Ministry of Education and Research of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Algorithm Analysis

Laboratory work #3

Study and Empirical Analysis of DFS and BFS Algorithms

Author:
Mihaela Catan
std. gr. FAF-231

Verified:
Cristofor Fiștic

Chișinău 2025

Contents

1 Objective

Study and analyze the following algorithms: Depth First Search (DFS) and Breadth First Search (BFS).

2 The Task

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.

3 Theorethical Notes

Empirical analysis is an evidence-based approach to the study and interpretation of information. Empirical evidence is information that can be gathered from experience or by the five senses. In a scientific context, it is called empirical research. Empirical analysis requires evidence to prove any theory.

Steps in Empirical Analysis:

1. Establish the purpose of the analysis.
2. Decide the efficiency metric.
3. Establish the properties of the input data in relation to which the analysis is performed.
4. Implement the algorithms in a programming language.
5. Generate multiple sets of inputs.
6. Analyze the results.

4 Introduction

Graph traversal is the process of visiting each vertex in a graph. Graph traversal algorithms are fundamental in graph theory and computer science. They provide systematic ways to explore graphs, ensuring that each vertex is visited in a specific order. This systematic approach helps in various tasks like searching, pathfinding, and analyzing the connectivity of the graph.

Different traversal algorithms have their own strategies for visiting vertices. Some algorithms, like Breadth-First Search (BFS), explore all neighboring vertices before moving to the next level. Others, like Depth-First Search (DFS), go as far as possible along each branch before backtracking. These methods help in solving different types of problems, from finding the shortest path to detecting cycles in the graph. For a deeper understanding of how these algorithms fit into the broader context of graph databases, check out this comprehensive guide on graph database models.

5 Types of Graph Traversal Algorithms

Some of the most popular graph traversal algorithms are:

- **Breadth First Search**

  Breadth-First Search (BFS) explores all neighboring vertices before moving to the next level. BFS uses a queue to keep track of the next vertex to visit. This ensures that all vertices at the current level are explored before moving to the next level.

  BFS is particularly useful for finding the shortest path in an unweighted graph. Since it explores all nodes at the present depth level before moving on, it guarantees that the first time it reaches the target node, it has found the shortest path. This makes BFS a go-to algorithm for scenarios like finding the shortest route in a maze or navigating through a network.

- **Depth First Search**

  Depth-First Search (DFS) explores as far as possible along each branch before backtracking. Think of it as diving deep into the graph, exploring each path to its end before moving on to the next path. DFS uses a stack, either explicitly or through recursion, to keep track of the vertices to visit next.

  DFS is useful for detecting cycles and exploring connected components. By diving deep into each branch, DFS can identify loops in the graph, making it effective for cycle detection. Additionally, DFS can help in identifying all vertices connected to a given vertex, which is useful in applications like finding clusters in social networks.

6 Complexity Analysis

In this analysis, the algorithms are evaluated based on their time complexity and space complexity.

- **Time Complexity:** The number of operations performed as a function of the input size n.
- **Space Complexity:** The amount of additional memory required by the algorithm.

7 Technical Implementation

This analysis implements two algorithms and evaluates their time and space complexities on 8 types of graphs: Sparse, Dense, Grid, Cycle, Star, Complete, Barabasi, Watts-Strogatz with sizes: 50, 100, 200, 300, 400, 500. The experiments are conducted on a backend hosted on Google Compute Engine, utilizing Python 3 and approximately 12 GB of RAM. While this environment offers a suitable platform for algorithmic analysis, several limitations and potential sources of error must be acknowledged. These include the possibility that certain algorithms may exhibit suboptimal performance on larger input sizes, as well as inherent inaccuracies in time and memory measurements due to factors such as Python's interpreter overhead and the effects of garbage collection.

7.1 Test Cases

A sparse graph is a type of graph in which the number of edges is significantly less than the maximum number of possible edges. In other words, only a few nodes (or vertices) are connected to each other compared to the total number of connections that could exist.

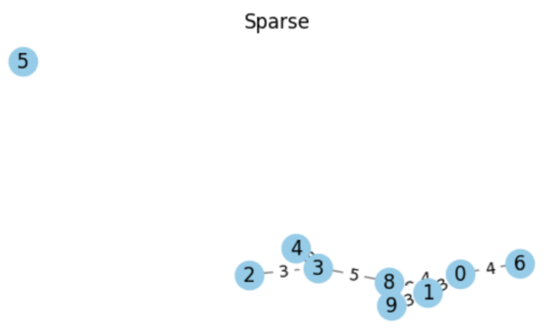

Figure 1: Sparse Graph with 10 nodes

A dense graph is a type of graph where the number of edges is close to the maximum number of possible edges. In simple terms, most of the vertices are connected to each other and leading to a high level of connectivity.

