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

REPORT

Laboratory work no. 4
DFS, BFS

Elaborated:
st. gr. FAF-213 Botnari Ciprian

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

ALGORITHM ANALYSIS

Objective

Study and analyze 2 algorithms for finding a node in graph: (Breadth-first search) BFS and (Depth-first search) DFS.

Tasks

1. Implement 2 algorithms for finding a node in graph;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical notes

An alternative approach to evaluating the complexity of an algorithm is through empirical analysis, which can provide insights on the complexity class of the algorithm, comparison of efficiency between algorithms solving the same problem, comparison of different implementations of the same algorithm, and performance on specific computers.

The process of empirical analysis typically involves:

1. Establishing the purpose of the analysis
2. Choosing the efficiency metric (number of operations or execution time)
3. Defining the properties of the input data
4. Implementing the algorithm in a programming language
5. Generating input data
6. Running the program with each set of data
7. Analyzing the results

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

Graph search is a fundamental problem in computer science, and there are several theories and algorithms for searching nodes in graphs. The two main categories of graph search algorithms are depth-first search (DFS) and breadth-first search (BFS).

DFS is a recursive algorithm that starts at the root node of a graph and explores as far as possible along each branch before backtracking. DFS can be implemented using a stack data structure or through recursive function calls. DFS is useful when searching for a path or a cycle in a graph, and can also be used for topological sorting.

BFS, on the other hand, explores all the neighboring nodes at the current depth before moving on to the next depth level. BFS can be implemented using a queue data structure. BFS is useful for finding the shortest path between two nodes in an unweighted graph, and it can also be used for finding connected components in a graph.

There are also more advanced graph search algorithms, such as Dijkstra's algorithm and A* search, which are used for finding the shortest path in weighted graphs. Dijkstra's algorithm is a modified BFS algorithm that takes into account the weights of the edges, while A* search is a heuristic search algorithm that uses an estimate of the distance to the goal node to guide the search.

The choice of a graph search algorithm depends on the specific problem being solved and the characteristics of the graph being searched. DFS is generally faster than BFS for large graphs, but BFS is more efficient for finding the shortest path in unweighted graphs. More advanced algorithms like Dijkstra's and A* search are useful for finding the shortest path in weighted graphs.

A balanced tree is a tree data structure where the height of the left and right subtrees of any node in the tree differ by at most one. In other words, a balanced tree is a tree where the difference in height between the left and right subtrees of any node is not more than a certain constant value.

An unbalanced tree is a tree where the height of the left and right subtrees of a node can differ significantly. It can arise due to poor insertion or deletion strategies, which can result in a skewed distribution of nodes.

In this laboratory work, I am going to tackle 2 algorithms: DFS and BFS.

Comparison Metric

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

Input Format

As input, 2 tree-like graphs will be provided: one balanced and one unbalanced.

