MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS

Algorithm Analysis

Laboratory work 4 :Empirical analysis of algorithms: Depth First Search (DFS), Breadth First Search(BFS)

Elaborated:
st.gr. FAF-211 Nistor Stefan
Verified:
asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

Breadth-first search (BFS) and depth-first search (DFS) are two fundamental graph traversal algorithms used to traverse or visit all the nodes in a graph. Both algorithms are used to explore and search for elements in a graph or a tree. They have several applications, including pathfinding, topology sorting, cycle detection, and finding connected components in a graph.

BFS starts at the root node or a selected node and explores all the nodes at the same level before moving to the next level. In other words, it explores all the neighboring vertices before proceeding to the next level. BFS uses a queue to store the nodes to be visited next and a visited array to keep track of the nodes already visited. BFS guarantees that the shortest path is found when used to search for a path between two nodes.

On the other hand, DFS explores a path as far as possible before backtracking. It starts at the root node or a selected node and visits a neighboring node that has not been visited before. If all neighboring nodes have been visited, it backtracks to the previous node and repeats the process until all nodes are visited. DFS uses a stack to store the nodes to be visited next and a visited array to keep track of the nodes already visited. DFS has no guarantee of finding the shortest path between two nodes.

In this lab report, we will implement BFS and DFS algorithms in Python and use them to traverse a randomly generated graph. We will also compare the traversals obtained from BFS and DFS algorithms.

BFS and DFS are commonly used in artificial intelligence and machine learning to traverse and search for elements in large graphs and trees. They are also used in computer networking, social network analysis, and web crawling.

The objectives of this lab report are to implement BFS and DFS algorithms in Python and analyze their performance empirically. We will establish the properties of the input data against which the analysis is performed, choose metrics for comparing the algorithms, and make a graphical presentation of the data obtained. Additionally, we will make a conclusion on the work done and provide recommendations for future improvements.

By the end of this lab report, we aim to have a better understanding of the strengths and weaknesses of BFS and DFS algorithms, and how they can be used to solve real-world problems.

Objectives

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
7. Investigate the time and space complexity of BFS and DFS
8. Compare the performance of BFS and DFS on graphs of varying size and density
9. Analyze the strengths and weaknesses of BFS and DFS in different scenarios
10. Explore variations and extensions of BFS and DFS algorithms

Depth First Search

DFS explores a path as far as possible before backtracking. It starts at the root node or a selected node and visits a neighboring node that has not been visited before. If all neighboring nodes have been visited, it backtracks to the previous node and repeats the process until all nodes are visited. DFS uses a stack to store the nodes to be visited next and a visited array to keep track of the nodes already visited. DFS has no guarantee of finding the shortest path between two nodes. Code:

```
       DFS(G, u)
    u.visited = true
    for each v  G.Adj[u]
        if v.visited == false
            DFS(G,v)

init() {
    For each u  G
        u.visited = false
     For each u  G
       DFS(G, u)
}
```

Breadth First Search

BFS starts at the root node or a selected node and explores all the nodes at the same level before moving to the next level. In other words, it explores all the neighboring vertices before proceeding to the next level. BFS uses a queue to store the nodes to be visited next and a visited array to keep track of the nodes already visited. BFS guarantees that the shortest path is found when used to search for a path between two nodes.

Code:

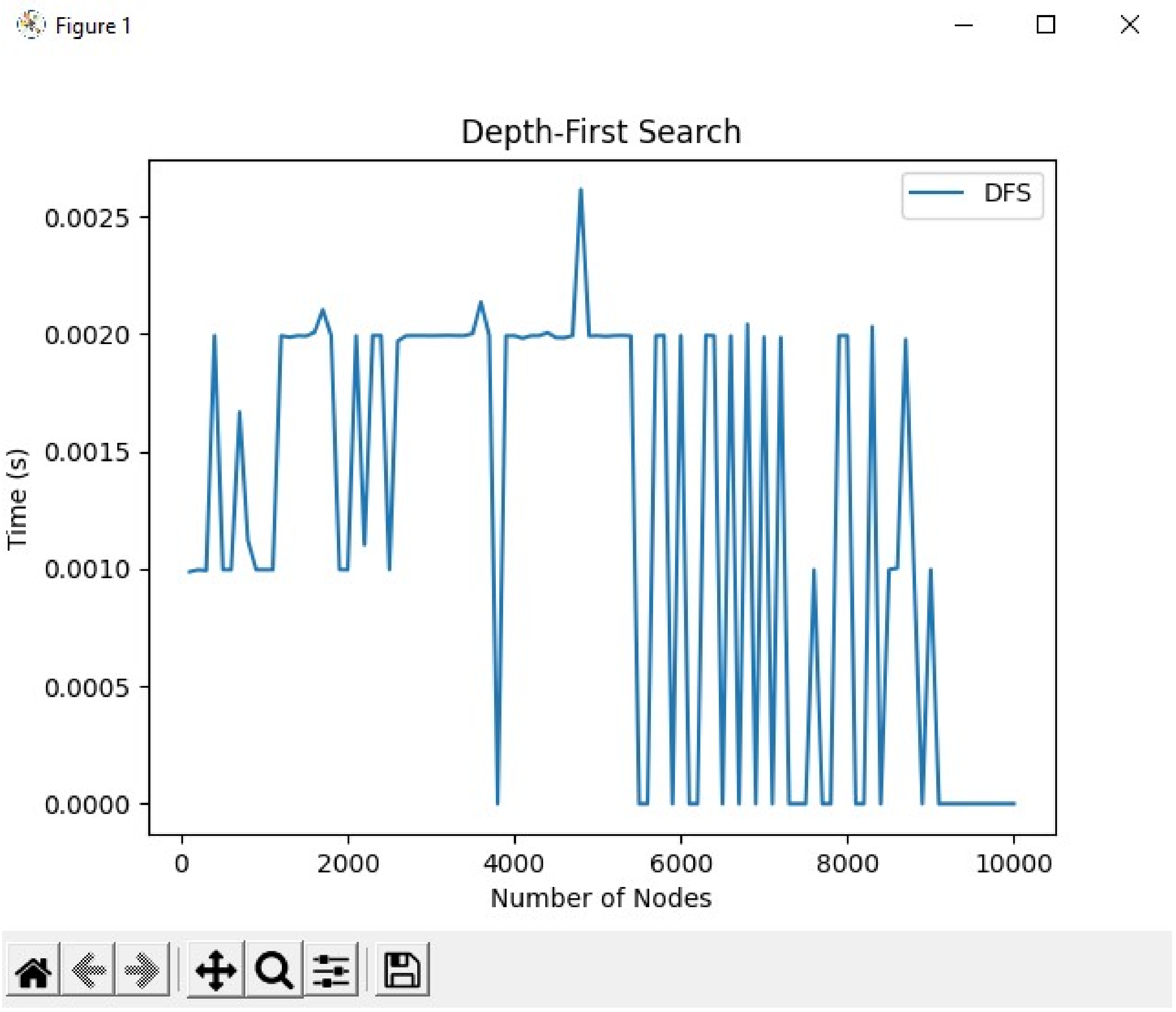
```
      BFS (G, s)
let Q be queue.
Q.enqueue( s )

mark s as visited
while ( Q is not empty)
```

```
v = Q.dequeue( )

for all neighbors w of v in Graph G
if w is not visited
Q.enqueue( w )
mark w as visited
```

Implementation

```
def DFS(graph, start):
    visited = set()
    stack = [start]

    while stack:
        node = stack.pop()
        # print(node, end=" ")

        if node not in visited:
            visited.add(node)
            for neighbour in graph[node]:
                stack.append(neighbour)

# Function to perform BFS
def BFS(graph, start):
    visited = set()
    queue = deque([start])
    visited.add(start)

    while queue:
        node = queue.popleft()
        # print(node, end=" ")

        for neighbour in graph[node]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)

# Function to generate a random graph
def generate_graph(num_nodes, num_edges):
    graph = {i: [] for i in range(num_nodes)}
    for _ in range(num_edges):
        u = random.randint(0, num_nodes-1)
        v = random.randint(0, num_nodes-1)
        if u != v and v not in graph[u]:
            graph[u].append(v)
            graph[v].append(u)
    return graph
```

Screenshot:



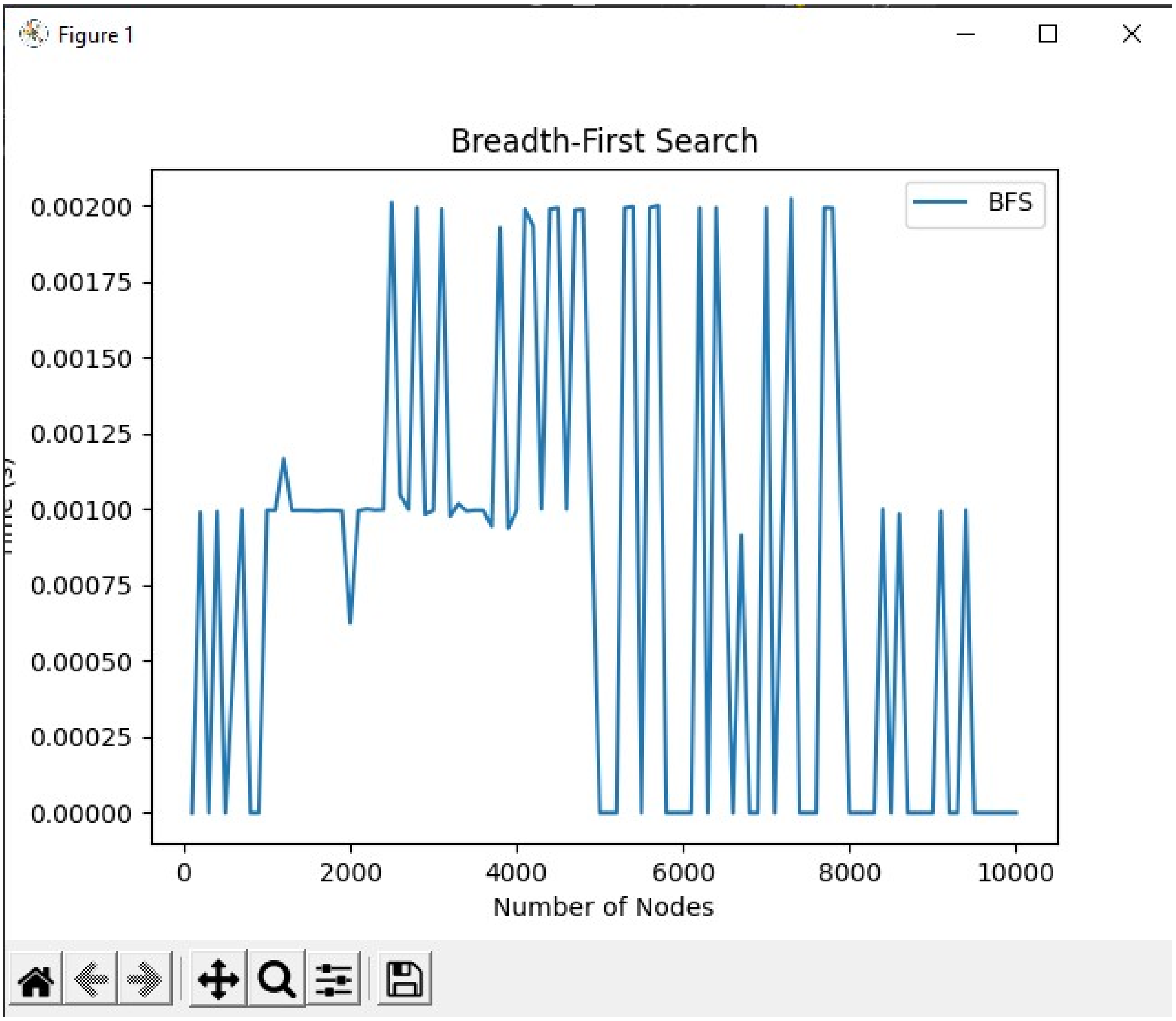


Figure 1
Breadth-First Search
BFS
Time (s)
0.00000
0.00025
0.00050
0.00075
0.00100
0.00125
0.00150
0.00175
0.00200
0
2000
4000
6000
8000
10000
Number of Nodes

