MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work 3:

Study and empirical analysis of Depth First Search(DFS) & Breadth First Search(BFS)

Checked by:

Fiștic Cristof, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

TABLE OF CONTENTS:

ALGORITHM ANALYSIS

Objective

Study and empirical analysis of sorting 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:

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 computer science, tree traversal (also known as tree search and walking the tree) is a form of graph traversal and refers to the process of visiting (e.g. retrieving, updating, or deleting) each node in a tree data structure, exactly once. Such traversals are classified by the order in which the nodes are visited. The following algorithms are described for a binary tree, but they may be generalized to other trees as well.

Unlike linked lists, one-dimensional arrays and other linear data structures, which are canonically traversed in linear order, trees may be traversed in multiple ways. They may be traversed in depth-first or breadth-first order. There are three common ways to traverse them in depth-first order: in-order, pre-order and post-order. Beyond these basic traversals, various more complex or hybrid schemes are possible, such as depth-limited searches like iterative deepening depth-first search. The latter, as well as breadth-first search, can also be used to traverse infinite trees.

Traversing a tree involves iterating over all nodes in some manner. Because from a given node there is more than one possible next node (it is not a linear data structure), then, assuming sequential computation (not parallel), some nodes must be deferred—stored in some way for later visiting. This is often done via a stack (LIFO) or queue (FIFO). As a tree is a self-referential (recursively defined) data structure, traversal can be defined by recursion or, more subtly, corecursion, in a natural and clear fashion; in these cases the deferred nodes are stored implicitly in the call stack. Depth-first search is easily implemented via a stack, including recursively (via the call stack), while breadth-first search is easily implemented via a queue, including co recursively.

Comparison Metric:

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

Input Format:

As input, each algorithm will receive different size graphs. We will generate different graphs (from 10 to 10000 nodes with the step of 10). Each algorithm will be called with the generated graph and the time execution will be saved in an array to later be plotted.

IMPLEMENTATION

We will make an empirical analysis of Depth-First Search (DFS) and Breadth-First Search (BFS) in python. All these algorithms will be implemented in their naive form in python and 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 on memory of the device used.

Depth-First Search (DFS):

Depth-first search (DFS) 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. Extra memory, usually a stack, is needed to keep track of the nodes discovered so far along a specified branch which helps in backtracking of the graph.

In depth-first search (DFS), the search tree is deepened as much as possible before going to the next sibling.

To traverse binary trees with depth-first search, perform the following operations at each node:

1. If the current node is empty then return.
2. Execute the following three operations in a certain order:
   N: Visit the current node.
   L: Recursively traverse the current node's left subtree.
   R: Recursively traverse the current node's right subtree.

The trace of a traversal is called a sequentialization of the tree. The traversal trace is a list of each visited node. No one sequentialization according to pre-, in- or post-order describes the underlying tree uniquely. Given a tree with distinct elements, either pre-order or post-order paired with in-order is sufficient to describe the tree uniquely. However, pre-order with post-order leaves some ambiguity in the tree structure.

Algorithm Description:

The naive depth-first search follows the algorithm as shown in the next pseudocode:

```
DFS(graph, start):
    Initialize an empty set called 'visited' to keep track of visited nodes
    Initialize a stack called 'stack' and push the starting node 'start' onto it

    while 'stack' is not empty:
```

```
        Pop a node 'node' from the top of the 'stack'
        If 'node' is not in 'visited':
            Add 'node' to 'visited'
            Push all unvisited neighbors of 'node' onto the 'stack'
    Return the set of 'visited' nodes
```

Implementation:

```python
def dfs(graph, start):
    visited = set()
    stack = [start]
    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            stack.extend([n for n in graph.neighbors(node) if n not in visited])
    return visited
```

Figure 1 depth-first search python implementation

Breadth-First Search (BFS):

Breadth-first search (BFS) is an algorithm for searching a tree data structure for a node that satisfies a given property. It starts at the tree root and explores all nodes at the present depth prior to moving on to the nodes at the next depth level. Extra memory, usually a queue, is needed to keep track of the child nodes that were encountered but not yet explored.