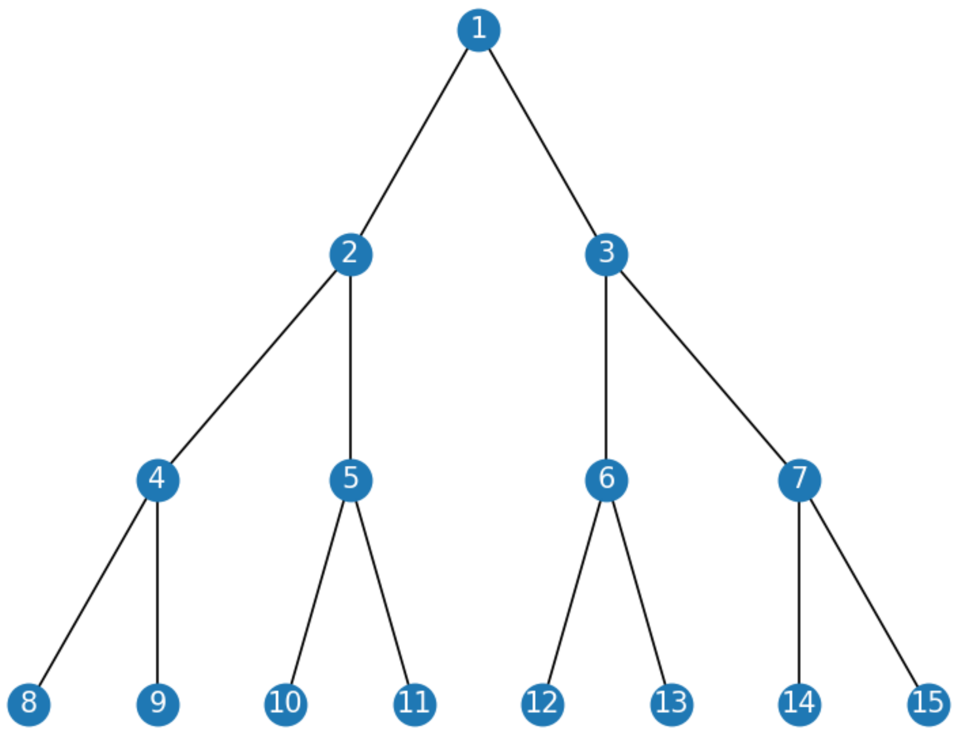


Figure 1. Balanced tree

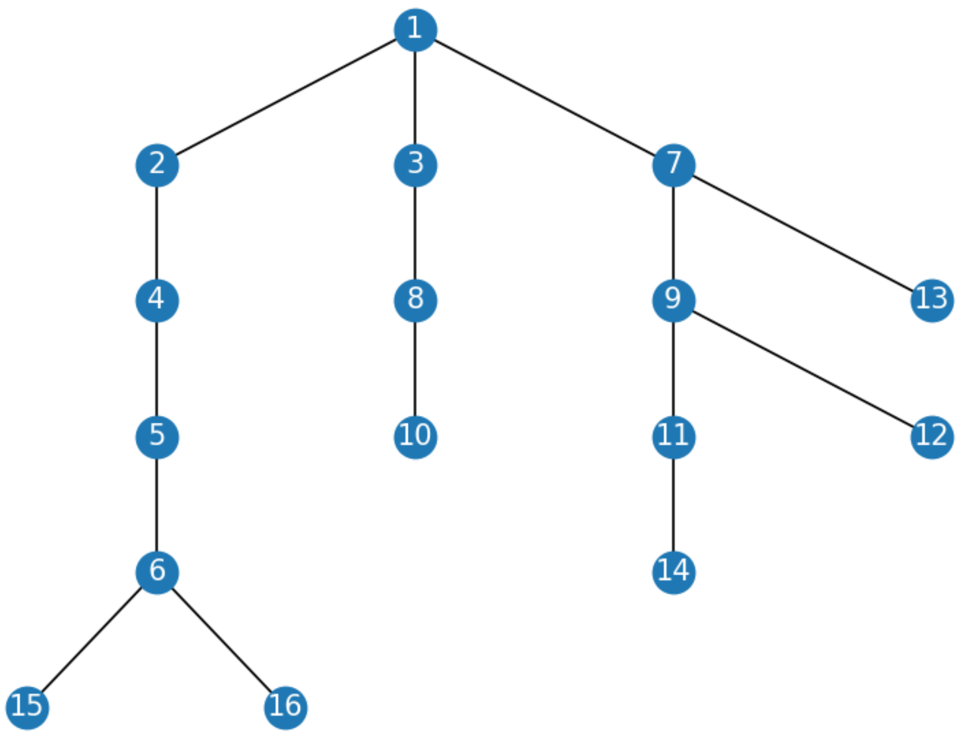


Figure 2. Unbalanced tree

IMPLEMENTATION

All variations of the algorithm will be put into practice in Python and objectively evaluated depending on how long it takes to complete each one. The particular efficiency in report with input will vary based on the memory of the device utilized, even though the overall trend of the results may be similar to other experimental data. As determined by experimental measurement, the error margin will be a few seconds.

