MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF THE REPUBLIC OF MOLDOVA

Technical University of Moldova Faculty of Computers, Informatics and Microelectronics Department of Software and Automation Engineering

Postoronca Dumitru FAF-233

Report

Laboratory Work №3

of AA

Checked by:
Fistic Cristofor, *university assistant*
DISA, FCIM, UTM

Chișinău – 2024

Conditions of the Task

Study and empirical analysis of sorting algorithms. Analysis of quickSort, mergeSort, heapSort, one of your choice (Counting Sort)

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.

Input data

To analyze and compare the sorting algorithms, we define the following input data conditions:

1. Wideness of the graph(number of direct children):
   - Small wideness ($n = 3$)
   - Large wideness ($n = 20$)
2. Depth of the the graph:
   - Low depth ($d = 3$)
   - Big depth ($d = 11$)
3. Edge Cases:
   - All elements identical
   - Cycled graph
   - Some directed graphs

Input data was generated by a special function written in JS that takes as input the wideness of the graph (number of direct children each node will have) and it's depth. Total number of nodes will be calculated by the formula:

$$N_{nodes} = width^{depth}$$

Metrics in Algorithm analysis

In this laboratory work I will try to test how each of the path finding algorithms work depending on the width and the depth of graph.

For this purpose I will run the same algorithm to find the path to the node from the depth 1 down to the node which is the leaf. In this way if the depth of generated graph is 5, I will show the metrics for: depth 1, depth 2, depth3, depth4, depth 5

Disabling Turbofan

In order to perform the algorithm analysis I've written Javascript programs that would track the execution time of each algorithm and run them in Node js. Node js uses V8 engine in order to execute the JavaScript programs. This engine consists of 2 components:

- Interpreting machine Ignition[1]
- Optimizing compiler **Turbofan**, that can optimize and translate some of the JavaScript code into the machine code.

To reduce the influence of Turbofan on the validity of data, I have run the algorithms with *–no-opt* flag to disable Turbofan. This decision is motivated by the unpredictability of the optimizing compiler which can optimize the code for an algorithm with a long execution time and not optimize the efficient algorithm. Thus I put all algorithms to equal execution conditions.

Algorithms Implementation

Results

After running of all the tests these are the results obtained:

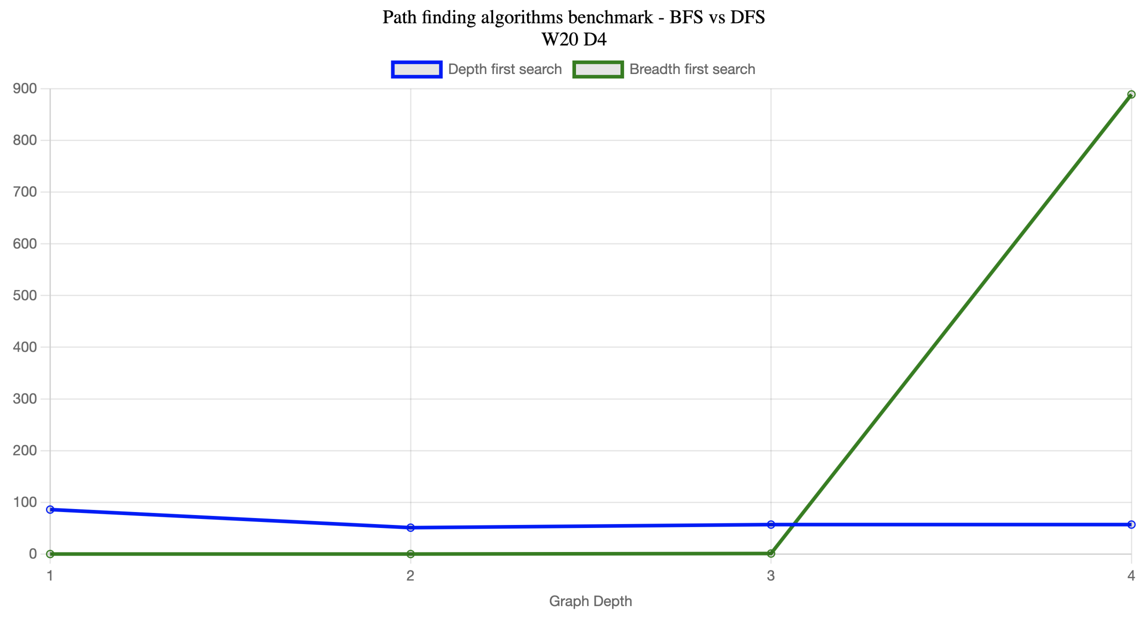


Figure 1: Path finding comparison for a graph with width 20 and depth 4

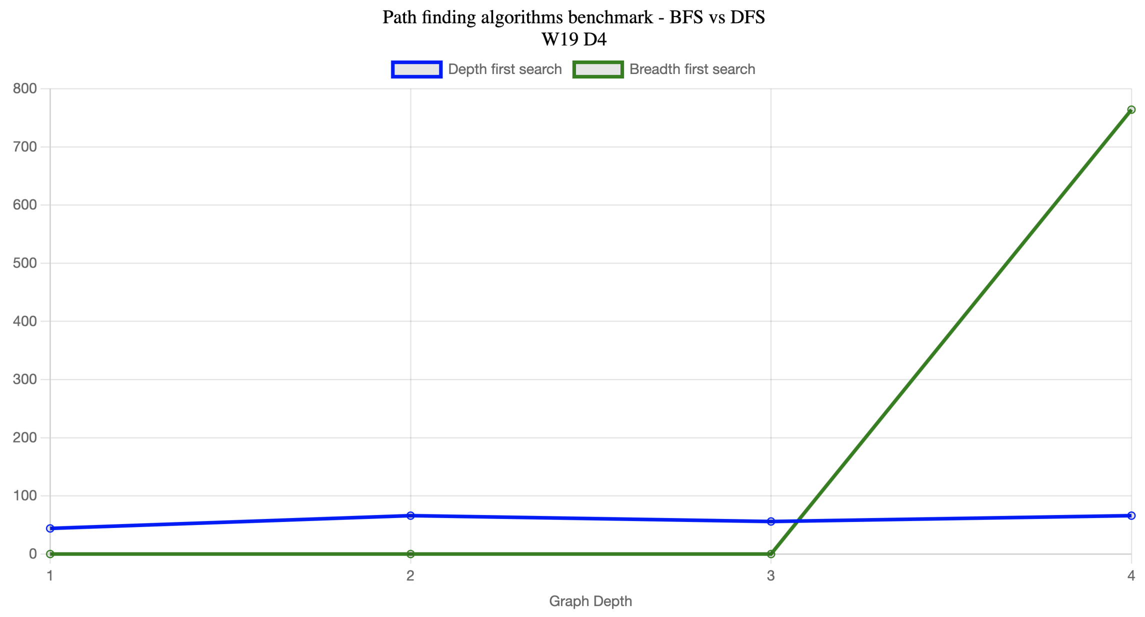


Figure 2: Path finding comparison for a graph with width 19 and depth 4

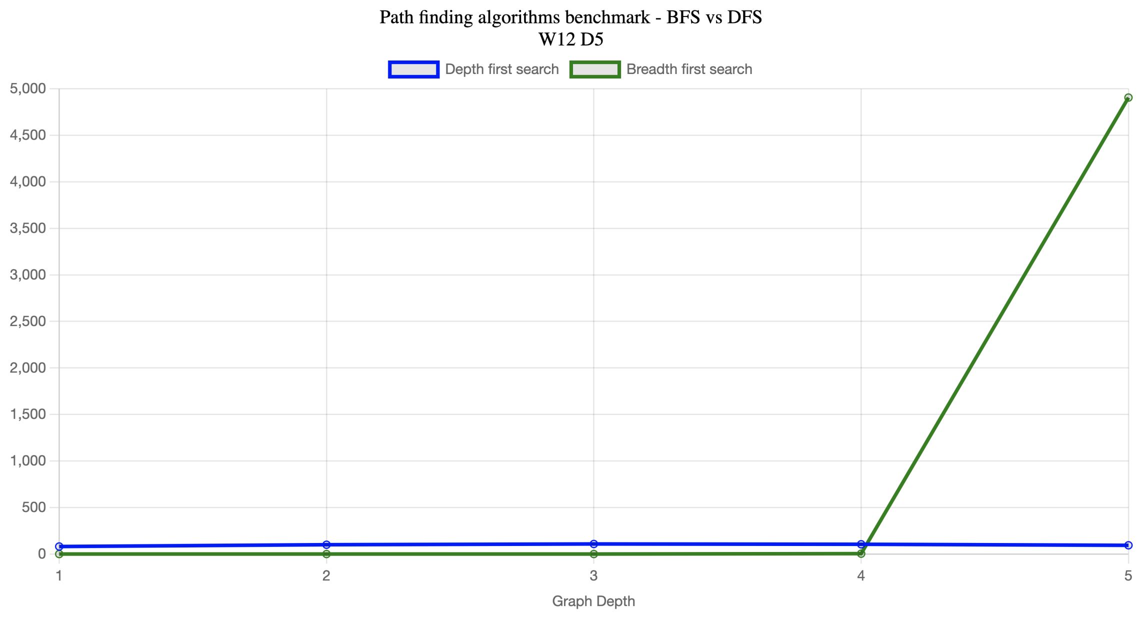