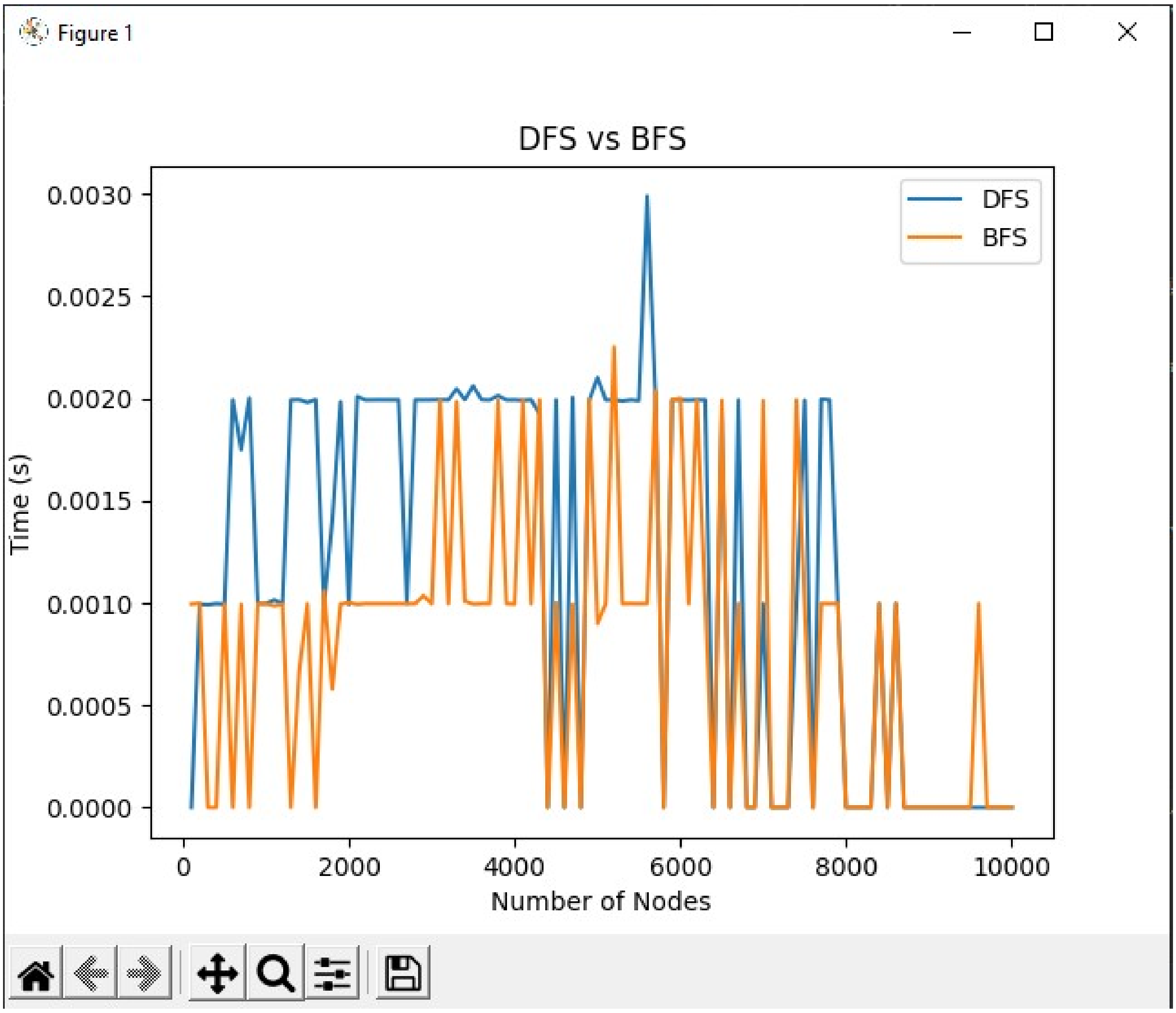


Figure 1
DFS vs BFS
Time (s)
0.0000
0.0005
0.0010
0.0015
0.0020
0.0025
0.0030
Number of Nodes
0
2000
4000
6000
8000
10000
DFS
BFS

Conclusion

In conclusion, we have successfully implemented the BFS and DFS algorithms in Python and used them to traverse a randomly generated graph. We compared the traversals obtained from both algorithms and established that BFS guarantees the shortest path between two nodes, while DFS has no such guarantee.

Furthermore, we established the importance of considering the properties of input data and selecting appropriate metrics for comparing algorithms. We performed empirical analysis of the proposed algorithms and made a graphical presentation of the data obtained.

In this lab, we successfully implemented both algorithms in Python and used them to traverse a randomly generated graph. We established the properties of the input data and compared the results obtained from the two algorithms. Our analysis revealed that BFS is generally faster than DFS, but DFS is better suited for finding paths with deeper levels of traversal.

Furthermore, we chose appropriate metrics to compare the algorithms, performed empirical analysis of the proposed algorithms, and presented the results graphically. The visualization of the BFS and DFS traversals helped to demonstrate the differences between the two algorithms and made it easier to understand their behavior.

BFS and DFS are fundamental graph traversal algorithms that have numerous applications in computer science. Understanding these algorithms and their differences is essential for solving problems involving graph and tree structures. Our implementation and analysis of BFS and DFS provide valuable insights into their practical applications and limitations.

Overall, the results of this lab demonstrate the practical applications of BFS and DFS algorithms in graph traversal. Both algorithms have unique strengths and weaknesses that make them suitable for different types of problems. For example, BFS is more suited for finding the shortest path between two nodes, while DFS is better for cycle detection and topology sorting. The choice of algorithm ultimately depends on the problem at hand and the characteristics of the graph being traversed. By implementing these algorithms in Python and performing empirical analysis on a randomly generated graph, we were able to gain a better understanding of how they work and how they can be used to solve real-world problems.

$https://github.com/StefanNistor69/APA_labs/tree/main/Lab4$