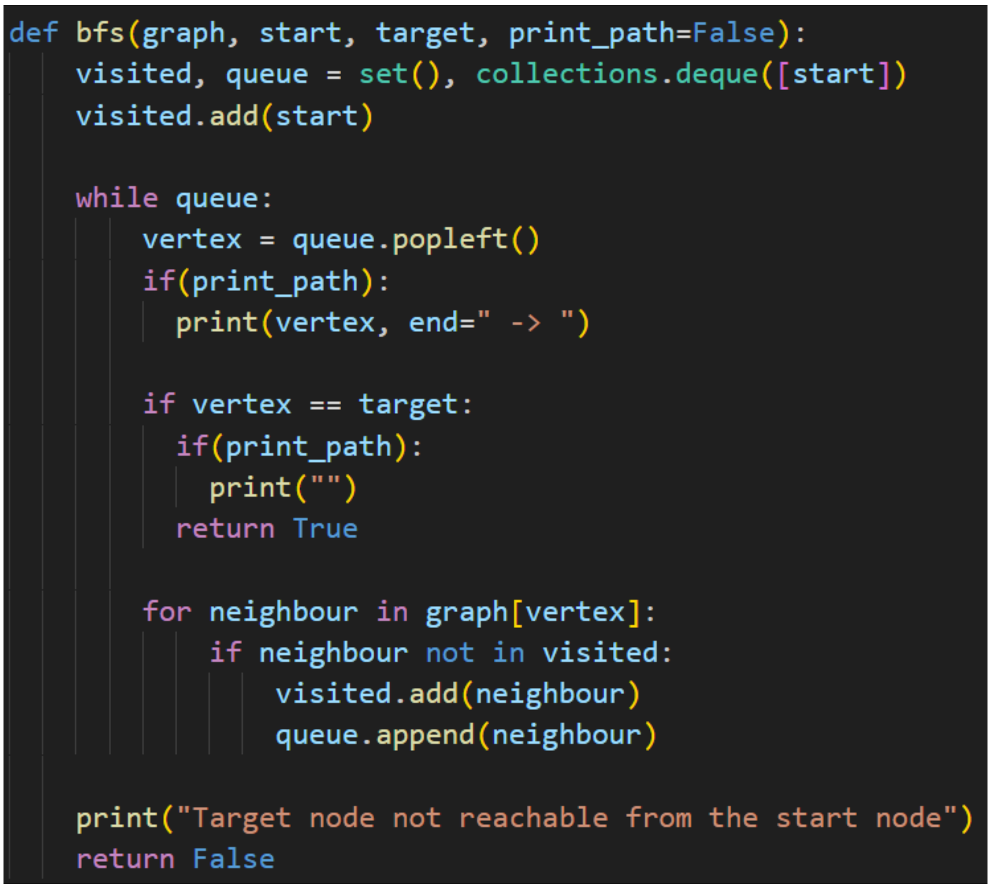
BFS

Algorithm Description:

The Breadth-first search (BFS) algorithm is used for searching a tree data structure for a specific node that meets a given criterion. It begins at the root of the tree and examines all the nodes at the current level before proceeding to the nodes at the next level. To keep track of the child nodes that have been encountered but not yet explored, an additional memory structure such as a queue is required.

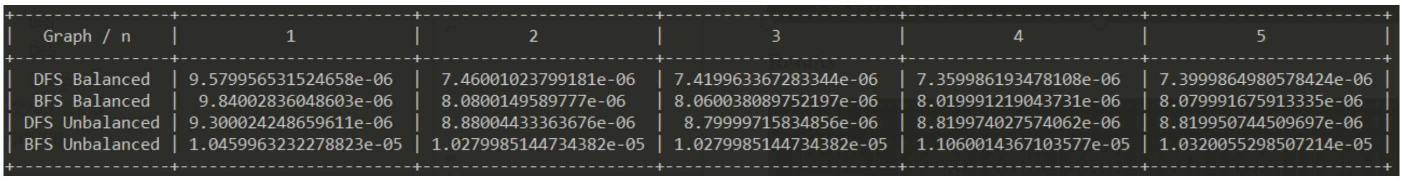
As an illustration, in a chess game, the chess engine may construct the game tree from the present position by trying all possible moves and employing breadth-first search to locate a winning position for white. Implicit trees like game trees may have an infinite size; however, breadth-first search is capable of locating a solution node if one exists.

