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 Lupan Lucian

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2024

Contents

ALGORITHM ANALYSIS

Study DFS and BFS and analyze the algorithms empirically.

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:

Graph search algorithms form the backbone of many applications, from social network analysis and route planning to data mining and recommendation systems. In this developer's guide, we will delve into the world of graph search algorithms, exploring their definition, significance, and practical applications.

At its core, a graph search algorithm is a technique used to traverse a graph, which is a collection of nodes connected by relationships. In various domains such as social networks, web pages, or biological networks, graph theory offers a powerful way to model complex interconnections.

The significance of graph search algorithms lies in their ability to efficiently explore and navigate these intricate networks. By traversing the graph, these algorithms can uncover hidden patterns, discover the shortest paths, and identify clusters.

Graph search algorithms:

1. **Breadth-first search:**

Breadth-first search (BFS) is an important graph search algorithm that is used to solve many problems including finding the shortest path in a graph and solving puzzle games (such as Rubik's Cubes). Many problems in computer science can be thought of in terms of graphs. For example, analyzing networks, mapping routes, and scheduling are graph problems. Graph search algorithms like breadth-first search are useful for analyzing and solving graph problems.

Breadth-first search starts by searching a start node, followed by its adjacent nodes, then all nodes that can be reached by a path from the start node containing two edges, three edges, and so on. Formally, the BFS algorithm visits all vertices in a graph G that are k edges away from the source vertex s before visiting any vertex k+1 edges away. This is done until no more vertices are reachable from s. The image below demonstrates exactly how this traversal proceeds:

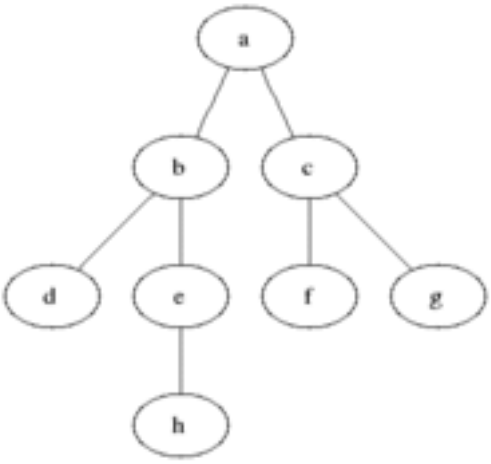

Figure 1. Breadth-first search algorithm representation.

For a graph G = (V, E) and a source vertex v, breadth-first search traverses the edges of G to find all reachable vertices from v. It also computes the shortest distance to any reachable vertex. Any path between two points in a breadth-first search tree corresponds to the shortest path from the root v to any other node s.

2. **Depth-first search:**

Depth-first search (DFS**)** is an algorithm for searching a graph or tree data structure. The algorithm starts at the root (top) node of a tree and goes as far as it can down a given branch (path), then backtracks until it finds an unexplored path, and then explores it. The algorithm does this until the entire graph has been explored. Many problems in computer science can be thought of in terms of graphs. For example, analyzing networks, mapping routes, scheduling, and finding spanning trees are graph problems. To analyze these problems, graph-search algorithms like depth-first search are useful.