This non-recursive implementation is similar to the non-recursive implementation of depth-first search, but differs from it in two ways:

1. it uses a queue (First In First Out) instead of a stack (Last In First Out) and
2. it checks whether a vertex has been explored before enqueueing the vertex rather than delaying this check until the vertex is dequeued from the queue.

If G is a tree, replacing the queue of this breadth-first search algorithm with a stack will yield a depth-first search algorithm. For general graphs, replacing the stack of the iterative depth-first search implementation with a queue would also produce a breadth-first search algorithm, although a somewhat nonstandard one. The Q queue contains the frontier along which the algorithm is currently searching.

Nodes can be labeled as explored by storing them in a set, or by an attribute on each node, depending on the implementation.

Note that the word node is usually interchangeable with the word vertex.

The parent attribute of each node is useful for accessing the nodes in a shortest path, for example by backtracking from the destination node up to the starting node, once the BFS has been run, and the predecessors nodes have been set.

Algorithm Description:

The naive breadth-first search follows the algorithm as shown in the next pseudocode:

```
BFS(graph, start):
    Initialize an empty set called 'visited' to keep track of visited nodes
    Initialize a queue called 'queue' and enqueue the starting node 'start'

    while 'queue' is not empty:
        Dequeue a node 'node' from the front of the 'queue'
        If 'node' is not in 'visited':
            Add 'node' to 'visited'
            Enqueue all unvisited neighbors of 'node' onto the 'queue'

    Return the set of 'visited' nodes
```

Implementation:

```python
def bfs(graph, start):
    visited = set()
    queue = [start]
    while queue:
        node = queue.pop(0)
        if node not in visited:
            visited.add(node)
            queue.extend([n for n in graph.neighbors(node) if n not in visited])
    return visited
```

Figure 2 breadth-first search python implementation

Results:

Algorithmic Complexity:

BFS employs a Queue data structure to systematically explore neighboring nodes level by level. In the worst-case scenario, where the graph is a dense tree and the target node resides at the farthest leaf, BFS has a time complexity of O(V + E), where V represents the number of vertices (nodes) and E represents the number of edges in the graph. This is because BFS needs to visit every single node (V) and traverse every edge (E) at least once to reach the target.

DFS, on the other hand, explores a single path deeply until it reaches a dead end or the target node. In the worst case, where the graph is a linked list and the target node is at the end, DFS has a time complexity of O(V + E) similar to BFS. However, in most practical scenarios, DFS exhibits an average-case time complexity of O(E), as it explores only a subset of edges depending on the chosen path.

Analysis of Results:

The provided table and the corresponding graph (which can be inferred from the table data) demonstrate a clear distinction in performance between BFS and DFS for random graphs. Here are the key observations:

1. Time Complexity Trends: As the graph size (number of nodes) increases, the execution time for both algorithms increases. However, the increase is significantly steeper for BFS compared to DFS. This aligns with the theoretical complexities discussed above. BFS systematically explores all neighboring nodes, leading to a larger number of node visits and edge traversals as the graph grows denser. In contrast, DFS can potentially reach the target node with fewer explorations, especially if the target is closer to the starting node.
2. Dominant Performance of DFS: Throughout the data range, DFS consistently exhibits faster execution times compared to BFS. This is evident from the consistently higher BFS time values in the table. For instance, at a graph size of 10,000 nodes, BFS takes roughly 0.22 seconds, whereas DFS completes the search in only 0.007 seconds. This significant difference highlights the advantage of DFS for exploring random graphs, particularly for larger datasets.

