Ministerul Educaţiei şi 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 Cuzmin Simion

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:

Quick Sort is a comparison-based sorting algorithm with an average and best-case time complexity of O(nlogn) and a worst-case time complexity of O(n**2). It operates by selecting a pivot element from the array and partitioning the array into two sub-arrays based on this pivot. Elements less than the pivot are placed to its left, while elements greater than the pivot are placed to its right. This process is recursively applied to the sub-arrays until the entire array is sorted. Quick Sort is efficient for large datasets and is widely used due to its average-case time complexity.

Merge Sort is a stable, comparison-based sorting algorithm with a time complexity of O(nlogn) in all cases. It operates by recursively dividing the array into halves until each sub-array contains only one element. Then, it merges the sub-arrays in a sorted manner to produce a single sorted array. Merge Sort is known for its predictable performance and is often used in scenarios where stability and worst-case performance are crucial.

Heap Sort is an in-place, comparison-based sorting algorithm with a time complexity of O(nlogn) in all cases. It involves building a max-heap from the input array and repeatedly extracting the maximum element from the heap, which results in a sorted array. Heap Sort offers stable performance and is often used in scenarios where memory constraints are a concern.

Selection Sort is a simple comparison-based sorting algorithm with a time complexity of O(n**2). It iterates through the array, selecting the smallest (or largest) element each time and swapping it with the element in the current position. Although easy to implement, Selection Sort is less efficient than other sorting algorithms, especially for large datasets.

Introduction:

Depth First Search (DFS) and Breadth First Search (BFS) are cornerstone algorithms in the realm of computer science, specifically in the study and manipulation of graphs and trees. Both algorithms are designed to traverse or search these data structures in a systematic and exhaustive manner. DFS dives deep into a data structure by exploring as far down a branch as possible before backtracking, making it an ideal choice for tasks that require exploring all paths or finding specific solutions deep within a graph. BFS, on the other hand, explores all neighbors at a given depth before moving on to the nodes at the next depth level, making it excellent for finding the shortest path between nodes or conducting a level-order traversal of a tree.

DFS's Utility:

Path Finding: DFS is particularly useful in scenarios where we are looking for a path between two points, and all potential paths need to be explored. Its ability to dive deep into a graph makes it suitable for applications like maze solving, where reaching the end point is the goal, regardless of the path taken.

Puzzle Solving: In puzzles where finding any solution is more important than finding the optimal one, DFS can be a powerful tool. It can systematically explore different states of the puzzle until it finds a solution.

Topological Sorting: DFS is instrumental in applications requiring an ordering of elements where certain elements must precede others. This is often seen in task scheduling and resolving dependencies in package management systems

BFS's Effectiveness:

Finding the Shortest Path: BFS excels in finding the shortest path between two nodes in a graph. This property is invaluable in numerous applications, such as routing and navigation systems, where the shortest route between two locations needs to be determined.

Level-Order Traversal: In tree data structures, BFS is used to traverse the nodes level by level. This approach is essential in scenarios where the hierarchy or level of nodes plays a crucial role, such as in organizing data in hierarchical systems or for breadth-first parsing of hierarchical structures like HTML DOM.

Comparison Metric:

The primary metric used for evaluating the performance of the algorithms is the execution time (T(n)). Execution time refers to the duration taken by an algorithm traversing through a graph, typically measured in seconds. By comparing the execution times of different algorithms across varying input sizes, we gain insights into their efficiency and scalability.

Input Format:

The input to each algorithm consists of random generated graphs. The number of edges is chosen by

random, while the number of vertices is consisted of an array: **vertices_number** = [10, 100, 200, 300, 400, 500, 600, 700, 800, 900]. In the end we will compare 10 graphs with number of vertices increasing and edges random between them.

IMPLEMENTATION

Algorithms:

```
class Graph:

    # Constructor
    russian17
    def __init__(self):

        # Default dictionary to store graph
        self.graph = defaultdict(list)

    # Function to add an edge to graph
    1 usage   russian17
    def addEdge(self, u, v):
        self.graph[u].append(v)

    # A function used by DFS
    2 usages   russian17
    def DFSUtil(self, v, visited):

        # Mark the current node as visited
        # and print it
        visited.add(v)
        print(v, end=' ')

        # Recur for all the vertices
        # adjacent to this vertex
        for neighbour in self.graph[v]:
            if neighbour not in visited:
                self.DFSUtil(neighbour, visited)
```

```python
def bfs(self, startNode):
    # Create a queue for BFS
    queue = deque()
    visited = [False] * (max(self.graph.keys()) + 1)

    # Mark the current node as visited and enqueue it
    visited[startNode] = True
    queue.append(startNode)

    # Iterate over the queue
    while queue:
        # Dequeue a vertex from queue and print it
        currentNode = queue.popleft()
        print(currentNode, end=" ")

        # Get all adjacent vertices of the dequeued vertex currentNode
        # If an adjacent has not been visited, then mark it visited and enqueue it
        for neighbor in self.graph[currentNode]:
            if not visited[neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)
```