Implementation



```python
def bfs(graph, start, target, print_path=False):
    visited, queue = set(), collections.deque([start])
    visited.add(start)

    while queue:
        vertex = queue.popleft()
        if(print_path):
            print(vertex, end=" -> ")

        if vertex == target:
            if(print_path):
                print("")
            return True

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

    print("Target node not reachable from the start node")
    return False
```

Results



```
+----------------+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
|   Graph / n    |           1           |           2           |           3           |           4           |           5           |
+----------------+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
|  DFS Balanced  | 9.579956531524658e-06 |  7.46001023799181e-06 | 7.419963367283344e-06 | 7.359986193478108e-06 | 7.3999864980578424e-06 |
|  BFS Balanced  |  9.84002836048603e-06 |  8.0800149589777e-06  | 8.060038089752197e-06 | 8.019991219043731e-06 | 8.079991675913335e-06 |
| DFS Unbalanced | 9.300024248659611e-06 |  8.88004433363676e-06 |  8.79999715834856e-06 | 8.819974027574062e-06 | 8.819950744509697e-06 |
| BFS Unbalanced | 1.0459963232278823e-05 | 1.0279985144734382e-05 | 1.0279985144734382e-05 | 1.1060014367103577e-05 | 1.0320055298507214e-05 |
+----------------+-----------------------+-----------------------+-----------------------+-----------------------+-----------------------+
```

The numbers in first row that each are themselves columns contain how many nodes we are looking for in the graph and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. The time complexity of this algorithm is T(V+E), where V is vertices and E is edges. To understand why this is the case, let's consider the steps involved in BFS:

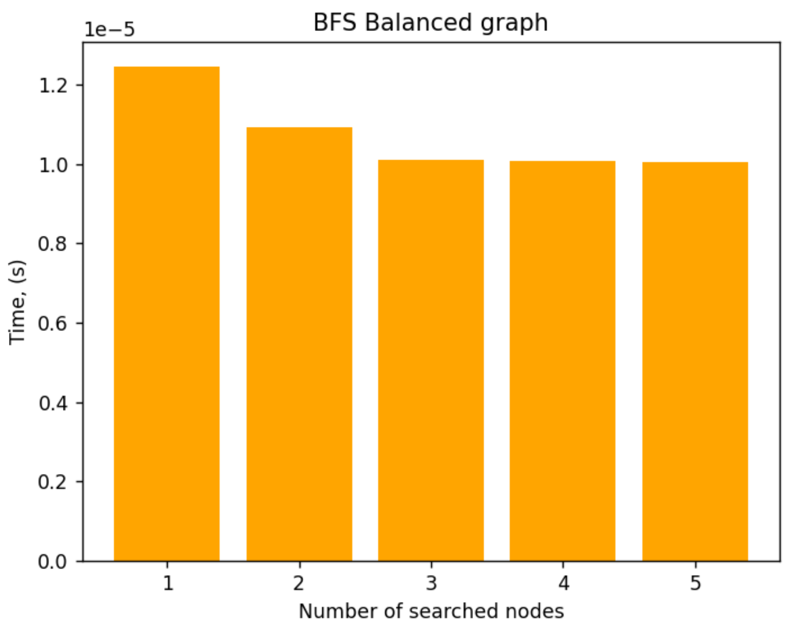
1. Initialize the queue with the starting vertex and mark it as visited.
2. While the queue is not empty, remove the vertex from the queue and explore all its neighbors.
3. If a neighbor has not been visited before, mark it as visited and add it to the queue.

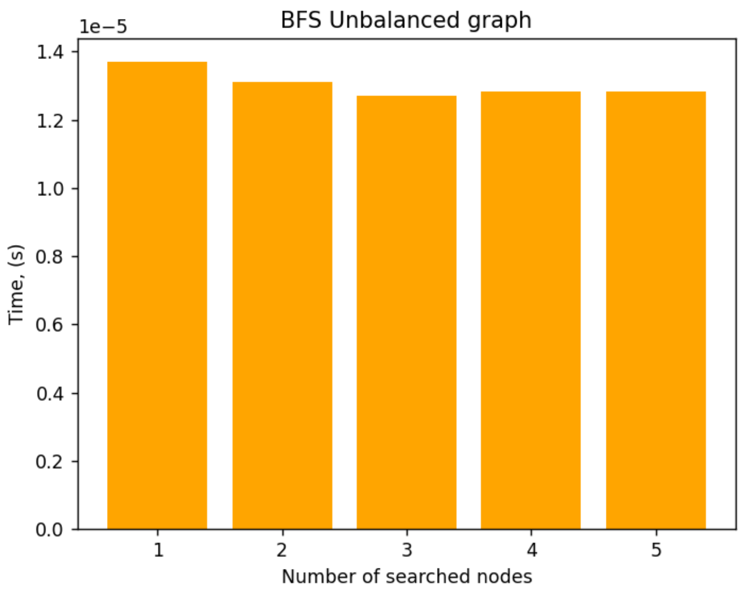
Step 1 takes constant time O(1) as it only involves initializing a queue and a set. Step 2 involves removing a vertex from the queue and checking its neighbors. Since each vertex can only be removed from the queue once and each edge is checked once, the time complexity of Step 2 is O(V+E).

