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

Laboratory Work 3

Study and empirical analysis of sorting algorithms
Analysis of Breadth-First Search, Depth-First Search

Elaborated:
st. gr. FAF-233 Moraru Patricia

Verified:
asist. univ. Fiștic Cristofor

Chișinău - 2025

Contents

Chapter 1

Introduction

1.1 Objective

The primary objective of this laboratory work is to study and analyze the efficiency of graph traversal algorithms, specifically Breadth-First Search (BFS) and Depth-First Search (DFS). The analysis will be conducted through empirical testing, evaluating the performance of these algorithms across various types of graphs with differing properties such as size, sparsity, directionality, and connectivity. The goal is to compare their efficiency using selected performance metrics (such as execution time and memory usage) and to visualize the results in order to draw meaningful conclusions about their behavior under different conditions.

1.2 Tasks

1. Implement the BFS and DFS algorithms in a programming language.
2. Establish the properties of the graph input data (e.g., directed/undirected, sparse/-dense, connected/disconnected).
3. Choose appropriate metrics for comparing algorithm performance (e.g., execution time, memory usage).
4. Perform empirical analysis of the BFS and DFS algorithms on various graph configurations.
5. Present the results graphically using plots or charts.
6. Draw conclusions based on the observed results and trends.

1.3 Mathematical vs. Empirical Analysis

Understanding algorithm performance involves two primary approaches: mathematical analysis and empirical analysis. The following table summarizes their differences:

Mathematical Analysis	Empirical Analysis
The algorithm is analyzed with the help of mathematical deviations and there is no need for specific input.	The algorithm is analyzed by taking some sample of input and no mathematical deviation is involved.
The principal weakness of these types of analysis is its **limited applicability**.	The principal strength of empirical analysis is that it is **applicable to any algorithm**.
The principal strength of mathematical analysis is that it is independent of any input or the computer on which the algorithm is running.	The principal weakness of empirical analysis is that it depends upon the sample input taken and the computer on which the algorithm is running.

Table 1.1: Comparison of Mathematical and Empirical Analysis

1.4 Theoretical Notes on Empirical Analysis

Empirical analysis in computer science refers to the experimental evaluation of algorithms to determine their performance characteristics based on observed data rather than solely relying on theoretical models. It is a crucial method when mathematical analysis is challenging or when practical performance under real-world conditions needs to be assessed.

1.4.1 Purpose of Empirical Analysis

- To obtain preliminary insights into the complexity class of an algorithm.
- To compare the efficiency of two or more algorithms solving the same problem.
- To evaluate different implementations of the same algorithm.
- To understand how an algorithm performs on specific hardware or software environments.

1.5 Complexity of Algorithms

There are two aspects of algorithmic performance:

1.5.1 Time Complexity

- Instructions take time.
- How fast does the algorithm perform?
- What affects its runtime?

1.5.2 Space Complexity

- Data structures take space.
- What kind of data structures can be used?
- How does the choice of data structure affect the runtime?

Here we will focus on **time**: How to estimate the time required for an algorithm.

T(n)	Name	Problems
O(1)	Constant	
O($\log n$)	Logarithmic	
O(n)	Linear	Easy-solved
O($n \log n$)	Linear-logarithmic	
O(n^2)	Quadratic	
O(n^3)	Cubic	
O(2^n)	Exponential	Hard-solved
O($n!$)	Factorial	Hard-solved

Table 1.2: Algorithm Complexity Categories

1.6 Empirical Analysis of Graph Traversal Algorithms

Empirical Analysis Process

1. **Define the Objective:** Assess and compare the performance of BFS and DFS in traversing graphs of different types and sizes.
2. **Select Performance Metrics:** Use execution time and memory usage as primary performance indicators.
3. **Determine Graph Characteristics:** Evaluate traversal performance across graph variations such as:
    - Directed vs. Undirected
    - Sparse vs. Dense
    - Connected vs. Disconnected
    - Weighted vs. Unweighted
    - Cyclic vs. Acyclic
    - Trees
4. **Implementation and Experiment Setup:** Implement both algorithms using consistent coding practices. Use graph generators to produce structured input data.
5. **Data Collection and Execution:** Run multiple experiments on graphs with varying nodes and edges. Collect timing and memory data.
6. **Analysis and Interpretation:** Use visualizations (e.g., bar charts, line graphs) to highlight algorithm trends under different graph scenarios.

Advantages of Empirical Analysis

- Provides practical performance data that complements theoretical complexity analysis.
- Identifies real-world issues such as cache misses, I/O overhead, and rounding errors.
- Helps validate theoretical predictions with observed behavior.

Challenges and Considerations

- **Hardware Dependence:** Performance results may vary significantly across different systems.
- **Implementation Bias:** Different coding styles or optimizations can affect results.
- **Measurement Noise:** External factors like background processes may introduce variability in results.

1.7 Graph Traversal Algorithms

Breadth-First Search (BFS) and **Depth-First Search (DFS)** are fundamental graph traversal algorithms that explore the nodes of a graph in distinct ways, serving as the building blocks for many complex graph algorithms.

BFS explores a graph layer by layer, visiting all the nodes at the current depth before progressing to the next layer, making it an optimal choice for finding the shortest path in unweighted graphs.

In contrast, **DFS** delves deep into the graph, traversing as far as possible along each branch before backtracking, proving effective for tasks like topological sorting and exploring all possible paths or configurations in a graph.

Both algorithms, with their unique approaches to graph traversal, offer versatile solutions to a myriad of problems in computer science, from network analysis to solving puzzles and mazes.

Each algorithm is studied in terms of time and space complexity, and their behavior is empirically evaluated on graphs with varying properties such as size, directionality, sparsity, and connectivity. The aim is to understand the practical performance of these algorithms and how different graph characteristics impact their efficiency.

1.8 Comparison Metrics

To assess the performance of graph traversal algorithms, we consider the following metrics:

- **Execution Time:** Measures the time required to sort datasets of varying sizes. Multiple runs are conducted to compute the average execution time and standard deviation for consistency.
- **Memory Usage:** Tracks the peak memory consumption during execution. This metric provides insight into the space complexity of each algorithm.

Performance measurements are obtained using:

- The `timeit` module to measure execution time over multiple repetitions.
- The `tracemalloc` module to capture peak memory usage during execution.

The collected data allows for an empirical comparison of how each sorting algorithm scales with increasing input sizes.

1.9 Input Format

The evaluation explores how BFS and DFS algorithms perform across various graph configurations. The dataset sizes correspond to different numbers of graph nodes, as follows:

$$\{10, 15, 50, 75, 100, 250, 500\} \tag{1.1}$$

For each input size, multiple graph structure variations were generated to simulate different real-world scenarios:

- **Directed vs Undirected:** Determines whether edges have a direction or not.
- **Weighted vs Unweighted:** Indicates if edges have associated weights (though BFS/DFS ignore weights, memory footprint changes).
- **Sparse vs Dense:** Varies the number of edges relative to nodes.
- **Cyclic vs Acyclic:** Includes graphs with cycles and Directed Acyclic Graphs (DAGs).
- **Connected vs Disconnected:** Evaluates performance on graphs with one or more components.
- **With Loops vs No Loops:** Includes or excludes self-loops in directed graphs.
- **Trees:** A special case of connected, acyclic graphs with minimal edges.

These input variations ensure a thorough empirical analysis of BFS and DFS behavior across diverse graph topologies.

Chapter 2

Implementation

This section outlines the implementation of the Breadth-First Search (BFS) and Depth-First Search (DFS) algorithms, along with the experimental setup designed to evaluate their performance across various graph types.

Experimental Setup

To perform a robust empirical evaluation, the experiments were run under the following setup:

- **Local Testing:** Experiments on graphs ranging from 10 to 500 nodes were conducted on a personal computer.
- **Graph Generator:** Custom functions were used to generate graphs with diverse characteristics—directed, undirected, sparse, dense, connected, disconnected, cyclic, acyclic, and tree-structured.
- **Performance Metrics:** Time complexity and memory usage were recorded for each algorithm and graph configuration.

Graph Traversal Algorithms Implementation

The subsequent sections detail the implementation of the BFS and DFS algorithms. Each implementation includes code and highlights core operational characteristics such as queue/stack usage, recursion, visited node tracking, and traversal order. Optimizations and considerations for handling large or disconnected graphs are also discussed.

Adjacency Matrix Representation

An **adjacency matrix** is a two-dimensional array used to represent a graph. It is well-suited for dense graphs and offers fast edge lookup. Each cell in the matrix corresponds to a potential edge between two vertices.

- The rows and columns represent the vertices in the graph.
- The entry at position `matrix[i][j]` indicates the presence of an edge between vertex i and vertex j.

- For unweighted graphs, values are typically 1 (edge exists) or 0 (no edge). For weighted graphs, the values represent edge weights.

Advantages of Adjacency Matrix

- **Fast Edge Lookup:** Determining if an edge exists between two vertices takes constant time, $\mathcal{O}(1)$.
- **Efficient for Dense Graphs:** Adjacency matrices handle dense graphs efficiently in terms of access and storage patterns.
- **Simple Implementation for Small Graphs:** Straightforward to implement and visualize for small graphs.

Disadvantages of Adjacency Matrix

- **Memory Inefficiency for Sparse Graphs:** Always allocates space for n^2 entries, even if many connections are missing.
- **Slower Neighbor Iteration:** Identifying all neighbors of a vertex requires scanning an entire row or column.
- **Limited Scalability:** Consumes a large amount of memory as the number of vertices increases, making it less suitable for large, sparse graphs.

Adjacency List Representation

An **adjacency list** is a common and efficient data structure used to represent graphs, particularly sparse ones. Each vertex in the graph maintains a list of its adjacent (neighboring) vertices.

- For each vertex, a list stores all the vertices directly connected to it by an edge.
- It can be implemented using:
  - An array of lists (or arrays), where the index represents the vertex, and the list contains its neighbors.
  - A dictionary (or hashmap), where keys represent vertices and values are lists of adjacent vertices.

Advantages of Adjacency List

- **Memory Efficiency:** Particularly effective for sparse graphs, as only existing edges are stored.
- **Easy Neighbor Iteration:** Direct access to a vertex's neighbors allows efficient iteration during traversals like BFS and DFS.
- **Scales Well for Sparse Graphs:** Requires significantly less memory than adjacency matrices when the graph has few edges.

Disadvantages of Adjacency List

- **Slower Edge Lookup:** Checking whether an edge exists between two vertices requires searching the list, which can be slower than in a matrix.
- **Less Optimal for Dense Graphs:** As the number of edges increases, adjacency lists may become less efficient than matrices, both in terms of memory and edge lookup time.

2.1 Depth-First Search

Depth-First Search (DFS) is a fundamental graph traversal algorithm that explores as far as possible along each branch before backtracking. Unlike breadth-first search, DFS dives deep into the graph, visiting the children of a node before moving to its siblings.

Simple Explanation

1. **Start at a Node:** Choose an arbitrary starting node.
2. **Go Deep:** Visit a neighbor of the current node, then visit a neighbor of that neighbor, and continue this process recursively, going as deep as possible.
3. **Backtrack:** If you reach a node with no unvisited neighbors, backtrack to the previous node and explore its remaining neighbors.
4. **Repeat:** Continue this process until all reachable nodes have been visited.

Visualization

DFS can be visualized as a person navigating a maze by taking each path as far as it goes. When a dead end is reached, the person backtracks to the last branching point and tries a different path.

Key Points

- DFS uses a **stack** data structure, which can be implemented explicitly or via recursion.
- It is particularly useful for:
  - Topological sorting of directed acyclic graphs (DAGs)
  - Detecting cycles in graphs
  - Solving puzzles or exploring all possible configurations

Python Implementation:

```python
def dfs_matrix(matrix, start_index):
    n = len(matrix)
    visited = [False] * n
    stack = [start_index]

    while stack:
        node = stack.pop()
        if not visited[node]:
            visited[node] = True
            for neighbor in range(n - 1, -1, -1):
                if matrix[node][neighbor] != 0 and not visited[
                    neighbor]:
                    stack.append(neighbor)
```

Listing 2.1: Depth-First Search

2.1.1 Optimized DFS (Adjacency List)

The optimized version of Depth-First Search uses an **adjacency list** representation instead of an adjacency matrix. This significantly improves memory efficiency, especially for sparse graphs, by only storing existing connections between vertices.

In this implementation, the algorithm uses a `set` to track visited nodes and a `stack` to manage the traversal order. The stack ensures that the algorithm explores as deep as possible before backtracking, consistent with DFS logic. The adjacency list allows direct access to a node's neighbors, making the traversal faster and more scalable for large or sparse graphs.

Reversing the list of neighbors during iteration ensures that the traversal order is consistent with recursive implementations, which helps with reproducibility and debugging. Overall, this approach offers better space complexity and often faster runtime compared to matrix-based DFS, particularly when the graph is large but sparsely connected.

Python Implementation:

```python
def dfs_list(adj_list, start_node):
    visited = set()
    stack = [start_node]

    while stack:
        node = stack.pop()
        if node not in visited:
            visited.add(node)
            for neighbor, _ in reversed(adj_list.get(node, [])):
                if neighbor not in visited:
                    stack.append(neighbor)
```

Listing 2.2: Optimized Depth-First Search with Adjacency List

2.2 Breadth-First Search

Breadth-First Search (BFS) is a graph traversal algorithm that explores all the nodes in a **breadthward** fashion. It visits all nodes at the current depth (or distance from the starting node) before moving on to nodes at the next depth level.

