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

Laboratory work 3: Empirical analysis of algorithms: Depth First Search (DFS), Breadth First Search(BFS)

Elaborated:
st. gr. FAF-221 Berco Andrei

Verified:
asist. univ. Fiștic Cristofor
prof. univ. Andrievschi-Bagrin Veronica

Chişinău - 2024

TABLE OF CONTENTS

Contents

ALGORITHM ANALYSIS

Objective

Study and analyze different graph traversing 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:

Empirical analysis provides an alternative approach to understanding the efficiency of algorithms when mathematical complexity analysis is impractical or insufficient. This method proves beneficial in various scenarios:

1. Initial Insights: It offers preliminary insights into an algorithm's complexity class, aiding in the understanding of its efficiency characteristics.
2. Comparative Analysis: It facilitates the comparison of multiple algorithms tackling the same problem, allowing for informed decisions regarding efficiency.
3. Implementation Comparison: Empirical analysis enables the comparison of different implementations of the same algorithm, providing insights into which may perform better in practice.
4. Hardware-specific Evaluation: It helps in assessing an algorithm's efficiency on a particular computing platform, taking into account hardware constraints and capabilities.

The empirical analysis of an algorithm typically involves the following steps:

Establishing Analysis Goals: Clearly define the objectives and scope of the analysis.

1. Choosing Efficiency Metrics: Select appropriate metrics, such as the number of operations executed or the execution time, based on the analysis goals.
2. Defining Input Data Properties: Determine the characteristics of the input data relevant to the analysis, including data size or specific attributes.
3. Implementation: Develop the algorithm in a programming language, ensuring it accurately reflects the intended logic.
4. Generating Input Data Sets: Create multiple sets of input data to cover a range of scenarios and edge cases.
5. Execution and Data Collection: Execute the program for each input data set,

recording relevant performance metrics.

6. Data Analysis: Analyze the collected data, either by computing synthetic quantities like mean and standard deviation or by plotting graphs to visualize the relationship between problem size and efficiency metrics.
7. The choice of efficiency measure depends on the analysis's objectives. For instance, if assessing complexity class or verifying theoretical estimates, counting the number of operations may be suitable. Conversely, if evaluating algorithm implementation behavior, measuring execution time becomes more relevant.

8. Post-execution, recorded results undergo analysis. This involves computing statistical measures or plotting graphs to visualize the algorithm's performance characteristics in terms of problem size and efficiency metrics. Such analyses aid in making informed decisions regarding algorithm selection and optimization strategies.

Introduction:

DFS:

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal.

Depth-first search 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.

Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

Auxiliary Space: O(V + E), since an extra visited array of size V is required, And stack size for iterative call to DFS function.

BFS:

Breadth First Search (BFS) is a fundamental graph traversal algorithm. It involves visiting all the connected nodes of a graph in a level-by-level manner.

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors. BFS is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.

Initialization: Enqueue the starting node into a queue and mark it as visited.

1. Exploration: While the queue is not empty:

2. Dequeue a node from the queue and visit it (e.g., print its value).
   For each unvisited neighbor of the dequeued node:
   Enqueue the neighbor into the queue.
   Mark the neighbor as visited.

3. Termination: Repeat step 2 until the queue is empty.

Time Complexity of BFS Algorithm: O(V + E)

BFS explores all the vertices and edges in the graph. In the worst case, it visits every vertex and edge once. Therefore, the time complexity of BFS is O(V + E), where V and E are the number of vertices and edges in the given graph.

Space Complexity of BFS Algorithm: O(V)

BFS uses a queue to keep track of the vertices that need to be visited. In the worst case, the queue can contain all the vertices in the graph. Therefore, the space complexity of BFS is O(V), where V and E are the number of vertices and edges in the given graph.

Comparison Metric:

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

Input Format:

As input, each algorithm will receive 8 series of numbers of nodes 4, 8, 16, 32, 64, 128, 256, ,512.

Next, using this numbers of nodes, it will be generated randomly graphs with that amount of nodes.

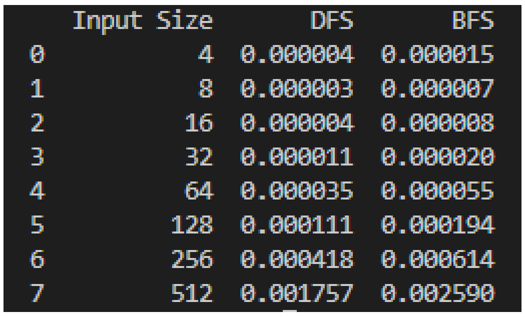
IMPLEMENTATION

All algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar toother experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used.

