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

Elaborated:

st. gr. FAF-221   Clepa Rodion

Verified:

asist. univ.   Fiştic Cristofor

Chişinău – 2024

Contents

ALGORITHM ANALYSIS

Objective
Study and empirical analysis of graph traversal algorithm.

Tasks:
Implement the algorithms listed above in a programming language

Establish the properties of the input data against which the analysis is performed

Choose metrics for comparing algorithms

Perform empirical analysis of the proposed algorithms

Make a graphical presentation of the data obtained

Make a conclusion on the work done.

Theoretical Notes
In the empirical analysis of an algorithm, the following steps are usually followed:

The purpose of the analysis is established.

Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.

The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).

The algorithm is implemented in a programming language.

Run the program for each input data set.

The obtained data are analyzed.

Input Format
The algorithms will receive a random generated graph. The number of nodes will start from 100 and go up to 2000 by 100. To generate graph will be used an function created by where each edge have to connect with other 50%.

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

IMPLEMENTATION
All 2 algorithms will be implemented in their naive form in Python an analyzed empirically based on the time required for their completion. To represent connection between nodes will be used dictionary. Example: {0: [1, 2], 1: [3, 4], 2: [4]}

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.

```python
def bfs(self, startNode, toPrint = 0):
        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)
        if toPrint == 1:
            print(visited)
```

Figure 1 Code for BFS

Input Validation: Checks if the **startNode** exists in the graph. If not, it returns "No such Node".

Initialization: Initializes a list called visited with the **startNode**, which keeps track of visited nodes. Also initializes a queue with the **startNode**.

BFS Traversal: While the queue is not empty: Checks if the first node in the queue has neighbors. If it has neighbors iterates through each neighbor of the current node (if a neighbor has not been visited yet) marks it as visited. Appends it to both the visited list and the queue. Removes the current node from the queue.

```
100 - 0.00010209083557128906
200 - 0.0020752429962158205
300 - 0.004286503791809082
400 - 0.006279706954956055
500 - 0.00723416805267334
600 - 0.01127769947052002
700 - 0.016883111000061034
800 - 0.02193140983581543
900 - 0.02773587703704834
1000 - 0.03535020351409912
```

Figure 2 Results for first 10 inputs BFS

In Figure 2 is represented the table of results for the 10 set of inputs. The left column denotes the number of elements in array. On right column we get the number of seconds that elapsed from when the function was run till when the function was executed.
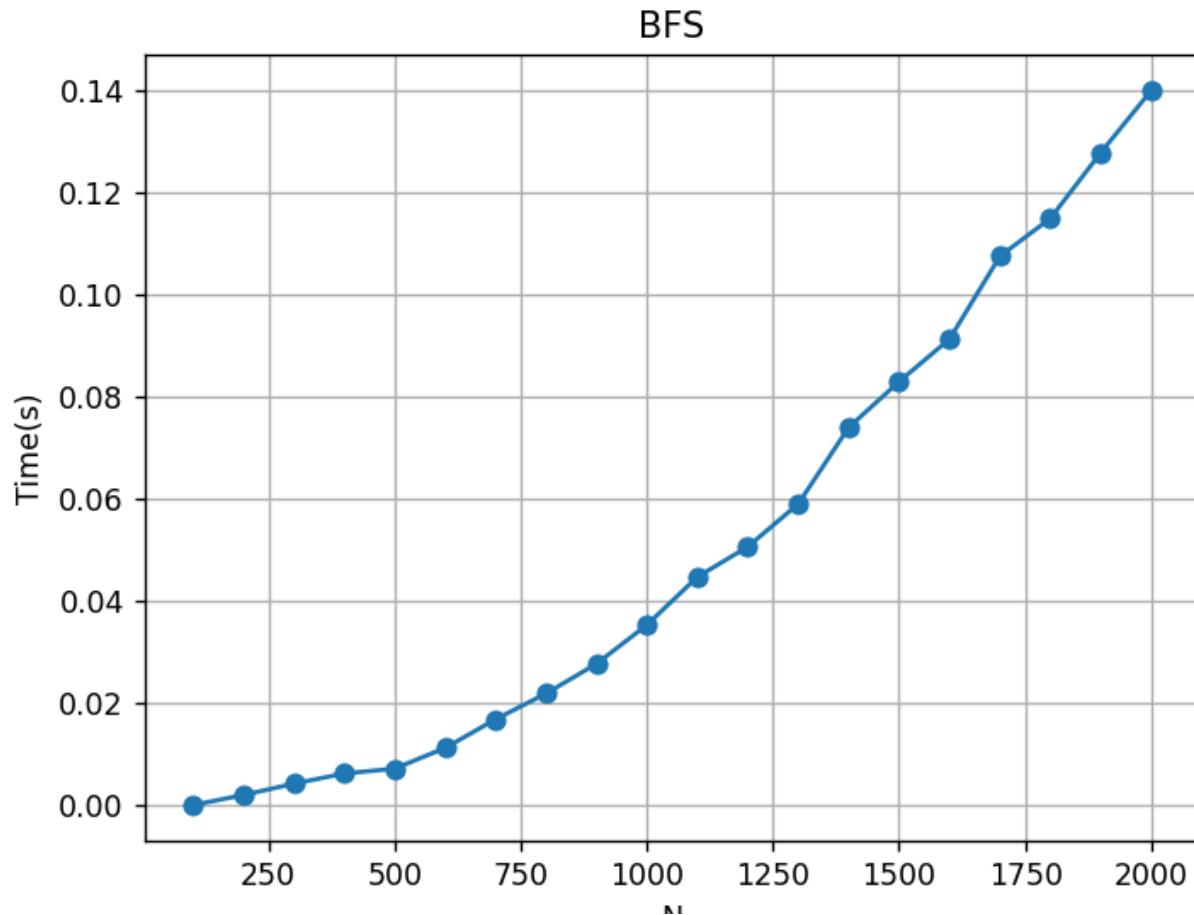