Simple Explanation

1. **Start at a Node:** Choose a starting node.
2. **Visit Neighbors:** Visit all direct neighbors of the starting node.
3. **Move to Next Level:** After visiting all neighbors, proceed to their unvisited neighbors.
4. **Repeat:** Continue this process until all reachable nodes have been visited.

Visualization

BFS can be visualized as a wave that spreads across the graph, beginning from the initial node and expanding level by level until all nodes have been explored.

Key Points

- BFS uses a **queue** data structure to keep track of the next nodes to visit.
- It is especially effective for:
  - Finding the shortest path in unweighted graphs
  - Level-order traversal in trees
  - Solving problems that require minimum steps or distances

Python Implementation:

```
def bfs_matrix(matrix, start_index):
    n = len(matrix)
    visited = [False] * n
    queue = deque([start_index])
    visited[start_index] = True

    while queue:
        node = queue.popleft()
        for neighbor in range(n):
            if matrix[node][neighbor] != 0 and not visited[
               neighbor]:
                visited[neighbor] = True
                queue.append(neighbor)
```

Listing 2.3: Breadth-First Search

2.2.1 Optimized BFS (Adjacency List)

The optimized version of Breadth-First Search utilizes an **adjacency list** representation for improved memory efficiency and traversal speed, especially in sparse graphs. By storing only existing edges, this approach avoids the unnecessary memory overhead of an adjacency matrix.

In this implementation, a `set` is used to track visited nodes, ensuring that each vertex is visited only once. A `deque` (double-ended queue) is employed to manage the traversal order efficiently, supporting constant-time enqueue and dequeue operations.

The adjacency list provides direct access to each node's neighbors, making neighbor iteration faster and more scalable for large graphs. This method is particularly well-suited for unweighted graphs where BFS is commonly used to determine the shortest path or the minimum number of steps between nodes.

Python Implementation:

```
def bfs_list(adj_list, start_node):
    visited = set()
    queue = deque([start_node])
    visited.add(start_node)

    while queue:
        node = queue.popleft()
        for neighbor, _ in adj_list.get(node, []):
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
```

Listing 2.4: Optimized BFS (Adjacency List)

2.3 Performance Evaluation

This section presents the empirical results of applying graph traversal algorithms (BFS and DFS) to both directed and undirected graphs. We compare their performance in terms of execution time and memory consumption. The evaluation includes both adjacency matrix and adjacency list implementations for varying graph sizes.

2.3.1 Directed vs Undirected Graphs

The experiment consisted of graphs of increasing sizes (10 to 500 nodes), using both adjacency matrix and adjacency list representations. For each configuration, execution time and memory usage were recorded for both DFS and BFS.

Empirical Results Table

The table below summarizes the collected data from running each algorithm on directed and undirected graphs.

Table 2.1: Execution Time and Memory Usage – Directed vs Undirected Graphs

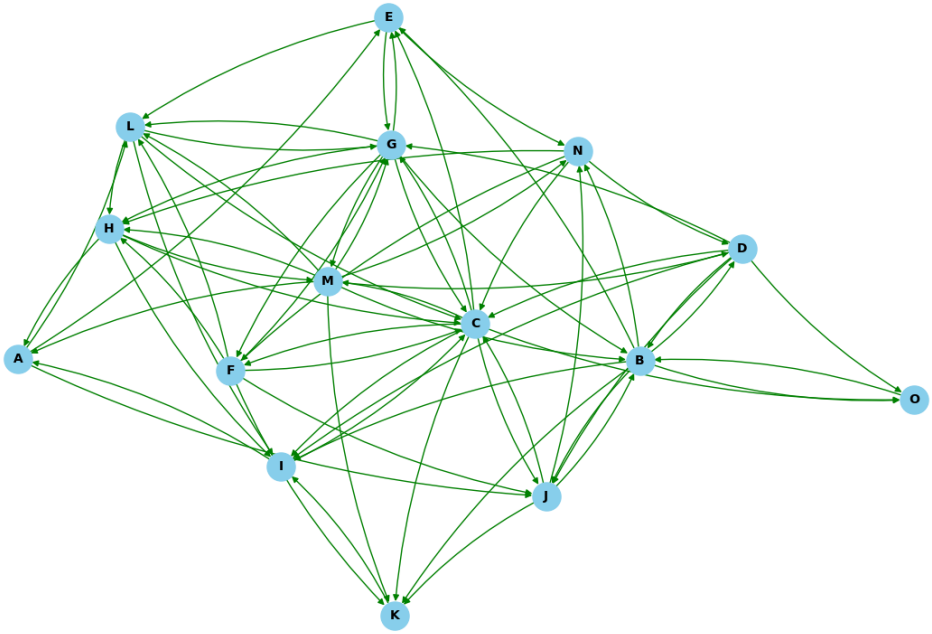
Algorithm	**Nodes**	**Type**	**Time (s)**	**Memory (B)**	**Result**
DFS (Matrix)	10	Undirected	6.4e-05	1,880	N/A
DFS (Matrix)	10	Directed	0.000164	2,240	N/A
DFS (Matrix)	15	Undirected	0.000115	3,528	N/A
DFS (Matrix)	15	Directed	0.000147	3,480	N/A
DFS (Matrix)	50	Undirected	0.000493	16,928	N/A
DFS (Matrix)	50	Directed	0.000780	24,040	N/A
DFS (Matrix)	75	Undirected	0.000499	26,240	N/A
DFS (Matrix)	75	Directed	0.001706	49,440	N/A
DFS (Matrix)	100	Undirected	0.001109	63,016	N/A
DFS (Matrix)	100	Directed	0.002079	88,544	N/A
DFS (Matrix)	250	Undirected	0.005702	353,848	N/A
DFS (Matrix)	250	Directed	0.010786	671,968	N/A
DFS (Matrix)	500	Undirected	0.027019	1,462,956	N/A
DFS (Matrix)	500	Directed	0.063535	2,058,452	N/A
BFS (Matrix)	10	Undirected	9.8e-05	1,880	N/A
BFS (Matrix)	10	Directed	0.000131	2,240	N/A
BFS (Matrix)	15	Undirected	9e-05	3,528	N/A
BFS (Matrix)	15	Directed	0.000244	3,480	N/A
BFS (Matrix)	50	Undirected	0.000491	16,928	N/A
BFS (Matrix)	50	Directed	0.000701	24,887	N/A
BFS (Matrix)	75	Undirected	0.000695	26,240	N/A
BFS (Matrix)	75	Directed	0.001772	49,440	N/A
BFS (Matrix)	100	Undirected	0.001201	63,016	N/A
BFS (Matrix)	100	Directed	0.002166	88,544	N/A
BFS (Matrix)	250	Undirected	0.006436	353,848	N/A
BFS (Matrix)	250	Directed	0.009898	524,040	N/A
BFS (Matrix)	500	Undirected	0.025267	1,462,060	N/A
BFS (Matrix)	500	Directed	0.053905	2,058,096	N/A
DFS (List)	10	Undirected	8.5e-05	1,960	N/A
DFS (List)	10	Directed	0.000156	2,448	N/A
DFS (List)	15	Undirected	0.000151	2,968	N/A
DFS (List)	15	Directed	0.000158	3,080	N/A
DFS (List)	50	Undirected	0.000229	7,720	N/A
DFS (List)	50	Directed	0.000441	11,296	N/A
DFS (List)	75	Undirected	0.000343	10,576	N/A
DFS (List)	75	Directed	0.000884	13,560	N/A
DFS (List)	100	Undirected	0.000412	20,312	N/A
DFS (List)	100	Directed	0.001081	28,144	N/A
DFS (List)	250	Undirected	0.001540	41,976	N/A
DFS (List)	250	Directed	0.002632	49,184	N/A
DFS (List)	500	Undirected	0.002303	245,215	N/A
DFS (List)	500	Directed	0.006515	301,892	N/A

Table 2.2: Execution Time and Memory Usage – Directed vs Undirected Graphs (Continuation)

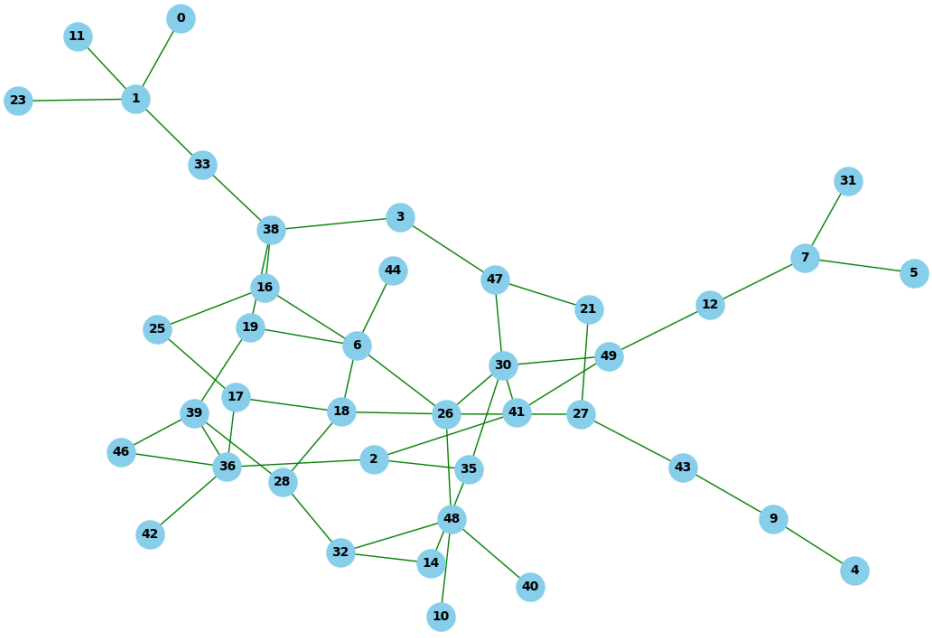
Algorithm	Nodes	Type	Time (s)	Memory (B)	Result
BFS (List)	10	Undirected	6.8e-05	2,512	N/A
BFS (List)	10	Directed	0.000117	2,944	N/A
BFS (List)	15	Undirected	6.4e-05	3,232	N/A
BFS (List)	15	Directed	0.000105	3,512	N/A
BFS (List)	50	Undirected	0.000254	8,400	N/A
BFS (List)	50	Directed	0.000578	11,592	N/A
BFS (List)	75	Undirected	0.000232	10,576	N/A
BFS (List)	75	Directed	0.000654	13,872	N/A
BFS (List)	100	Undirected	0.000443	21,520	N/A
BFS (List)	100	Directed	0.000756	173,991	N/A
BFS (List)	250	Undirected	0.000796	41,976	N/A
BFS (List)	250	Directed	0.002262	48,376	N/A
BFS (List)	500	Undirected	0.001964	98,751	N/A
BFS (List)	500	Directed	0.004668	148,928	N/A

Visual Graph Examples

Figure 2.1 shows visual representations of some generated graphs used in the experiments.



(a) Directed graph with 15 nodes



(b) Undirected graph with 50 nodes

Figure 2.1: Example of directed vs. undirected graph structures

Graphical Analysis of Traversal Performance

The following plot (Figure 2.2) compares execution time and memory usage for each algorithm across both graph types.