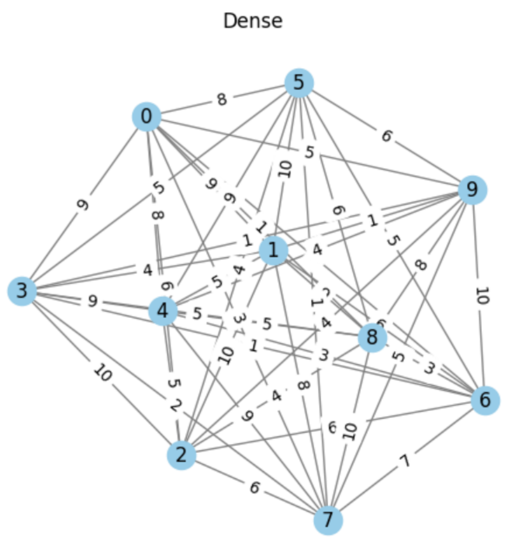

Figure 2: Dense Graph with 10 nodes

A grid graph is a type of graph in mathematics and computer science that represents a grid or lattice structure. It consists of vertices and edges arranged in a rectangular or square grid pattern. Each vertex corresponds to a point in the grid, and edges connect vertices that are adjacent horizontally or vertically. Grid graphs are often used in problems involving pathfinding, optimization, and spatial analysis.

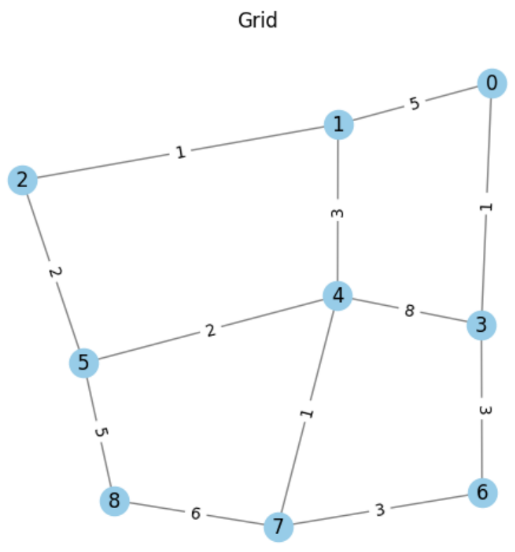

Figure 3: Grid Graph with 10 nodes

A cycle graph is a type of graph in mathematics and graph theory that consists of a single cycle. In simpler terms, it is a closed chain of vertices where each vertex is connected to exactly two other vertices.

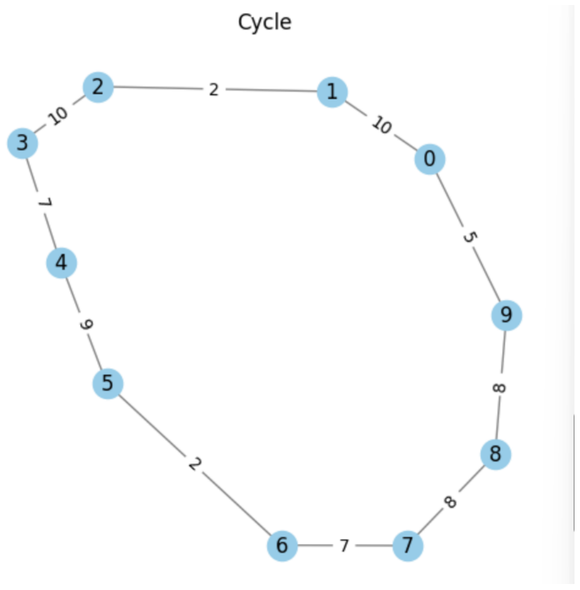

Figure 4: Cycle Graph with 10 nodes

A star graph is a type of graph in mathematics and graph theory that resembles a star shape. It consists of one central vertex connected to several outer vertices, with no connections between the outer vertices themselves.

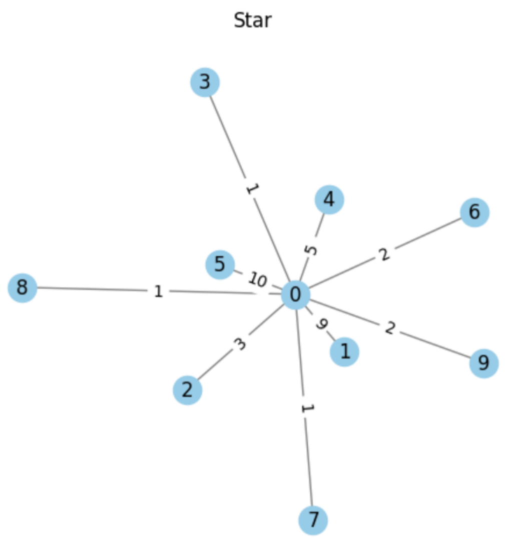

Figure 5: Star Graph with 10 nodes

A complete graph is a type of graph in mathematics and graph theory where every pair of distinct vertices is connected by a unique edge.

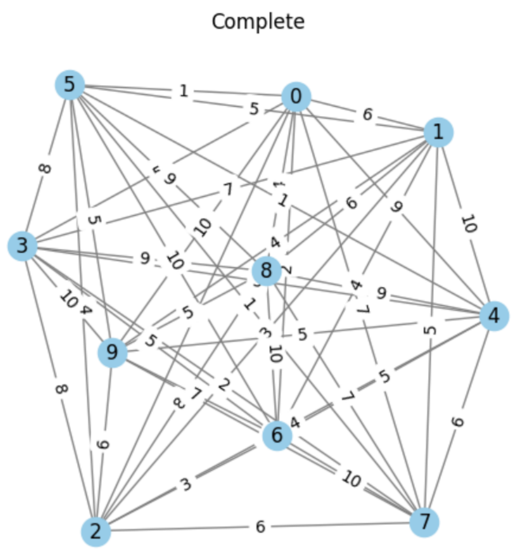

Figure 6: Complete Graph with 10 nodes

A Barabási graph, often referred to as a Barabási–Albert graph, is a type of graph generated using the Barabási–Albert model. This model is used to create scale-free networks, which are networks where some nodes (called hubs) have significantly more connections than others. The graph grows over time, and new nodes preferentially attach to existing nodes with higher degrees (more connections). This mechanism is known as preferential attachment.