Figure 3: Path finding comparison for a graph with width 12 and depth 5

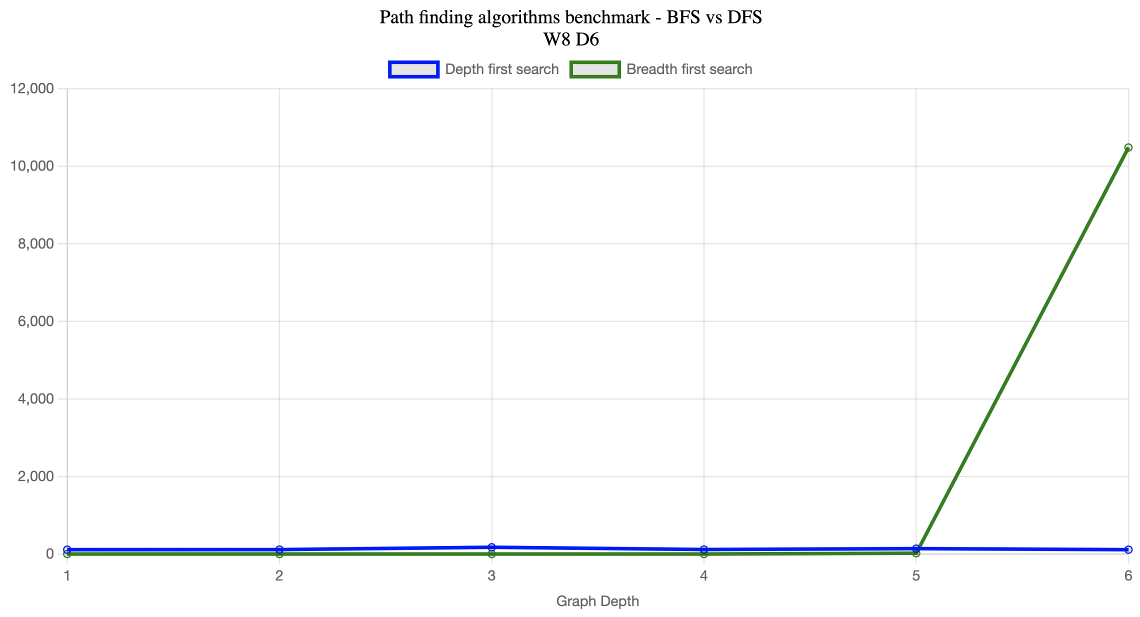


Figure 4: Path finding comparison for a graph with width 8 and depth 6

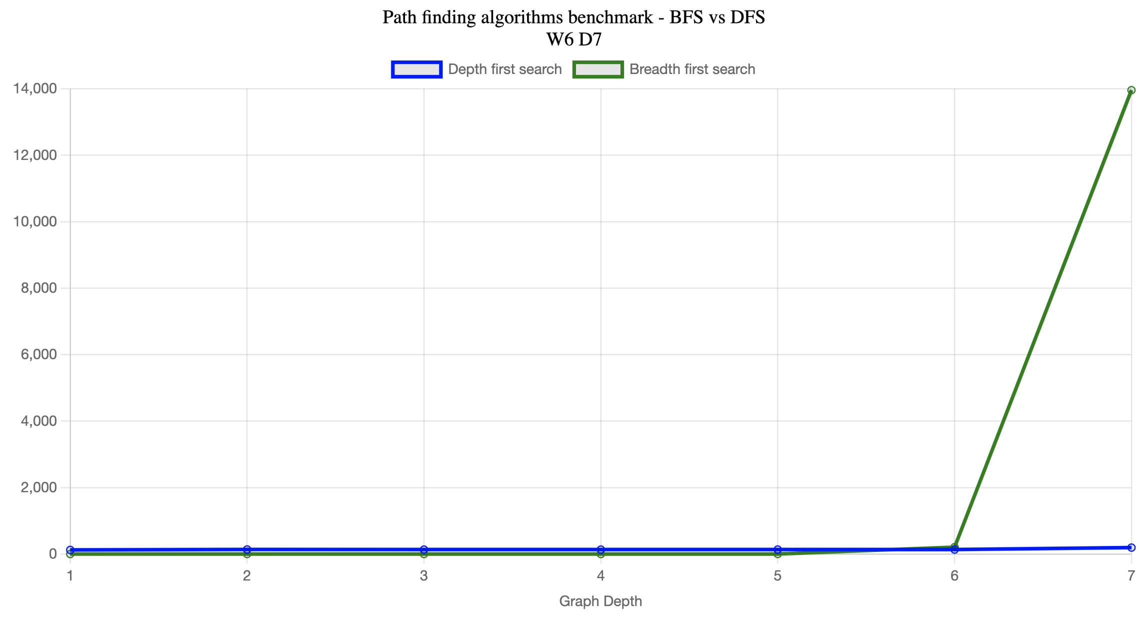