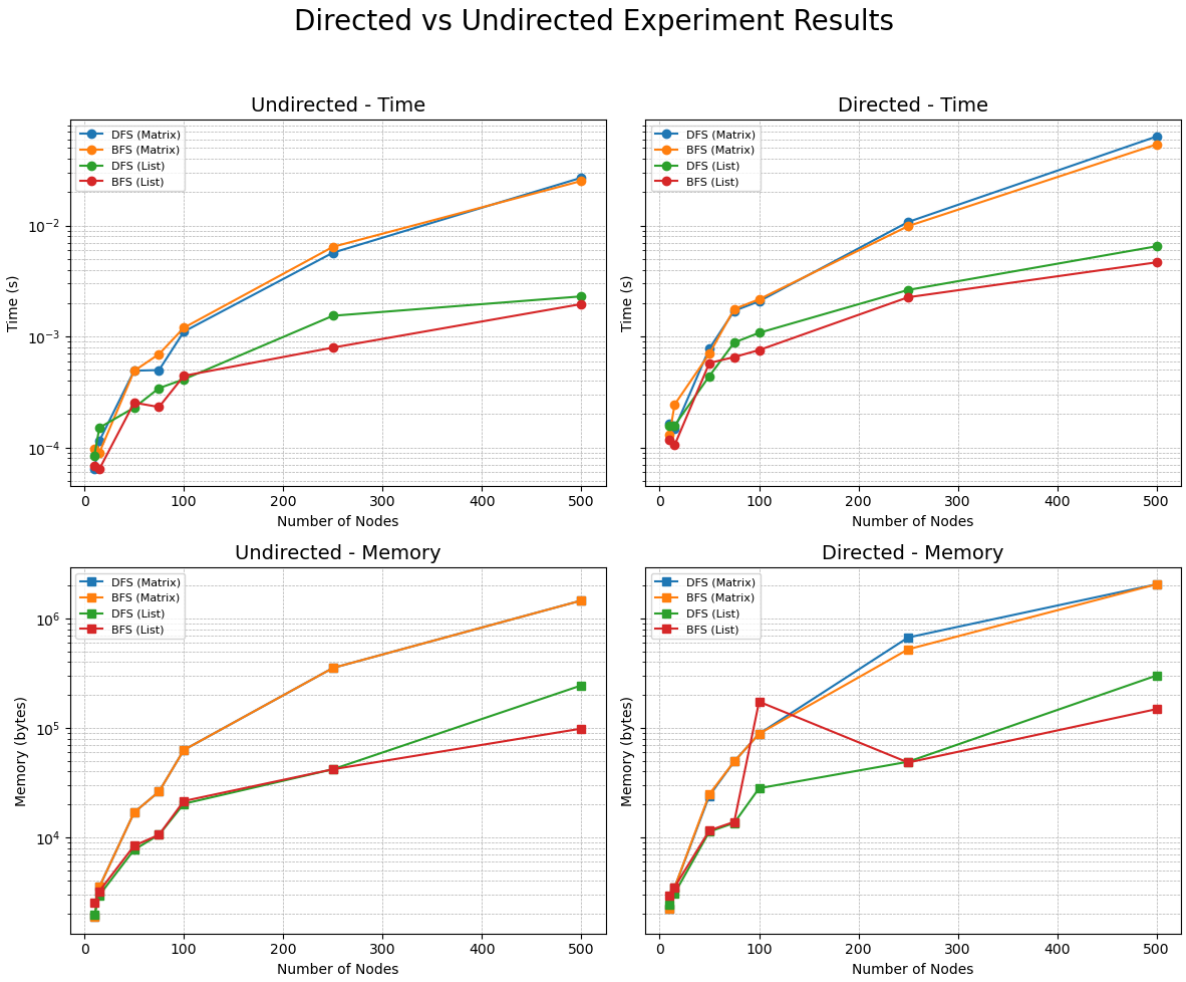


Figure 2.2: Directed vs Undirected: BFS and DFS Time and Memory Analysis

Graph Type Impact (Directed vs Undirected): The structure of the graph—whether directed or undirected—has a significant effect on traversal outcomes. In undirected graphs, each edge allows traversal in both directions, often resulting in a more connected graph and faster full exploration. In contrast, directed graphs can have asymmetric connections and isolated nodes (nodes with only incoming or only outgoing edges), which can increase the traversal depth and complexity. As seen in our results, traversal algorithms on directed graphs generally exhibited higher execution times and memory usage, especially in matrix representations. This is because the algorithms may need to process more steps to explore disconnected or one-way paths fully.

Analysis

- **Adjacency matrix** implementations consistently consumed more memory than adjacency lists due to their fixed size (n×n) representation.
- **Execution time** increased with graph size, and directed graphs generally resulted in longer processing times—particularly noticeable with matrix-based DFS and BFS.
- **Adjacency lists** showed more efficient performance on large, sparse graphs, both in speed and memory footprint.

- BFS (List) was the most memory-efficient approach and also scaled better across increasing node sizes.

2.3.2 Weighted vs Unweighted Graphs

This experiment evaluates how edge weights influence graph traversal performance. Using graphs of varying sizes (10 to 500 nodes), both weighted and unweighted variants were generated based on the Erdős–Rényi model. For each graph, both adjacency matrix and adjacency list representations were used. Execution time and memory usage were recorded for DFS and BFS algorithms under each condition.

Empirical Results Tables

The data collected is split below for readability.

Table 2.3: Execution Time and Memory Usage – Weighted vs Unweighted (Matrix)

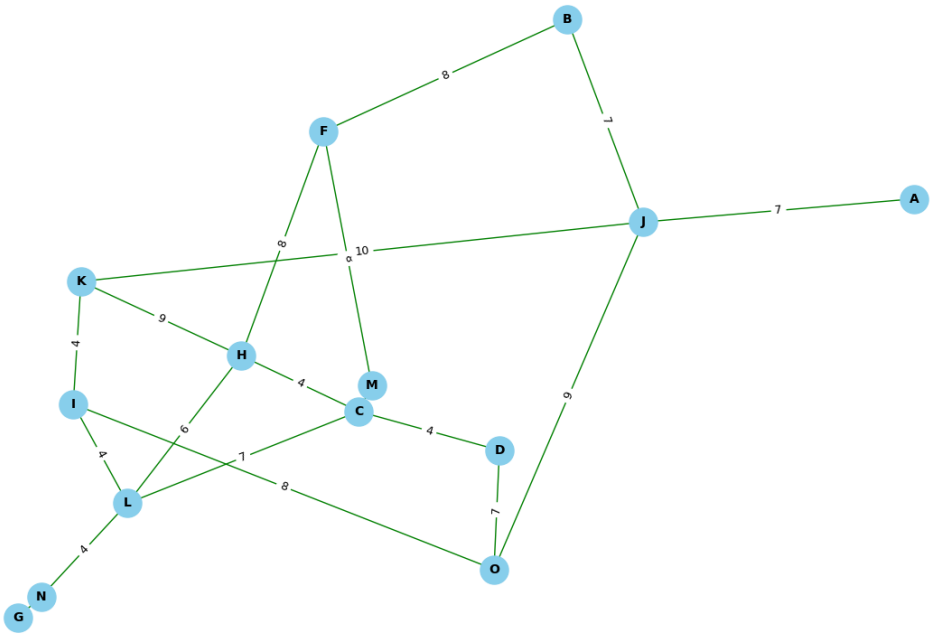
Algorithm	Nodes	Type	Time (s)	Memory (B)	Result
DFS (Matrix)	10	Unweighted	4.3e-05	1,880	N/A
DFS (Matrix)	10	Weighted	5e-05	1,880	N/A
DFS (Matrix)	15	Unweighted	6e-05	3,528	N/A
DFS (Matrix)	15	Weighted	9.4e-05	3,528	N/A
DFS (Matrix)	50	Unweighted	0.000339	16,928	N/A
DFS (Matrix)	50	Weighted	0.000331	16,928	N/A
DFS (Matrix)	75	Unweighted	0.000485	26,943	N/A
DFS (Matrix)	75	Weighted	0.000444	26,240	N/A
DFS (Matrix)	100	Unweighted	0.001124	63,016	N/A
DFS (Matrix)	100	Weighted	0.000602	63,016	N/A
DFS (Matrix)	250	Unweighted	0.005625	353,848	N/A
DFS (Matrix)	250	Weighted	0.003198	353,848	N/A
DFS (Matrix)	500	Unweighted	0.016395	1,463,084	N/A
DFS (Matrix)	500	Weighted	0.020126	1,462,788	N/A
BFS (Matrix)	10	Unweighted	5.8e-05	1,880	N/A
BFS (Matrix)	10	Weighted	7.5e-05	1,880	N/A
BFS (Matrix)	15	Unweighted	0.000137	3,528	N/A
BFS (Matrix)	15	Weighted	0.000154	3,528	N/A
BFS (Matrix)	50	Unweighted	0.000406	16,928	N/A
BFS (Matrix)	50	Weighted	0.000338	16,928	N/A
BFS (Matrix)	75	Unweighted	0.000479	26,240	N/A
BFS (Matrix)	75	Weighted	0.000604	26,240	N/A
BFS (Matrix)	100	Unweighted	0.001262	63,016	N/A
BFS (Matrix)	100	Weighted	0.001356	63,016	N/A
BFS (Matrix)	250	Unweighted	0.006536	353,848	N/A
BFS (Matrix)	250	Weighted	0.006106	353,848	N/A
BFS (Matrix)	500	Unweighted	0.028444	1,462,148	N/A
BFS (Matrix)	500	Weighted	0.018428	1,362,268	N/A

Table 2.4: Execution Time and Memory Usage – Weighted vs Unweighted (Continuation - List)

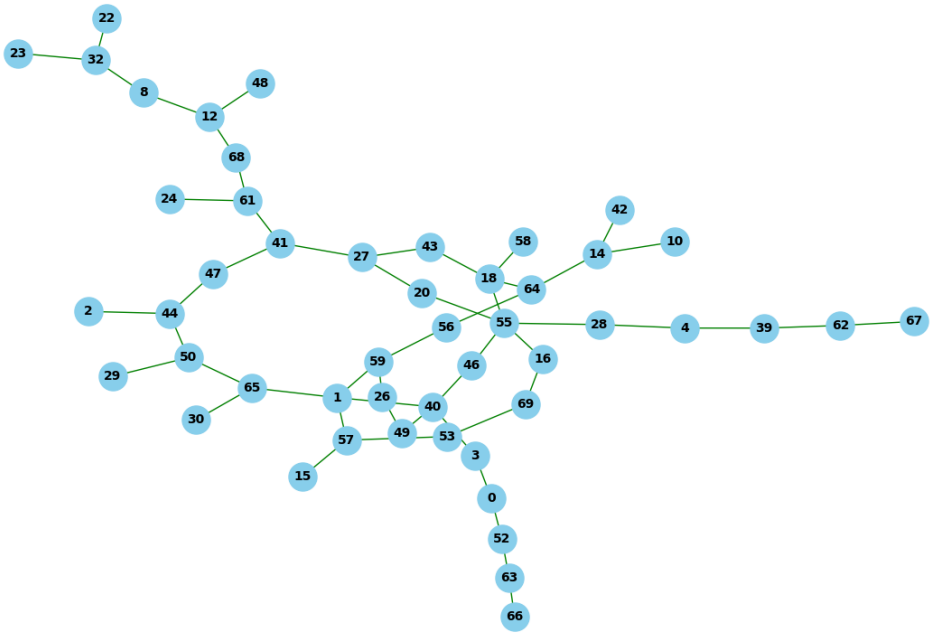
Algorithm	Nodes	Type	Time (s)	Memory (B)	Result
DFS (List)	10	Unweighted	3.7e-05	1,960	N/A
DFS (List)	10	Weighted	4.9e-05	1,960	N/A
DFS (List)	15	Unweighted	5.4e-05	2,968	N/A
DFS (List)	15	Weighted	0.0001	2,968	N/A
DFS (List)	50	Unweighted	0.000215	7,720	N/A
DFS (List)	50	Weighted	0.000152	7,720	N/A
DFS (List)	75	Unweighted	0.000225	10,576	N/A
DFS (List)	75	Weighted	0.000136	10,576	N/A
DFS (List)	100	Unweighted	0.000413	20,312	N/A
DFS (List)	100	Weighted	0.000244	20,312	N/A
DFS (List)	250	Unweighted	0.000577	41,976	N/A
DFS (List)	250	Weighted	0.000865	41,976	N/A
DFS (List)	500	Unweighted	0.001157	96,992	N/A
DFS (List)	500	Weighted	0.001149	96,992	N/A
BFS (List)	10	Unweighted	4.3e-05	2,512	N/A
BFS (List)	10	Weighted	4.1e-05	2,512	N/A
BFS (List)	15	Unweighted	4.8e-05	3,232	N/A
BFS (List)	15	Weighted	7.9e-05	3,232	N/A
BFS (List)	50	Unweighted	0.000142	8,400	N/A
BFS (List)	50	Weighted	0.000119	8,400	N/A
BFS (List)	75	Unweighted	0.000145	10,576	N/A
BFS (List)	75	Weighted	0.000126	10,576	N/A
BFS (List)	100	Unweighted	0.000343	21,520	N/A
BFS (List)	100	Weighted	0.000338	21,520	N/A
BFS (List)	250	Unweighted	0.000544	41,976	N/A
BFS (List)	250	Weighted	0.000545	41,976	N/A
BFS (List)	500	Unweighted	0.001287	97,672	N/A
BFS (List)	500	Weighted	0.001115	246,130	N/A

Visual Graph Examples

Figure 2.3 shows examples of graphs used in the experiment for both weighted and unweighted cases.