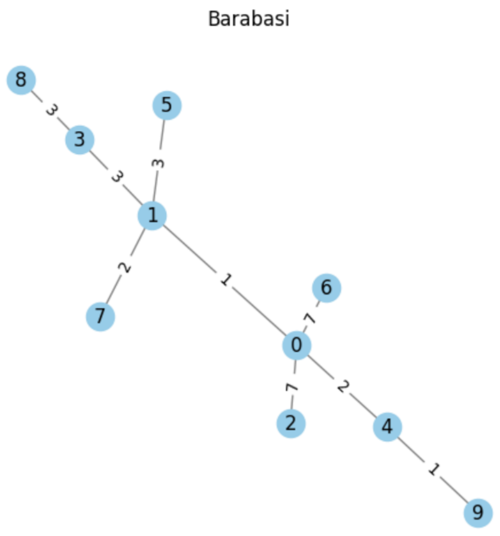

Figure 7: Barabasi Graph with 10 nodes

A Watts-Strogatz graph is a type of graph generated using the Watts-Strogatz model, which creates small-world networks. These networks are characterized by short average path lengths and high clustering, resembling many real-world networks like social networks or neural networks.

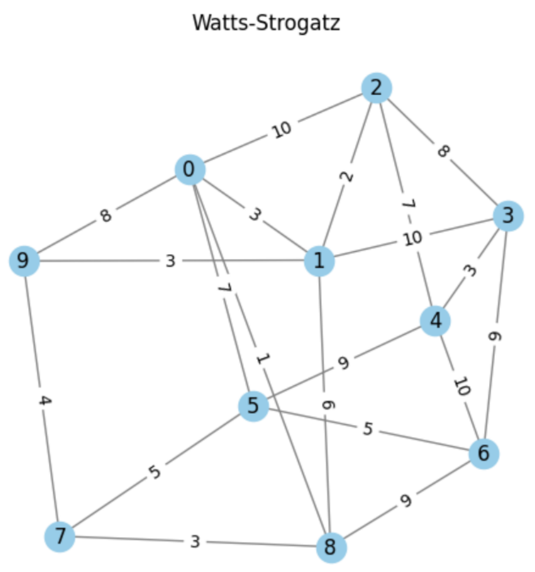

Figure 8: Watts-Strogatz Graph with 10 nodes

7.2 Depth First Search

Depth first Search or Depth first traversal is a recursive algorithm for searching all the vertices of a graph or tree data structure. Traversal means visiting all the nodes of a graph.

A standard DFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

7.2.1 How it works:

1. Start by putting any one of the graph's vertices on top of a stack.
2. Take the top item of the stack and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the top of the stack.
4. Keep repeating steps 2 and 3 until the stack is empty.

7.2.2 Python Implementation:

```
def dfs_optimized(graph, start, visited=None):
  if visited is None:
    visited = set()
  visited.add(start)
  print("Visiting", start)
  for next in graph[start] - visited:
    dfs_optimized(graph, next, visited)
  return visited
```

The time complexity of the DFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

7.3 Breadth First Search

Traversal means visiting all the nodes of a graph. Breadth First Traversal or Breadth First Search is a recursive algorithm for searching all the vertices of a graph or tree data structure.

A standard BFS implementation puts each vertex of the graph into one of two categories:

- Visited
- Not visited

The purpose of the algorithm is to mark each vertex as visited while avoiding cycles.

7.3.1 How it works:

1. Start by putting any one of the graph's vertices at the back of a queue.
2. Take the front item of the queue and add it to the visited list.
3. Create a list of that vertex's adjacent nodes. Add the ones which aren't in the visited list to the back of the queue.
4. Keep repeating steps 2 and 3 until the queue is empty.

The graph might have two different disconnected parts so to make sure that we cover every vertex, we can also run the BFS algorithm on every node.

7.3.2 Python Implementation:

```
from collections import deque
def bfs_optimized(graph, start):
    visited = set()
    queue = deque([start])

    while queue:
        node = queue.popleft()
        if node not in visited:
            print("Visiting", node)
            visited.add(node)
            queue.extend(graph[node] - visited)

    return visited
```

The time complexity of the BFS algorithm is represented in the form of O(V + E), where V is the number of nodes and E is the number of edges.

The space complexity of the algorithm is O(V).

8 Performance Results:

In Sparse graphs, BFS outperforms DFS in both execution time and memory usage starting from size 50. This is because Sparse graphs have fewer edges, making BFS's systematic traversal across breadth levels faster and more efficient. DFS, which dives deeper into graph paths, ends up expending more time and memory on recursive operations even in lightly connected structures.

