**Ministry of Education and Research of the Republic of Moldova**

**Technical University of Moldova**

**Faculty of Computers, Informatics and Microelectronics**

**REPORT**

Laboratory work no. 4

*Empirical analysis of algorithms:*

*Depth First Search Breadth First Search*

Did:

St. gr. FAF-211 Andrei Ceban

Checked:

asist. univ. Cristofor Fistic

Chişinău - 2023

**Laboratory work no. 4**

**Objective:**

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.

**INTRODUCTION**

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

For example, in a chess endgame a chess engine may build the game tree from the current position by applying all possible moves, and use breadth-first search to find a win position for white. Implicit trees (such as game trees or other problem-solving trees) may be of infinite size; breadth-first search is guaranteed to find a solution node[1] if one exists. Breadth-first search can be generalized to graphs, when the start node is explicitly given, and precautions are taken against following a vertex twice.

Depth-first search (DFS) is an algorithm for traversing or searching tree or graph data structures. The algorithm starts at the root node (selecting some arbitrary node as the root node in the case of a graph) and explores as far as possible along each branch before backtracking. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

For applications of DFS in relation to specific domains, such as searching for solutions in artificial intelligence or web-crawling, the graph to be traversed is often either too large to visit in its entirety or infinite (DFS may suffer from non-termination). In such cases, search is only performed to a limited depth; due to limited resources, such as memory or disk space, one typically does not use data structures to keep track of the set of all previously visited vertices.

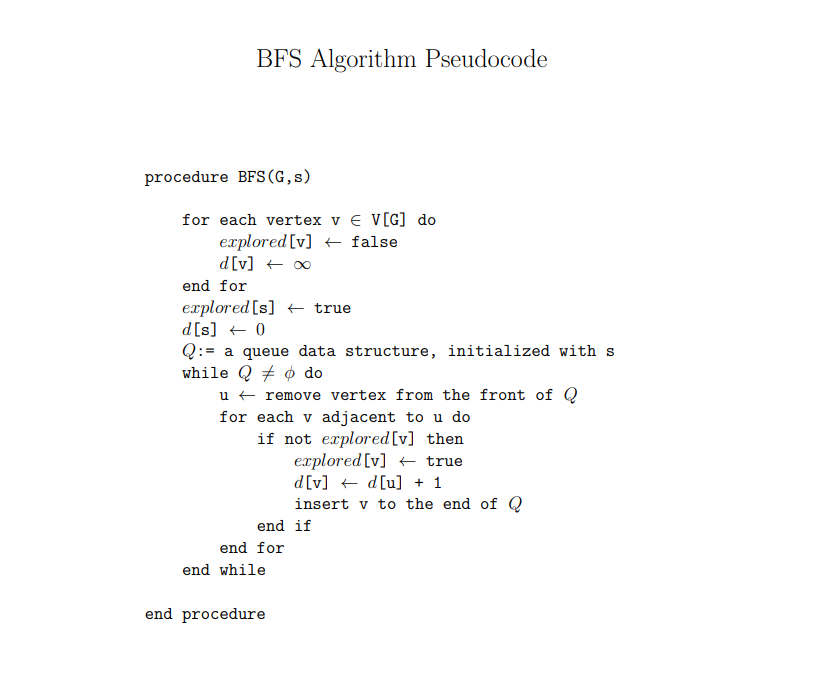
It may also be used to collect a sample of graph nodes. However, incomplete DFS, similarly to incomplete BFS, is biased towards nodes of high degree. The result of a depth-first search of a graph can be conveniently described in terms of a spanning tree of the vertices reached during the search.