Step 3 involves adding a new vertex to the queue, and marking it as visited. The time complexity of this step is O(1) for each vertex, since each vertex is added to the queue and visited only once.

Therefore, the overall time complexity of BFS algorithm for searching a graph is O(V+E), as Step 2 dominates the running time of the algorithm. In practice, this time complexity is generally considered to be efficient, especially for sparse graphs where E is much smaller than V^2.

Graph



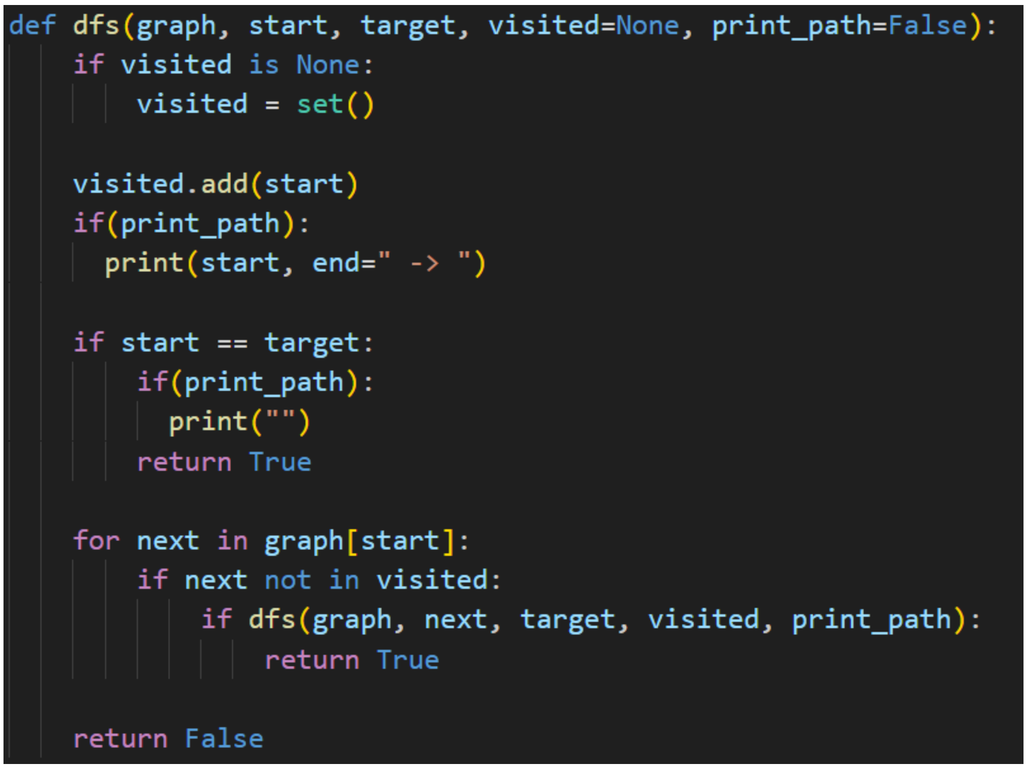


DFS

Algorithm Description:

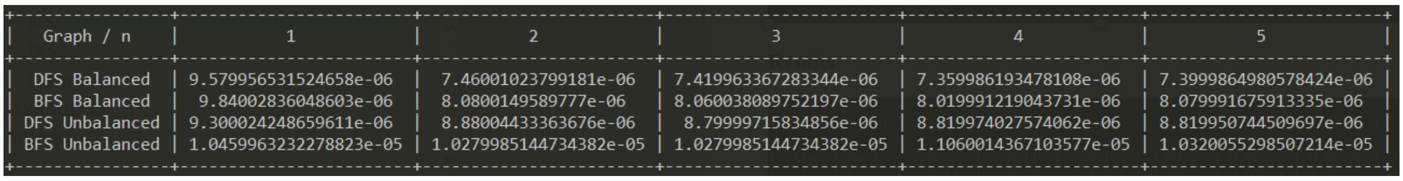
The Depth-First Search (DFS) algorithm is utilized for exploring or searching tree or graph data structures. To start the algorithm, the root node is chosen, and in the case of a graph, any arbitrary node may serve as the root. The algorithm explores as far as possible along each branch before backtracking. To keep track of the nodes encountered thus far along a particular branch and facilitate backtracking of the graph, extra memory, usually a stack, is required.

Implementation



```python
def dfs(graph, start, target, visited=None, print_path=False):
    if visited is None:
        visited = set()

    visited.add(start)
    if(print_path):
        print(start, end=" -> ")

    if start == target:
        if(print_path):
            print("")
        return True

    for next in graph[start]:
        if next not in visited:
            if dfs(graph, next, target, visited, print_path):
                return True

    return False
```

Results



```
+----------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+
|   Graph / n    |            1            |            2            |            3            |            4            |            5            |
+----------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+
|  DFS Balanced  |  9.579956531524658e-06  |  7.46001023799181e-06   |  7.419963367283344e-06  |  7.359986193478108e-06  |  7.3999864980578424e-06 |
|  BFS Balanced  |   9.84002836048603e-06  |  8.0800149589777e-06    |  8.060038089752197e-06  |  8.019991219043731e-06  |  8.079991675913335e-06  |
| DFS Unbalanced |  9.300024248659611e-06  |  8.88004433363676e-06   |   8.79999715834856e-06  |  8.819974027574062e-06  |  8.819950744509697e-06  |
| BFS Unbalanced | 1.0459963232278823e-05  | 1.0279985144734382e-05  | 1.0279985144734382e-05  | 1.1060014367103577e-05  | 1.0320055298507214e-05  |
+----------------+-------------------------+-------------------------+-------------------------+-------------------------+-------------------------+
```

The numbers in first row that each are themselves columns contain how many nodes we are looking for in the graph and the values in the table represent the time in seconds and I measure that using the function **perf_counter** from **time** module. The time complexity of this algorithm is T(V+E), where V is vertices and E is edges. To understand why this is the case, let's consider the steps involved in DFS:

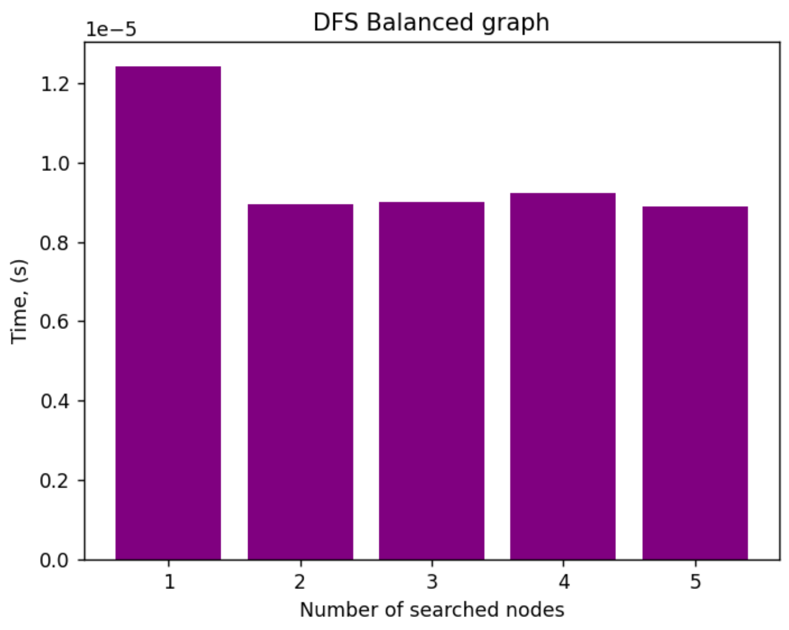
1. Choose a starting vertex and mark it as visited.
2. Explore all unvisited neighbors of the vertex, recursively.
3. When no more unvisited neighbors are left, backtrack to the previous vertex and repeat step 2 for that vertex.
4. If all vertices have been visited, terminate the search.

