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-212 Brinza Cristian

Verified:
asist. Univ. Fiștic Cristofor

Chișinău – 2023

GitHub link: https://github.com/CristianBrinza/UTM/tree/main/year2/aa/labs/lab4

BASIC TASK:
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.

**Depth-First Search (DFS):**

DFS, or Depth-First Look, is an algorithm utilized to navigate a chart or tree information structure. It begins at the root hub (or any subjective hub) and investigates as distant as conceivable along each department some time recently backtracking.

The calculation keeps up a stack to keep track of the hubs to visit. When going by a node, the algorithm marks the hub as gone to and pushes all unvisited neighbors of the hub onto the stack. It at that point pops the another hub from the stack and rehashes the method until the stack is purge. DFS is regularly utilized to check in the event that a chart is associated and to discover all associated components of a chart. It can moreover be utilized to discover ways between hubs in a chart.

```python
# Implementation of DFS Algorithm in Python
def dfs(graph, start):
    visited, stack = set(), [start]
    while stack:
        vertex = stack.pop()
        if vertex not in visited:
            visited.add(vertex)
            stack.extend(graph[vertex] - visited)
    return visited
```

This function is an implementation of the DFS (Depth-First Search) algorithm in Python. It takes two parameters: graph, which is a dictionary that represents the graph as an adjacency list, and start, which is the starting node for the search.

The function initializes two variables: visited and stack. visited is a set that stores the nodes that have been visited during the search, and stack is a list that stores the nodes that still need to be visited.

The algorithm starts by adding the starting node start to the stack. It then enters a loop that continues until the stack is empty. In each iteration of the loop, the function pops the last node from the stack and stores it in the vertex variable.

If the node vertex has not been visited yet, the function adds it to the visited set and extends the stack with the unvisited neighbors of vertex. The graph[vertex] - visited expression returns the set of neighbors of vertex that have not been visited yet. The - operator is used to subtract the visited set from the set of neighbors, ensuring that only unvisited neighbors are added to the stack.

Finally, the function returns the visited set, which contains all the nodes that were visited during the search.

**Breadth-First Search (BFS):**

BFS, or Breadth-First Search, is another algorithm used to traverse a graph or tree data structure. It starts at the root node (or any arbitrary node) and explores all the neighboring nodes at the current depth level before moving on to the nodes at the next depth level.

The algorithm maintains a queue to keep track of the nodes to visit. When visiting a node, the algorithm marks the node as visited and adds all unvisited neighbors of the node to the end of the queue. It then dequeues the next node from the queue and repeats the process until the queue is empty. BFS is often used to find the shortest path between nodes in a graph and to solve other graph problems such as finding the minimum spanning tree or performing topological sorting.

```python
# Implementation of BFS Algorithm in Python
def bfs(graph, start):
    visited, queue = set(), [start]
    while queue:
        vertex = queue.pop(0)
        if vertex not in visited:
            visited.add(vertex)
            queue.extend(graph[vertex] - visited)
    return visited
```

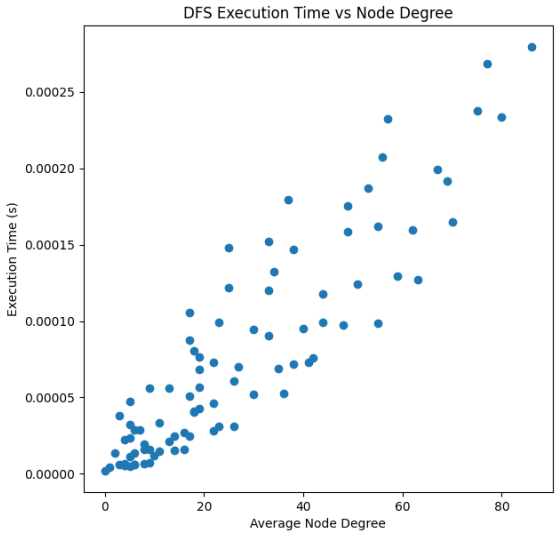
This function is an implementation of the BFS (Breadth-First Search) algorithm in Python. It takes two parameters: graph, which is a dictionary that represents the graph as an adjacency list, and start, which is the starting node for the search.

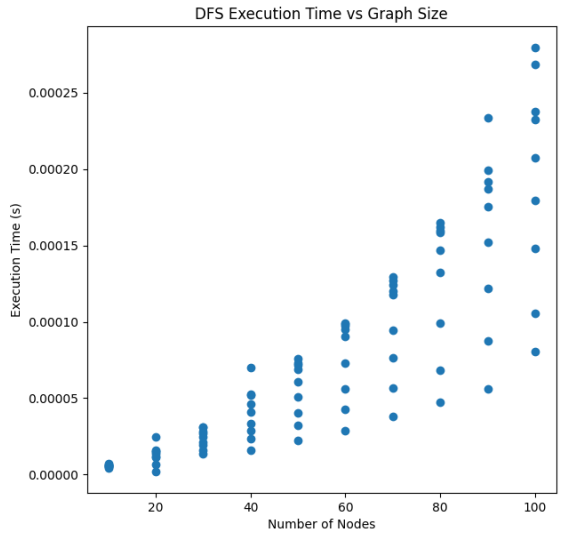
The function initializes two variables: visited and queue. visited is a set that stores the nodes that have been visited during the search, and queue is a list that stores the nodes that still need to be visited.