	Graph Size	BFS Time (seconds)	DFS Time (seconds)
0	10	0.000000	0.000000
1	20	0.000000	0.000000
2	30	0.000000	0.000000
3	40	0.000000	0.000000
4	50	0.000000	0.000000
..	...	...	...
995	9960	0.226007	0.009996
996	9970	0.233998	0.007001
997	9980	0.221958	0.011520
998	9990	0.222432	0.006001
999	10000	0.225322	0.007000

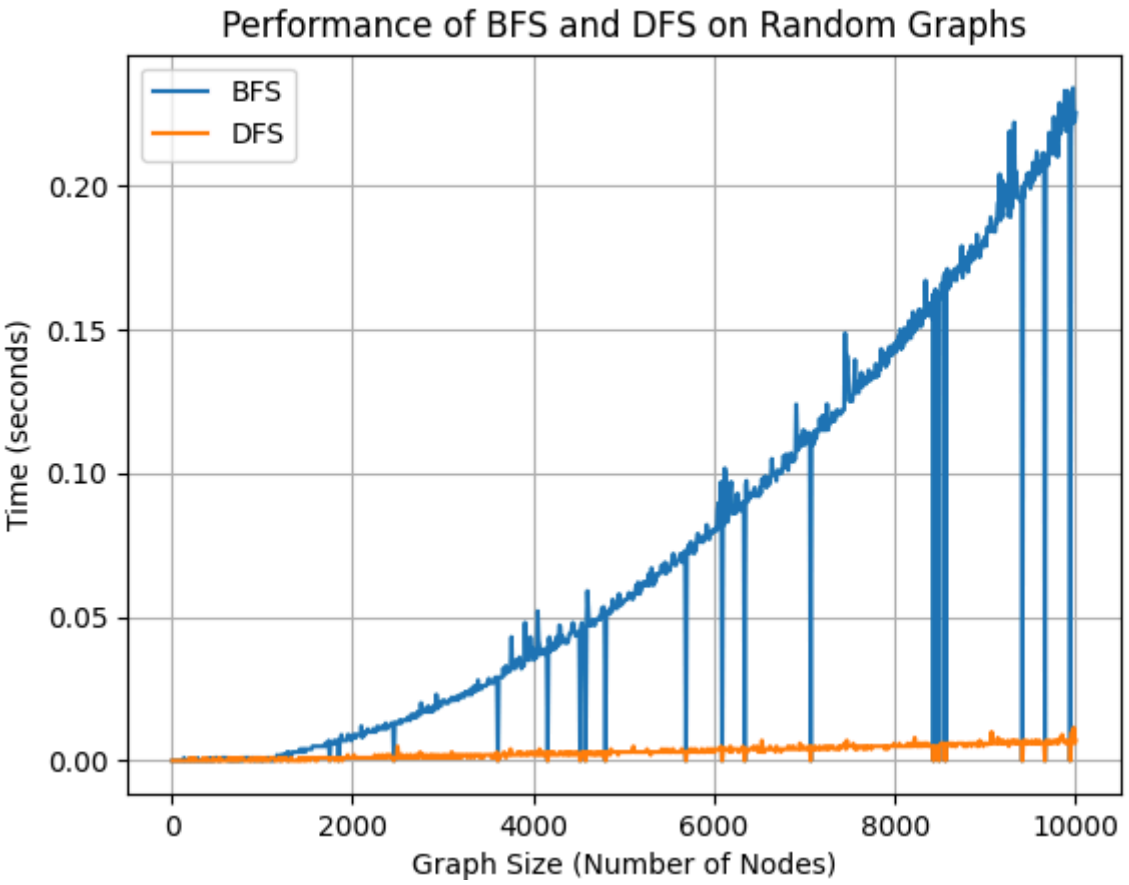

Figure 3 benchmark of bfs and dfs

CONCLUSION:

Based on the analysis, DFS emerges as the preferred choice for traversing random graphs, especially for larger graphs. This is primarily due to its faster execution times and potentially lower memory usage compared to BFS. However, the choice between BFS and DFS depends on the specific problem being addressed: Finding the Shortest Path: If the objective is to find the shortest path between two nodes in an unweighted graph, BFS remains the optimal choice due to its guaranteed exploration of all nodes at a given level before moving to the next. General Graph Exploration: For general graph exploration tasks where finding the shortest path is not a priority, DFS offers significant performance advantages, particularly for larger and denser graphs. This analysis underscores the importance of understanding the algorithmic complexities of BFS and DFS when selecting the appropriate graph traversal method for a given problem and dataset size.