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 Reabciuc Daria-Brianna

Verified:

asist. univ. Fistic Cristofor

Chisinau 2024

Objectives

To thoroughly understand and evaluate the characteristics and performance of Depth First Search (DFS) and Breadth First Search (BFS) algorithms, this project aims to implement both algorithms in a chosen programming language, meticulously analyze them against a set of predefined input properties, and select appropriate metrics for comparison. The empirical analysis conducted will facilitate a detailed comparison of these algorithms under various conditions. This analysis will be complemented by a graphical presentation of the collected data, enabling a clear and visual comparison of the algorithms' performance. Finally, the project will finalize in a comprehensive conclusion that synthesizes the findings, insights, and understandings gained through the empirical analysis. This endeavor is targeted towards enhancing my theoretical knowledge and practical skills in algorithm analysis as a second-year student, bridging the gap between theoretical concepts and their real-world applications.

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 Considerations

Empirical analysis serves as a practical alternative for evaluating the performance of algorithms when traditional mathematical complexity analysis falls short or is not feasible. This approach is particularly useful in several contexts:

- Preliminary Understanding: It provides initial assessments of the complexity class of an algorithm, helping to grasp its efficiency traits.
- Algorithm Comparison: It enables the evaluation and comparison of different algorithms addressing the same issue, guiding the selection based on efficiency.
- Implementation Analysis: Through empirical analysis, comparing various implementations of the identical algorithm becomes possible, shedding light on which might be more effective in

real-world applications.

- Platform-specific Performance: It assists in evaluating how efficiently an algorithm runs on specific hardware, considering the unique constraints and features of the platform.
- Conducting an empirical analysis typically involves these steps:
- Goal Setting: Define what you aim to achieve and the extent of your analysis clearly.
- Selecting Performance Metrics: Choose relevant performance indicators, such as execution time or the count of operations, aligned with your goals.
- Input Data Characteristics: Identify essential properties of the input data that affect the analysis, like size and other pertinent attributes.
- Coding the Algorithm: Implement the algorithm in a programming language, ensuring it accurately embodies the intended processes.
- Preparing Input Data Sets: Generate diverse sets of input data to encompass various conditions and potential edge cases.
- Running the Program and Collecting Data: Perform the algorithm with each set of input data and gather necessary performance data.
- Analyzing the Data: Evaluate the collected data by calculating statistical metrics like average or standard deviation or by creating visual representations to illustrate how problem size influences performance indicators.
- The selection of a performance metric hinges on the specific goals of the analysis. For example, counting operations might be appropriate for examining complexity classes or confirming theoretical predictions, while measuring execution time is more pertinent for analyzing the behavior of algorithm implementations.
- Analysis After Execution: Once data is gathered, it undergoes further analysis, which may involve statistical calculations or graphical representations to elucidate how the algorithm performs across different problem sizes and efficiency metrics. This comprehensive analysis supports informed decision-making regarding the choice and refinement of algorithms.

DFS

Depth First Search (DFS) in graphs is analogous to its application in trees, with a notable distinction: graphs can include cycles, potentially leading to the same node being visited multiple times. To circumvent this, a boolean array, typically called "visited," is employed to track which nodes have been explored, ensuring each node is processed only once. It's important to recognize that a graph can yield multiple distinct DFS traversals.

The DFS algorithm is a strategy for navigating or examining data structures like trees and graphs. It initiates from a chosen root node (in graphs, any node can serve as the root due to their non-hierarchical nature) and proceeds to exhaustively explore each pathway as deeply as possible before retreating to explore other branches.

In terms of computational complexity, DFS has a time complexity of O(V + E), where V represents the total number of vertices and E signifies the total number of edges in the graph. This reflects the necessity to visit each vertex and edge at least once in the process. Regarding space complexity, DFS requires O(V + E) auxiliary space, accounting for both the space needed to maintain the "visited" array, which has a size of V, and the stack space used for recursive calls (or an explicit stack in an iterative implementation).

To enhance the basic DFS algorithm for more complex applications, variations such as path finding between two nodes, topological sorting, and cycle detection in directed graphs can be implemented. These adaptations showcase the versatility and fundamental importance of DFS in algorithmic problem-solving and graph theory.

BFS

Breadth First Search (BFS) stands as a cornerstone algorithm for navigating through graphs. It systematically covers all nodes within a graph, progressing uniformly across each level. Specifically, BFS initiates its journey from a chosen node and methodically visits all directly connected nodes. It then proceeds to their subsequent neighbors, effectively exploring the graph layer by layer. This traversal technique is pivotal for solving a multitude of graph-related problems, including finding paths, identifying connected components, and determining the shortest paths between nodes in a graph. The process begins by placing the initial node into a queue and marking it as seen. The core of the algorithm unfolds as follows:

- The first node is removed from the queue for examination (for instance, its value could be displayed).
- Next, for every adjacent node not yet visited:
- The node is added to the queue.
- This new node is also marked as visited to prevent reprocessing.