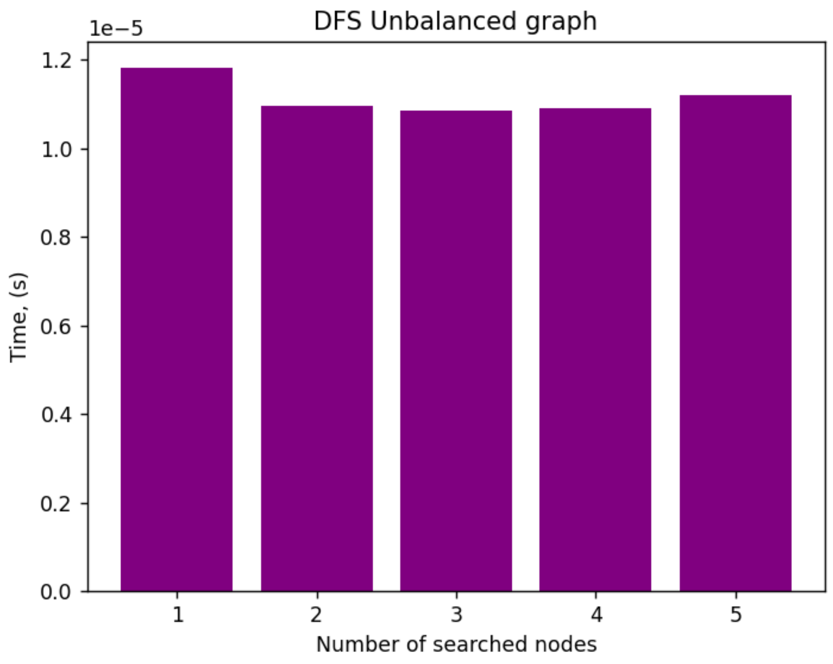
Step 1 takes constant time O(1) as it only involves initializing a set to mark visited vertices. Step 2 involves recursively exploring all unvisited neighbors of a vertex, which requires visiting each edge at most once. Thus, the time complexity of Step 2 is O(V+E).

Steps 3 and 4 involve backtracking and termination, respectively, and their time complexity is negligible compared to Step 2.

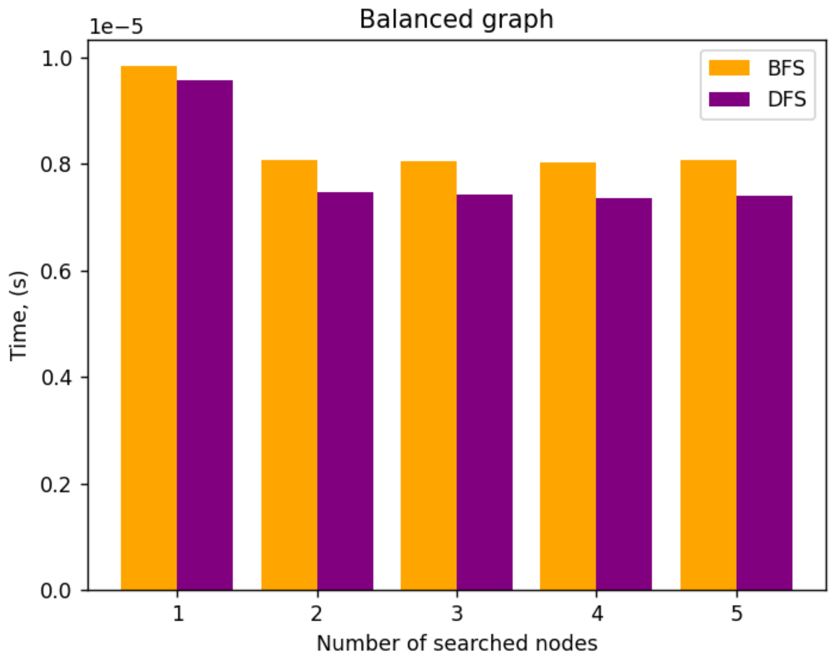
Therefore, the overall time complexity of DFS algorithm for searching a graph is O(V+E), as Step 2 dominates the running time of the algorithm. In practice, this time complexity is generally considered to be efficient, especially for sparse graphs where E is much smaller than V^2.

