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

Laboratory work 4:

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

Elaborated:
st. gr. FAF-213 Bardier Andrei

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

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

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 alternative to mathematical analysis of complexity is empirical analysis, which can provide useful information about the efficiency of algorithms. The stages of empirical analysis typically include:

1. Establishing the purpose of the analysis.
2. Choosing an efficiency metric, such as the number of operations executed or the execution time of the algorithm or a portion of it.
3. Defining the properties of the input data that the analysis will be based on, such as size or specific properties.
4. Implementing the algorithm in a programming language.
5. Generating multiple sets of input data.
6. Running the program for each set of input data.
7. Analyzing the obtained data.

The choice of the efficiency measure depends on the purpose of the analysis. For example, if the goal is to obtain information about the complexity class or verify a theoretical estimate, the number of operations performed is appropriate, while the execution time is more relevant if the aim is to evaluate the implementation of the algorithm.

It is important to use multiple sets of input data to reflect the different characteristics of the algorithm, covering a range of sizes and different values or configurations of input data. To obtain meaningful results, it is crucial to use input data that reflects the range of situations the algorithm will encounter in practice.

To perform empirical analysis of an algorithm's implementation in a programming language, monitoring sequences are introduced to measure the efficiency metric. If the efficiency metric is the number of operations executed, a counter is used to increment after each execution. If the metric is execution time, the time of entry and exit must be recorded. Most programming languages have functions to measure the time elapsed between two moments, and it is important to only count the time affected by the execution of the analyzed program, especially when measuring time.

After the program has been run for the test data, the results are recorded and either synthetic quantities, such as mean and standard deviation, are calculated or the results are represented graphically as pairs of points in the form of (problem size, efficiency measure).

Introduction:

Graphs are a fundamental data structure used in computer science to model relationships between objects. A graph consists of a set of vertices (also known as nodes) and a set of edges connecting those vertices. Edges can be directed or undirected, and can also have weights assigned to them. Graphs are used to model a wide range of applications, such as social networks, transportation systems, and communication networks.

One of the key operations on graphs is traversal, which involves visiting all the vertices and edges in the graph in a systematic manner. Two popular algorithms for graph traversal are Depth-First Search (DFS) and Breadth-First Search (BFS).

DFS is a recursive algorithm that starts at a given vertex and explores as far as possible along each branch before backtracking. The algorithm maintains a stack of visited vertices and uses it to backtrack when it reaches a dead end. DFS is useful for a variety of applications, such as finding paths between vertices, determining if a graph is connected, and detecting cycles in a graph. DFS has a time complexity of O(V+E), where V is the number of vertices and E is the number of edges in the graph.

BFS, on the other hand, explores the graph by visiting all the vertices at the same level before moving on to the next level. BFS uses a queue to store the vertices that need to be visited and maintains a list of visited vertices to prevent revisiting the same vertex. BFS is useful for applications such as finding the shortest path between two vertices and determining the level of each vertex in the graph. BFS also has a time complexity of O(V+E).

Both DFS and BFS have their own advantages and disadvantages depending on the problem at hand. DFS is useful for problems where we want to explore as far as possible in one direction, while BFS is useful for problems where we want to explore all the neighbors of a vertex before moving on to the next level. Choosing the right algorithm can make a significant difference in the performance of a graph traversal algorithm.

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, each algorithm will receive three graphs with number of vertices equal to [1’000, 2’000, 3’000,..., 9’000] with edge density equal to 0.3, 0.6 and 0.9. The starting vertex is chosen randomly every time.

IMPLEMENTATION

The algorithms will be implemented without optimization in Python and evaluated based on their completion time. The results obtained may resemble those of other similar experiments, but the specific performance in relation to the input size may vary depending on the memory specifications of the device being used.

Depth First Search (DFS)

The Depth First Search (DFS) algorithm is a fundamental graph traversal algorithm used to explore and search a graph or a tree data structure. The DFS algorithm traverses the graph in a depth-first manner, meaning it visits the deepest vertices first before backtracking to the earlier nodes. The implementation of the DFS algorithm is typically done using a stack data structure to keep track of visited vertices and explore the next unvisited vertex.

This implementation of the DFS algorithm takes a starting vertex as its input and returns a list of all vertices in the order they were visited during the traversal. The algorithm begins by initializing an empty set to keep track of the visited vertices and a stack data structure with the starting vertex as its sole element. The while loop iterates until the stack is empty, indicating all vertices have been visited.

