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

Laboratory work 4:

Empirical analysis of Depth First Search and Breadth First Search.

Elaborated:
st. gr. FAF-213 Konjevic Alexandra

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

ALGORITHM ANALYSIS

Objective:

Study and analyze algorithms for searching in graphs or trees: Depth First Search and Breadth First Search.

Tasks:

1. Implement the listed algorithms 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.

Introduction:

Depth First Search (DFS) and Breadth First Search (BFS) are two of the most fundamental and widely used algorithms in computer science. These algorithms are commonly used to traverse graphs, trees, and other data structures in order to find specific information, determine connectivity, or perform other useful computations.

DFS and BFS are both algorithms for exploring a graph, but they differ in how they traverse the graph. DFS explores a graph by visiting as far as possible along each branch before backtracking, while BFS explores the graph by visiting all the neighboring nodes at the current depth before moving on to the next depth.

In this report, we will provide an overview of the DFS and BFS algorithms, explain their differences, discuss their applications, and provide examples of their implementation in various scenarios. Additionally, we will compare the performance of DFS and BFS and provide insights on when to use each algorithm based on the problem requirements.

Comparison Metric:

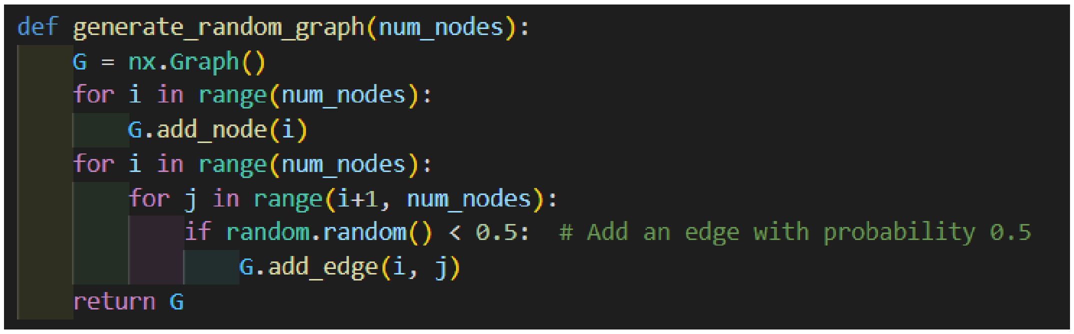
The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)).

Input Format:

I generated random graphs of lengths from 500 vertexes to 7000. For each graph I measured the time of executing of functions `bfs` and `dfs`.

IMPLEMENTATION

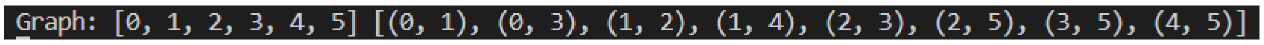
First of all, I used python library `networkx` to generate random graphs, via the method:



```python
def generate_random_graph(num_nodes):
    G = nx.Graph()
    for i in range(num_nodes):
        G.add_node(i)
    for i in range(num_nodes):
        for j in range(i+1, num_nodes):
            if random.random() < 0.5:  # Add an edge with probability 0.5
                G.add_edge(i, j)
    return G
```

Figure 1. Method to generate random graphs in python

Using the method `draw` of this library, I also visualized a random graph:



```
Graph: [0, 1, 2, 3, 4, 5] [(0, 1), (0, 3), (1, 2), (1, 4), (2, 3), (2, 5), (3, 5), (4, 5)]
```