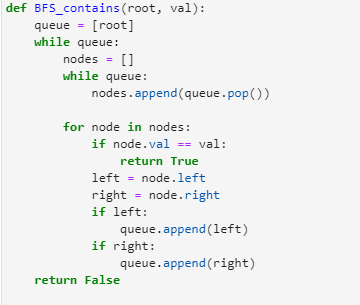
**IMPLEMENTATION**

**Breadth First Search**

**Algorithm explanation :**

1. Choose a starting node: The algorithm starts with a source node. This can be any node in the graph.
2. Initialize a queue and mark the starting node as visited: Create an empty queue and enqueue the starting node. Also, mark the starting node as visited so that we don't visit it again.
3. Explore neighbors of the current node: Dequeue the first node from the queue and explore all its neighbors. Visit each neighbor that hasn't been visited before, mark it as visited, and enqueue it to the queue.
4. Repeat until all nodes have been visited: Repeat step 3 until there are no more nodes to visit in the queue. If the queue becomes empty and there are still unvisited nodes in the graph, then the algorithm chooses another unvisited node as the starting node and repeats the process.
5. Return the result once all nodes have been visited



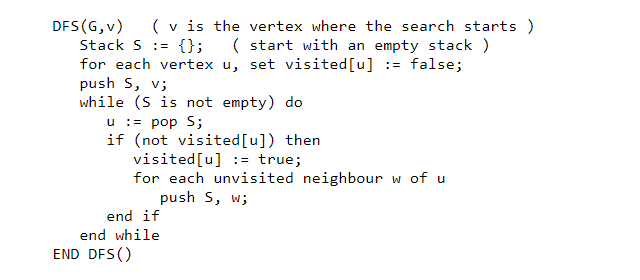


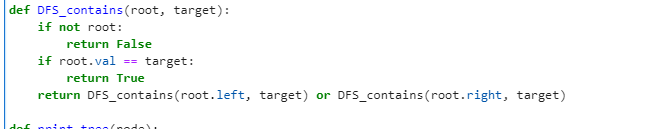
**Deaph First Search**

**Algorithm explanation :**

1. Choose a starting node: The algorithm starts with a source node. This can be any node in the graph.
2. Initialize a stack and mark the starting node as visited: Create an empty stack and push the starting node onto it. Also, mark the starting node as visited so that we don't visit it again.
3. Explore the neighbors of the current node: Pop the top node from the stack and explore all its neighbors. Visit each neighbor that hasn't been visited before, mark it as visited, and push it onto the stack.
4. Repeat until all nodes have been visited: Repeat step 3 until there are no more nodes to visit in the stack. If the stack becomes empty and there are still unvisited nodes in the graph, then the algorithm backtracks to the last node that has unvisited neighbors and repeats the process.
5. Return the result once all nodes have been visited