(a) Weighted graph with 15 nodes



(b) Unweighted graph with 75 nodes

Figure 2.3: Example graphs used in the weighted vs unweighted experiment

Graphical Analysis of Traversal Performance

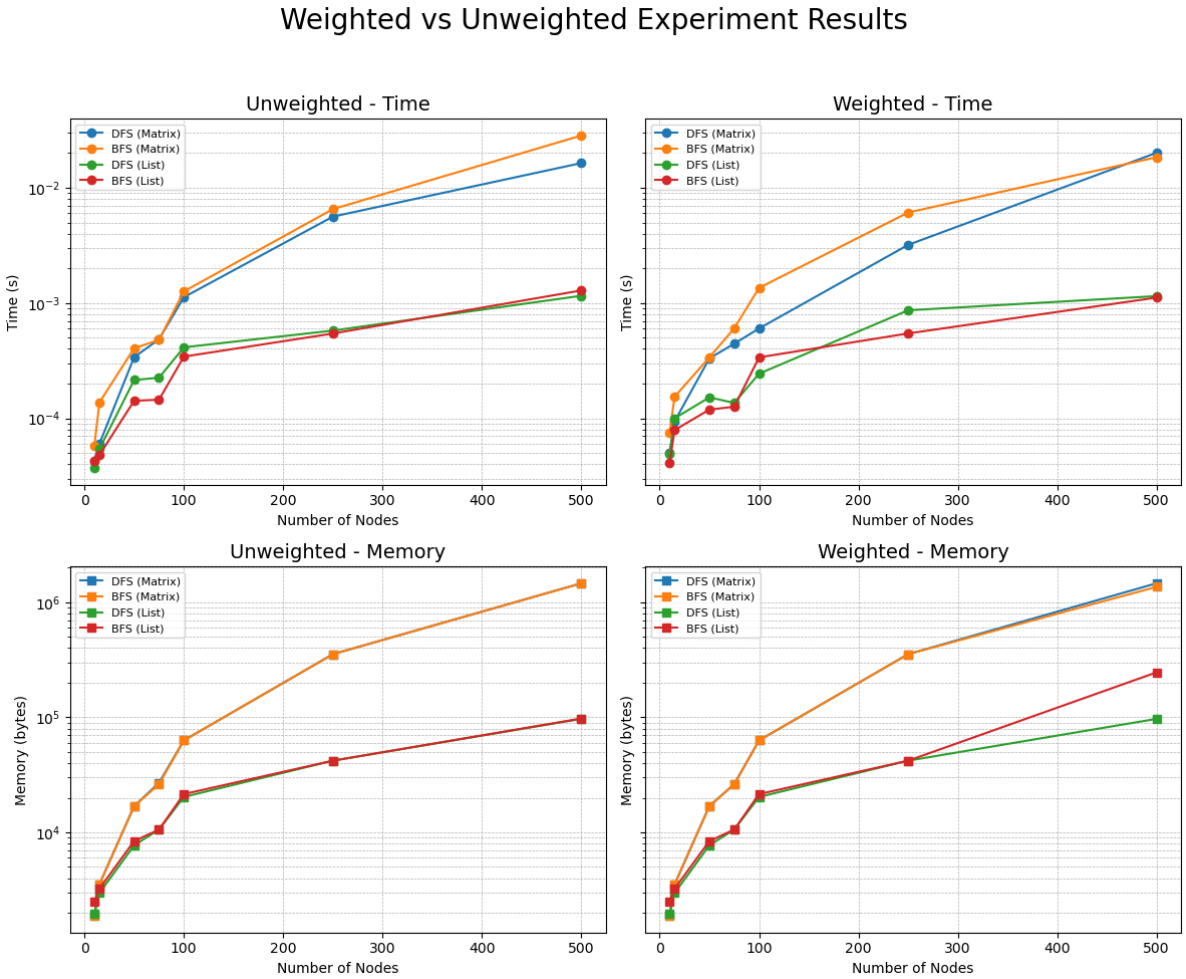


Figure 2.4: Weighted vs Unweighted: BFS and DFS Time and Memory Analysis

Graph Weight Impact (Weighted vs Unweighted): The presence of weights on edges does not significantly influence memory usage, as graph size dominates memory complexity. Importantly, standard BFS and DFS are **weight-agnostic algorithms** — they do not use edge weights to determine traversal paths. Whether an edge has a weight of 1 or 1000, these algorithms only consider **which nodes are connected**, not the cost of reaching them. Therefore, from a theoretical standpoint, weighted and unweighted traversals should perform identically.

However, in practice, execution time may show slight variations due to implementation-specific factors. These include: (i) how edge weights are stored (e.g., as part of a tuple in adjacency lists), which can introduce minor overhead during iteration, (ii) changes in graph structure or connectivity introduced during weighted graph generation (even with the same parameters), and (iii) unpacking operations during traversal loops when edges include weights.

Analysis

- Weighted and unweighted graphs consumed similar memory, confirming that node count and representation (matrix or list) dominate space usage more than edge attributes.
- BFS and DFS execution times may differ slightly due to metadata overhead or structural changes, despite both ignoring weights during traversal logic.
- Adjacency list versions again performed better overall for memory efficiency, especially on sparse graphs.
- BFS (List) remained the fastest and most memory-efficient among all tested combinations.

2.3.3 Sparse vs Dense Graphs

This experiment explores how graph density affects traversal algorithms. Graphs of increasing sizes (from 10 to 500 nodes) were created using the Erdős–Rényi model. The 'sparse' flag determined edge probability. All graphs were undirected and unweighted, with a connectivity check to retain only the largest connected component if needed.

Empirical Results Table

The table below shows execution time and memory usage for DFS and BFS using matrix and list representations across sparse and dense graphs.

Empirical Results Tables

The data collected is split below for readability.

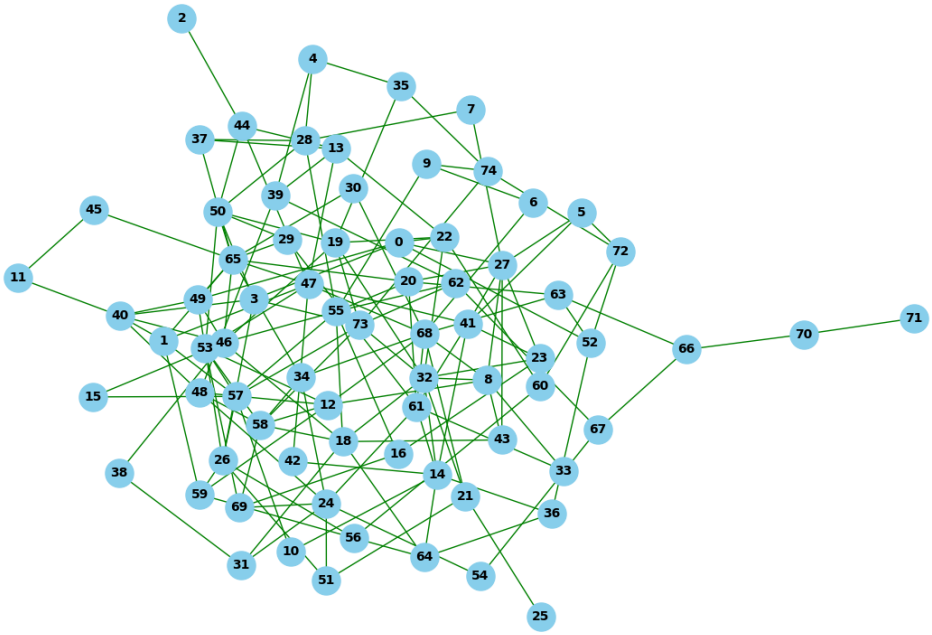
Table 2.5: Execution Time and Memory Usage – Weighted vs Unweighted (Matrix)

Algorithm	Nodes	Type	Time (s)	Memory (B)	Result
DFS (Matrix)	10	Sparse	4e-05	1,880	N/A
DFS (Matrix)	10	Dense	9.7e-05	2,248	N/A
DFS (Matrix)	15	Sparse	7.6e-05	3,528	N/A
DFS (Matrix)	15	Dense	8.1e-05	3,776	N/A
DFS (Matrix)	50	Sparse	0.000287	16,928	N/A
DFS (Matrix)	50	Dense	0.000552	27,032	N/A
DFS (Matrix)	75	Sparse	0.000339	26,240	N/A
DFS (Matrix)	75	Dense	0.000556	51,224	N/A
DFS (Matrix)	100	Sparse	0.000837	63,016	N/A
DFS (Matrix)	100	Dense	0.001069	95,056	N/A
DFS (Matrix)	250	Sparse	0.002931	353,848	N/A
DFS (Matrix)	250	Dense	0.004902	525,480	N/A
DFS (Matrix)	500	Sparse	0.021657	1,462,988	N/A
DFS (Matrix)	500	Dense	0.03	2,198,092	N/A

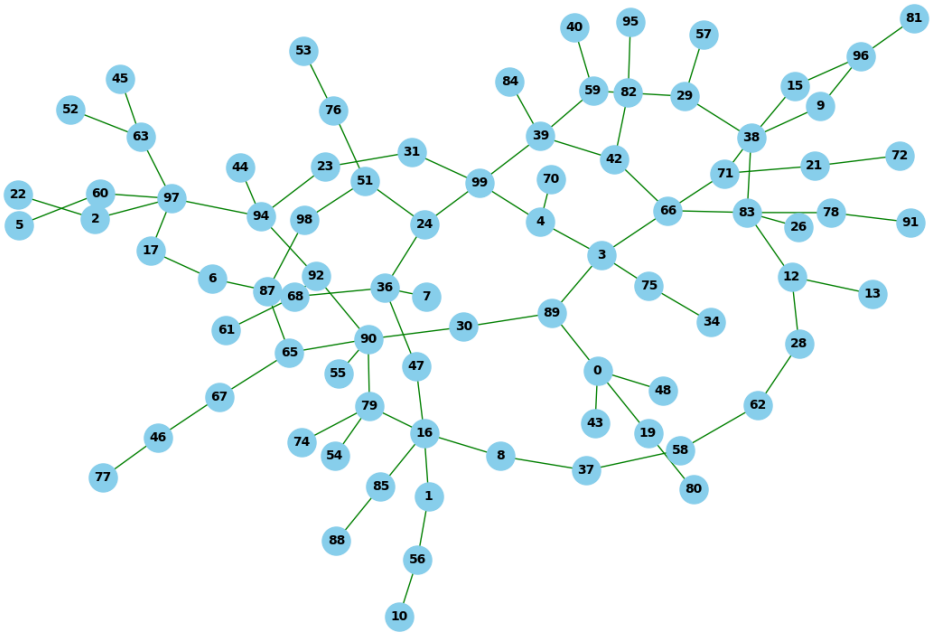
Table 2.6: Execution Time and Memory Usage – Weighted vs Unweighted (Continuation - List)

Algorithm	**Nodes**	**Type**	**Time (s)**	**Memory (B)**	**Result**
BFS (Matrix)	10	Sparse	4.8e-05	1,880	N/A
BFS (Matrix)	10	Dense	9.5e-05	2,248	N/A
BFS (Matrix)	15	Sparse	0.000105	3,528	N/A
BFS (Matrix)	15	Dense	0.000105	3,776	N/A
BFS (Matrix)	50	Sparse	0.000333	16,928	N/A
BFS (Matrix)	50	Dense	0.000644	27,032	N/A
BFS (Matrix)	75	Sparse	0.000524	26,240	N/A
BFS (Matrix)	75	Dense	0.000947	51,224	N/A
BFS (Matrix)	100	Sparse	0.00082	63,016	N/A
BFS (Matrix)	100	Dense	0.001357	95,056	N/A
BFS (Matrix)	250	Sparse	0.003808	353,848	N/A
BFS (Matrix)	250	Dense	0.005462	525,480	N/A
BFS (Matrix)	500	Sparse	0.025296	1,462,220	N/A
BFS (Matrix)	500	Dense	0.021721	2,177,412	N/A
DFS (List)	10	Sparse	0.000102	1,960	N/A
DFS (List)	10	Dense	8.3e-05	2,480	N/A
DFS (List)	15	Sparse	0.000111	2,968	N/A
DFS (List)	15	Dense	9.6e-05	3,840	N/A
DFS (List)	50	Sparse	0.000182	7,720	N/A
DFS (List)	50	Dense	0.000512	11,880	N/A
DFS (List)	75	Sparse	0.000159	10,576	N/A
DFS (List)	75	Dense	0.000419	13,184	N/A
DFS (List)	100	Sparse	0.000318	20,312	N/A
DFS (List)	100	Dense	0.000802	27,656	N/A
DFS (List)	250	Sparse	0.000546	41,976	N/A
DFS (List)	250	Dense	0.008337	49,232	N/A
DFS (List)	500	Sparse	0.002188	97,751	N/A
DFS (List)	500	Dense	0.002903	151,288	N/A
BFS (List)	10	Sparse	3.3e-05	2,512	N/A
BFS (List)	10	Dense	5.5e-05	2,976	N/A
BFS (List)	15	Sparse	4.8e-05	3,232	N/A
BFS (List)	15	Dense	7.7e-05	3,672	N/A
BFS (List)	50	Sparse	0.000153	8,400	N/A
BFS (List)	50	Dense	0.000265	11,880	N/A
BFS (List)	75	Sparse	0.000119	10,576	N/A
BFS (List)	75	Dense	0.00044	13,496	N/A
BFS (List)	100	Sparse	0.000353	21,520	N/A
BFS (List)	100	Dense	0.000516	27,616	N/A
BFS (List)	250	Sparse	0.000575	41,976	N/A
BFS (List)	250	Dense	0.001021	49,232	N/A
BFS (List)	500	Sparse	0.001156	97,672	N/A
BFS (List)	500	Dense	0.003637	147,600	N/A