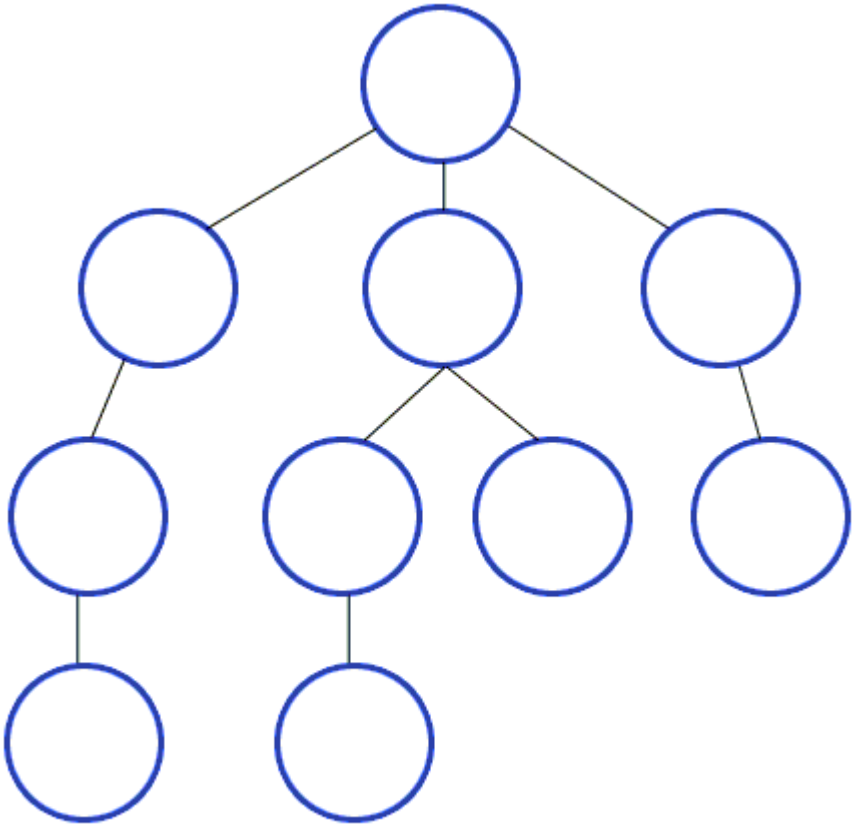

Figure 2. Depth-first search algorithm representation.

The main strategy of depth-first search is to explore deeper into the graph whenever possible. Depth-first search explores edges that come out of the most recently discovered vertex, s. Only edges going to unexplored vertices are explored. When all of s's edges have been explored, the search backtracks until it reaches an unexplored neighbor. This process continues until all of the vertices that are reachable from the original source vertex are discovered. If there are any unvisited vertices, depth-first search selects one of them as a new source and repeats the search from that vertex. The algorithm repeats this entire process until it has discovered every vertex. This algorithm is careful not to repeat vertices, so each vertex is explored once. DFS uses a stack data structure to keep track of vertices.

Empirical Comparison:

Empirical comparison involves testing sorting algorithms on real-world datasets to observe their performance characteristics. Here's why it is crucial:

1. **Practical Efficiency:**
    - Theoretical analyses provide insights into the worst-case, average-case, and best-case scenarios. However, real-world data often exhibits unique patterns that may affect the actual performance of an algorithm.
2. **Adaptability:**
    - Different sorting algorithms may excel in specific scenarios. Empirical comparison helps identify which algorithm performs best under particular conditions, enabling the selection of the most suitable algorithm for a given task.
3. **Implementation Factors:**
    - The actual implementation of an algorithm can impact its performance. Empirical comparisons help identify subtle implementation details or optimizations that can significantly influence the practical efficiency of an algorithm.
4. **Hardware Dependencies:**
    - The efficiency of sorting algorithms can be influenced by the underlying hardware architecture. Empirical testing on different machines and environments provides a more realistic understanding of their performance.
5. **Scaling Behavior:**
    - The scalability of sorting algorithms is crucial, especially when dealing with large datasets. Empirical studies reveal how algorithms scale as the size of the input data increases.
6. **Benchmarking:**
    - Empirical comparison facilitates benchmarking, which is essential for assessing the relative merits of various algorithms in a specific context. Benchmark results guide the selection of the most efficient sorting algorithm for a particular application.

Introduction:

Graph search algorithms are fundamental techniques employed to traverse and explore the vertices and edges of a graph efficiently. These algorithms play a pivotal role in various fields such as computer science, operations research, and social network analysis. Primarily, graph search algorithms aim to navigate through the vertices and edges of a graph to solve various problems, including pathfinding, reachability analysis, and network flow optimization.

Commonly used graph search algorithms include breadth-first search (BFS) and depth-first search (DFS), each with its unique characteristics and applications. BFS systematically explores the vertices of the graph level by level, making it suitable for finding shortest paths and exploring all connected components. Conversely, DFS explores as far as possible along each branch before backtracking, making it useful for topological sorting, cycle detection, and maze solving. These algorithms serve as foundational tools in graph theory and computational problem-solving, providing versatile solutions to a wide range of graph-related challenges.

