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

Laboratory work 3:

Study and Empirical Analysis of Algorithms: Depth First Search (DFS), Breadth First Search(BFS)

Elaborated:
st. gr. FAF-221 Chichioi Iuliana

Verified:

asist. univ. Fiștic Cristofor

prof. univ. Andrievschi-Bagrin Veronica

Chişinău - 2023

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and analyse different 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.
This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.
In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analysed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behaviour of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

Introduction:

Depth First Search (DFS) and Breadth First Search (BFS) are fundamental algorithms used for traversing graphs, exploring vertices, and discovering paths between nodes. These algorithms play crucial roles in various applications, including network routing, pathfinding, maze solving, and graph analysis.

DFS is a recursive algorithm that explores as far as possible along each branch before backtracking. It traverses a graph in depth, exploring the deepest node of a branch before moving on to explore adjacent branches. DFS is characterised by its depth-first nature, making it particularly useful for tasks such as finding connected components, detecting cycles in graphs, and performing topological sorting.

BFS, on the other hand, is an iterative algorithm that explores a graph level by level. It starts at a chosen vertex and explores all its neighbours before moving on to the next level of vertices. BFS traverses a graph breadth-first, prioritising the exploration of nodes at the current level before moving deeper into the graph. BFS is commonly used for finding shortest paths in unweighted graphs, solving puzzles with multiple states, and traversing trees or graphs with uniform edge costs.

While both DFS and BFS aim to explore all vertices in a graph, they differ in their strategies and traversal orders. Understanding the characteristics, advantages, and limitations of DFS and BFS is essential for effectively applying them to various graph-related problems and optimising algorithmic performance.

In this report, we will delve deeper into the mechanics of DFS and BFS, explore their implementations, analyse their runtime complexities, and compare their performance on different types of graphs. Through empirical analysis and experimental results, we aim to provide insights into the behaviour and efficiency of DFS and BFS, aiding in the selection of the most suitable algorithm for specific graph traversal tasks.

Comparison Metric:

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

Input Format:

The input format for the analysis consists of a list of integers representing the number of nodes in the graphs to be analysed. The list values contains the following values:

values = [10, 50, 100, 200, 300, 400, 500]

Each value in the list represents the number of nodes in a graph. These values are used to generate graphs of varying sizes for performance analysis of the Depth First Search (DFS) and Breadth First Search (BFS) algorithms.

IMPLEMENTATION

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

Depth First Search(DFS):

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.

Algorithm Description:

Step1: Initially stack and visited arrays are empty.

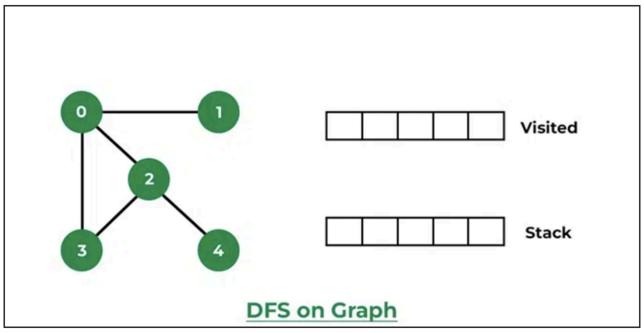


Figure 1 Stack and visited arrays are empty initially.

Step 2: Visit 0 and put its adjacent nodes which are not visited yet into the stack.

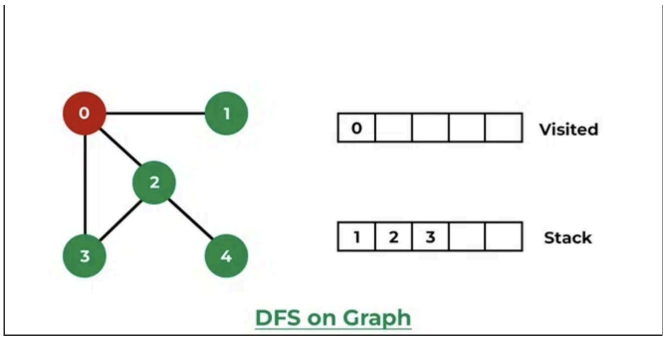


Figure 2 Visit node 0 and put its adjacent nodes (1, 2, 3) into the stack

Step 3: Now, Node 1 at the top of the stack, so visit node 1 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