Figure 5: Path finding comparison for a graph with width 6 and depth 7

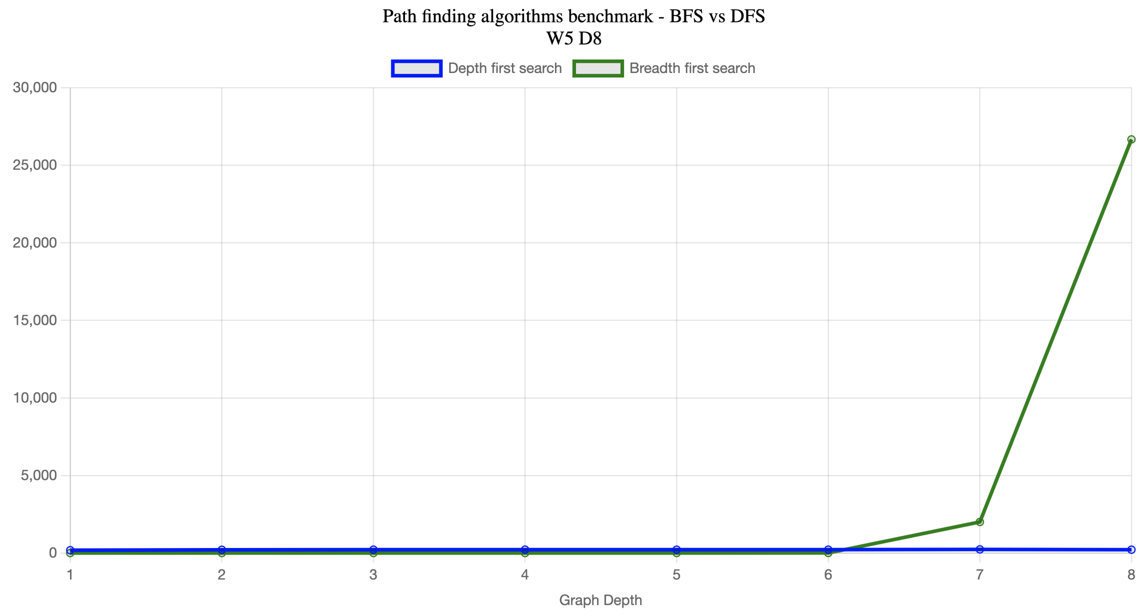


Figure 6: Path finding comparison for a graph with width 5 and depth 8

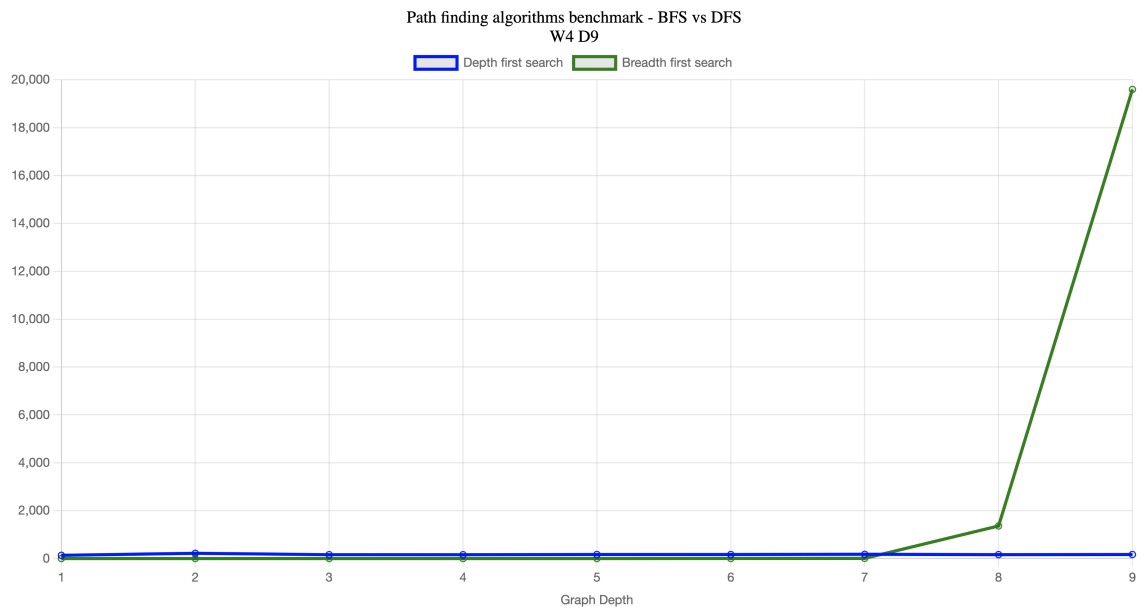