Figure 2. Random graph

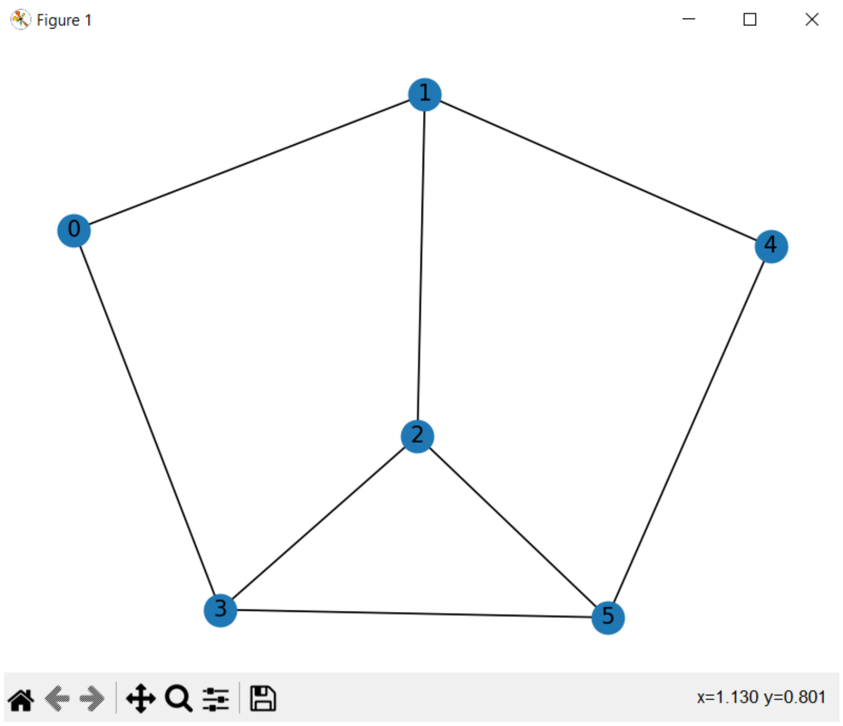
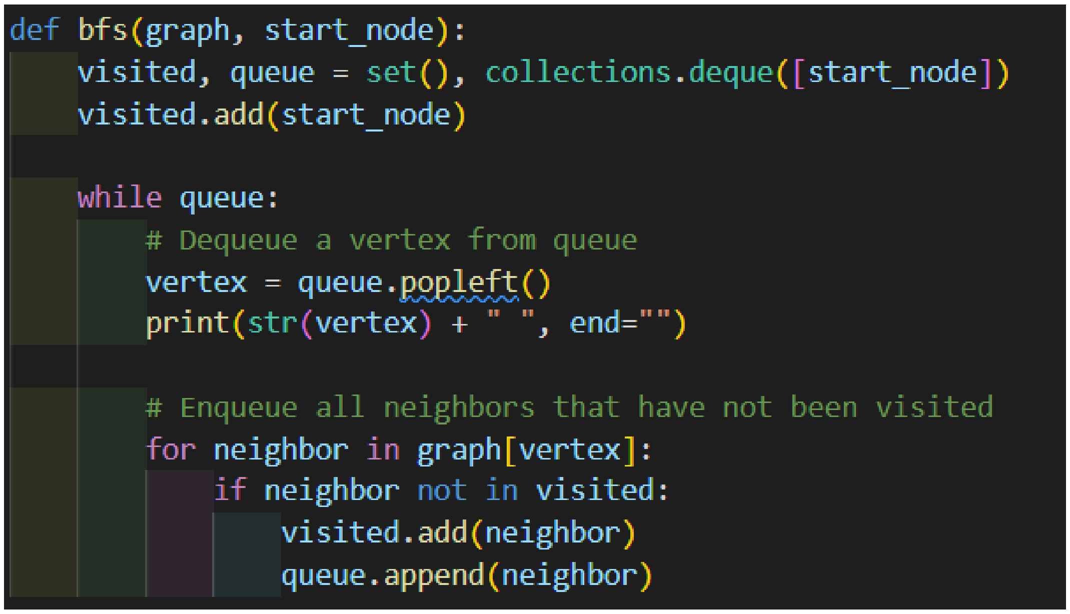


Figure 3. Visualization of random graph

1. BFS method implementation



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

    while queue:
        # Dequeue a vertex from queue
        vertex = queue.popleft()
        print(str(vertex) + " ", end="")

        # Enqueue all neighbors that have not been visited
        for neighbor in graph[vertex]:
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

Figure 4. BFS implementation in python

Algorithm Description:

The algorithm traverses the graph in a breadth-first manner, meaning that it visits all the nodes at the same level before moving on to the next level.