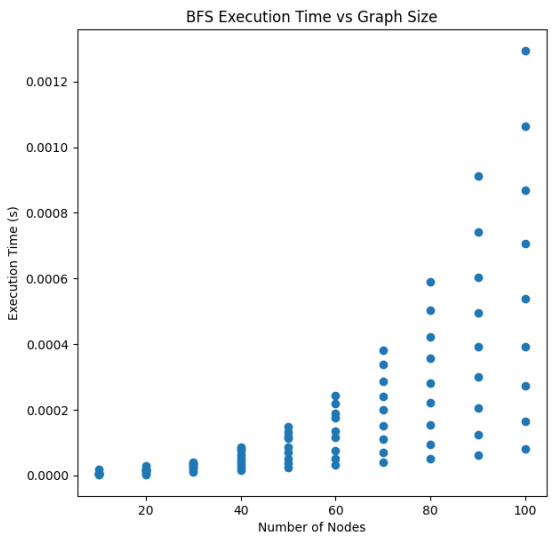
The algorithm starts by adding the starting node start to the queue. It then enters a loop that continues until the queue is empty. In each iteration of the loop, the function dequeues the first node from the queue and stores it in the vertex variable.

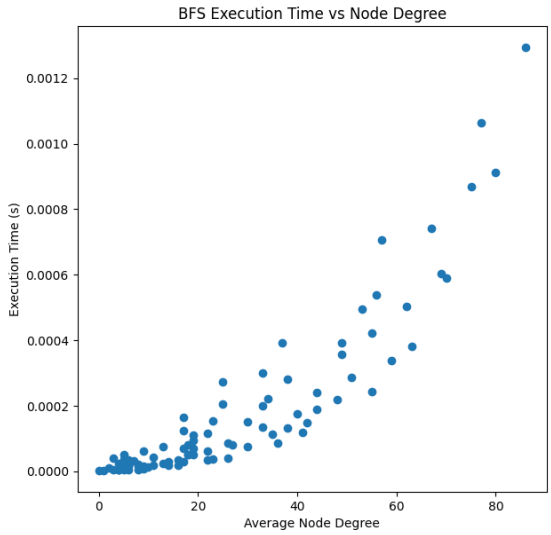
If the node vertex has not been visited yet, the function adds it to the visited set and extends the queue with the unvisited neighbors of vertex. The graph[vertex] - visited expression returns the set of neighbors of vertex that have not been visited yet, ensuring that only unvisited neighbors are added to the queue.

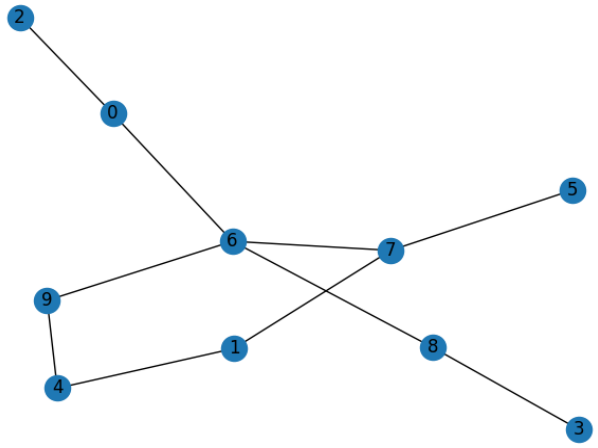
Finally, the function returns the visited set, which contains all the nodes that were visited during the search.


DFS Execution Time vs Node Degree
Execution Time (s)
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
0
20
40
60
80
Average Node Degree


DFS Execution Time vs Graph Size
Execution Time (s)
0.00025
0.00020
0.00015
0.00010
0.00005
0.00000
20
40
60
80
100
Number of Nodes


BFS Execution Time vs Graph Size
Execution Time (s)
0.0012
0.0010
0.0008
0.0006
0.0004
0.0002
0.0000
20
40
60
80
100
Number of Nodes




## Conclusions:

Algorithm Comparison: The Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms have been compared based on their execution times in traversing random graphs of varying sizes and edge densities. This comparison helps in understanding the relative performance of these two popular graph traversal algorithms.

Execution Time Trends: By analyzing the scatter plots, trends in the execution times of DFS and BFS can be observed as the graph size and average node degree change. This provides valuable insights into how these algorithms scale with respect to the number of nodes and the density of edges in the graphs.

Iterative Implementation: The iterative implementation of both DFS and BFS, using a stack and a queue, respectively, helps prevent stack overflow issues for large graphs. This demonstrates the practicality of these implementations in real-world applications where large graphs are common.

Random Graph Generation: The create_graph() function generates random graphs with a specified number of nodes and edges, ensuring no self-loops or duplicate edges. This feature allows for the testing of the algorithms on a diverse set of graph structures, providing a broader perspective on their performance.

Visualization: The visualization of the results using scatter plots helps in identifying trends and patterns in the execution times of DFS and BFS. These visualizations aid in the effective communication of the performance differences between the two algorithms.

Practical Implications: The findings from this study can guide the selection of an appropriate graph traversal algorithm for specific applications. Depending on the observed performance trends, developers can choose between DFS and BFS based on the characteristics of the graphs they are working with, such as size and edge density.

In conclusion, both DFS and BFS are widely used in computer science and have applications in many areas such as web crawling, pathfinding, and network analysis. They have different strengths and weaknesses and can be used in different scenarios depending on the problem at hand. DFS is generally faster and uses less memory than BFS for large, sparse graphs, but may not find the shortest path between nodes. BFS is guaranteed to find the shortest path between nodes and is more suitable for smaller, denser graphs.

GitHub link: https://github.com/CristianBrinza/UTM/tree/main/year2/aa/labs/lab4