Comparison Metric:

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

Input Format:

As input, the algorithms will take in a randomized graph data set, with an initial node_count of 1000, increment of 500 and test_cases of 20. The graph data types have randomly defined edges, with a 1/10 chance of the current vertex adding an edge to a random vertex that's already defined, another 1/10 chance of the next vertex adding an edge to another randomly defined vertex, and a 1/10 chance of adding an edge between 2 randomly chosen defined vertices.

IMPLEMENTATION

Both of the algorithms are implemented in Python, making use of the random, time, collections and matplotlib.pyplot libraries in order to both ensure the functionality of the algorithms, analyze the algorithms empirically and visually representing the execution times.

Depth-first search:

Algorithm Description:

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.

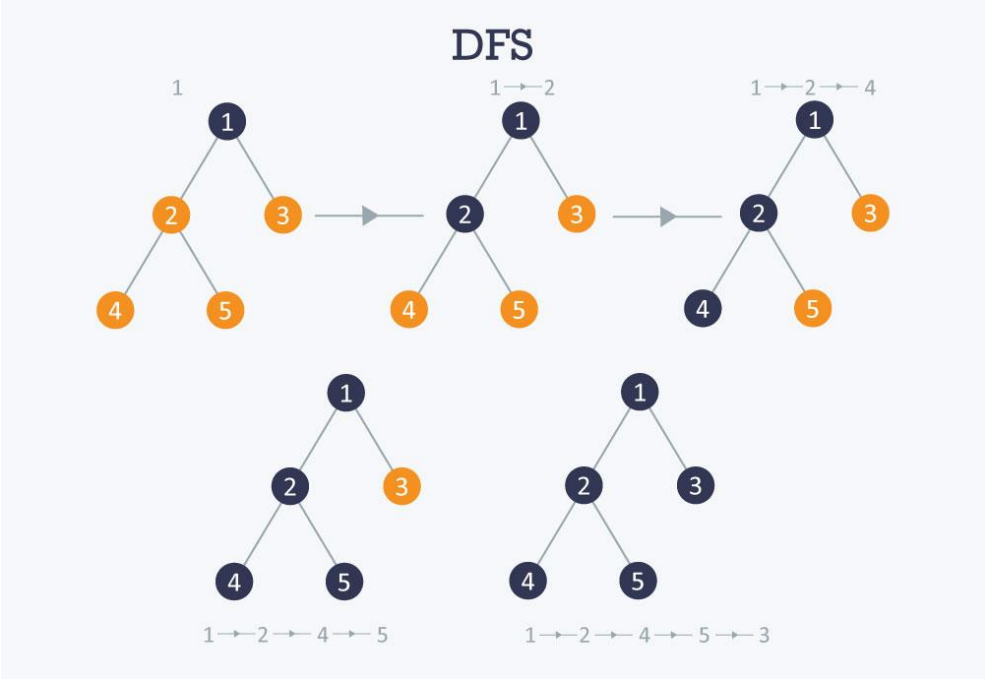

Figure 3. DFS algorithm explanation.

The complexity of DFS is as follows:

- **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:

```
class Graph:
    def __init__(self):
        self.edges = {}

    def addEdge(self, a, b):
        if a not in self.edges:
            self.edges[a] = []
        self.edges[a].append(b)

    def dfs(self, startNode):
        visited = set()
        stack = deque([startNode])

        while stack:
            node = stack.pop()
            if node in visited:
                continue
            if node not in self.edges:
                continue
            visited.add(node)
            for neighbor in reversed(self.edges[node]):
                if neighbor not in visited:
                    stack.append(neighbor)
```

Figure 4. DFS code in Python.

Breadth-first search:
Algorithm Description:

Merge sort is defined as a sorting algorithm that works by dividing an array into smaller subarrays, sorting each subarray, and then merging the sorted subarrays back together to form the final sorted array.