Visual Graph Examples



(a) Dense graph with 75 nodes



(b) Sparse graph with 100 nodes

Figure 2.5: Examples of sparse and dense graph structures

Graphical Analysis of Traversal Performance

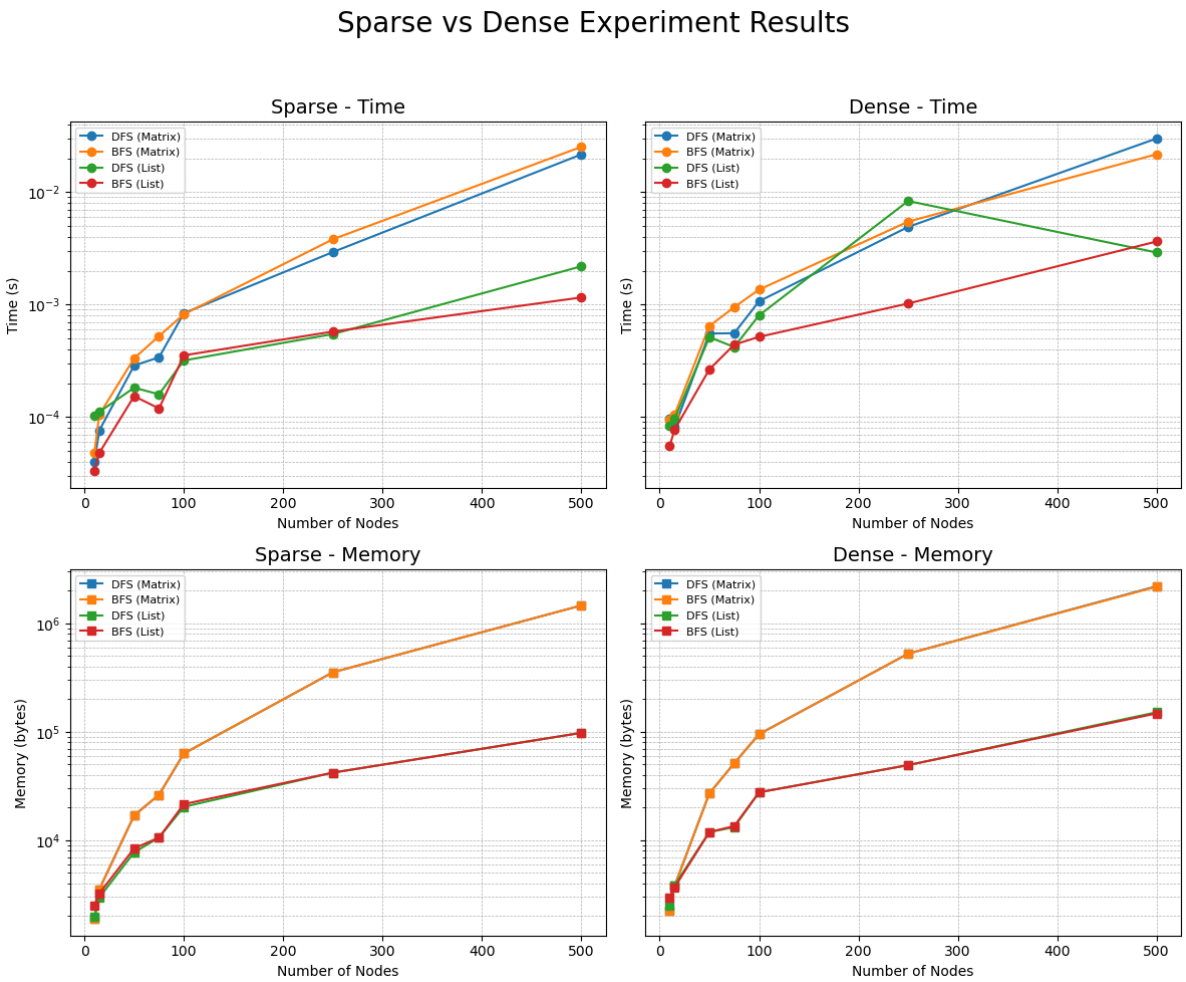


Figure 2.6: Sparse vs Dense: BFS and DFS Time and Memory Analysis

Graph Density Impact (Sparse vs Dense): Graph density significantly affects traversal performance. In dense graphs, nodes have more neighbors, leading to increased branching during search. This impacts both time and memory, especially with matrix representations that grow quadratically. Sparse graphs, by contrast, maintain fewer edges, reducing traversal operations and benefiting list-based representations.

Analysis

- **Dense graphs** increase memory and time usage, notably for adjacency matrix due to its $\mathcal{O}(n^2)$ storage.
- **Sparse graphs** benefit list representations the most, resulting in the smallest memory footprint and fastest traversal.
- **DFS and BFS** scale differently: DFS sees greater time impact on dense graphs due to recursive depth, while BFS's breadth expansion grows due to more adjacent edges.

- Matrix-based BFS in dense graphs performed worst in memory usage, confirming it is less suited for large dense structures.

2.3.4 Cyclic vs Acyclic Graphs

This experiment investigates how the presence of cycles impacts the performance of graph traversal algorithms. Graphs were generated using the Erdős–Rényi model for cyclic structures and a custom DAG generator for acyclic graphs. All graphs are directed and unweighted, and the size varied from 10 to 500 nodes.

Empirical Results Tables

The results below are separated into matrix and list representations for DFS and BFS.

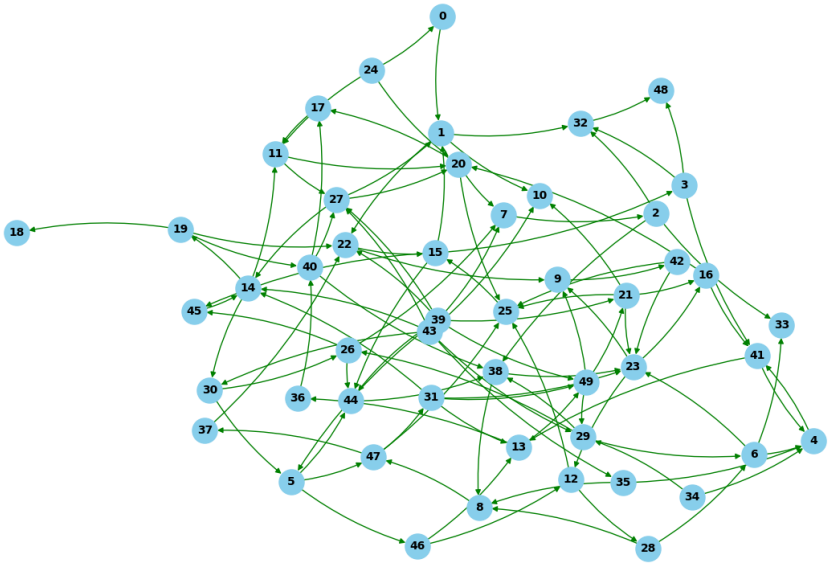
Table 2.7: Execution Time and Memory Usage – Cyclic vs Acyclic

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
DFS (Matrix)	10	Cyclic	8.9e-05	2,240	N/A
DFS (Matrix)	10	Acyclic	7.7e-05	2,320	N/A
DFS (Matrix)	50	Cyclic	0.000318	24,040	N/A
DFS (Matrix)	50	Acyclic	0.000247	24,040	N/A
DFS (Matrix)	250	Cyclic	0.004378	672,157	N/A
DFS (Matrix)	250	Acyclic	0.001621	524,040	N/A
DFS (Matrix)	500	Cyclic	0.024152	2,184,196	N/A
DFS (Matrix)	500	Acyclic	0.004117	2,058,240	N/A
BFS (Matrix)	10	Cyclic	8.3e-05	2,240	N/A
BFS (Matrix)	10	Acyclic	7.9e-05	2,320	N/A
BFS (Matrix)	50	Cyclic	0.000423	24,040	N/A
BFS (Matrix)	50	Acyclic	0.000221	24,040	N/A
BFS (Matrix)	250	Cyclic	0.003824	524,040	N/A
BFS (Matrix)	250	Acyclic	0.00189	524,040	N/A
BFS (Matrix)	500	Cyclic	0.031825	2,183,108	N/A
BFS (Matrix)	500	Acyclic	0.003605	2,181,303	N/A
DFS (List)	10	Cyclic	6.5e-05	2,160	N/A
DFS (List)	10	Acyclic	6e-05	2,184	N/A
DFS (List)	50	Cyclic	0.000385	9,280	N/A
DFS (List)	50	Acyclic	0.00026	9,960	N/A
DFS (List)	250	Cyclic	0.001184	39,792	N/A
DFS (List)	250	Acyclic	0.001167	42,352	N/A
DFS (List)	500	Cyclic	0.010088	99,848	N/A
DFS (List)	500	Acyclic	0.00327	84,336	N/A

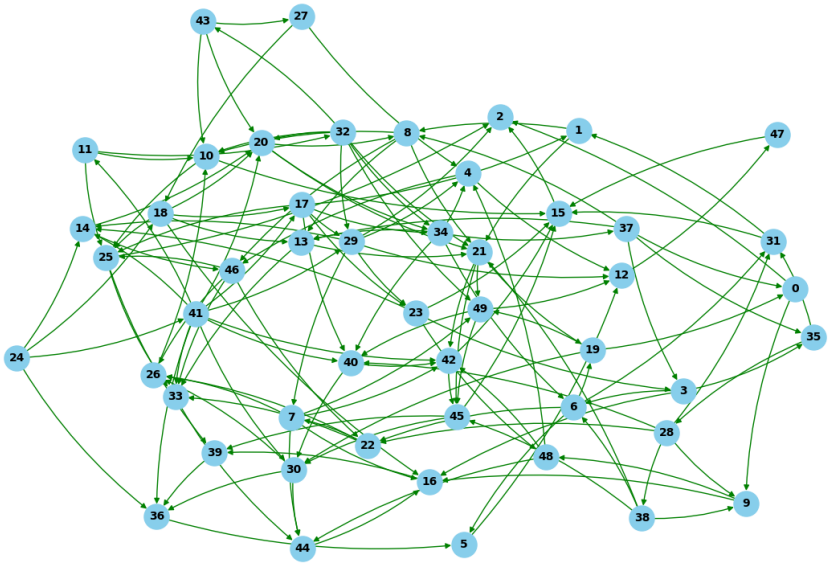
Table 2.8: Execution Time and Memory Usage – Cyclic vs Acyclic (Continuation)

Algorithm	**Nodes**	**Graph Type**	**Time (s)**	**Memory (B)**	**Result**
BFS (List)	10	Cyclic	6.8e-05	2,720	N/A
BFS (List)	10	Acyclic	6.4e-05	2,744	N/A
BFS (List)	50	Cyclic	0.000143	9,960	N/A
BFS (List)	50	Acyclic	0.000149	10,512	N/A
BFS (List)	250	Cyclic	0.000643	40,576	N/A
BFS (List)	250	Acyclic	0.000611	42,352	N/A
BFS (List)	500	Cyclic	0.001439	100,528	N/A
BFS (List)	500	Acyclic	0.002054	84,336	N/A

Visual Graph Examples



(a) Cyclic graph with 50 nodes



(b) Acyclic (DAG) graph with 50 nodes

Figure 2.7: Examples of cyclic and acyclic graph structures

Graphical Analysis of Traversal Performance

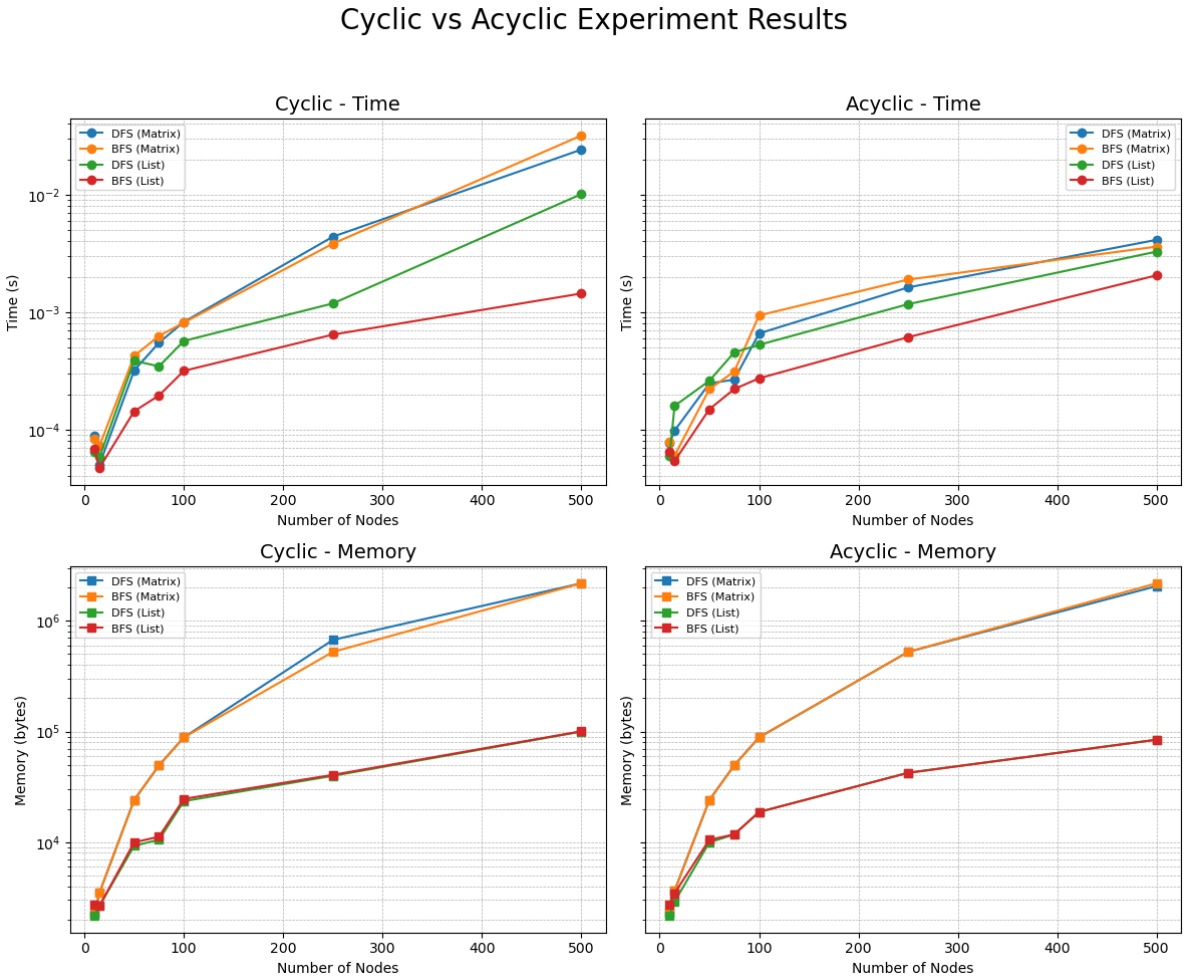


Figure 2.8: Cyclic vs Acyclic: BFS and DFS Time and Memory Analysis

Impact of Cycles on Traversal: Cycle presence affects traversal differently depending on the algorithm and graph representation. Acyclic graphs limit the depth and revisit behavior, resulting in lower memory and faster traversal. Cyclic graphs may reprocess nodes and increase recursion depth (DFS) or queue expansion (BFS).