Figure 7: Path finding comparison for a graph with width 4 and depth 9

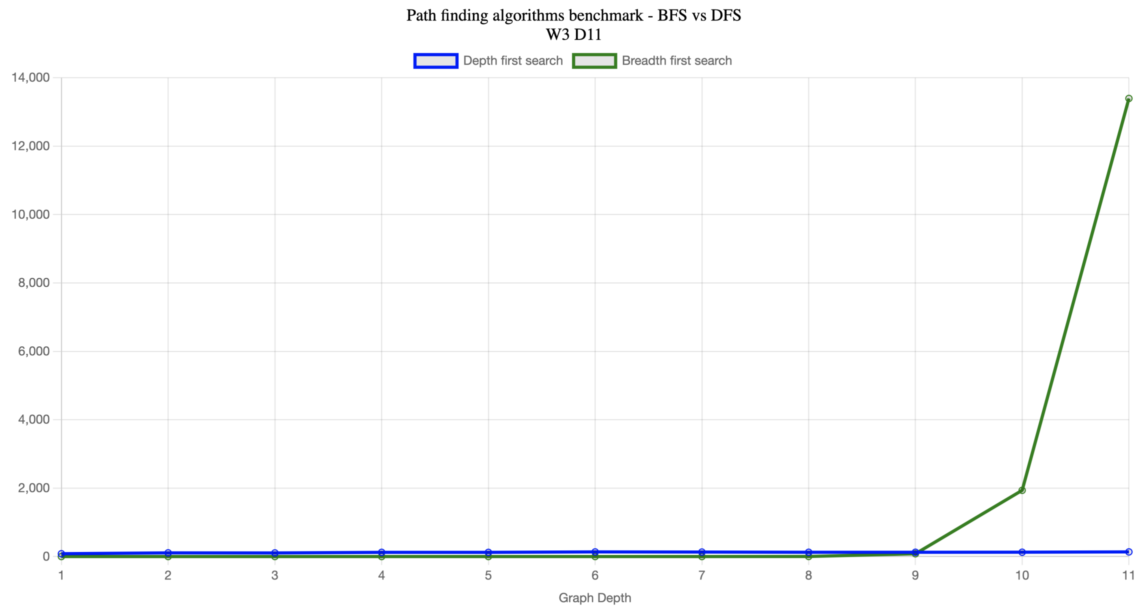


Figure 8: Path finding comparison for a graph with width 3 and depth 11

As you can observe, the BFS algorithm works blazingly fast when the node is not very deep in the graph.