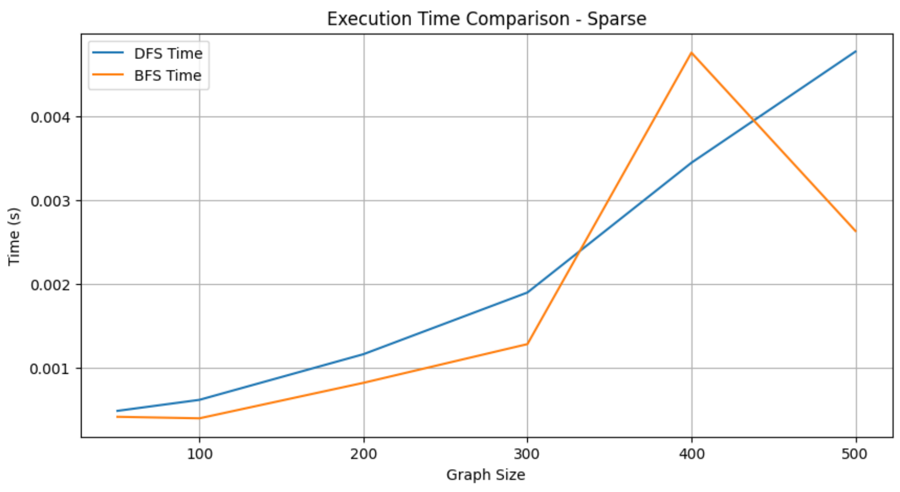

Figure 9: Time Usage on Sparse Graph

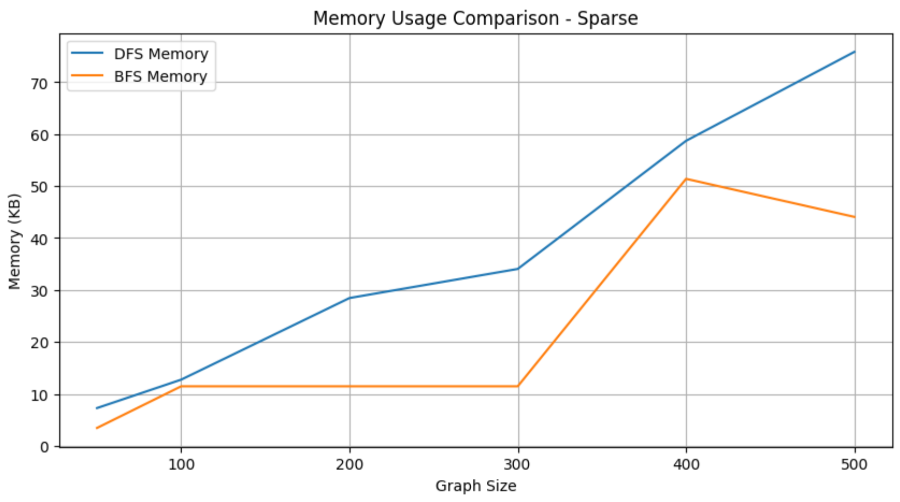

Figure 10: Space Usage on Sparse Graph

Dense graphs show BFS outperforming DFS from size 50 onward in both performance metrics. The high number of edges in dense graphs creates complex paths that DFS navigates exhaustively, increasing both time and memory consumption. BFS, on the other hand, benefits from traversing breadth-first levels, minimizing repeated processing of interconnected nodes.