Analysis

- **Cyclic graphs** result in higher traversal time, especially in matrix-based DFS due to repeated visits and deeper recursion.
- **Acyclic graphs** benefit from simpler paths and minimal backtracking, reducing both memory and execution time.
- **List representations** remain more memory-efficient than matrix ones, but still show time penalties in cyclic traversals.
- **DFS** is more sensitive to cycles due to recursive depth, while BFS shows increased queue size in denser, cyclic scenarios.

2.3.5 Connected vs Disconnected Graphs

Graph connectivity significantly impacts traversal algorithms such as BFS and DFS. In a connected graph, every node is reachable from any other node, making traversal straightforward. However, disconnected graphs consist of isolated components, requiring multiple traversal initiations to explore all nodes. This section analyzes the effect of connectivity on performance and memory usage across multiple graph sizes using both adjacency list and matrix representations.

Empirical Results Tables

The results below present execution time and memory usage for BFS and DFS on connected and disconnected graphs. The measurements reflect **full traversal**, meaning the algorithm was rerun on all disconnected components where applicable.

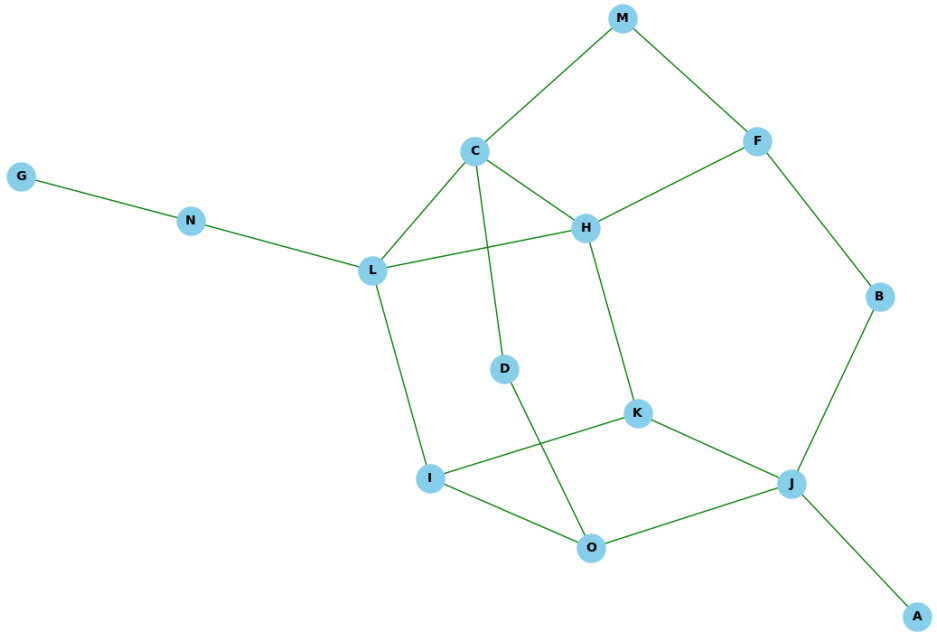
Table 2.9: Execution Time and Memory Usage – Connected vs Disconnected (Matrix)

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
DFS (Matrix, Full)	10	Connected	5.8e-05	1,880	N/A
DFS (Matrix, Full)	10	Disconnected	5e-05	2,456	N/A
DFS (Matrix, Full)	15	Connected	9.3e-05	3,528	N/A
DFS (Matrix, Full)	15	Disconnected	6.8e-05	4,192	N/A
DFS (Matrix, Full)	50	Connected	0.000285	16,928	N/A
DFS (Matrix, Full)	50	Disconnected	0.000281	27,032	N/A
DFS (Matrix, Full)	75	Connected	0.000301	26,240	N/A
DFS (Matrix, Full)	75	Disconnected	0.000683	52,432	N/A
DFS (Matrix, Full)	100	Connected	0.000881	63,016	N/A
DFS (Matrix, Full)	100	Disconnected	0.000787	95,056	N/A
DFS (Matrix, Full)	250	Connected	0.003521	353,848	N/A
DFS (Matrix, Full)	250	Disconnected	0.004573	537,592	N/A
DFS (Matrix, Full)	500	Connected	0.010551	1,463,848	N/A
DFS (Matrix, Full)	500	Disconnected	0.016147	2,085,472	N/A
BFS (Matrix, Full)	10	Connected	3.5e-05	1,880	N/A
BFS (Matrix, Full)	10	Disconnected	3.9e-05	2,672	N/A
BFS (Matrix, Full)	15	Connected	8.1e-05	3,528	N/A
BFS (Matrix, Full)	15	Disconnected	9.9e-05	4,192	N/A
BFS (Matrix, Full)	50	Connected	0.000285	16,928	N/A
BFS (Matrix, Full)	50	Disconnected	0.000245	27,032	N/A
BFS (Matrix, Full)	75	Connected	0.000272	26,240	N/A
BFS (Matrix, Full)	75	Disconnected	0.000627	52,432	N/A
BFS (Matrix, Full)	100	Connected	0.000767	63,016	N/A
BFS (Matrix, Full)	100	Disconnected	0.001045	95,056	N/A
BFS (Matrix, Full)	250	Connected	0.002966	353,848	N/A
BFS (Matrix, Full)	250	Disconnected	0.003941	537,592	N/A
BFS (Matrix, Full)	500	Connected	0.009774	1,462,148	N/A
BFS (Matrix, Full)	500	Disconnected	0.028168	2,182,356	N/A

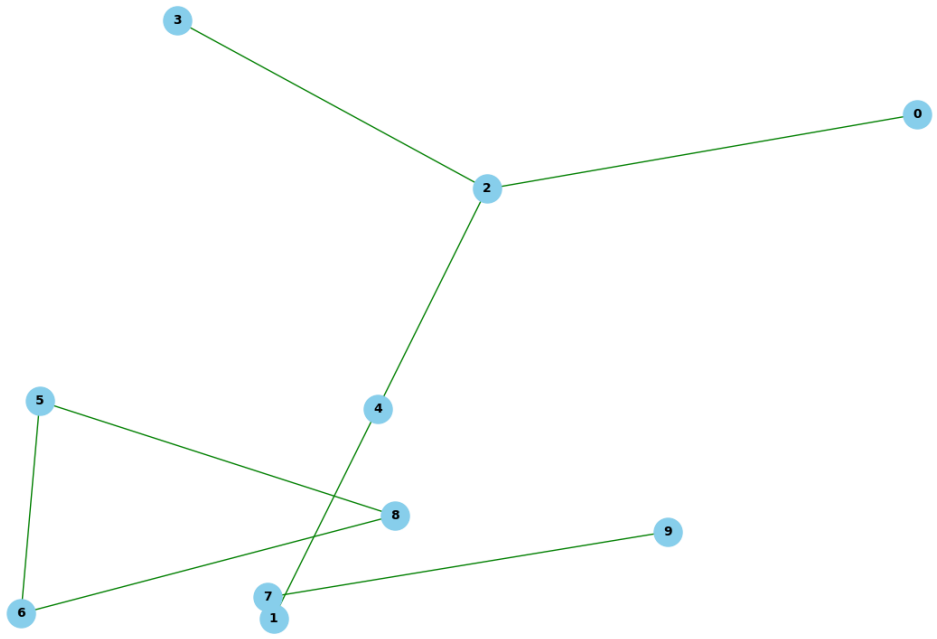
Table 2.10: Execution Time and Memory Usage – Connected vs Disconnected (List)

Algorithm	**Nodes**	**Graph Type**	**Time (s)**	**Memory (B)**	**Result**
DFS (List, Full)	10	Connected	4.5e-05	1,960	N/A
DFS (List, Full)	10	Disconnected	5.3e-05	2,296	N/A
DFS (List, Full)	15	Connected	7e-05	2,968	N/A
DFS (List, Full)	15	Disconnected	6.6e-05	3,208	N/A
DFS (List, Full)	50	Connected	9.7e-05	7,792	N/A
DFS (List, Full)	50	Disconnected	0.00014	10,376	N/A
DFS (List, Full)	75	Connected	0.00011	10,576	N/A
DFS (List, Full)	75	Disconnected	0.000148	10,824	N/A
DFS (List, Full)	100	Connected	0.000207	20,384	N/A
DFS (List, Full)	100	Disconnected	0.000203	23,696	N/A
DFS (List, Full)	250	Connected	0.000588	41,976	N/A
DFS (List, Full)	250	Disconnected	0.000481	39,568	N/A
DFS (List, Full)	500	Connected	0.001117	97,064	N/A
DFS (List, Full)	500	Disconnected	0.001246	247,415	N/A
BFS (List, Full)	10	Connected	4.5e-05	2,584	N/A
BFS (List, Full)	10	Disconnected	3e-05	3,608	N/A
BFS (List, Full)	15	Connected	6.2e-05	3,304	N/A
BFS (List, Full)	15	Disconnected	4.9e-05	4,152	N/A
BFS (List, Full)	50	Connected	0.000151	8,472	N/A
BFS (List, Full)	50	Disconnected	0.000139	10,376	N/A
BFS (List, Full)	75	Connected	0.000169	10,576	N/A
BFS (List, Full)	75	Disconnected	0.000198	11,760	N/A
BFS (List, Full)	100	Connected	0.000174	21,592	N/A
BFS (List, Full)	100	Disconnected	0.000284	24,312	N/A
BFS (List, Full)	250	Connected	0.000626	41,976	N/A
BFS (List, Full)	250	Disconnected	0.000353	39,568	N/A
BFS (List, Full)	500	Connected	0.001359	97,744	N/A
BFS (List, Full)	500	Disconnected	0.003267	100,312	N/A

Visual Graph Examples



(a) Example of a Connected Graph (15 nodes)



(b) Example of a Disconnected Graph (10 nodes)

Figure 2.9: Graph structures used for connected vs disconnected comparisons

Graphical Analysis of Traversal Performance

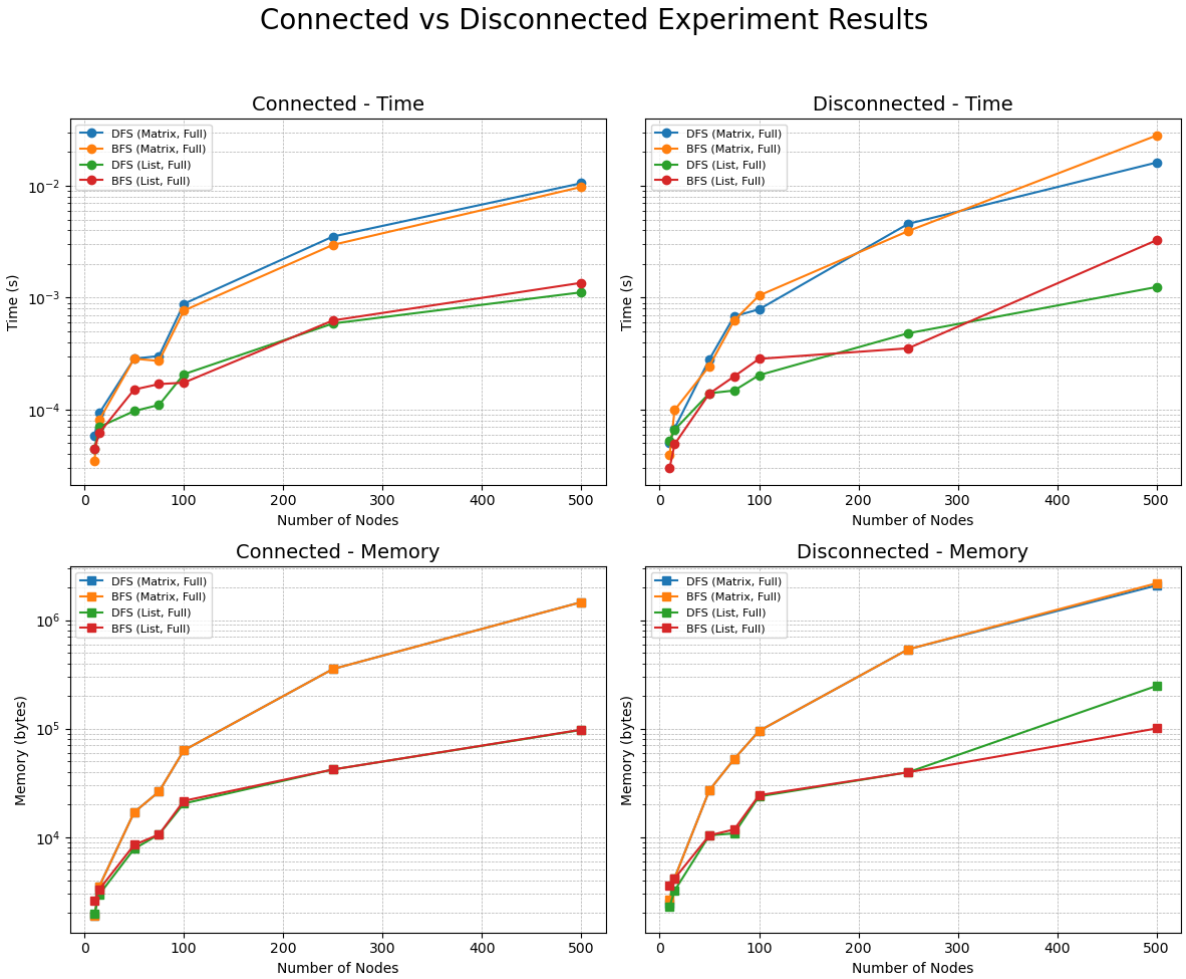


Figure 2.10: Connected vs Disconnected: BFS and DFS Time and Memory Analysis

Handling Disconnected Graphs in BFS/DFS:

- In disconnected graphs, traversal from a single start node is insufficient. All unvisited nodes must be processed separately to ensure complete coverage.
- **DFS** explores each component deeply. Recursive depth is limited per component, but must restart for each.
- **BFS** processes reachable nodes level-by-level. In disconnected graphs, the queue resets for each new component.
- Experiments used full traversal, where BFS/DFS were restarted for each unvisited node to ensure total exploration.

