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

Laboratory work 3: Study and empirical analysis of searching algorithms. Analysis of Depth First Search (DFS), Breadth First Search(BFS)

Elaborated:
st. gr. FAF-222 Martiniuc Artiom

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2024

TABLE OF CONTENTS

Objective

ALGORITHM ANALYSIS

Study and analyze different searching algorithms.

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 analyzed.

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 behavior 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:

In this laboratory exercise, we explore and analyze two different searching algorithms: Depth First Search (DFS), Breadth First Search(BFS). These traversal methods are pivotal in computer science, serving a multitude of purposes ranging from pathfinding in networks to analyzing connected components in graphs. We intend to analyse these algorithms' operational dynamics in terms of time and space complexity by coding them and performing an empirical investigation on their performance. Selecting the best algorithm to navigate and analyse a given network structure or problem requires understanding the behaviour, efficiency, and restrictions of the different algorithms.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations).

Within this laboratory, we will be analyzing the 2 naïve algorithms empirically.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)), The amount of memory used by the algorithm as the graph size increases.

Input Format:

The input for both algorithms is a graph. We can represent a graph as a dictionary where keys are nodes and values are sets of neighbors. We will generate a graph where each node is connected to 4 other nodes randomly.

Example of input:

```
graph = {
   'A': {'B', 'C'},
   'B': {'A', 'D', 'E'},
   'C': {'A', 'F'},
   'D': {'B'},
   'E': {'B', 'F'},
   'F': {'C', 'E'}
}
```

IMPLEMENTATION

Two algorithms will be implemented in their native form in python an analyzed 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 o memory of the device used.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

Depth First Search

Depth First Search (DFS) is a graph traversal algorithm that explores as far as possible along each branch before backtracking. It starts at a selected node (root if a tree) and explores as deep as possible along each branch before backtracking. This means that in a DFS, one starts at the root (selecting some arbitrary node as the root in the case of a graph) and explores as far as possible along each branch before backtracking.

Key Characteristics:

- Utilizes a stack data structure, either through recursion or using an explicit stack.
- Explores possible vertices (from a supplied root) down each branch before backtracking.
- Has a time complexity of O(V + E) for a graph represented using an adjacency list, where V is the number of vertices and E is the number of edges.
- Space complexity can go up to O(V) in the worst case to store the call stack.

Implementation:

```
def dfs(graph, start, visited=None):
    if visited is None:
        visited = set()
    visited.add(start)

    for next_node in graph[start] - visited:
        dfs(graph, next_node, visited)
    return visited
```

Figure 1 DFS in Python

Breadth First Search

Breadth First Search (BFS) is another graph traversal algorithm that starts at a selected node and explores all the neighbor nodes at the present depth prior to moving on to the nodes at the next depth level. It employs a queue to keep track of the next location to visit.

Key Characteristics:

- Utilizes a queue to keep track of the nodes that are next in line to explore.
- Explores all neighbors of a node before moving to the neighbors' neighbors.
- Time complexity is O(V + E) for a graph represented using an adjacency list, similar to DFS.
- Space complexity can also be up to O(V) in the worst case due to the need to store a queue of vertices to explore.

Implementation:

```
def bfs(graph, start):
    visited, queue = set(), deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()

        for neighbour in graph[vertex]:
            if neighbour not in visited:
                visited.add(neighbour)
                queue.append(neighbour)
    return visited
```

Figure 2 DFS in Python

RESULTS

The graph below compares the execution time of Depth First Search (DFS) and Breadth First Search (BFS) as the number of nodes in the graph increases. As you can see, both algorithms exhibit a trend where the time increases with the number of nodes. However, the exact performance and efficiency depend on the structure of the graph and how nodes are connected.

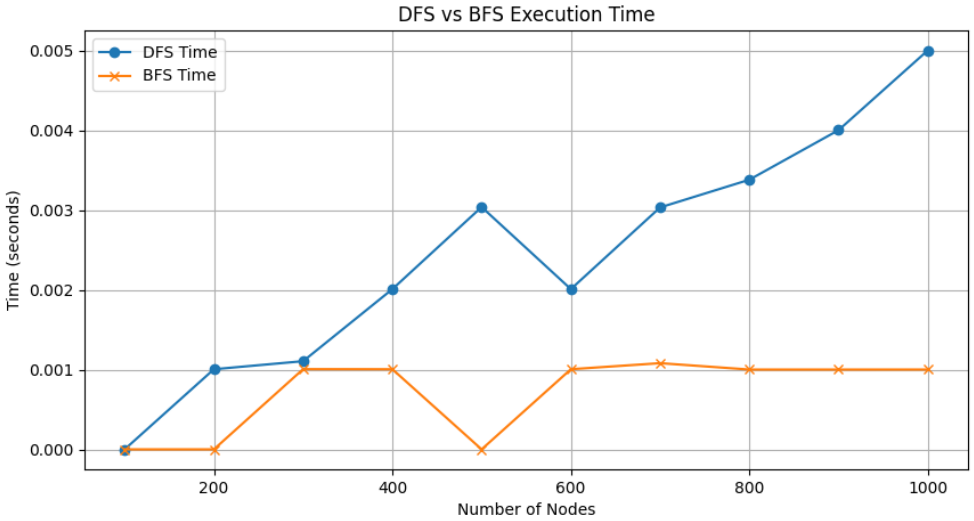

Figure 3 Graph of Searching Algorithms Performance

The execution time for DFS increases with the number of nodes, suggesting that the time complexity is likely O(V + E), where V is the number of vertices and E is the number of edges. The increase is not linear, displaying some fluctuations. This could be due to the nature of DFS, which can have variable performance based on the graph's structure and the starting point.

BFS shows a relatively flat trend, which implies that it is less affected by increases in the number of nodes in this instance. This could suggest that BFS is efficiently handling the graph data structure used in this test, possibly due to more uniform branching or less depth in the graph.

In comparison to DFS, BFS appears to be more consistent and stable in terms of execution time as the graph size grows. BFS might be a more reliable choice for graphs where the traversal time needs to be predictable and less variable.

CONCLUSION

In conclusion, the decision to use either DFS or BFS should consider the specific characteristics of the graph and the requirements of the application. While DFS can be quicker on some graphs, especially those with fewer branching paths, BFS provides a more consistent execution time across different sizes, which can be particularly beneficial for applications requiring performance predictability.

If the choice of algorithm depends on performance consistency as the graph grows, BFS might be preferred. However, if the specific structure of the graph and the application allows for deeper searches with backtracking, DFS could be utilized.