PSEUDOCODE



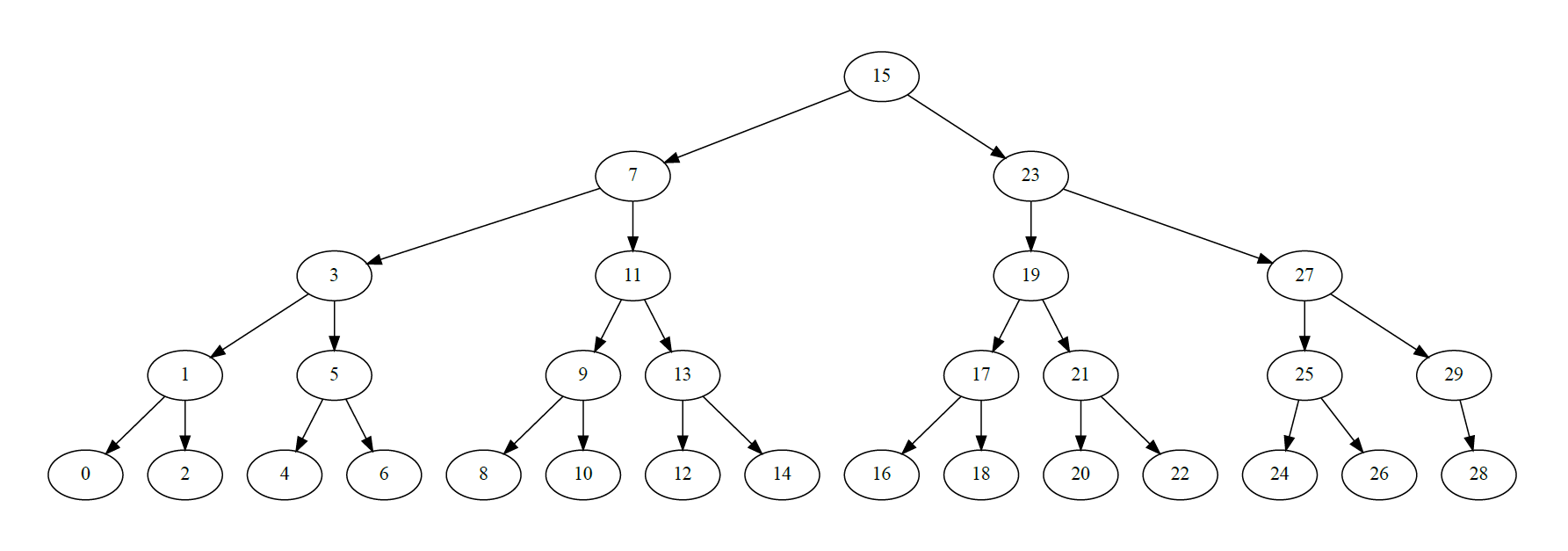


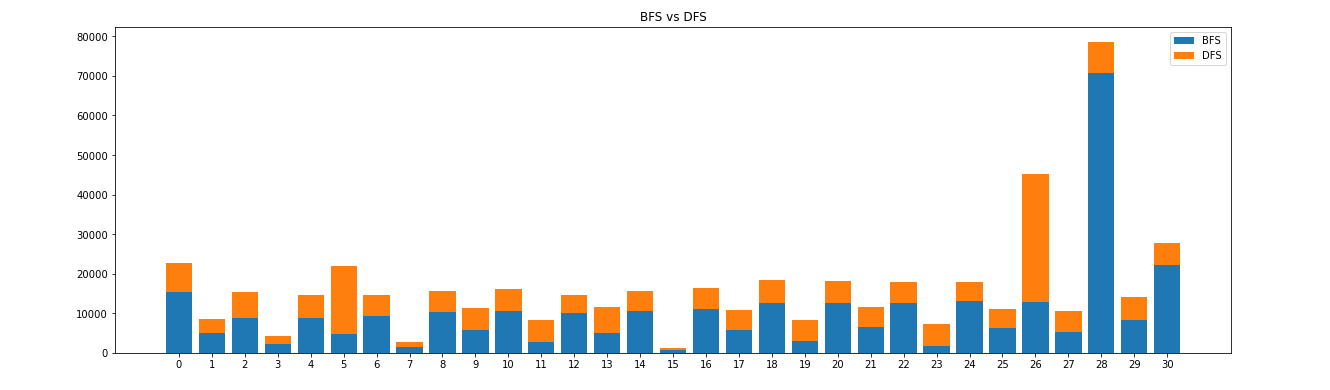
**RESULTS**

The execution speeds of the Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms in traversing randomly generated graphs of various sizes and edge densities have been compared. Understanding the relative performance of these two well-known graph traversal methods is made easier by this comparison.

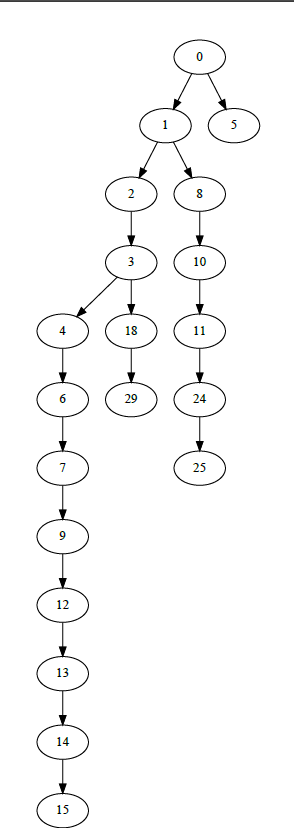
Execution Time Trends: As the graph size and average node degree change, patterns in the execution timings of DFS and BFS can be seen by examining the scatter plots. Insights into how these algorithms scale in terms of the quantity of nodes and the density of edges in the graphs are gained as a result.