Figure 3 Graph of BFS

We can notice how, as the number of elements increases, the algorithm begins to require more time to execute.

Depth First Search

```python
def dfs(self, startNode, toPrint = 0):
    visited = [startNode]
    stack = [startNode]
    while stack:
        node = stack.pop(0)
        if node not in self.edges:
            continue
        for neighbor in self.edges[node]:
            if neighbor not in visited:
                stack.insert(0, neighbor)
                visited.append(neighbor)
```

Figure 4 Code for DFS

Initialization: Initializes a list called visited with the **startNode**, which keeps track of visited nodes. Initializes a stack with the **startNode**.

DFS Traversal: While the stack is not empty: Pops the top node from the stack. If the node does not have any edges (neighbors), it skips to the next iteration. Iterates through each neighbor of the current node. If a neighbor has not been visited yet: Pushes it onto the stack. Marks it as visited by appending it to the visited list.

```
100 - 1.02090835557128906e-05
200 - 0.00020752429962158205
300 - 0.0004286503791809082
400 - 0.0006279706954956055
500 - 0.000723416805267334
600 - 0.0011277699470522002
700 - 0.0016883111000061033
800 - 0.002193140983581543
900 - 0.002773587703704834
1000 - 0.0035350203514099123
```

Figure 5 Results for first 10 inputs DFS

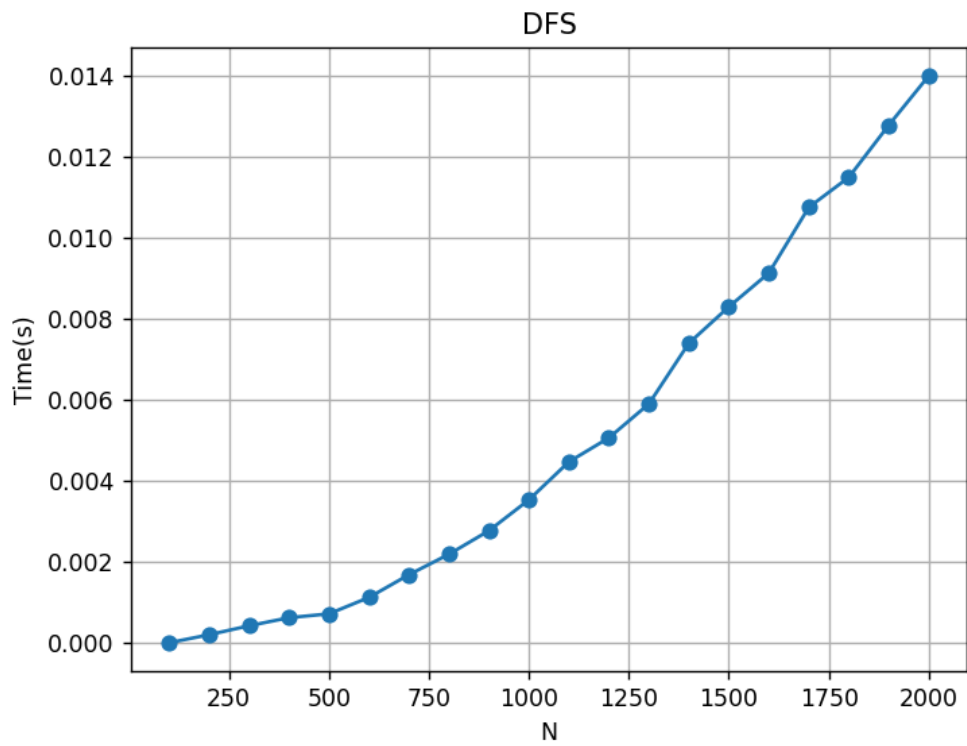As we can see DFS have the similar results as BFS.