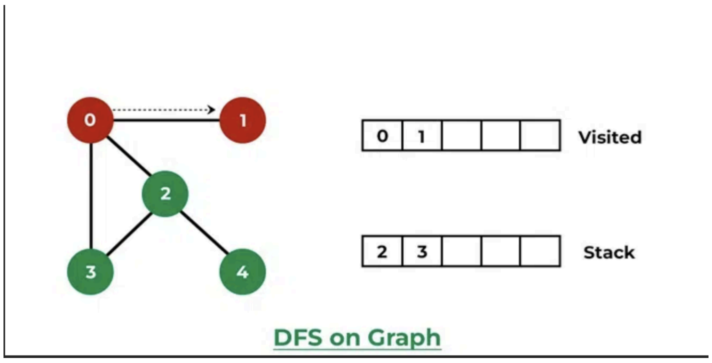


Figure 3 Visit node 1

Step 4: Now, Node 2 at the top of the stack, so visit node 2 and pop it from the stack and put all of its adjacent nodes which are not visited (i.e, 3, 4) in the stack.

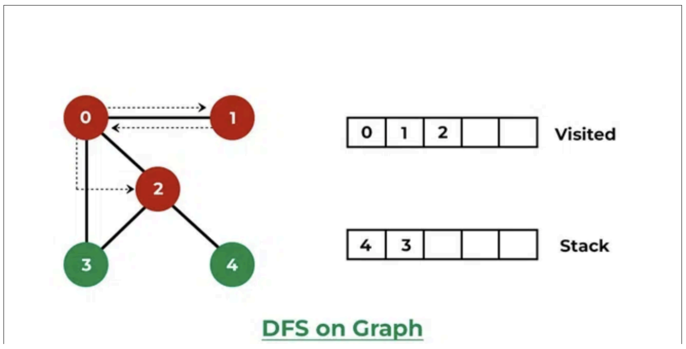


Figure 4 Visit node 2 and put its unvisited adjacent nodes (3, 4) into the stack

Step 5: Now, Node 4 at the top of the stack, so visit node 4 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

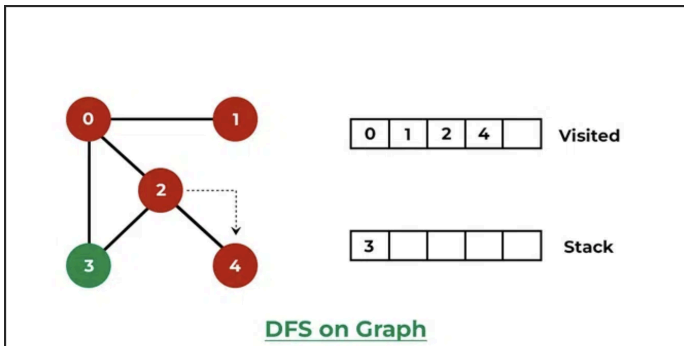


Figure 5 Visit node 4

Step 6: Now, Node 3 at the top of the stack, so visit node 3 and pop it from the stack and put all of its adjacent nodes which are not visited in the stack.

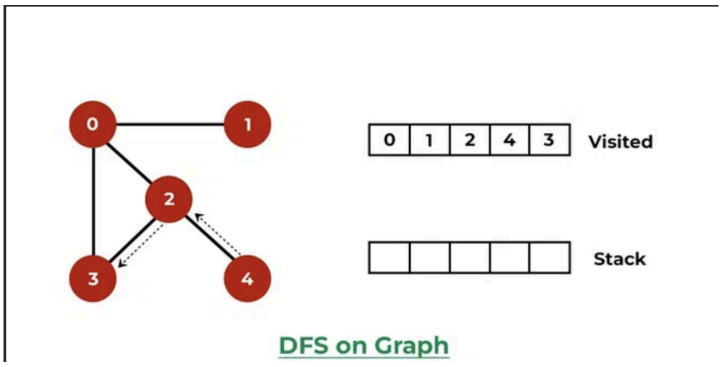


Figure 6 Visit node 3

Now, Stack becomes empty, which means we have visited all the nodes and our DFS traversal ends.