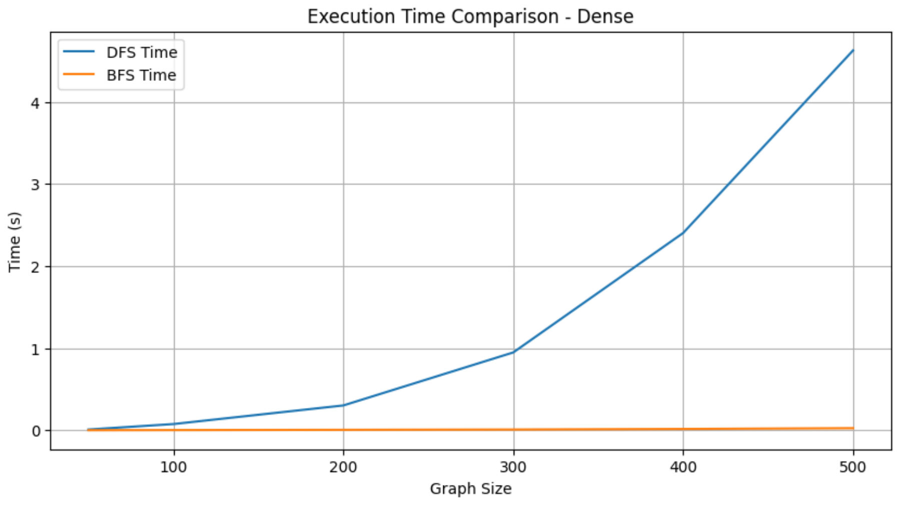

Figure 11: Time Usage on Dense Graph

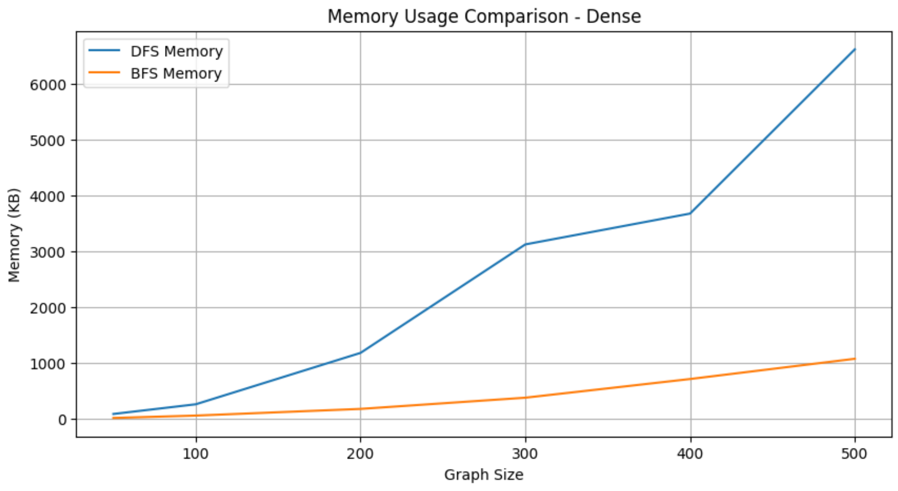

Figure 12: Space Usage on Dense Graph

In Grid graphs, BFS outperforms DFS starting from size 50 for execution time and memory usage. The systematic and evenly distributed structure of grids allows BFS to traverse breadth levels efficiently, while DFS expends additional resources on recursive path navigation through the grid's uniform layout.

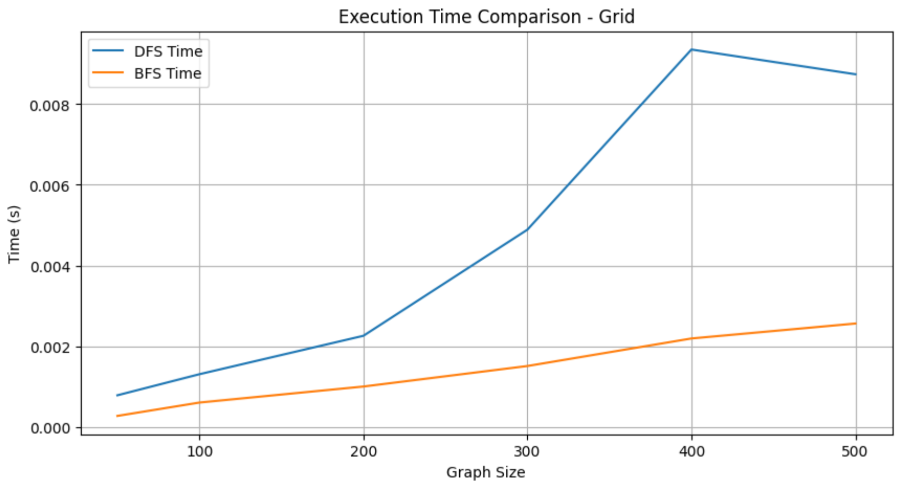

Figure 13: Time Usage on Grid Graph

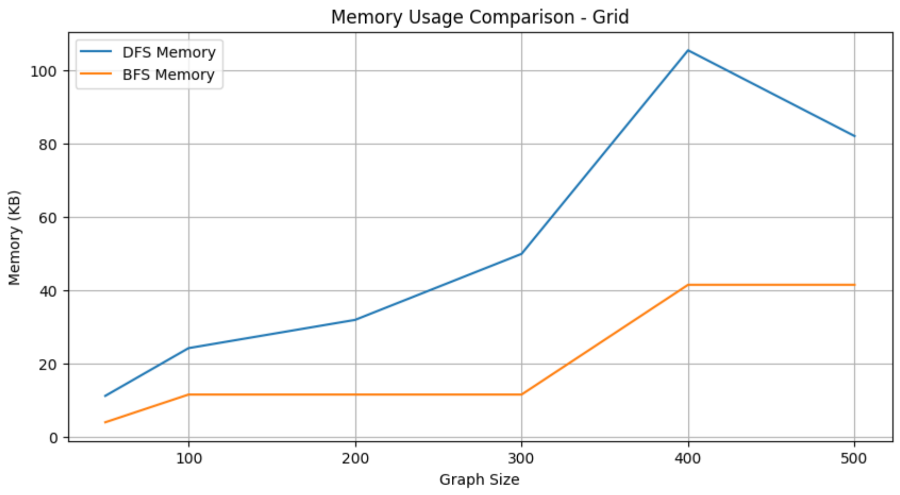

Figure 14: Space Usage on Grid Graph

Cycle graphs demonstrate BFS's superiority over DFS from size 50 onward. Since cycles involve repeated paths, BFS's breadth-first traversal avoids redundant depth explorations that DFS encounters. This efficiency translates into lower time and memory usage for BFS.