Depth First Search

Depth First Traversal (or DFS) for a graph is similar to Depth First Traversal of a tree. The only catch here is, that, unlike trees, graphs may contain cycles (a node may be visited twice). To avoid processing a node more than once, use a boolean visited array. A graph can have more than one DFS traversal

```
class GraphDFS:

    def __init__(self):
        self.graph = defaultdict(list)

    def addEdge(self, u, v):
        self.graph[u].append(v)

    def DFSUtil(self, v, visited):

        visited.add(v)

        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)

    def DFS(self, v):

        visited = set()
        self.DFSUtil(v, visited)
```



```
   Input Size       DFS       BFS
0           4  0.000004  0.000015
1           8  0.000003  0.000007
2          16  0.000004  0.000008
3          32  0.000011  0.000020
4          64  0.000035  0.000055
5         128  0.000111  0.000194
6         256  0.000418  0.000614
7         512  0.001757  0.002590
```

Figure 1. Time table for algorithms

As observed in the table, with small numbers of nodes, DFS is faster more than 2 times for up to 32 nodes. After that, it is also faster than BFS, but less than 2 times

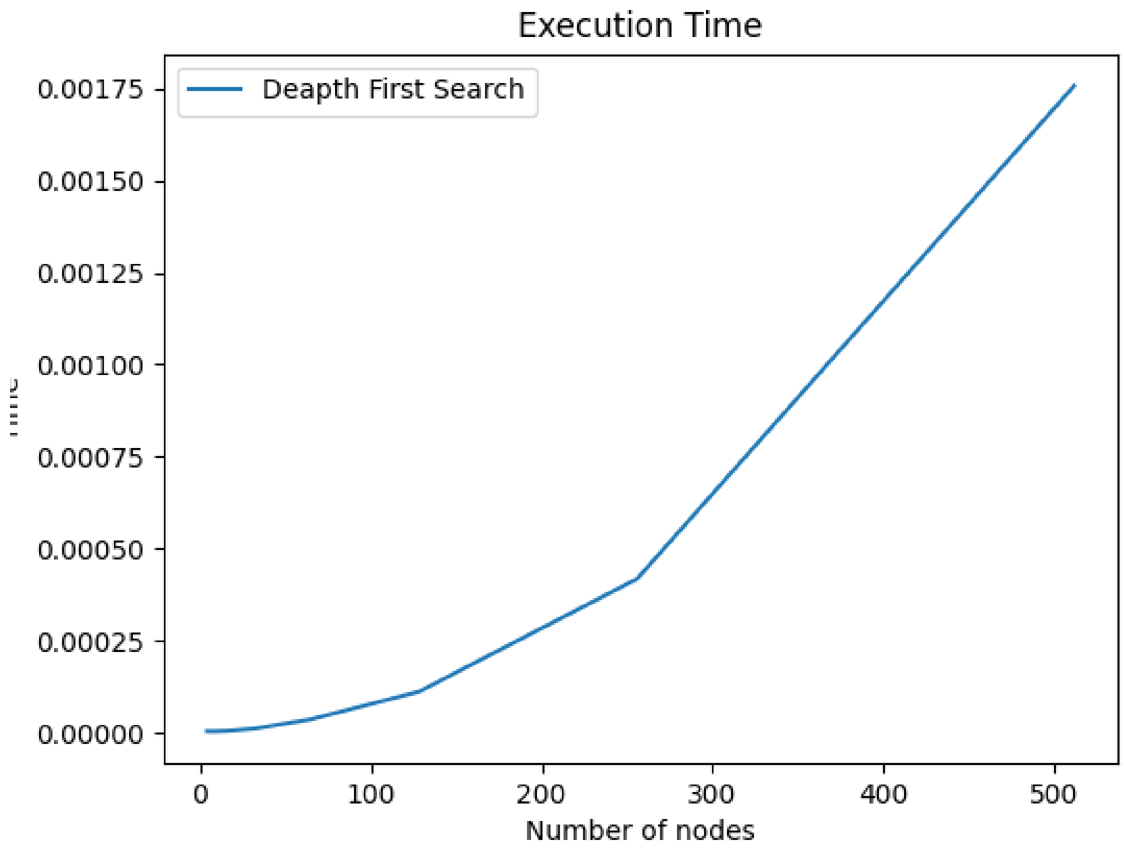


Figure 2. Graph for DFS

The graph shows a positive correlation between the number of nodes and the execution time. This means that it takes longer to perform a depth-first search on a graph with more nodes. This is because a depth-first search has to explore all of the edges of a graph, and a graph with more nodes will have more edges to explore.