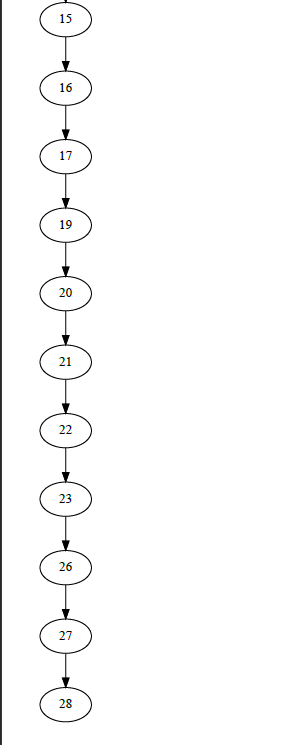
**BALANCED GRAPH**

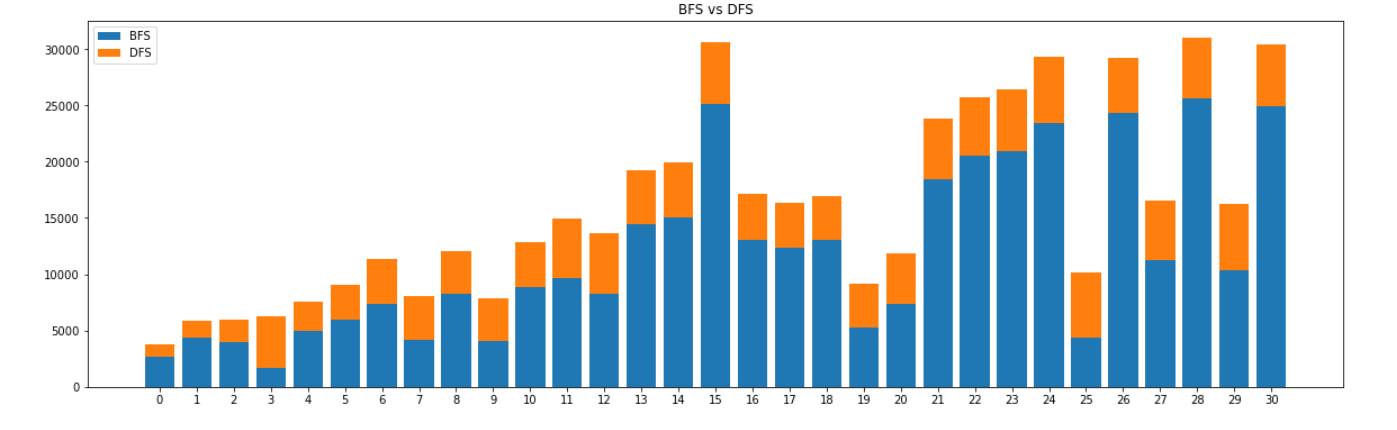




**UNBALANCED GRAPH**







**CONCLUSION**

Both BFS and the DFS are graph-searching techniques that take the same amount of time to run but use different amounts of space. DFS uses linear space as it has to remember a single path to unexplored vertices, whereas BFS maintains every vertex in the memory. DFS produces many solutions and isn’t optimal, but it appears to work well if the solutions are dense, whereas BFS is very optimal because it searches for the best goal first.

For implementation, BFS uses a queue data structure, while DFS uses a stack. BFS uses a larger amount of memory because it expands all children of a vertex and keeps them in memory. DFS is faster than BFS. Time Complexity of BFS = O(V+E) where V is vertices and E is edges. Time Complexity of DFS is also O(V+E) where V is vertices and E is edges. BFS requires more memory space.

A BFS will find the shortest path between the starting point and any other reachable node. A depth-first search will not necessarily find the shortest path.

Git Repo : https://github.com/andeiceban0352/Labs-Anul2/tree/main/Lab%20APA/Lab4