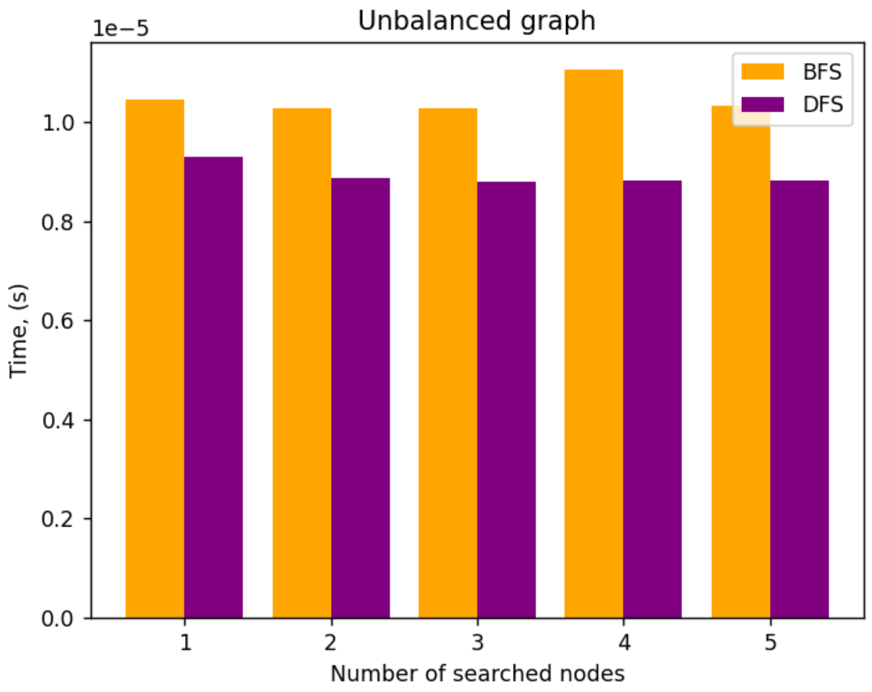
Graph


DFS Balanced graph
1e−5
Time, (s)
Number of searched nodes
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1
2
3
4
5


DFS Unbalanced graph
1e−5
Time, (s)
Number of searched nodes
0.0
0.2
0.4
0.6
0.8
1.0
1.2
1
2
3
4
5

Comparison





DFS is often faster than BFS on an unbalanced graph because it goes deeper into a path before backtracking, which can reduce the number of recursive calls and memory usage. On the other hand, BFS is better on a balanced graph because it explores all the nodes at the same level before moving on to the next level, resulting in a more uniform traversal.

DFS is also better suited for solving problems that require finding a path or a solution that is deep in the graph, as it can quickly traverse to the bottom of a path. In contrast, BFS is better for finding the shortest path between two nodes, as it guarantees that the first path found is the shortest one.

However, DFS may not always find the shortest path or the optimal solution, as it can get stuck in a local maximum or minimum. BFS, on the other hand, always finds the shortest path or solution as it explores all the nodes at a given level before moving on to the next level.

CONCLUSION

From the conducted analysis, it can be concluded that the choice between DFS and BFS depends on the characteristics of the graph and the specific problem being solved. While DFS outperforms BFS on unbalanced graphs, BFS has a slight edge on balanced graphs, and both algorithms have their strengths and weaknesses depending on the problem at hand.

If the goal is to find a solution deep in the graph or traverse quickly to the bottom of a path, DFS is a better choice. However, if the goal is to find the shortest path or the optimal solution, BFS is the better algorithm, as it always explores all the nodes at a given level before moving on to the next level.
It's important to note that DFS may not always find the shortest path or the optimal solution, as it can get stuck in a local maximum or minimum. On the other hand, BFS guarantees that the first path found is the shortest one.

In conclusion, the choice between DFS and BFS should be based on the problem requirements and the graph characteristics. Both algorithms have their strengths and weaknesses, and a careful consideration of the problem at hand will lead to the selection of the most appropriate algorithm.

GITHUB

Sufferal/Algorithms: Analysis of algorithms (github.com)