In simple terms, we can say that the process of merge sort is to divide the array into two halves, sort each half, and then merge the sorted halves back together. This process is repeated until the entire array is sorted.

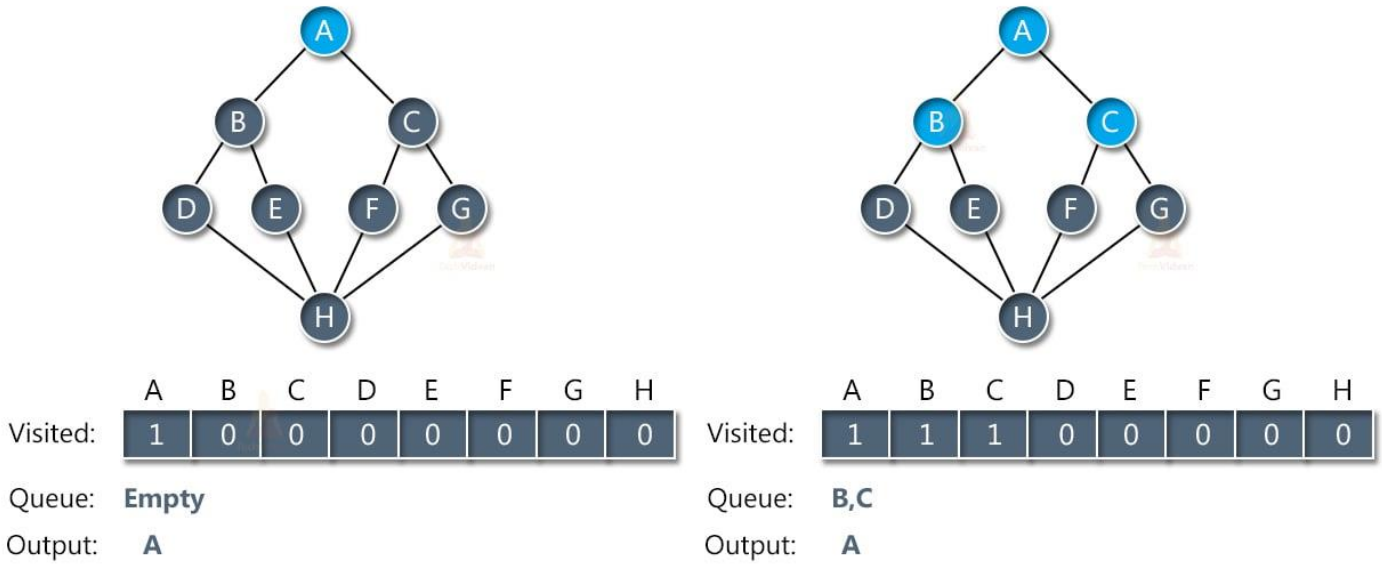

Figure 5. BFS Algorithm Explanation.

The complexity of BFS is as follows:

- **Time complexity:** O (V + E), where V and E are the number of vertices and edges in the given graph.
- **Space complexity:** O(V), where V and E are the number of vertices and edges in the given graph.

Implementation:

```
class Graph:
    def __init__(self):
        self.edges = {}

    def addEdge(self, a, b):
        if a not in self.edges:
            self.edges[a] = []
        self.edges[a].append(b)

    def bfs(self, startNode):
        if startNode not in self.edges:
            return "No such Node"

        visited = set()
        queue = deque([startNode])

        while queue:
            node = queue.popleft()
            visited.add(node)

            for neighbor in self.edges[node]:
                if neighbor not in visited:
                    queue.append(neighbor)
                    visited.add(neighbor)
```

Figure 6. BFS Code in Python.