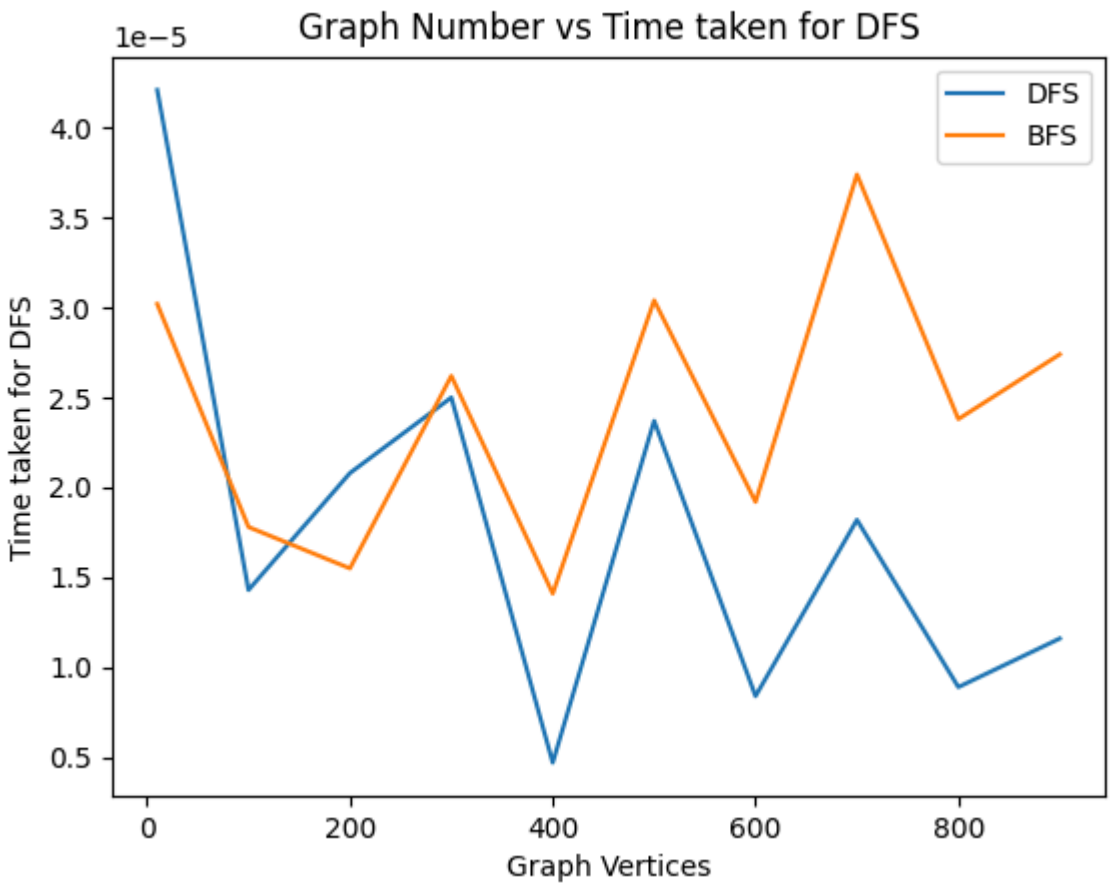
```python
def DFS(self, v):

    # Create a set to store visited vertices
    visited = set()

    # Call the recursive helper function
    # to print DFS traversal
    self.DFSUtil(v, visited)
```

Comparison:

```
graphs = []
vertices_number = [10, 100, 200, 300, 400, 500, 600, 700, 800, 900]
graph_vertices = []
# Creating graphs
for i in range(10):
    g = Graph()  # Create a new Graph instance
    num_vertices = vertices_number[i]
    graph_vertices.append(num_vertices)
    # To ensure there are as many vertices as edges, we create a pair of edges for each vertex
    for j in range(num_vertices):
        # Add edges with random connections ensuring we do not exceed the vertices count
        u = random.randint( a: 0, num_vertices - 1)
        v = random.randint( a: 0, num_vertices - 1)
        g.addEdge(u, v)

    graphs.append(g)

# Plotting with graphviz
for i, g in enumerate(graphs):
    dot = graphviz.Digraph(comment='The Round Table')
    for node in g.graph:
        for neighbor in g.graph[node]:
            dot.edge(str(node), str(neighbor))
    dot.render('graph' + str(i))

timeDFS = []
timeBFS = []
```

```
# Iterating over each graph and measuring DFS execution time
for graph in graphs:
    start = time.perf_counter()
    graph.DFS(1)
    end = time.perf_counter()
    timeDFS.append(end - start)

    start = time.perf_counter()
    graph.bfs(1)
    end = time.perf_counter()
    timeBFS.append(end - start)


# Plotting the graph
plt.plot( *args: graph_vertices, timeDFS, label="DFS")
plt.plot( *args: graph_vertices, timeBFS, label="BFS")
plt.xlabel('Graph Vertices')
plt.ylabel('Time taken for DFS')
plt.title('Graph Number vs Time taken for DFS')
plt.legend()
plt.show()
```

Plots:



CONCLUSION

In conclusion, this laboratory work's analysis of DFS and BFS algorithms provides clear insights into their efficiency and suitability for different graph traversal tasks. Through testing on various graph sizes and densities, it was observed that DFS is highly effective for exhaustive search tasks, like finding paths in mazes or specific deep solutions, due to its depth-oriented strategy. BFS, with its approach of exploring all neighbors at a given depth before moving deeper, proves to be better for applications needing the shortest path solutions, such as routing or level-order data processing.

The comparison based on execution time showed the direct impact of graph characteristics on algorithm performance. While DFS is more suited to complex and dense graphs, BFS excels in sparse graphs where finding the shortest path is essential. These findings highlight the importance of choosing the right algorithm based on the problem's nature.

Graphical data presentation further illustrated how both algorithms scale with graph size, offering practical guidance for their application in real-world scenarios. This work underscores the necessity of algorithm selection in solving graph-related problems and enriches the understanding of DFS and BFS's operational strengths.

Overall, this study reinforces the significance of tailored algorithm selection in computational tasks involving graph theory and provides a foundation for future exploration in algorithm optimization and application.

The link to github: https://github.com/russian17/AA/tree/main/Lab3