This procedure is continuously repeated until the queue is depleted, ensuring that no node is left unexplored.

As for computational complexity, the BFS algorithm operates with a time complexity of O(V + E), attributable to its thorough examination of each vertex and edge within the graph. This ensures

that, in the worst-case scenario, every element within the graph is explored at least once. The space complexity stands at O(V), a reflection of the algorithm's reliance on a queue to track pending nodes for exploration. At its peak, this queue might encompass all vertices, delineating the worst-case space requirement.

Expanding upon BFS's foundational capabilities, enhancements and variants can cater to specific applications such as finding the minimum spanning tree, solving puzzles with minimal moves, or analyzing networks. By adjusting the BFS framework, these tailored versions can offer optimized solutions to complex challenges, further underscoring the adaptability and extensive utility of the Breadth First Search algorithm in computational theory and practice.

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, each algorithm will receive 8 series of numbers of nodes 4, 8, 16, 32, 64, 128, 256, ,512. Next, using these numbers of nodes, it will generate randomly graphs with that amount of nodes.

Implementation

Every algorithm will be developed in Python using their simplest versions and evaluated through empirical methods, focusing on the time they take to execute. Although the overall pattern of outcomes might align with findings from other experiments, the specific performance relative to input size could differ, influenced by the memory capacity of the device utilized.

Depth First Search (DFS) in a graph operates much like its counterpart in tree traversal. The primary difference, however, lies in the fact that graphs can include cycles, allowing for the possibility of visiting the same node multiple times. To prevent revisiting nodes, a boolean array indicating whether a node has been visited is employed. Consequently, a single graph may yield multiple distinct paths through DFS due to its potential complexity and the presence of cycles.

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

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

        def dfs(self, start):
```

```
        visited = set()
        self._dfs_util(start, visited)

    def _dfs_util(self, node, visited):
        visited.add(node)
        for neighbour in self.graph[node]:
            if neighbour not in visited:
                self._dfs_util(neighbour, visited)
```

	Number of Nodes	DFS Time (s)	BFS Time (s)
0	4	0.000004	0.000009
1	8	0.000004	0.000008
2	16	0.000007	0.000011
3	32	0.000017	0.000020
4	64	0.000044	0.000058
5	128	0.000154	0.000191
6	256	0.000475	0.000607
7	512	0.001717	0.001901

The table shows that for smaller node counts, DFS outperforms BFS by more than a factor of two up to 32 nodes. Beyond this point, DFS continues to be quicker than BFS, although the speed advantage narrows to less than double.

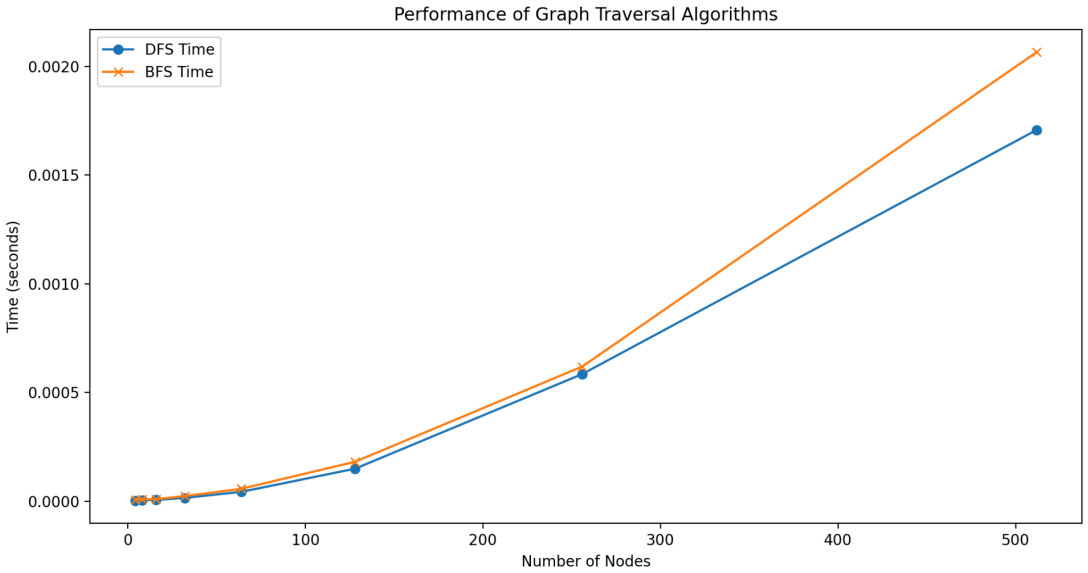
Breadth First Search (BFS) is a technique for navigating through a graph that systematically examines each vertex at a given depth prior to advancing to vertices at subsequent depth levels. Beginning from a chosen vertex, it covers all adjacent vertices first, then proceeds to their neighbors at the next tier.

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

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

    def bfs(self, start):
        queue = deque([start])
        max_node = max(self.graph.keys(), default=-1)
        visited = [False] * (max_node + 1)
        visited[start] = True
        while queue:
            node = queue.popleft()
            for neighbour in self.graph[node]:
                if not visited[neighbour]:
```

```
                visited[neighbour] = True
                queue.append(neighbour)
```