Breadth First Search:

Breadth First Search (BFS) is a graph traversal algorithm that explores all the vertices in a graph at the current depth before moving on to the vertices at the next depth level. It starts at a specified vertex and visits all its neighbors before moving on to the next level of neighbors.

```
class GraphBFS:
    def __init__(self):
        self.adjList = defaultdict(list)

    def addEdge(self, u, v):
        self.adjList[u].append(v)
        self.adjList[v].append(u)

    def show(self, adjList):
        for key, value in adjList.items():
            print(key, "   ", value,"\n")

    def bfs(self, startNode):
        queue = deque()
        max_node = max(self.adjList.keys(), default=-1)
        visited = [False] * (max_node + 1)

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

        while queue:

            currentNode = queue.popleft()

            for neighbor in self.adjList[currentNode]:
                if not visited[neighbor]:
                    visited[neighbor] = True
                    queue.append(neighbor)
```

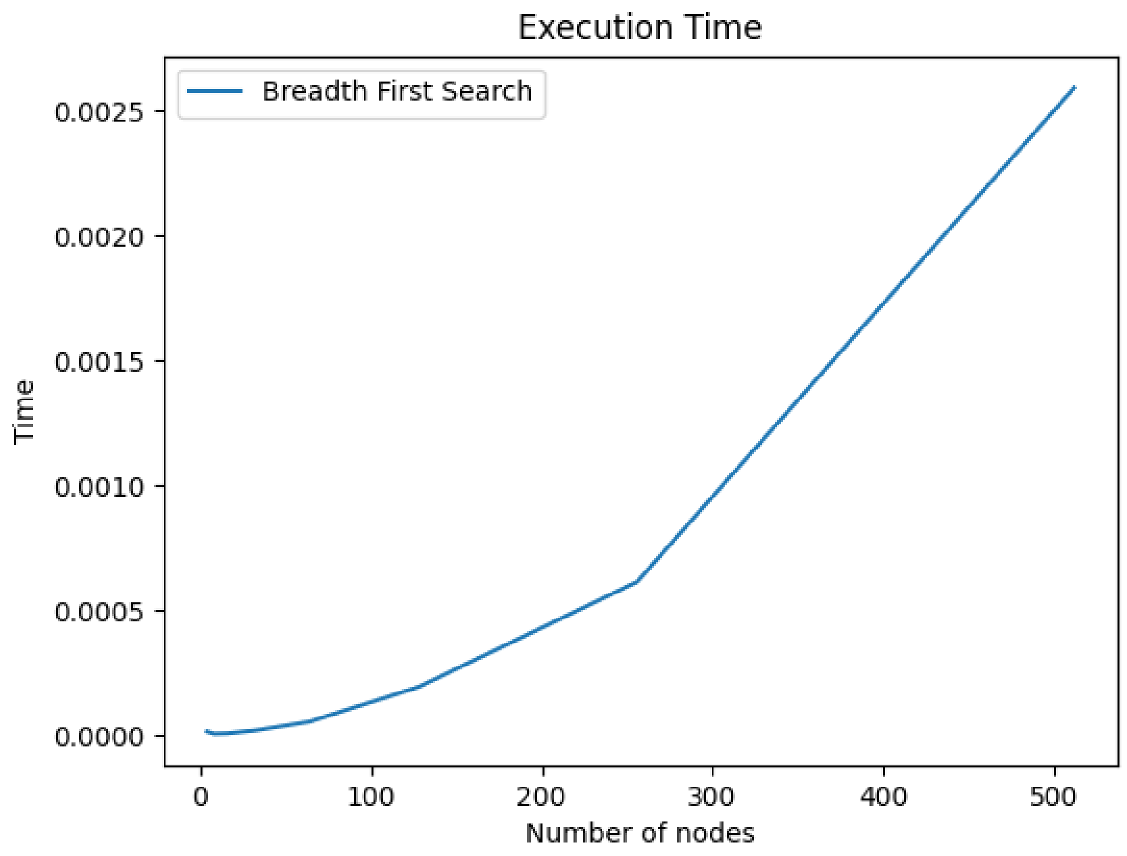
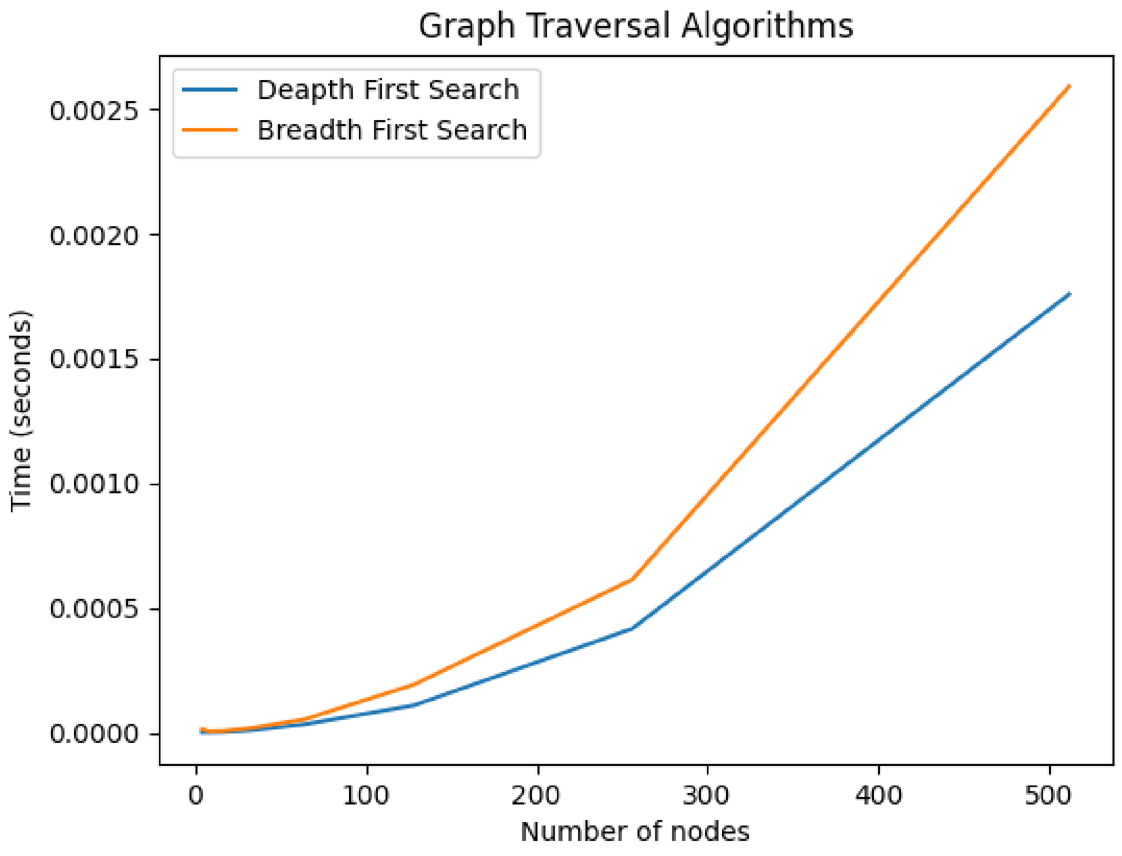