Analysis

- **Disconnected graphs** exhibit increased traversal time due to multiple entry points and repeated overhead.

- **Connected graphs** benefit from single-pass traversal, minimizing redundant operations.
- **Matrix representations** are memory-heavy in all cases, with disconnected graphs showing higher memory usage due to denser component distribution.
- **List representations** are significantly more efficient in sparse graphs, particularly for large disconnected topologies.
- The impact of graph connectivity is most noticeable in matrix-based BFS, where both time and memory grow steeply in disconnected scenarios.

2.3.6 No Loops vs With Loops

This section analyzes the influence of self-loops on the performance of BFS and DFS algorithms. A self-loop is an edge that connects a node to itself. While such edges do not add connectivity across components, they may still impact traversal cost and memory behavior, especially in matrix representations. We compare both adjacency matrix and adjacency list implementations over graphs of increasing size with and without self-loops.

Empirical Results Tables

The table below summarizes execution time and memory usage for each algorithm. The graphs labeled "With Loops" contain a small percentage of randomly placed self-loops, while "No Loops" graphs are loop-free.

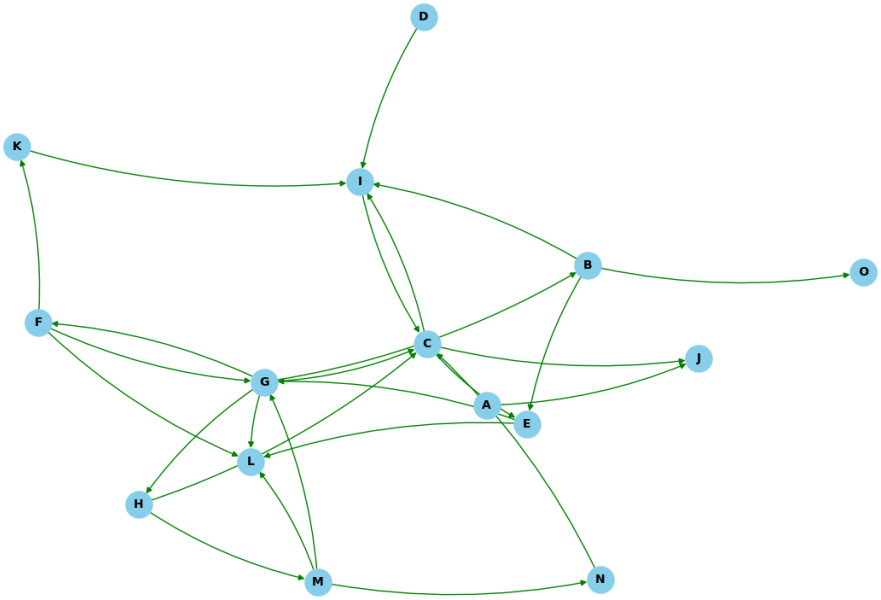
Table 2.11: Execution Time and Memory Usage – No Loops vs With Loops

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
DFS (Matrix)	10	No Loops	5.8e-05	2,240	N/A
DFS (Matrix)	10	With Loops	6.3e-05	2,240	N/A
DFS (Matrix)	15	No Loops	5.5e-05	3,480	N/A
DFS (Matrix)	15	With Loops	6e-05	3,480	N/A
DFS (Matrix)	50	No Loops	0.000306	24,040	N/A
DFS (Matrix)	50	With Loops	0.000393	24,040	N/A
DFS (Matrix)	75	No Loops	0.000561	47,056	N/A
DFS (Matrix)	75	With Loops	0.000598	47,056	N/A
DFS (Matrix)	100	No Loops	0.000985	88,544	N/A
DFS (Matrix)	100	With Loops	0.001035	88,544	N/A
DFS (Matrix)	250	No Loops	0.005018	495,992	N/A
DFS (Matrix)	250	With Loops	0.004646	495,992	N/A
DFS (Matrix)	500	No Loops	0.021399	2,126,606	N/A
DFS (Matrix)	500	With Loops	0.022610	2,104,324	N/A
BFS (Matrix)	10	No Loops	7.9e-05	2,240	N/A
BFS (Matrix)	10	With Loops	4.4e-05	2,240	N/A
BFS (Matrix)	15	No Loops	6.6e-05	3,480	N/A
BFS (Matrix)	15	With Loops	6.2e-05	3,480	N/A
BFS (Matrix)	50	No Loops	0.000443	24,040	N/A
BFS (Matrix)	50	With Loops	0.000387	24,040	N/A

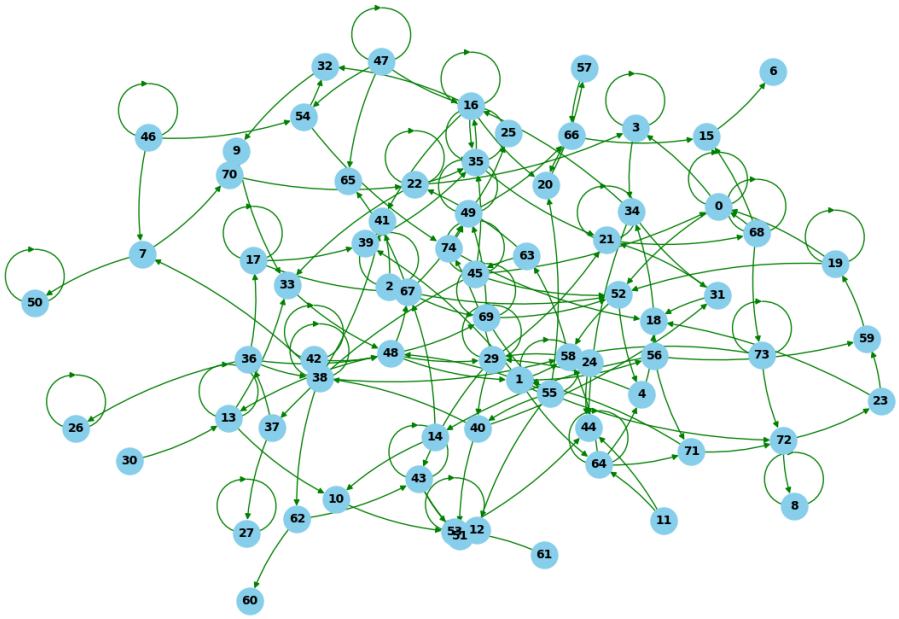
Table 2.12: Execution Time and Memory Usage – No Loops vs With Loops (Continuation)

Algorithm	**Nodes**	**Graph Type**	**Time (s)**	**Memory (B)**	**Result**
BFS (Matrix)	75	No Loops	0.000665	47,056	N/A
BFS (Matrix)	75	With Loops	0.000570	47,056	N/A
BFS (Matrix)	100	No Loops	0.001085	88,544	N/A
BFS (Matrix)	100	With Loops	0.001178	88,544	N/A
BFS (Matrix)	250	No Loops	0.005400	495,992	N/A
BFS (Matrix)	250	With Loops	0.005496	496,136	N/A
BFS (Matrix)	500	No Loops	0.017363	2,126,412	N/A
BFS (Matrix)	500	With Loops	0.024447	2,126,046	N/A
DFS (List)	10	No Loops	6.6e-05	2,160	N/A
DFS (List)	10	With Loops	8.6e-05	2,160	N/A
DFS (List)	15	No Loops	8.2e-05	2,648	N/A
DFS (List)	15	With Loops	7.1e-05	2,648	N/A
DFS (List)	50	No Loops	0.000224	9,280	N/A
DFS (List)	50	With Loops	0.000242	9,432	N/A
DFS (List)	75	No Loops	0.000315	10,512	N/A
DFS (List)	75	With Loops	0.000324	10,992	N/A
DFS (List)	100	No Loops	0.000439	23,480	N/A
DFS (List)	100	With Loops	0.000436	23,848	N/A
DFS (List)	250	No Loops	0.000993	187,871	N/A
DFS (List)	250	With Loops	0.001130	41,527	N/A
DFS (List)	500	No Loops	0.002374	99,848	N/A
DFS (List)	500	With Loops	0.001976	249,552	N/A
BFS (List)	10	No Loops	6.8e-05	2,720	N/A
BFS (List)	10	With Loops	7.4e-05	2,720	N/A
BFS (List)	15	No Loops	6.1e-05	2,648	N/A
BFS (List)	15	With Loops	7.7e-05	2,648	N/A
BFS (List)	50	No Loops	0.000237	9,960	N/A
BFS (List)	50	With Loops	0.000256	10,112	N/A
BFS (List)	75	No Loops	0.000268	11,208	N/A
BFS (List)	75	With Loops	0.000351	11,688	N/A
BFS (List)	100	No Loops	0.000371	24,624	N/A
BFS (List)	100	With Loops	0.000460	24,992	N/A
BFS (List)	250	No Loops	0.001019	40,520	N/A
BFS (List)	250	With Loops	0.001016	41,440	N/A
BFS (List)	500	No Loops	0.001903	101,287	N/A
BFS (List)	500	With Loops	0.002092	102,056	N/A

Visual Graph Examples



(a) Graph without self-loops (15 nodes)



(b) Graph with self-loops (75 nodes)

Figure 2.11: Structural comparison between loop-free and self-loop-enhanced graphs

Graphical Analysis of Traversal Performance

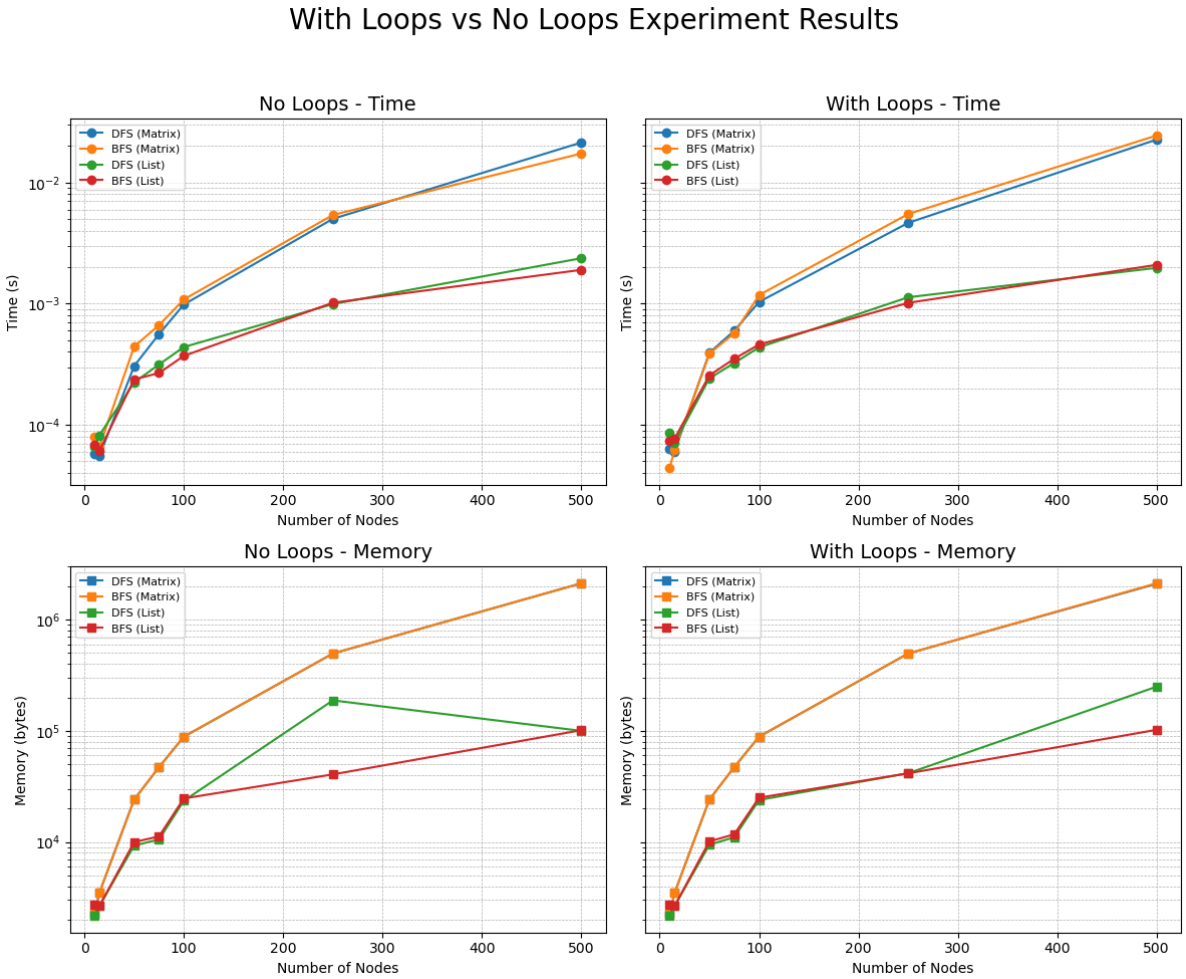


Figure 2.12: With Loops vs No Loops: BFS and DFS Time and Memory Analysis

Effect of Self-Loops in Traversal:

- Self-loops do not enhance connectivity, but in matrix form they still occupy memory and may increase lookup overhead during traversal.
- DFS skips previously visited nodes, so self-loops may cause negligible additional cost unless counted multiple times.
- BFS can briefly enqueue self-looped nodes but will recognize and skip them due to visited flags.
- In list-based representations, self-loops have minimal memory impact. In matrix form, they affect both density and size.