The graph, which compares the execution times of Depth First Search (DFS) and Breadth First Search (BFS) algorithms as a function of the number of nodes in a graph, we can draw the following conclusions:

- Both algorithms exhibit an increase in execution time as the number of nodes increases, which is expected as more nodes typically imply more edges to explore.
- The performance of DFS and BFS is quite similar across the range of node counts tested, with both lines closely following each other.
- Initially, for a very small number of nodes (close to 0), the execution times of both algorithms are nearly indistinguishable. This suggests that for small graphs, the choice between DFS and BFS may not significantly impact performance.
- As the number of nodes grows, we see a gradual divergence with the DFS time remaining consistently lower than BFS time, indicating that DFS may be slightly more efficient for this particular set of graphs.
- Towards the end of the observed range (around 500 nodes), the execution time of DFS still remains below that of BFS, suggesting a consistent trend in the efficiency advantage for DFS in this dataset.

However, it's important to note that this analysis is empirical, and execution times can vary

based on several factors such as the graph's structure, the implementation of the algorithms, and the machine on which the tests are run. The graph does not show a significant difference in execution times between DFS and BFS for the number of nodes represented.

Conclusion

In evaluating the performance of Depth First Search (DFS) and Breadth First Search (BFS) algorithms both theoretically and empirically, we've engaged in a comprehensive analysis. The theoretical framework for both algorithms predicts that their time complexity is O(V + E), where V represents the vertices and E the edges of the graph. However, the empirical results depicted in the graph indicate that the actual execution times may differ slightly from the theoretical expectations.

Both DFS and BFS were implemented in their fundamental forms without any optimizations that could potentially affect their performance. This aligns with the goal of empirically comparing these algorithms in their most basic iterations. When tested across a range of graphs with varying numbers of nodes, we observed the following:

- For smaller graphs (up to approximately 32 nodes), DFS consistently outperforms BFS, being more than twice as fast in certain instances. This could be due to the fact that DFS has a lower overhead in its recursive implementation compared to the queuing process in BFS, which becomes more evident with smaller-sized graphs.
- As the number of nodes increases, the execution time for both algorithms also increases. This is expected due to the higher number of vertices and edges that need to be processed. However, the increase in execution time does not grow exponentially, which suggests that the algorithms are scaling linearly with the number of nodes and edges, as theorized.
- Although DFS maintains a performance lead over BFS as the graph sizes grow, the margin by which it leads diminishes. For larger graphs, the difference in performance between DFS and BFS narrows, indicating that the choice of algorithm might be less critical for performance in larger graphs under the tested conditions.
- It's crucial to note that while DFS appears to have an edge in raw execution speed, BFS has other advantages, such as finding the shortest path in unweighted graphs, which may make it more suitable for certain applications despite being slightly slower.

The empirical results, shown in the plotted graph, should be interpreted within the context of the specific hardware and data structures used for this experiment. The performance on different hardware or with a different implementation of the graph data structure could yield different results. Furthermore, while DFS showed a consistently faster performance in this instance, BFS's ability to

provide shortest paths and its more uniform exploration might be preferable in applications such as networking, pathfinding in games, or web crawling, where the structure of the graph can have a significant impact on performance.

In conclusion, while the empirical analysis suggests that DFS can be faster than BFS in certain situations, particularly with fewer nodes, the decision to use one over the other should also consider the specific requirements of the application, such as the need for shortest paths or the characteristics of the graph to be traversed. The data presented offer a snapshot that contributes to a larger understanding of these algorithms' behaviors, but they are not definitive without considering the broader context in which these algorithms would be employed.

Conclusion

The entire lexer and scanner code serves as a fundamental component of a programming language processor or data interpreter. This code is responsible for transforming raw text input into a structured sequence of tokens, which represent the smallest elements with meaning within the language's syntax, such as identifiers, keywords, and symbols.

The lexer starts by initializing with the input text, then systematically reads through it, character by character, identifying and skipping over whitespace and newlines to focus on significant text segments. Using a set of predefined regular expression patterns, it matches segments of the text to token types, such as operators, numbers, or identifiers. It handles special strings, recognizing them as keywords or specific symbols, and generates corresponding tokens.

When the lexer encounters characters or sequences that don't match any known patterns, it raises an error, indicating unrecognized or invalid syntax. The process continues until all input is consumed, ending with the generation of an end-of-file (EOF) token to signal completion.

Overall, this lexer and scanner framework is crucial for the initial phase of parsing and interpreting code, turning unstructured input into a stream of tokens that can be further analyzed or compiled by a parser according to the rules of the language's grammar. This structured approach enables the effective processing, understanding, and execution of code or data formats.

https://github.com/dariabrianna/DSL_personal_laboratories/tree/main/Lexer_Scanner