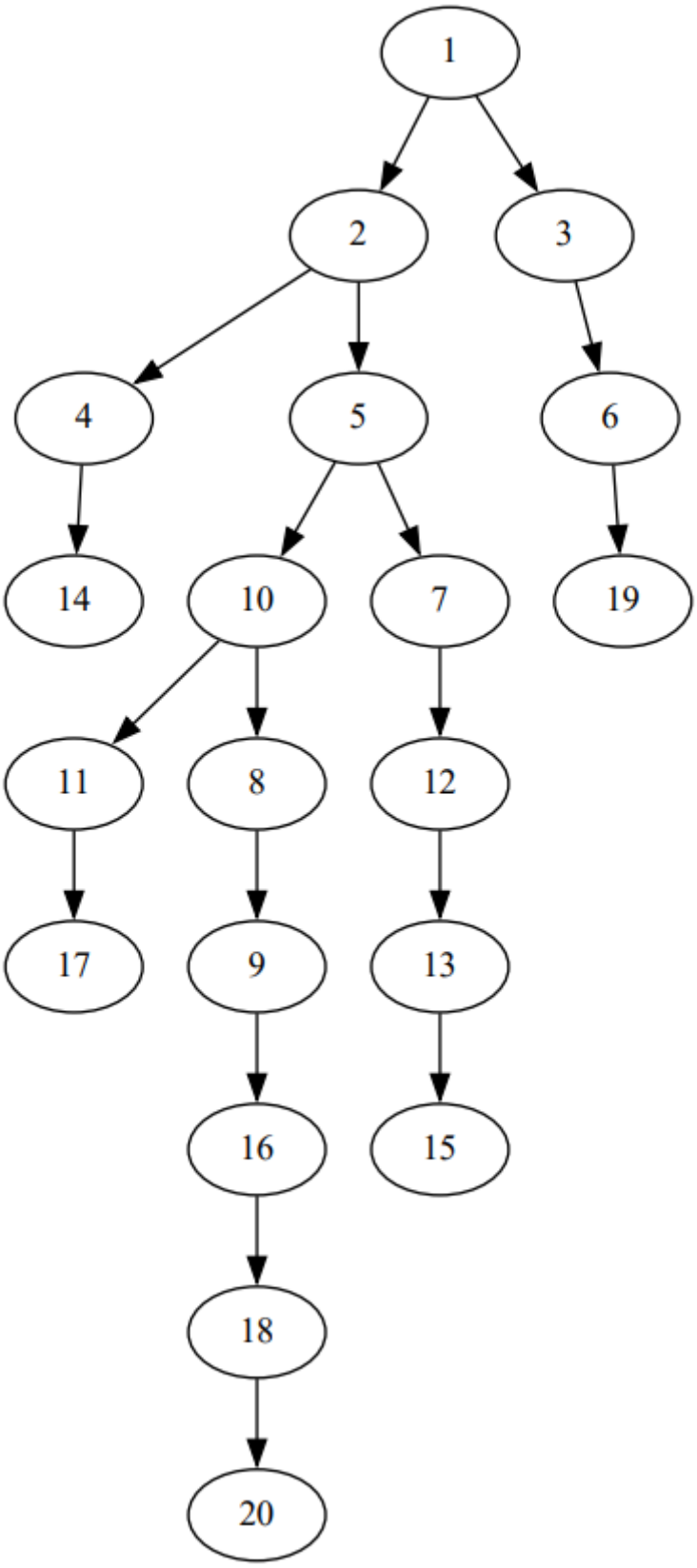
The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)).

Input Format

As input, we will give two binary trees, one balanced and another unbalanced. We will search for 5 nodes for each tree, each tree will have a total of 20 nodes, values variating from 1 to 20.



Balanced Tree



Unbalanced Tree

IMPLEMENTATION

Both algorithms will be implemented in their naive form in python an 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 rapport with input will vary depending on performance of the device used.