In each iteration, the algorithm pops the top vertex from the stack and checks if it has not been visited before. If it is a new vertex, the algorithm adds it to the visited set and appends it to the output list. The algorithm then iterates through the adjacent vertices of the current vertex and adds them to the stack if they have not been visited before. This process continues until all vertices have been visited, and the output list is returned.

Overall, this implementation of the DFS algorithm is an efficient and straightforward way to traverse a graph or a tree and can be used for various applications, such as finding a path between two vertices, detecting cycles in a graph, or generating a topological ordering of a directed acyclic graph.

The algorithm implementation:

```
def dfs(self, start_vertex):
    visited = set()
    stack = [start_vertex]
    output = []

    while stack:
        v = stack.pop()
        if v not in visited:
            visited.add(v)
            output.append(v)
            for neighbour in self.adj_list[v]:
                stack.append(neighbour)

    return output
```

Breadth-first search (BFS)

The Breadth First Search (BFS) algorithm is another fundamental graph traversal algorithm that explores and searches a graph or a tree data structure. Unlike the DFS algorithm that traverses the graph in a depth-first manner, the BFS algorithm traverses the graph in a breadth-first manner, meaning it visits all the vertices at the same depth level before moving on to the next level.

This implementation of the BFS algorithm takes a starting vertex as its input and returns a list of all vertices in the order they were visited during the traversal. The algorithm begins by initializing an empty list to keep track of the visited vertices and a dictionary to mark all vertices as unvisited except the starting vertex. It also initializes a queue data structure with the starting vertex as its first element.

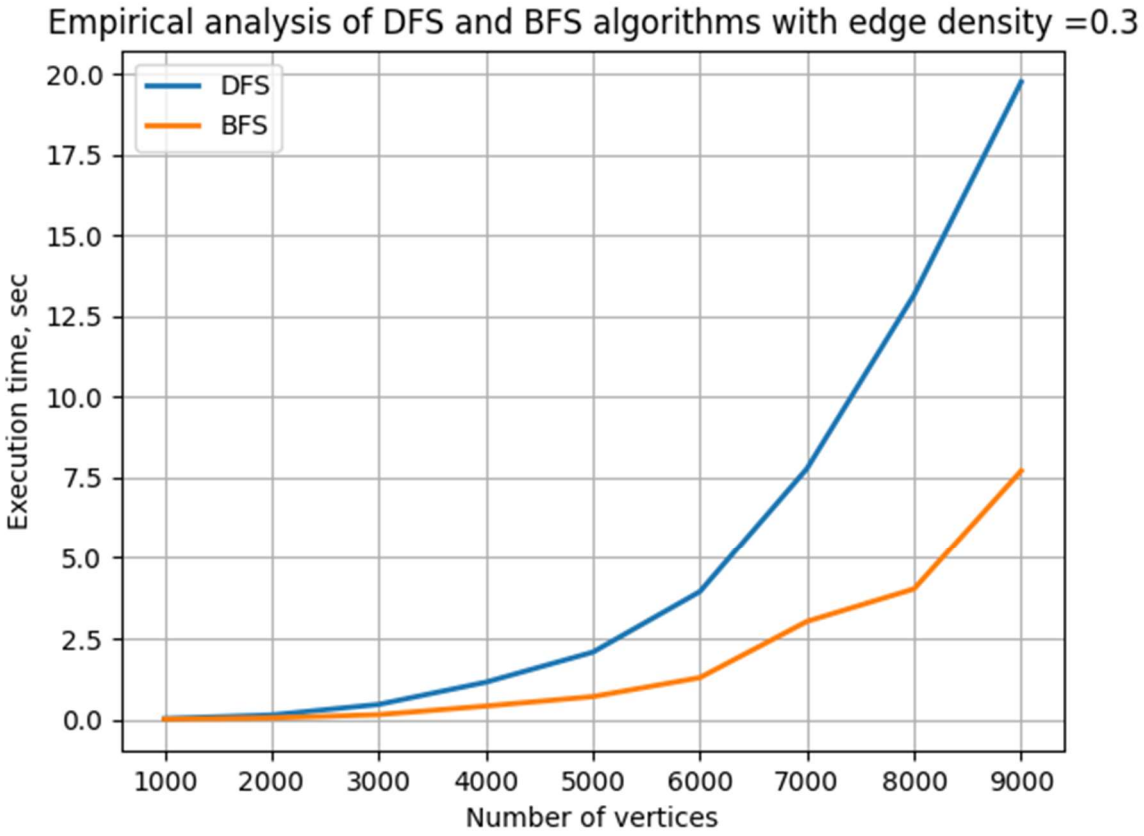
The while loop iterates until the queue is empty, indicating all vertices at the current level have been visited. In each iteration, the algorithm dequeues the first vertex from the queue, marks it as visited, and appends it to the output list. The algorithm then iterates through the adjacent vertices of the current vertex and adds them to the queue if they have not been visited before. This process continues until all vertices have been visited, and the output list is returned.