Algorithm comparison results:

After performing the results on randomized datasets (according to the input data), saving the execution times, and plotting them on a graph, the results are as follows:

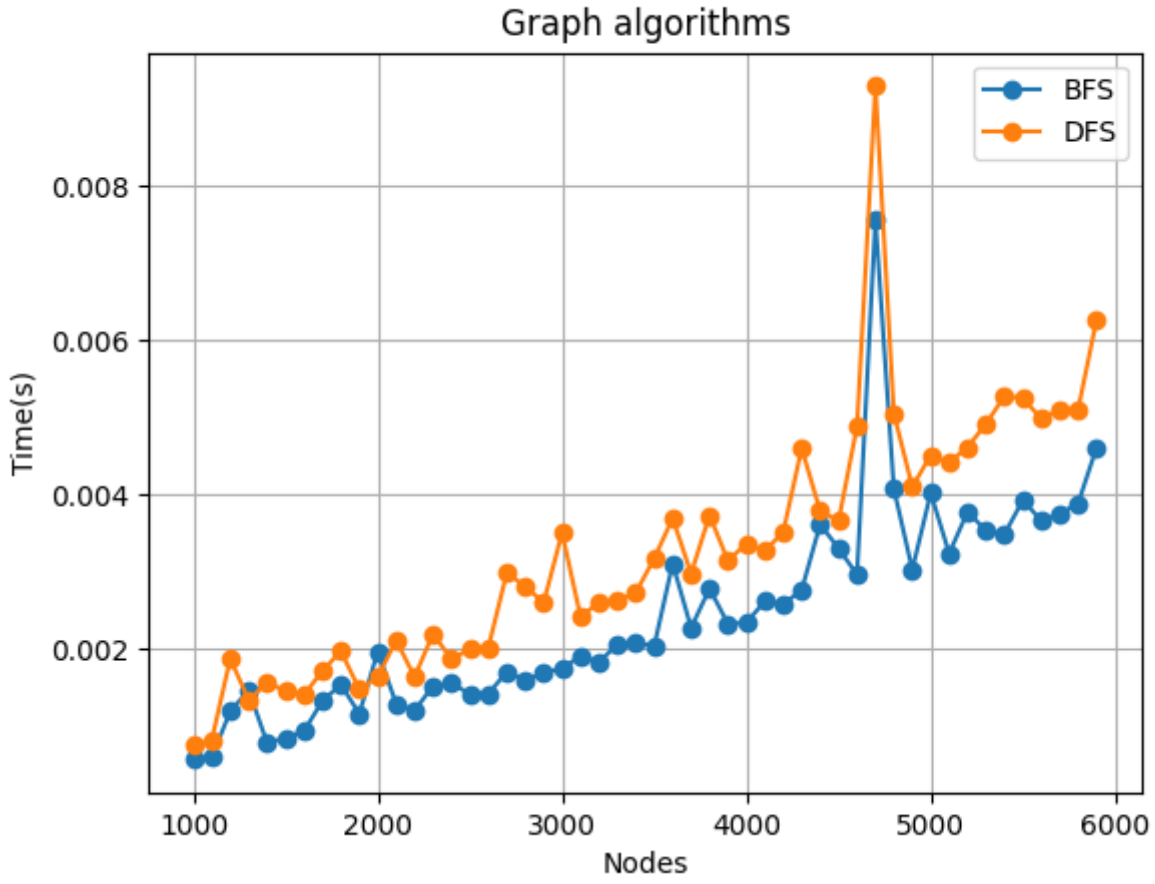

Figure 7. Algorithm comparison, first execution.

In figure 7, we can see that both DFS and BFS run very fast. Even with 3000 vertices, the algorithms run incredibly fast, as both of their complexities are relatively similar - O (V + E). From these results, we can also draw the conclusion that DFS is a bit faster than BFS.