Depth-First Search

Pseudocode:

```
DFS(G, v):
  while stack is not empty:
    node = stack.pop()

    if node not in visited:
      visited.add(node)

      for neighbor in G.adjacent_nodes(node):
        if neighbor not in visited:
          stack.push(neighbor)
```

Implementation:

```python
def dfs(node, target, visited=None, path=None):
    if visited is None:
        visited = set()
    if path is None:
        path = []

    if node.value == target:
        return path + [node]

    visited.add(node)
    for child in node.children:
        if child not in visited:
            result = dfs(child, target, visited, path + [node])
            if result is not None:
                return result

    return None
```

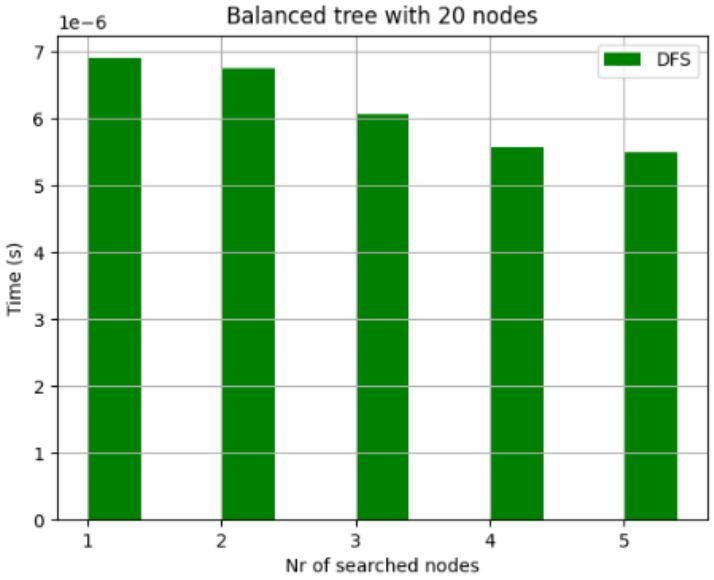
The time complexity of this DFS algorithm is **O(V+E)**, where **V** is the number of vertices and **E** is the number of edges in the graph. In each call to **dfs**, we do a constant amount of work to check if the current node matches the target, add the node to the **visited** set, and iterate through its children. The for loop iterates over all children of the current node, so in the worst case scenario, where every node is connected to every other node, the loop executes **E** times. However, we also add nodes to the **visited** set and append them to the **path** list, which take constant time. Since each node is visited and added to the **path** list at most once, these operations take **O(V)** time. In the worst case scenario the complexity will be **O(n)**, and on the best case scenario it will be **O(1)**. Therefore, the overall time complexity of the algorithm is **O(V+E)**. This is the same as the time complexity of traversing the entire graph, since we need to visit each vertex and edge at least once.

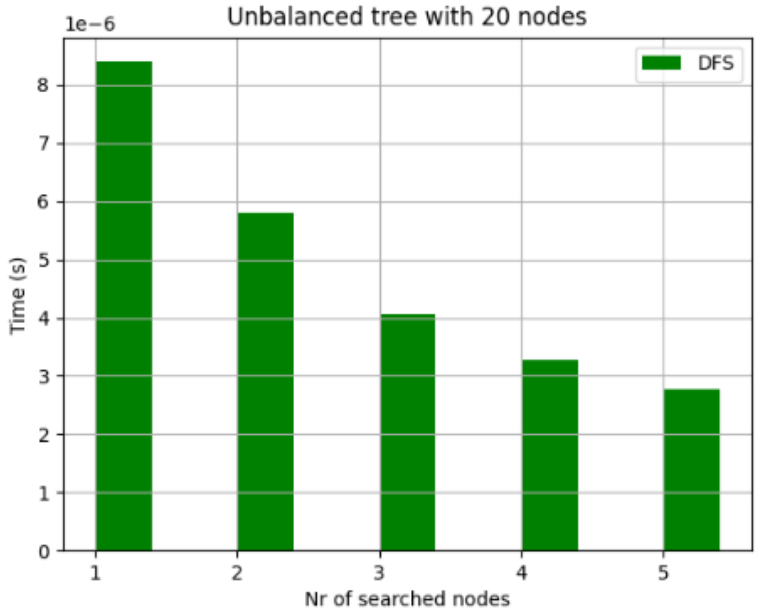
Results:

Nr of Nodes searched	1	2	3	4	5
Balanced DFS	6.900001608300954e-06	6.750000466126949e-06	6.066666780194889e-06	5.5750006140442565e-06	5.499999679159373e-06