Complexity Analysis of Depth First Search(DFS):

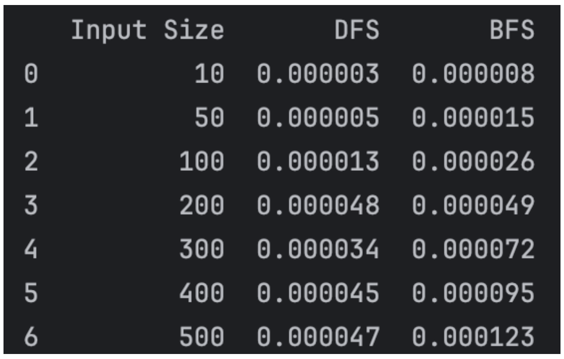
Time complexity: O(V + E), where V is the number of vertices and E is the number of edges in the graph.

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

Implementation:

```
from collections import defaultdict
class Graph_DFS:
    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)
```

Results:



```
   Input Size       DFS       BFS
0          10  0.000003  0.000008
1          50  0.000005  0.000015
2         100  0.000013  0.000026
3         200  0.000048  0.000049
4         300  0.000034  0.000072
5         400  0.000045  0.000095
6         500  0.000047  0.000123
```

Figure 7 Results of program

As the input size increases from 10 to 500, the execution time of DFS shows a slight fluctuation, but overall remains relatively low. The execution time of DFS appears to increase slightly with larger input sizes, indicating a mild increase in computational complexity. DFS traversal is efficient for smaller graphs (up to 200 nodes), with execution times consistently below 0.00005 seconds. However, for larger graphs (300 nodes and above), the execution time of DFS shows a slight increase, reaching up to 0.000047 seconds for 500 nodes.

Overall, DFS demonstrates consistent and efficient performance across varying input sizes, making it suitable for applications where exploration of deep branches is essential.

Graph:

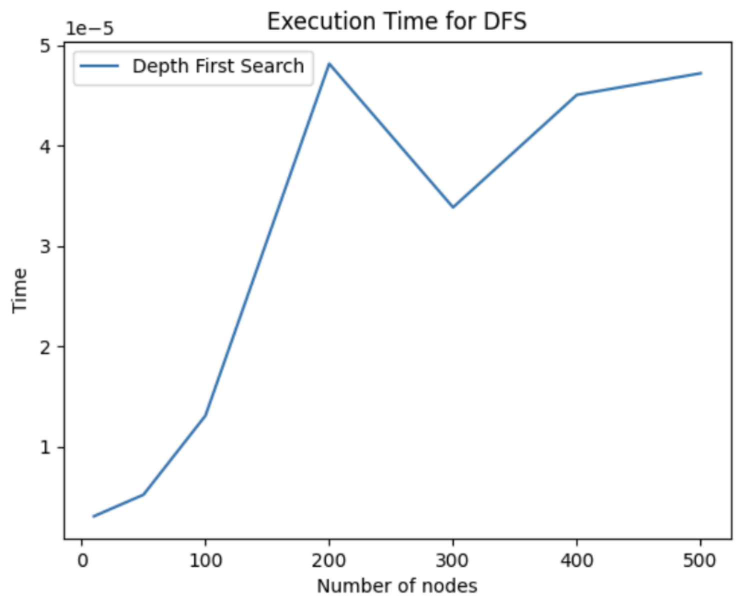


Figure 8 Graphical Representation

The graph representing the execution times for DFS traversal shows a relatively stable trend with increasing input sizes. The execution times for DFS traversal remain consistently low across all input sizes, indicating efficient performance. As the input size increases, there is a slight fluctuation in execution times, but the overall trend suggests that DFS traversal maintains its efficiency. DFS traversal is well-suited for applications where exploration of deep branches is essential, as it efficiently traverses through the graph depth-first.

Breadth First Search(BFS):

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 neighbours before moving on to the next level of neighbours. BFS is commonly used in algorithms for pathfinding, connected components, and shortest path problems in graphs.

Algorithm Description:

1. Initialization: Enqueue the starting node into a queue and mark it as visited.
2. Exploration: While the queue is not empty:
    - Dequeue a node from the queue and visit it (e.g., print its value).
    - For each unvisited neighbour of the dequeued node:
        - Enqueue the neighbour into the queue.
        - Mark the neighbour as visited.
3. Termination: Repeat step 2 until the queue is empty.

This algorithm ensures that all nodes in the graph are visited in a breadth-first manner, starting from the starting node.

Step1: Initially the queue and visited arrays are empty.