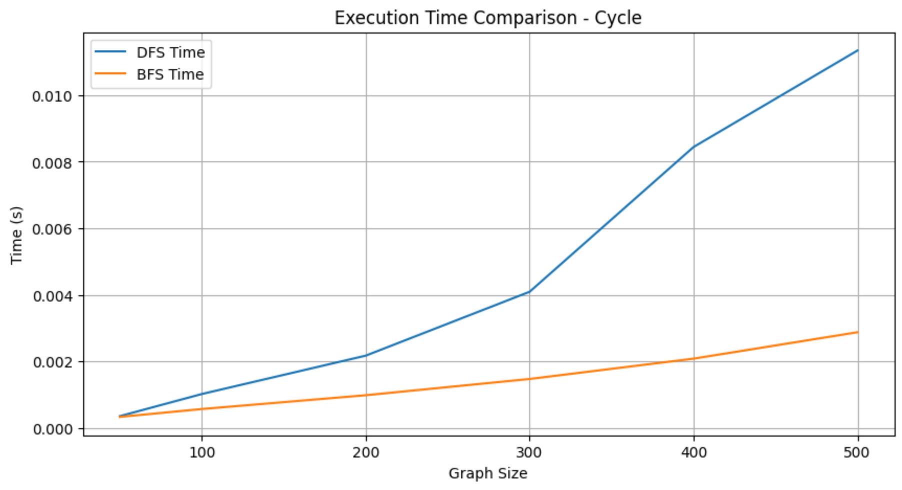

Figure 15: Time Usage on Cycle Graph

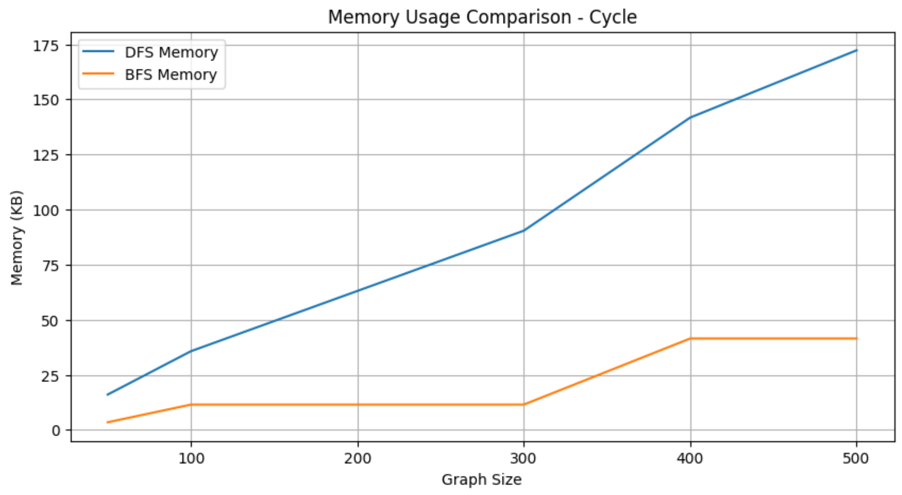

Figure 16: Space Usage on Cycle Graph

In Star graphs, BFS uses less memory than DFS starting from size 100, while execution time remains comparable in smaller sizes. This is because the centralized hub structure of Star graphs makes DFS's deeper recursive calls more memory-intensive, while BFS handles breadth exploration efficiently without deep recursions. But the execution time is higher for the BFS traversal than for the DFS traversal.

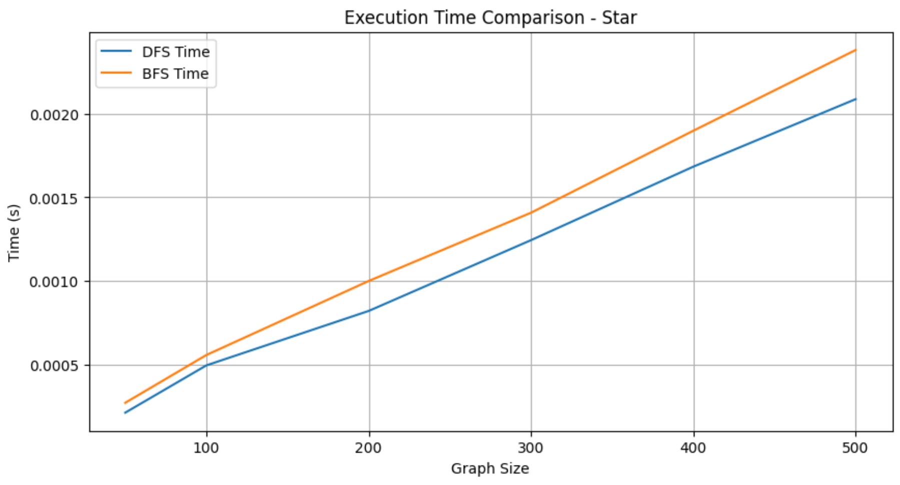

Figure 17: Time Usage on Star Graph

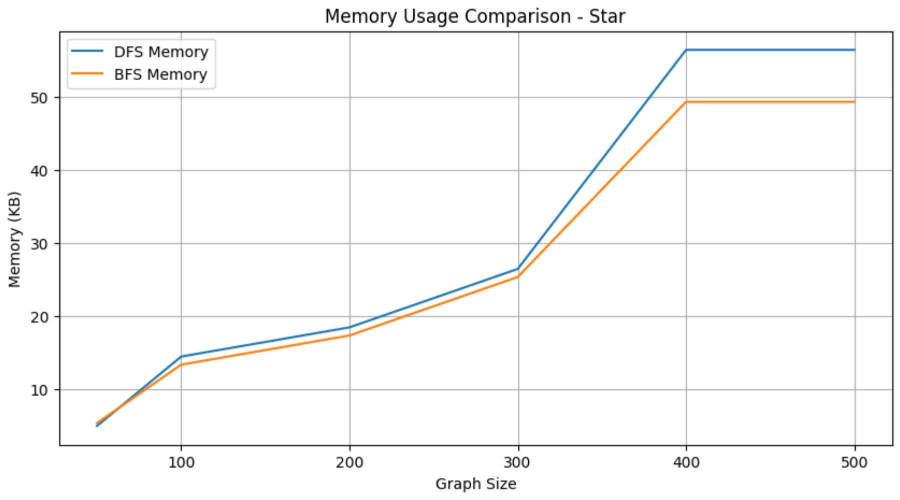

Figure 18: Space Usage on Star Graph

Complete graphs, with all nodes interconnected, present the largest differences in performance metrics. BFS outperforms DFS in both execution time and memory usage starting from size 50. The exhaustive depth-first traversal performed by DFS becomes extremely resource-intensive due to the graph's dense connectivity, whereas BFS processes breadth levels more systematically and avoids redundant depth operations.