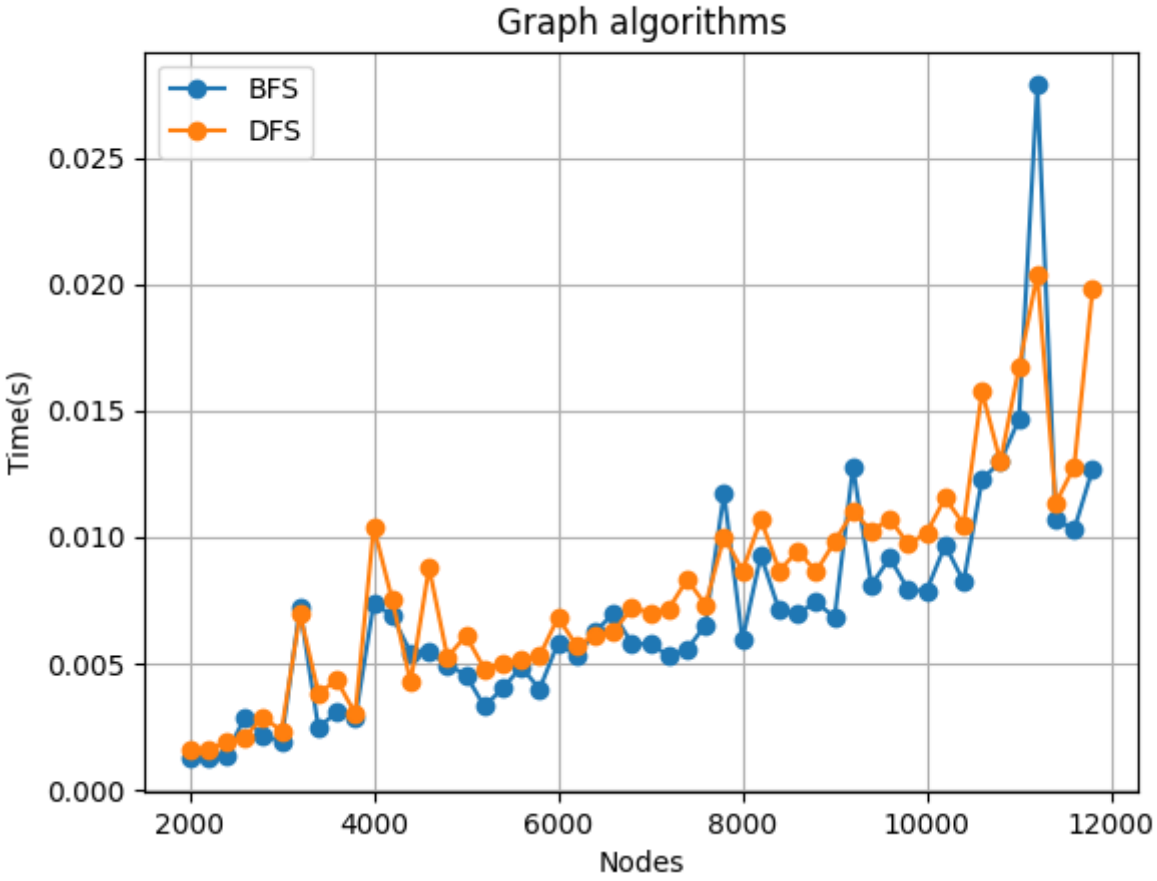

Figure 8. Algorithm comparison, second execution.

In figure 8, we operate on a larger data set, using up to a graph with 12000 nodes to compute the execution times of the DFA and BFS algorithms. We can see that the execution times are roughly the same, still incredibly fast, with the BFS algorithm being slightly faster than DFS on average.

CONCLUSION

After finishing this laboratory work, I learnt about the 2 most popular graph search algorithms – depth first search and breadth first search. I also learnt more about Python programming, improved my problem-solving skills when it comes to dealing with randomly creating graphs, appending edges, and ensuring there's no repeats in the connections between the vertices.

All the algorithms for this laboratory work ran at O (V+E) time, with V representing the number of vertices and E – the number of edges. All the algorithms ran incredibly fast, with DFS being slightly faster (on average, about 5-10%) than BFS. It's also important to note that although both algorithms basically focus on the same thing, their implementation for real-world tasks differs. For example, BFS could be used to find the shortest path between nodes in an unweighted graph, or for network routing, whereas DFS is used for topological sorting and scheduling problems.

One last observation for this laboratory work is that when the data set is increased – i.e., increasing the number of nodes, the algorithms computation times scale with each other, and most of the program time is spent on creating the graph objects themselves. For example, going from 2000 nodes to 12000 nodes increases the computation time by about 5 times (to ~0.010 seconds), which is still incredibly fast.

GitHub repository link - https://github.com/Lucian-Lu/AA-Labs