Here's how the algorithm works step-by-step:

- The algorithm creates an empty set called visited to keep track of the nodes that have already been visited, and a queue data structure initialized with the starting node.
- The starting node is added to the visited set.
- The algorithm enters a loop that continues until the queue is empty.
- In each iteration of the loop, the algorithm dequeues a vertex from the left side of the queue using the popleft() method, which returns the leftmost item in the deque and removes it from the deque.
- The algorithm then prints the vertex that has just been dequeued.
- The algorithm then enqueues all the neighbors of the dequeued vertex that have not yet been visited. This is done by iterating over the list of neighbors of the vertex obtained from the graph parameter, and checking if each neighbor is in the visited set. If a neighbor is not in the visited set, it is added to the visited set and to the right end of the queue using the append() method.
- The loop continues until the queue is empty, at which point the algorithm has visited all the nodes reachable from the starting node in a breadth-first order.

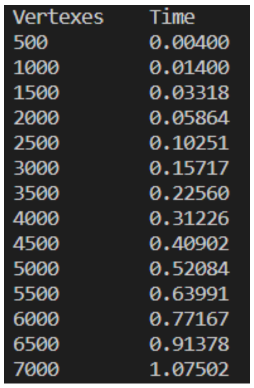

```
Vertexes    Time
500         0.00400
1000        0.01400
1500        0.03318
2000        0.05864
2500        0.10251
3000        0.15717
3500        0.22560
4000        0.31226
4500        0.40902
5000        0.52084
5500        0.63991
6000        0.77167
6500        0.91378
7000        1.07502
```

Figure 5. Results for BFS

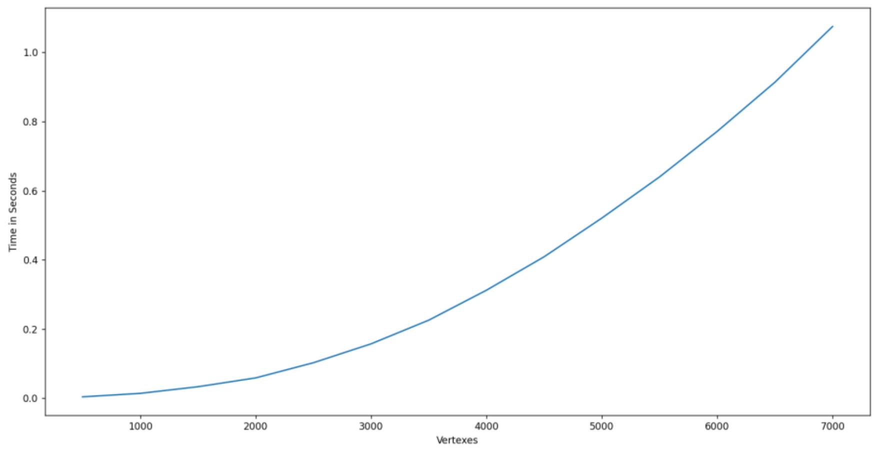
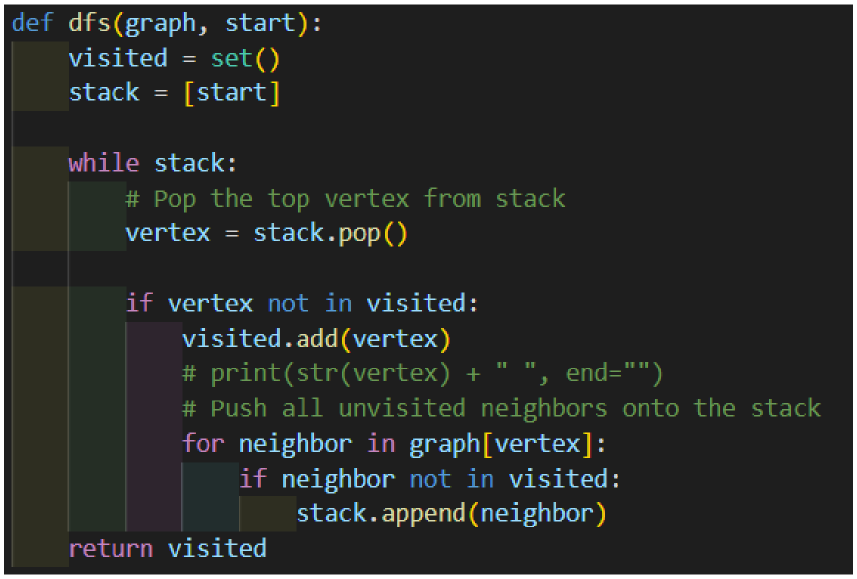