Nr of Nodes searched	1	2	3	4	5
Unbalanced DFS	8.39999847812578e-06	5.800000508315861e-06	4.066666103123377e-06	3.2749994716141373e-06	2.7799993404187263e-06

Graphs:





Breadth-First Search

Pseudocode:

BFS(G, v):
  while queue is not empty:
    node = queue.dequeue()

    if node not in visited:
      visited.add(node)

      for neighbor in G.adjacent_nodes(node):
        if neighbor not in visited:
          queue.enqueue(neighbor)

Implementation:

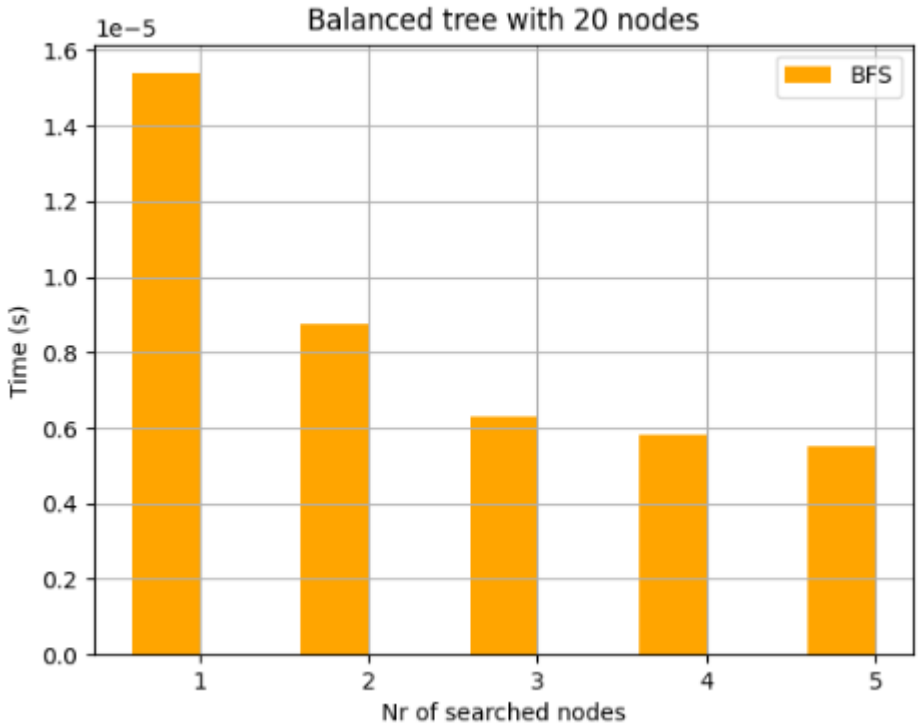
```python
def bfs(node, target):
    queue = [(node, [])]
    visited = set()

    while queue:
        current, path = queue.pop(0)
        if current.value == target:
            return path + [current]
        visited.add(current)

        for child in current.children:
            if child not in visited:
                queue.append((child, path + [current]))

    return None
```