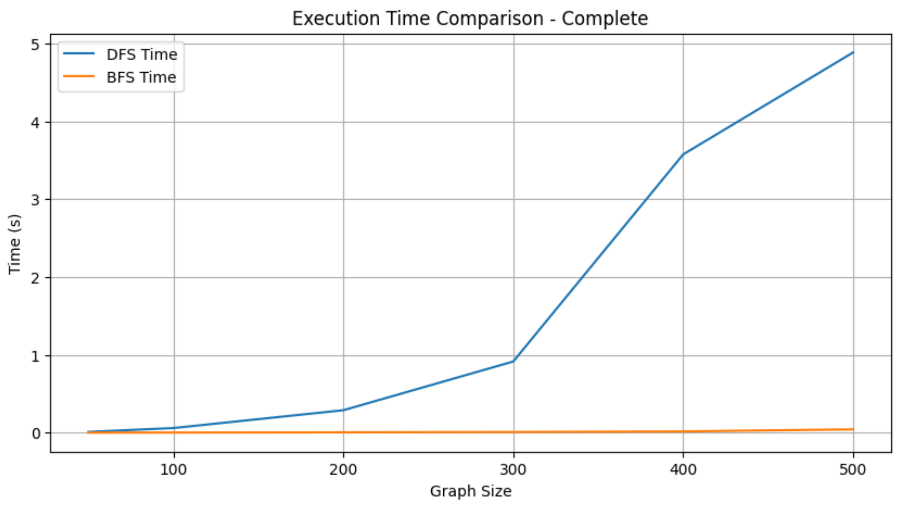

Figure 19: Time Usage on Complete Graph

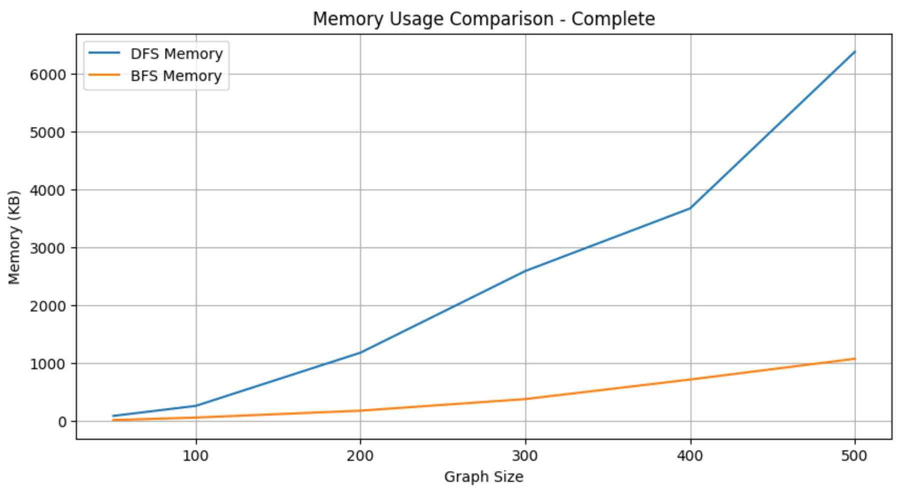

Figure 20: Space Usage on Complete Graph

Barabási graphs exhibit BFS's advantages from size 50 in execution time and memory usage. The preferential attachment property of these graphs leads DFS to navigate highly connected hubs with costly recursive operations, while BFS handles hub traversals efficiently by level.