Figure 3. Graph for BFS

We can see that the number of nodes executed increases as time goes on. This makes sense because a BFS algorithm explores outward from a starting node, visiting all its neighbors before moving to the neighbors of those neighbors, and so on. So, as time goes on, the BFS algorithm explores more and more of the graph.

CONCLUSION



This examination sheds light on the characteristics and performance of Depth-First Search (DFS) and Breadth-First Search (BFS) traversal algorithms on graphs. While both algorithms efficiently navigate graph structures, their underlying mechanisms and exploration strategies lead to distinct time complexities in various scenarios.

The graph demonstrates that DFS execution time increases proportionally with the number of nodes in the graph. This aligns with the inherent nature of DFS, where it delves deeply into a single path at a time, exploring all branches before backtracking and proceeding down alternative paths. This characteristic makes DFS a suitable choice for tasks like finding specific nodes within a graph or detecting cycles. However, for graphs with expansive branching structures, DFS can lead to inefficiencies as it might revisit previously explored areas.

The graph depicting BFS showcases a steady rise in the number of nodes explored over time. This reflects the BFS method of systematically visiting all neighboring nodes at the current level before moving down to the next level. This comprehensive exploration makes BFS ideal for finding the shortest path between two nodes in a graph or identifying all connected components within a graph. However, for sparse graphs with a low number of edges, BFS might explore a significant number of unnecessary nodes, potentially leading to higher execution times compared to DFS.

In conclusion, the selection between DFS and BFS hinges on the specific graph structure and the problem at hand. DFS excels at finding specific nodes or cycles, while BFS shines in tasks involving shortest path determination or connected component identification. By understanding the strengths and weaknesses of each algorithm, we can leverage them effectively for graph traversal tasks.

References:

1. https://github.com/KaBoomKaBoom/Algorithm-Analysis.git