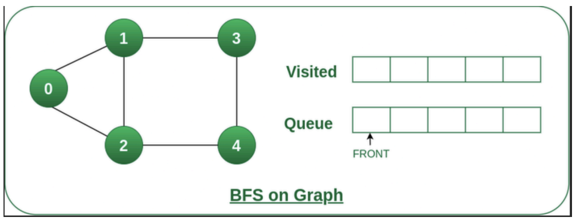


Figure 9 Queue and visited arrays are empty initially.

Step 2: Push node 0 into the queue and mark it visited.

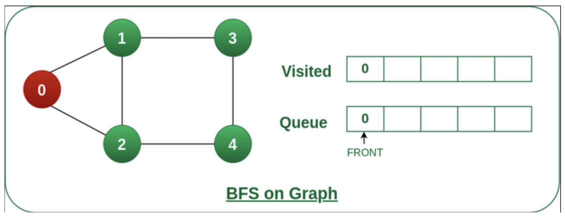


Figure 10 Push node 0 into the queue and mark it visited.

Step 3: Remove node 0 from the front of the queue and visit the unvisited neighbours and push them into the queue.

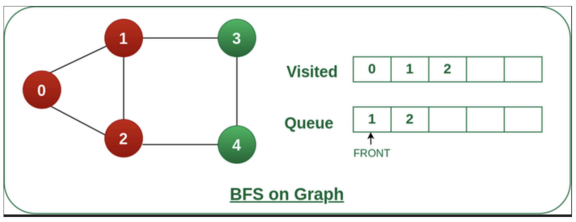


Figure 11 Remove node 0 from the front of the queue and visit the unvisited neighbours and push into the queue.

Step 4: Remove node 1 from the front of the queue and visit the unvisited neighbours and push them into the queue.

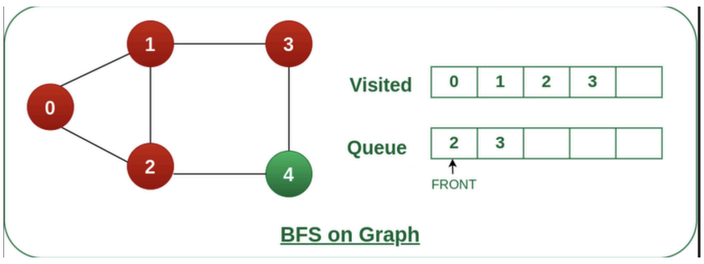


Figure 12 Remove node 1 from the front of queue and visited the unvisited neighbours and push

Step 5: Remove node 2 from the front of the queue and visit the unvisited neighbours and push them into the queue.

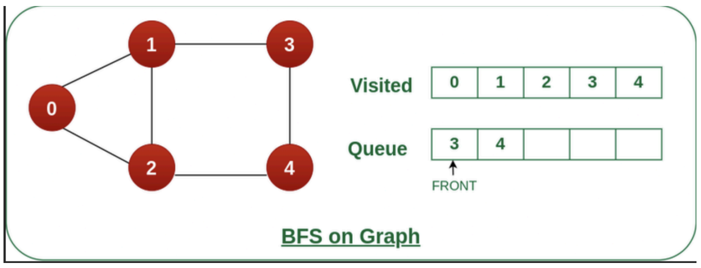


Figure 13 Remove node 2 from the front of the queue and visit the unvisited neighbours and push them into the queue.

Step 6: Remove node 3 from the front of the queue and visit the unvisited neighbours and push them into the

queue.
As we can see that every neighbour of node 3 is visited, so move to the next node that is in the front of the queue.

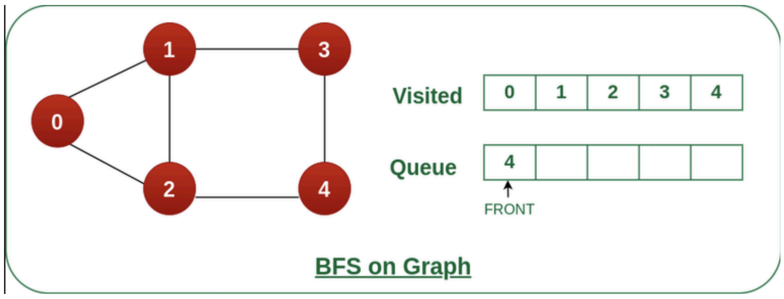


Figure 14 Remove node 3 from the front of the queue and visit the unvisited neighbours and push them into the queue.