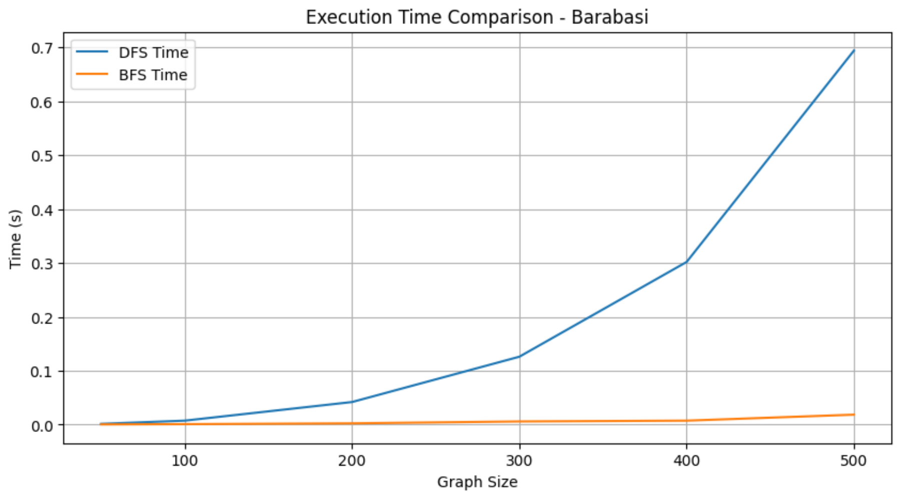

Figure 21: Time Usage on Barabasi Graph

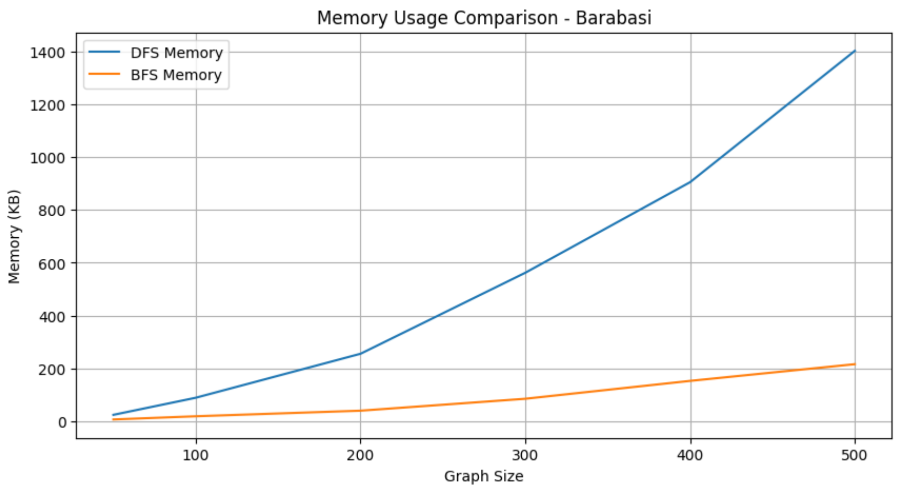

Figure 22: Space Usage on Barabasi Graph

Watts-Strogatz graphs also show BFS outperforming DFS from size 50 onward. These graphs feature small-world properties and short average path lengths that BFS leverages for faster breadth exploration, whereas DFS navigates deeper paths at a higher computational cost.

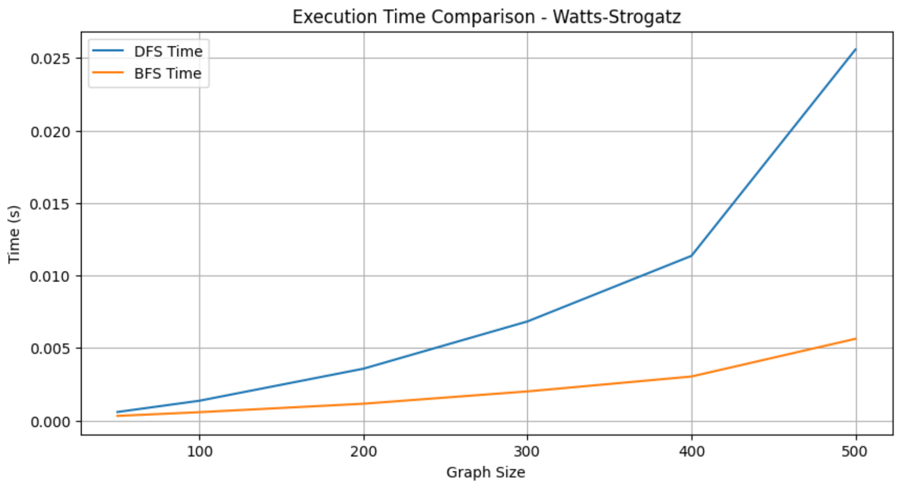

Figure 23: Time Usage on Watts-Strogatz Graph

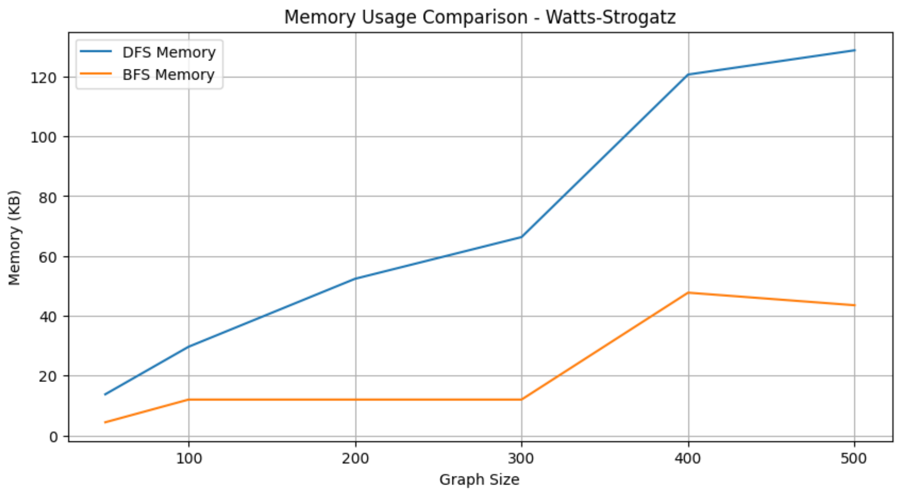

Figure 24: Space Usage on Watts-Strogatz Graph

Conclusion

This structured analysis highlights the cause-and-effect relationship between graph topology and algorithm performance. BFS's breadth-first traversal provides systematic exploration, making it more efficient in execution time and memory usage across graph types, especially as size and complexity increase. DFS's depth-first recursive approach, while thorough, struggles with higher memory demands and redundant operations in interconnected or cyclic structures. This study emphasizes the importance of understanding graph properties when selecting an algorithm to optimize performance.

9 Bibliography

- The code is available at: Google Colab Notebook
- GitHub Repository: GitHub