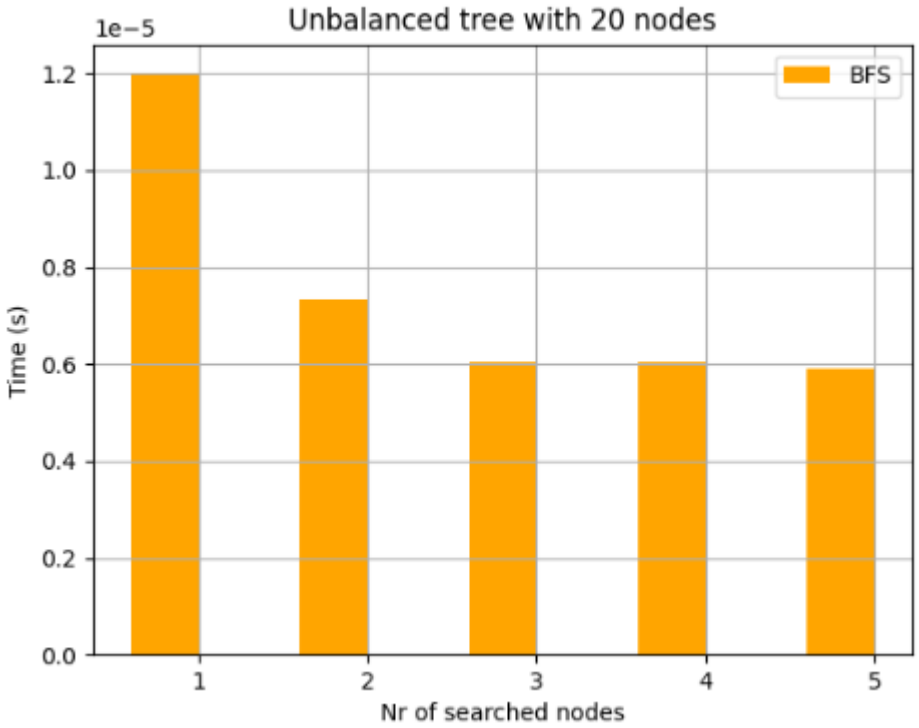
The time complexity of this BFS algorithm is **O(V+E)**, where **V** is the number of vertices and **E** is the number of edges in the graph. In each iteration of the while loop, we dequeue a node from the queue and examine all its children. Since each node is added to the queue at most once, and each edge is traversed at most twice (once for each of its adjacent nodes), the total number of node and edge examinations is **O(V+E)**. Furthermore, in each iteration, we add the current node and its path to the **visited** set and **path** list, respectively, which takes constant time. Since each node is visited and added to the **path** list at most once, these operations take **O(V)** time. Like in previous algortihm the best case scenario is **O(1)** and worst case **O(n)**. Therefore, the overall time complexity of the algorithm is **O(V+E)**. This is the same as the time complexity of traversing the entire graph, since we need to visit each vertex and edge at least once.

Results:

Nr of Nodes searched	1	2	3	4	5
Balanced BFS	1.5400000847876072e-05	8.750001143198460e-06	6.300001890243341e-06	5.825002517667599e-06	5.500001134350896e-06
Nr of Nodes searched	1	2	3	4	5
Unbalanced BFS	1.200000406242907e-05	7.350001396844163e-06	6.033335618364314e-06	6.050002411939204e-06	5.920001422055066e-06

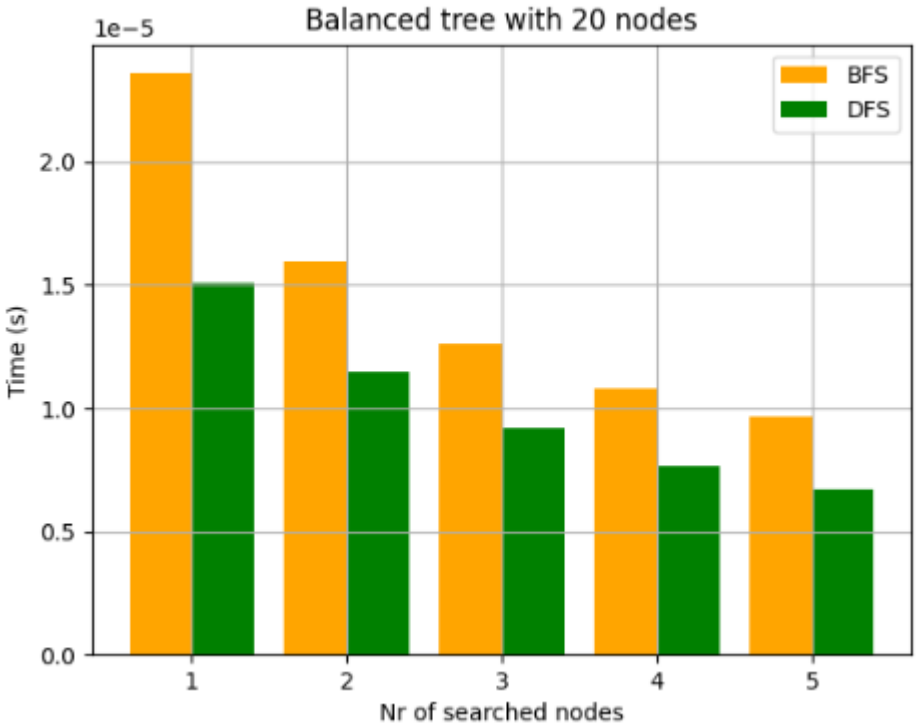
Graph:

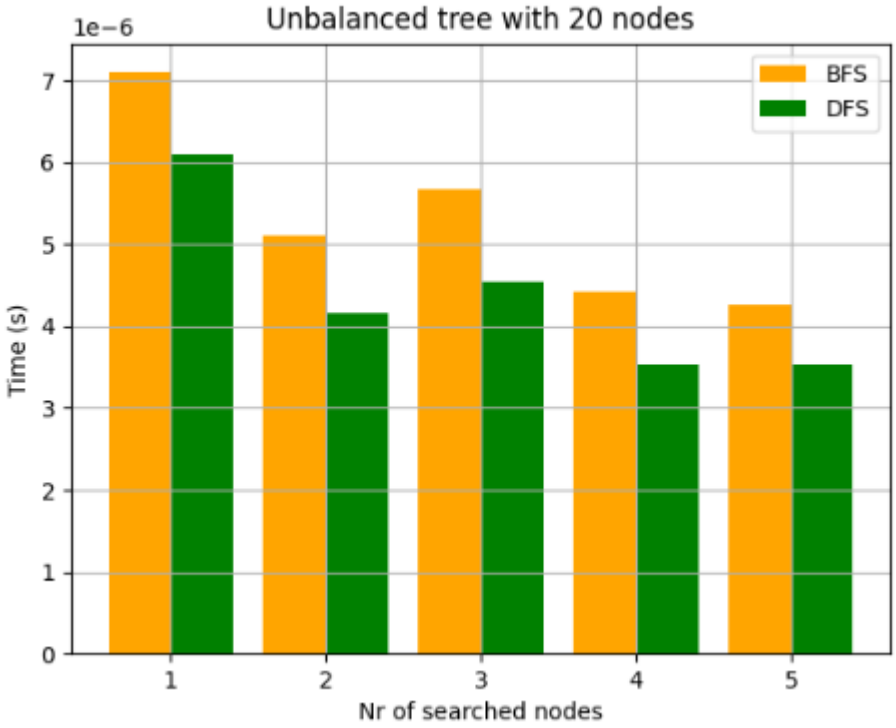




Both algorithms compared:

Graphs:





Time Table:

Nr of Nodes searched	1	2	3	4	5
Balanced BFS	9.399998816661537e-06	7.399998139590025e-06	5.8333316701464355e-06	6.249998477869667e-06	5.299998156260699e-06
Balanced DFS	6.000002031214535e-06	4.0000013541430235e-06	3.6999990697950125e-06	4.424999133334495e-06	3.739999374374747e-06
Unbalanced BFS	7.100003131199628e-06	5.099998816149309e-06	5.666666159716745e-06	4.424999133334495e-06	4.2599989683367315e-06
Unbalanced DFS	6.100002792663872e-06	4.150002496317029e-06	4.533333897901078e-06	3.5249995562480763e-06	3.539999306667596e-06

CONCLUSION

The laboratory work conducted an empirical analysis of DFS and BFS algorithms and presented the results graphically. The purpose of the experiment was to compare the performance of these two algorithms on different types of trees.

The results of the experiment showed that the performance of DFS and BFS varied depending on the characteristics of the tree. For example, DFS was faster than BFS. These findings are consistent with the theoretical time complexities of these algorithms. Both DFS and BFS having a same time complexity of O(n) with n, nr of nodes.

The graphical presentation of the results was effective in conveying the findings of the experiment. The graphs allowed us to see the relative performance of DFS and BFS on different types of graphs at a glance, and to easily compare the results of different experiments.

Overall, the laboratory work demonstrated the importance of empirical analysis in evaluating the performance of algorithms, and showed how graphical presentation can be an effective tool for communicating the results of experiments. The findings of the experiment can be useful in guiding the choice of algorithm for different types of graphs in practical applications.

REFERENCES

AA-Labs/Lab4 at main · Syn4z/AA-Labs (github.com)