Step *7:* Remove node 4 from the front of the queue and visit the unvisited neighbours and push them into the queue.
As we can see that every neighbour of node 4 is visited, so move to the next node that is in the front of the queue.

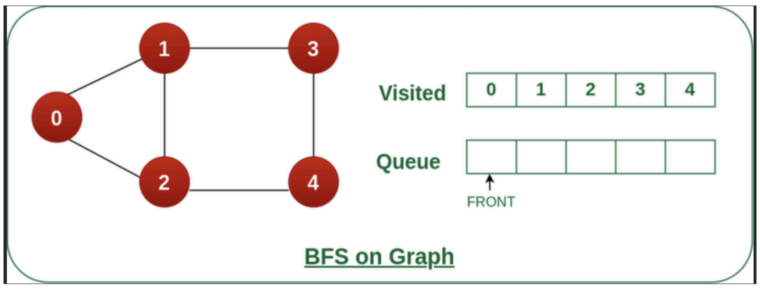


Figure 15 Remove node 4 from the front of the queue and visit the unvisited neighbours and push them into the queue.

Now, Queue becomes empty, So, terminate this process of iteration.

Complexity Analysis of Breadth First Search(BFS):

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)

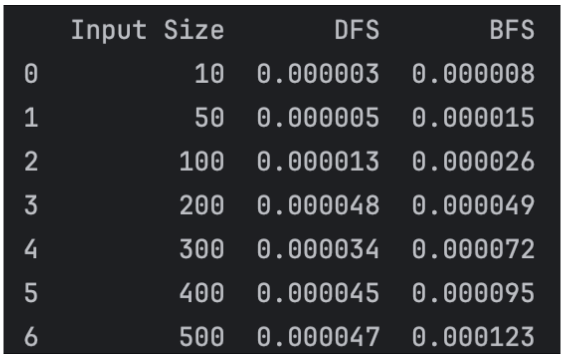
BFS uses a queue to keep track of the vertices that need to be visited. In the worst case, the queue can contain all the vertices in the graph. Therefore, the space complexity of BFS is O(V), where V and E are the number of vertices and edges in the given graph.

Implementation:

```
from collections import defaultdict, deque
class Graph_BFS:
    def __init__(self):
        self.adjList = defaultdict(list)
    def addEdge(self, u, v):
        self.adjList[u].append(v)
        self.adjList[v].append(u)
```

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

Results:



```
   Input Size       DFS       BFS
0          10  0.000003  0.000008
1          50  0.000005  0.000015
2         100  0.000013  0.000026
3         200  0.000048  0.000049
4         300  0.000034  0.000072
5         400  0.000045  0.000095
6         500  0.000047  0.000123
```

Figure 7 Results of program

The execution time of BFS exhibits a more noticeable increase as the input size grows from 10 to 500. BFS traversal generally takes longer than DFS, especially for larger graphs. Execution times for BFS range from 0.000008 seconds for 10 nodes to 0.000123 seconds for 500 nodes. There is a significant increase in execution time for BFS as the input size increases, indicating a higher computational complexity compared to DFS.

Despite the increased execution time, BFS offers the advantage of finding the shortest path in unweighted graphs, making it suitable for applications requiring pathfinding and network analysis.

Graph:

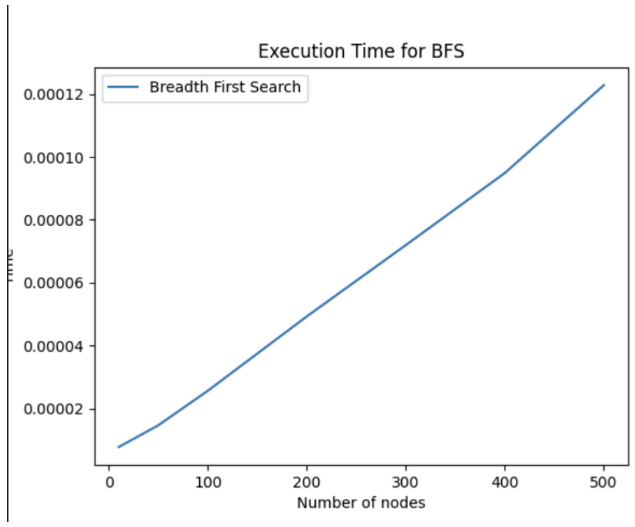


Figure 16 Graphical Representation

In contrast, the graph representing the execution times for BFS traversal displays a noticeable increase in execution times with larger input sizes. BFS traversal generally takes longer than DFS, especially for larger graphs, as indicated by the upward trend in execution times.The execution times for BFS traversal exhibit a steeper increase as the input size grows, highlighting its higher computational complexity compared to DFS.