Figure 6 Graph of DFS

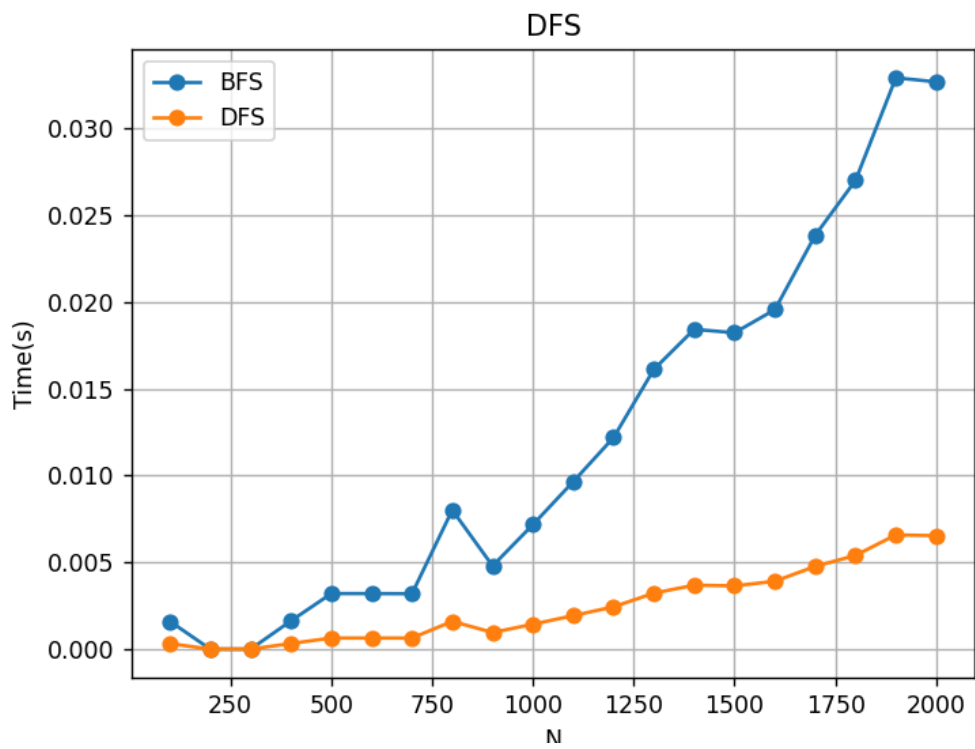Now will be represent results when node connection is 10%

Figure 7 Graph of DFS and BFS 10%

Conclusion

Both algorithms have same time complexity of O(V+E), where V is the number of vertices (nodes) and E is the number of edges in the graph. For BFS this is because it visits each vertex and edge once. DFS explores a graph by traversing as deeply as possible along each branch before backtracking. Space Complexity, BFS generally requires more memory compared to DFS because it uses a queue to store nodes, which can potentially grow to hold all the nodes at a given level of the graph. DFS, on the other hand, uses a stack to store nodes, which can lead to a smaller memory footprint compared to BFS, especially for graphs with many levels or branching factors.

Git Link

https://github.com/RodionClepa/APA/tree/Lab3