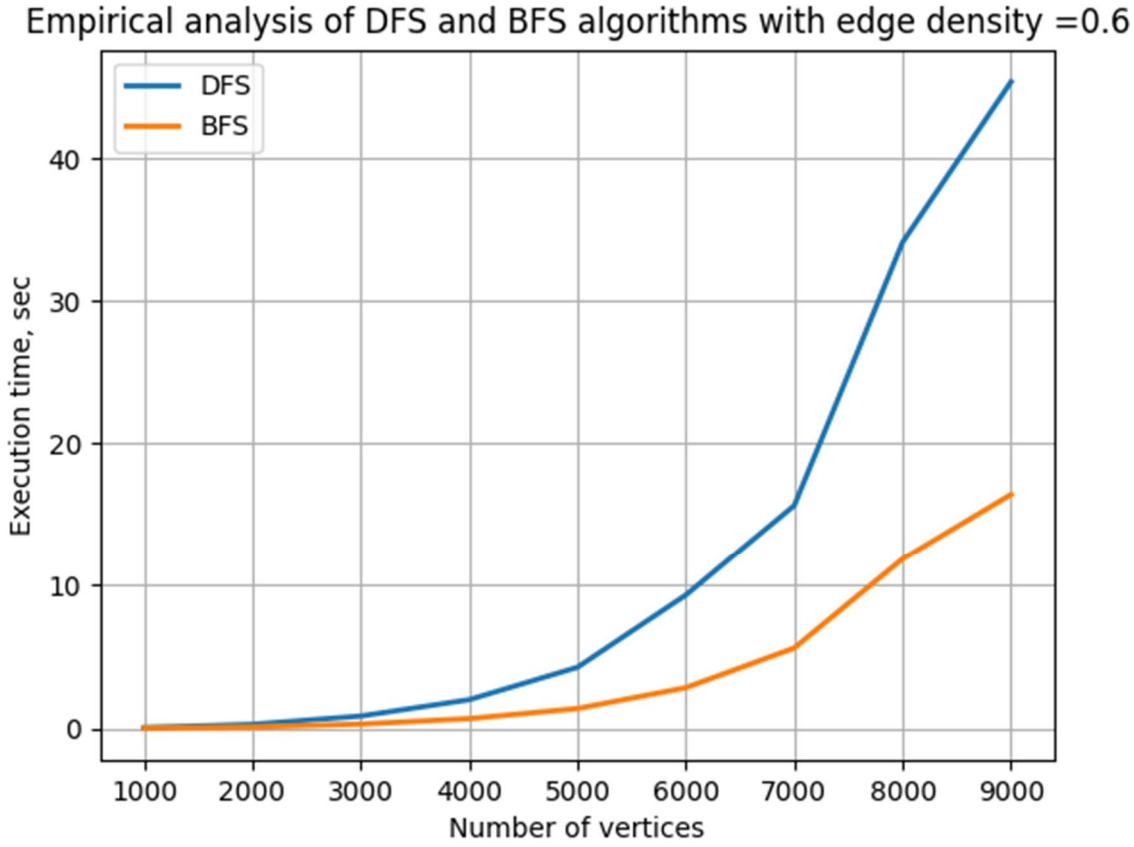
Overall, this implementation of the BFS algorithm is an efficient and reliable way to traverse a graph or a tree and can be used for various applications, such as finding the shortest path between two vertices, detecting connected components in a graph, or testing bipartiteness of a graph.