This happens because the of the principle according to which BFS works. It searches in the graph level by level, checking all the nearest neighbors.

DFS would have a chance to find the node, but because it dives directly into the deepest level in graph it may perform redundant computations until it will procedd with searching in desired subtree.

Because of this particularities you can see how much slower BFS becomes as soon as the searched node is deeper in the graph. While DFS searches there, BFS still checks all it's neighbours

Even if the time complexity of both algorithms is:

$$O(n) = V + E$$

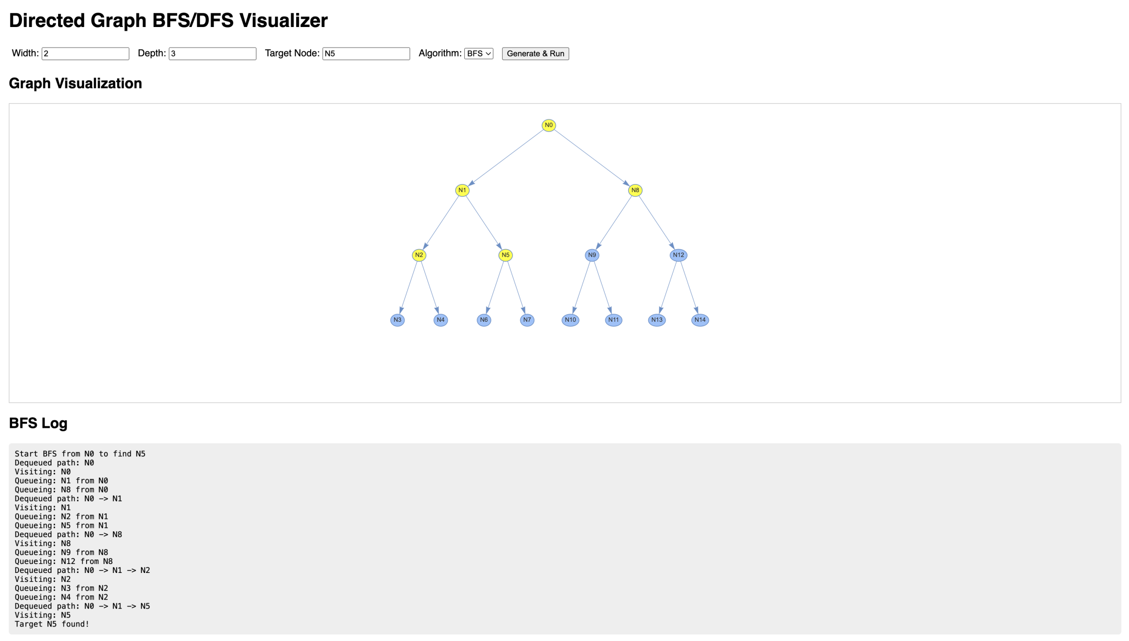
Depending on the edge case, one algorithm can be much faster than another.

Graphical Representation

Graphical representation was done to showcase how the DFS and BFS algorithms would look on practice and can be accessed online[3]

Here user have the following functionalities:

- Choose the depth and width of the graph
- Chosing the node to search
- Switching between algorithms
- Display of the logs for each algorithm



```
Start BFS from N0 to find N5
Dequeued path: N0
Visiting: N0
Queueing: N1 from N0
Queueing: N8 from N0
Dequeued path: N0 -> N1
Visiting: N1
Queueing: N2 from N1
Queueing: N5 from N1
Dequeued path: N0 -> N8
Visiting: N8
Queueing: N9 from N8
Queueing: N12 from N8
Dequeued path: N0 -> N1 -> N2
Visiting: N2
Queueing: N3 from N2
Queueing: N4 from N2
Dequeued path: N0 -> N1 -> N5
Visiting: N5
Target N5 found!
```

Figure 9: Graphical representation of DFS and BFS

References

[1] Franceska Hinkelman (2017) - Understanding bytecode *Medium*

[2] GitHub repository of current laboratory work

[3] Graphical representation of BFS and DFS