Despite the increased execution times, BFS traversal offers the advantage of finding the shortest path in unweighted graphs, making it suitable for tasks requiring pathfinding and network analysis. However, this efficiency comes at the cost of longer execution times, particularly for larger graphs.

CONCLUSION

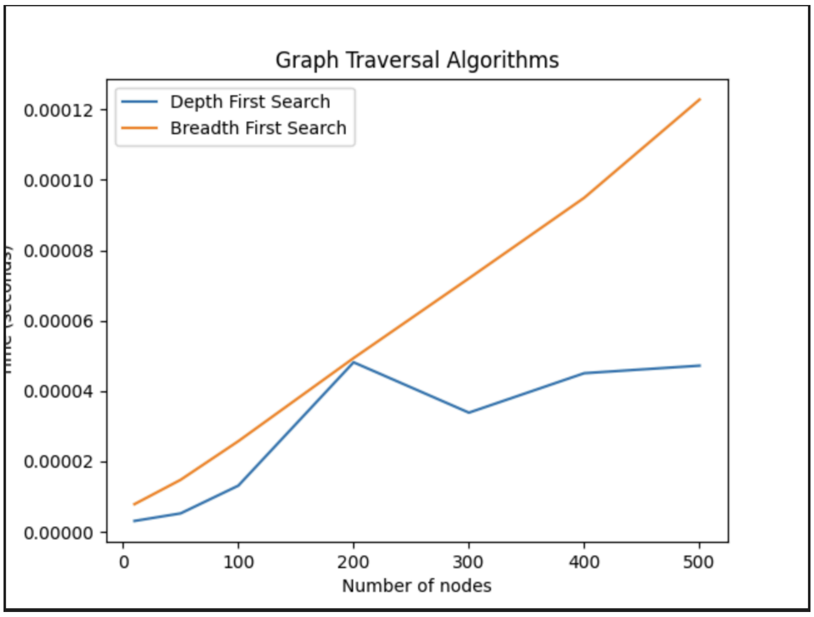


Figure 17 Graphical Representation Comparison

Depth First Search (DFS) and Breadth First Search (BFS) are fundamental graph traversal algorithms with distinct strategies and applications. Through empirical analysis, we have gained insights into the performance characteristics of DFS and BFS, allowing us to draw the following conclusions:

DFS Performance:
- DFS traversal demonstrates consistent and efficient performance across varying input sizes.
- The execution times for DFS remain relatively low, even for larger graphs, indicating its suitability for tasks where exploration of deep branches is essential.
- Despite a slight increase in execution times with larger input sizes, DFS maintains its efficiency and stability throughout the analysis.

BFS Performance:
- In contrast, BFS traversal exhibits a noticeable increase in execution times as the input size grows.
- BFS traversal generally takes longer than DFS, especially for larger graphs, highlighting its higher computational complexity.
- Despite longer execution times, BFS offers the advantage of finding the shortest path in unweighted graphs, making it suitable for tasks requiring pathfinding and network analysis.

Graphical Representation:
- The graphical representations of execution times for DFS and BFS traversal illustrate distinct trends.
- DFS traversal shows a relatively stable trend with minimal fluctuations in execution times, while BFS traversal displays a steeper increase in execution times with larger input sizes.

Algorithm Selection:

- The choice between DFS and BFS depends on the specific requirements of the problem and the characteristics of the input data.
- DFS is preferable for tasks where exploration of deep branches is critical and execution time is a priority.
- BFS is suitable for applications requiring shortest path calculations and network analysis, despite its higher computational complexity.

In conclusion, DFS and BFS are powerful tools for graph traversal and analysis, each offering unique advantages and performance characteristics. By understanding the behaviour and efficiency of these algorithms, we can make informed decisions when selecting the most suitable approach for solving graph-related problems.

References:

[1] GitHub Repository: Algorithms Analysis Laboratory Works, Retrieved from GitHub - Chiuliana/Algorithms_Analysis_Laboratory_Works