Figure 6. Graph for BFS

2. DFS method implementation



```python
def dfs(graph, start):
    visited = set()
    stack = [start]

    while stack:
        # Pop the top vertex from stack
        vertex = stack.pop()

        if vertex not in visited:
            visited.add(vertex)
            # print(str(vertex) + " ", end="")
            # Push all unvisited neighbors onto the stack
            for neighbor in graph[vertex]:
                if neighbor not in visited:
                    stack.append(neighbor)
    return visited
```

Figure 7. DFS implementation in python

Algorithm Description:

The algorithm takes two inputs: a graph data structure represented as a dictionary and a starting vertex from where the DFS traversal begins.

Here are the step-by-step instructions for the DFS algorithm:

- Create an empty set called visited to keep track of visited vertices.

- Create a list called stack and add the starting vertex to it.
- While the stack is not empty, do the following steps:

    a. Pop the top vertex from the stack and assign it to the variable vertex.

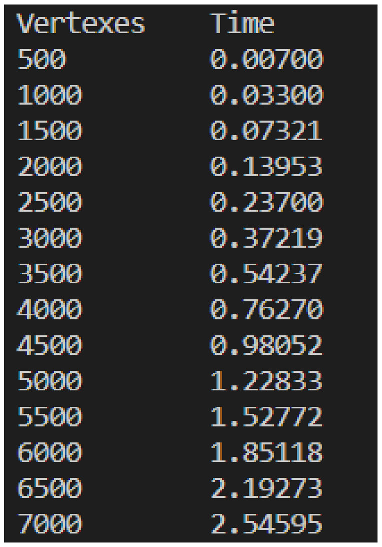
    b. If the vertex has not been visited yet, then add it to the visited set.

    c. For each unvisited neighbor of the vertex, do the following:

    i. Add the neighbor to the stack.

- When the while loop terminates, return the visited set containing all visited vertices.

The DFS algorithm works by traversing a path as far as possible before backtracking. The algorithm uses a stack data structure to keep track of the vertices to visit in the next step. It starts with the given starting vertex and explores the neighboring vertices by adding them to the stack. It continues to explore the neighboring vertices until it reaches a vertex with no unvisited neighbors, and then backtracks to the previous vertex in the stack. The algorithm repeats this process until it visits all the vertices reachable from the starting vertex.



```
Vertexes     Time
500          0.00700
1000         0.03300
1500         0.07321
2000         0.13953
2500         0.23700
3000         0.37219
3500         0.54237
4000         0.76270
4500         0.98052
5000         1.22833
5500         1.52772
6000         1.85118
6500         2.19273
7000         2.54595
```