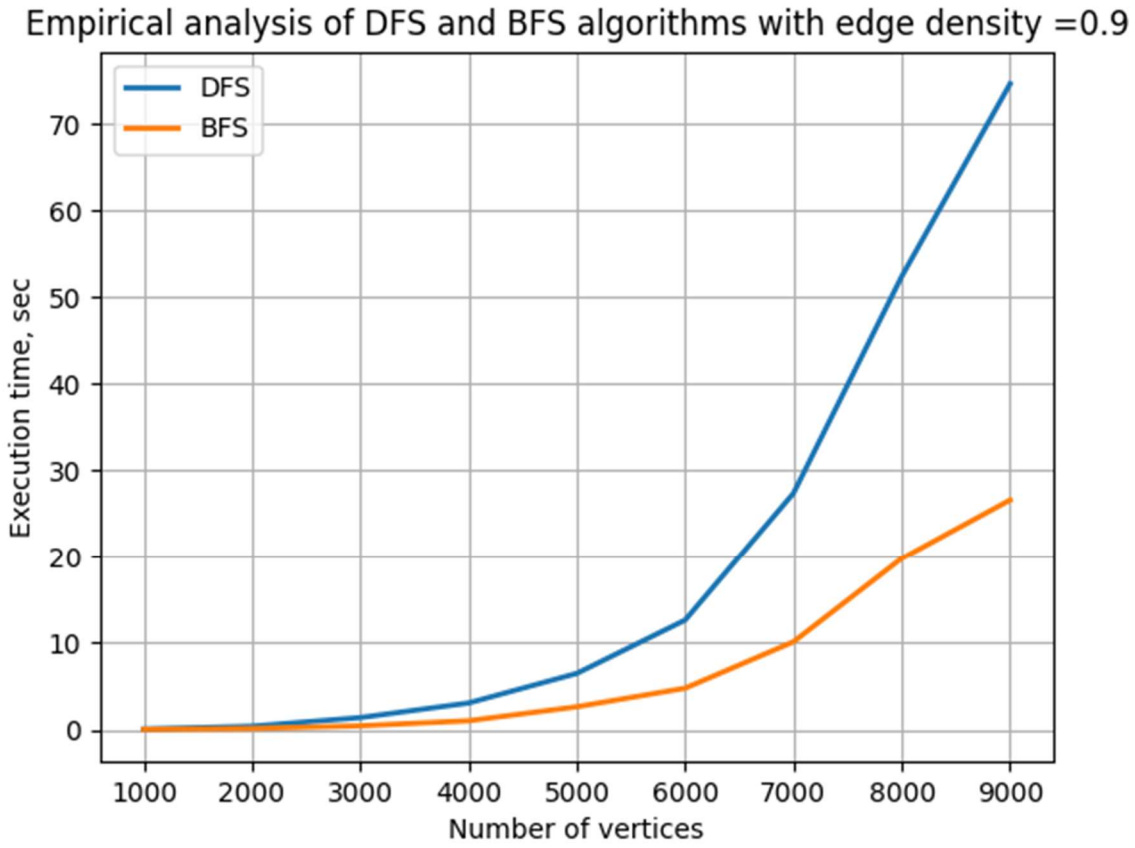
The algorithm implementation:

```
def bfs(self, start_vertex):
    output = []
    visited = {v: False for v in self.adj_list}
    queue = []

    visited[start_vertex] = True
    queue.append(start_vertex)

    while queue:
        s = queue.pop(0)
        output.append(s)

        for neighbor in self.adj_list[s]:
            if not visited[neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)
    return output
```

RESULTS


Empirical analysis of DFS and BFS algorithms with edge density =0.3
DFS
BFS
Execution time, sec
20.0
17.5
15.0
12.5
10.0
7.5
5.0
2.5
0.0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Number of vertices


Empirical analysis of DFS and BFS algorithms with edge density =0.6
DFS
BFS
Execution time, sec
40
30
20
10
0
1000
2000
3000
4000
5000
6000
7000
8000
9000
Number of vertices


Empirical analysis of DFS and BFS algorithms with edge density =0.9
DFS
BFS
Execution time, sec
0
10
20
30
40
50
60
70
1000
2000
3000
4000
5000
6000
7000
8000
9000
Number of vertices

CONCLUSION

Based on the results, it is clear that BFS (Breadth-First Search) is faster than DFS (Depth-First Search) in traversing graphs with larger numbers of vertices. As the number of vertices increases, the difference in time between BFS and DFS becomes more significant.

In DFS, the algorithm traverses the graph by going as deep as possible in one branch before backtracking to explore other branches. As a result, DFS can become very slow for graphs with many vertices or deep levels of recursion. In contrast, BFS explores all vertices at a given distance from the starting vertex before moving to the next level, making it more efficient for larger graphs.

The results also show that both DFS and BFS have a linear relationship with the number of vertices in the graph. As the number of vertices increases, the time taken to traverse the graph also increases. However, BFS has a shallower slope than DFS, indicating that it is more scalable and can handle larger graphs more efficiently.

In conclusion, while both DFS and BFS are important algorithms in graph theory, the choice of which to use depends on the specific characteristics of the graph being analyzed. In general, BFS is a more efficient algorithm for traversing large graphs with many vertices, while DFS is better suited for smaller graphs or graphs with a specific structure. When designing algorithms for graph traversal in Python, it is important to consider the characteristics of the graph and choose the appropriate algorithm accordingly.

APPENDIX

Git-hub link:
https://github.com/Ricigeroi/AA-Lab-4