Figure 8. Results for DFS

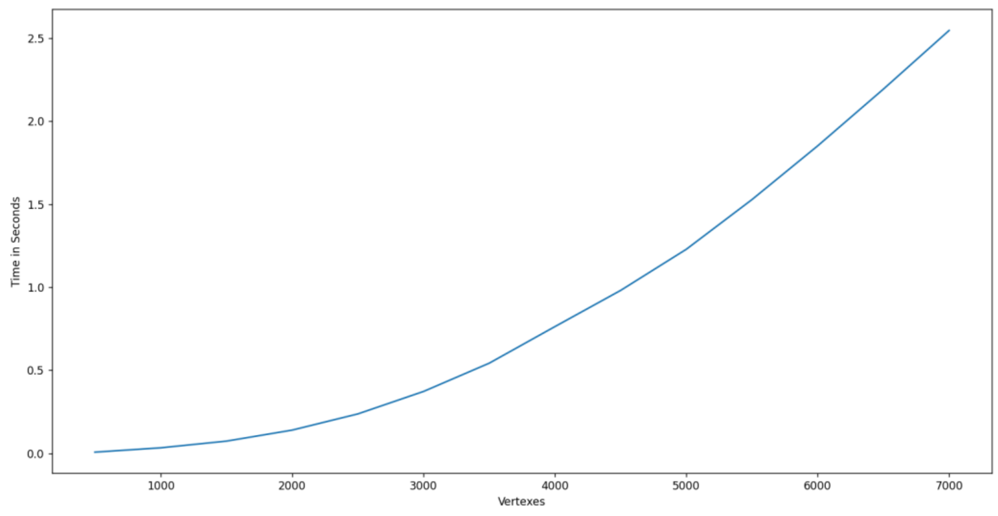


Figure 9. Graph for DFS

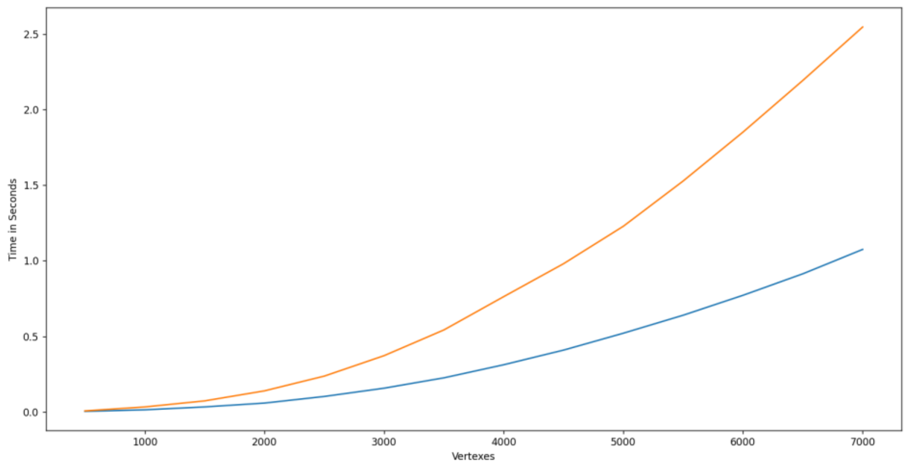


Figure 10. Graph with both BFS and DFS

● – DFS ● – BFS

Conclusion

In conclusion, the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms are two fundamental and widely used graph traversal algorithms in computer science. Through the implementation of these algorithms in Python, we have demonstrated their practical applications in graph analysis and search.

BFS explores all the vertices at the same level before moving on to the next level, making it useful for finding the shortest path or distances between nodes in unweighted graphs. On the other hand, DFS explores the graph by going as deep as possible, making it useful for applications such as topological sorting and cycle detection.

Overall, the efficiency of BFS and DFS depends on the structure and size of the graph being traversed. BFS has a higher memory requirement and can be slower than DFS for dense graphs, while DFS can get stuck in infinite loops or take longer to find a solution for disconnected graphs.

Despite their limitations, BFS and DFS remain important tools in graph analysis and search, and their implementation in Python provides a useful framework for further experimentation and exploration of these algorithms. Further research can investigate the optimization of BFS and DFS for specific use cases or explore more advanced graph traversal algorithms.

Github repository: https://github.com/alya1007/Labs-semester-4/tree/master/AA