Analysis

- **List-based traversals** remain largely unaffected by self-loops in terms of time and memory.

- **Matrix representations** experience small performance shifts, particularly in memory, due to added edge entries.
- **DFS** is slightly more susceptible to performance noise with loops due to recursion overhead on revisits.
- Overall, self-loops have a **minor but measurable impact** when the representation is dense or the graph is large.

2.3.7 Trees as Graph Structures

Trees represent a specialized class of graphs with no cycles and exactly one path between any two connected nodes. These properties make trees ideal for analyzing best-case performance for traversal algorithms. In this section, we measure how BFS and DFS perform on trees of various sizes, using both adjacency matrix and list representations.

Empirical Results Tables

The tables below show the execution time and memory consumption for DFS and BFS on tree structures generated using a randomized spanning tree approach.

Table 2.13: Execution Time and Memory Usage – Trees

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
DFS (Matrix)	10	Tree	5.2e-05	2,456	N/A
DFS (Matrix)	15	Tree	5.1e-05	4,192	N/A
DFS (Matrix)	50	Tree	0.000325	27,032	N/A
DFS (Matrix)	75	Tree	0.000641	52,432	N/A
DFS (Matrix)	100	Tree	0.000801	95,056	N/A
DFS (Matrix)	250	Tree	0.005251	537,592	N/A
DFS (Matrix)	500	Tree	0.017566	2,180,900	N/A
BFS (Matrix)	10	Tree	5e-05	2,456	N/A
BFS (Matrix)	15	Tree	4.6e-05	4,192	N/A
BFS (Matrix)	50	Tree	0.000278	27,032	N/A
BFS (Matrix)	75	Tree	0.000389	52,432	N/A
BFS (Matrix)	100	Tree	0.000755	95,056	N/A
BFS (Matrix)	250	Tree	0.004456	537,592	N/A
BFS (Matrix)	500	Tree	0.022740	2,181,620	N/A

Table 2.14: Execution Time and Memory Usage – Trees (Continuation)

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
DFS (List)	10	Tree	5.1e-05	2,344	N/A
DFS (List)	15	Tree	4e-05	3,504	N/A
DFS (List)	50	Tree	0.000187	10,632	N/A
DFS (List)	75	Tree	0.000258	12,304	N/A
DFS (List)	100	Tree	0.000331	24,008	N/A
DFS (List)	250	Tree	0.000609	44,376	N/A
DFS (List)	500	Tree	0.001210	104,968	N/A
BFS (List)	10	Tree	5.3e-05	2,768	N/A
BFS (List)	15	Tree	6.4e-05	3,520	N/A
BFS (List)	50	Tree	0.000230	10,632	N/A
BFS (List)	75	Tree	0.000226	12,464	N/A
BFS (List)	100	Tree	0.000282	25,312	N/A
BFS (List)	250	Tree	0.000696	44,376	N/A
BFS (List)	500	Tree	0.001768	253,470	N/A

Visual Tree Example

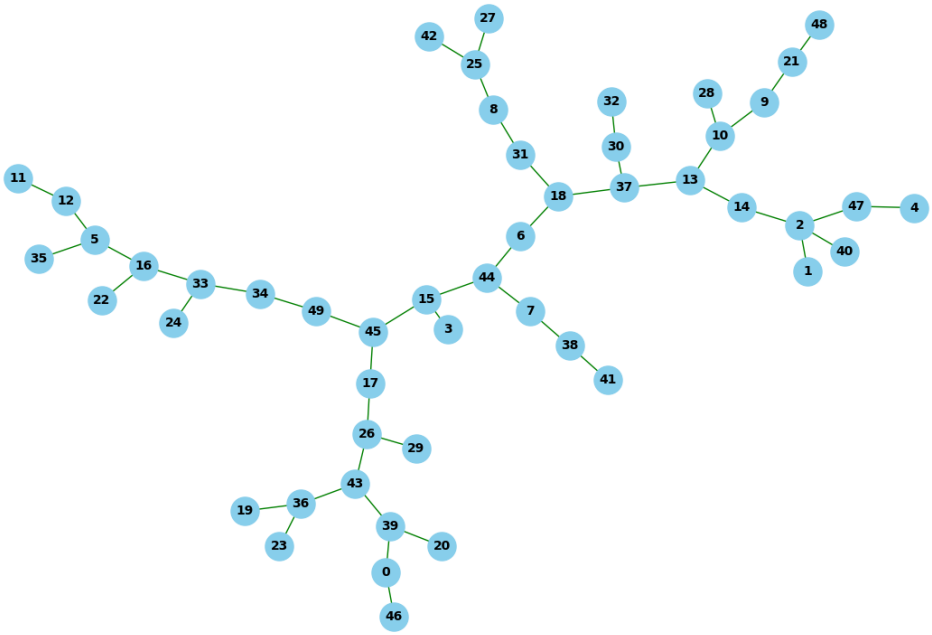


Figure 2.13: Example of a randomly generated tree with 50 nodes

Graphical Analysis of Traversal Performance

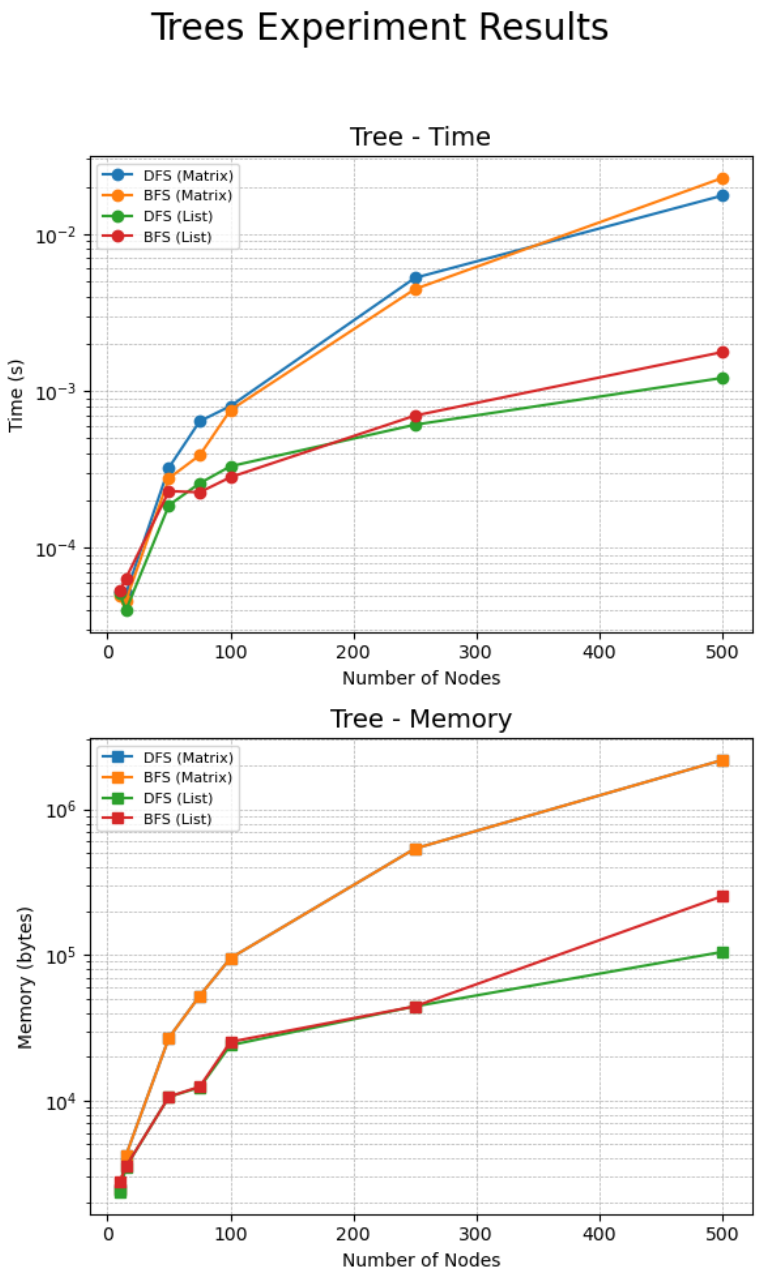


Figure 2.14: Tree Structure: BFS and DFS Time and Memory Analysis

Behavior of BFS and DFS on Trees:

- Trees contain no cycles and exactly $n - 1$ edges, allowing both DFS and BFS to traverse each node exactly once.
- This yields **optimal linear time complexity** in practice for both DFS and BFS.
- Memory usage in list form remains minimal due to sparse connectivity.
- Matrix representations, though sparse, still allocate $O(n^2)$ memory regardless of tree sparsity.

Analysis

- **Traversal time** for DFS and BFS on trees is consistently faster than on general graphs with cycles or dense connectivity.
- **List-based structures** clearly outperform matrix representations in both time and memory due to the sparse nature of trees.
- **DFS (List)** exhibits the best overall performance on trees in terms of efficiency.
- These results confirm that traversal algorithms perform optimally on acyclic, minimally connected structures.

2.3.8 Star Graph Structure

A star graph is a special case of a tree with one central node (the hub) directly connected to all other nodes (the spokes), and no other edges between non-central nodes. It represents an idealized structure for analyzing algorithms in shallow, high-fanout scenarios, common in hub-and-spoke systems such as transportation, broadcasting, or centralized networks.

Traversal performance in star graphs can reveal how well algorithms handle immediate expansion from a single root to many leaf nodes. BFS in particular benefits from this fan-out pattern, reaching all nodes in just one step, whereas DFS still behaves efficiently due to limited depth, though its stack-based mechanism may impose unnecessary overhead in comparison.

Empirical Results Table

The following table summarizes the execution time and memory usage for DFS and BFS using both matrix and list representations applied to star graphs.

Table 2.15: Execution Time and Memory Usage – Star Graph

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
DFS (Matrix)	10	Star	4.8e-05	2,456	N/A
DFS (Matrix)	15	Star	7.3e-05	4,192	N/A
DFS (Matrix)	50	Star	0.000339	27,032	N/A
DFS (Matrix)	75	Star	0.00103	52,432	N/A
DFS (Matrix)	100	Star	0.001583	95,056	N/A
DFS (Matrix)	250	Star	0.007282	537,592	N/A
DFS (Matrix)	500	Star	0.029606	2,192,028	N/A
BFS (Matrix)	10	Star	4.8e-05	2,456	N/A
BFS (Matrix)	15	Star	6.1e-05	4,192	N/A
BFS (Matrix)	50	Star	0.000344	27,032	N/A
BFS (Matrix)	75	Star	0.000719	52,432	N/A
BFS (Matrix)	100	Star	0.001375	95,056	N/A
BFS (Matrix)	250	Star	0.006086	537,592	N/A
BFS (Matrix)	500	Star	0.034801	2,192,236	N/A

Table 2.16: Execution Time and Memory Usage – Star Graph (continuatio)

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
DFS (List)	10	Star	0.000153	2,440	N/A
DFS (List)	15	Star	0.000178	3,568	N/A
DFS (List)	50	Star	0.000376	11,016	N/A
DFS (List)	75	Star	0.000649	13,408	N/A
DFS (List)	100	Star	0.001276	25,096	N/A
DFS (List)	250	Star	0.001684	195,967	N/A
DFS (List)	500	Star	0.003016	258,072	N/A
BFS (List)	10	Star	4.1e-05	2,864	N/A
BFS (List)	15	Star	8.2e-05	3,584	N/A
BFS (List)	50	Star	0.000135	11,016	N/A
BFS (List)	75	Star	0.000187	13,408	N/A
BFS (List)	100	Star	0.000368	26,112	N/A
BFS (List)	250	Star	0.000812	47,944	N/A
BFS (List)	500	Star	0.002287	110,736	N/A

Graphical Illustration of a Star Graph

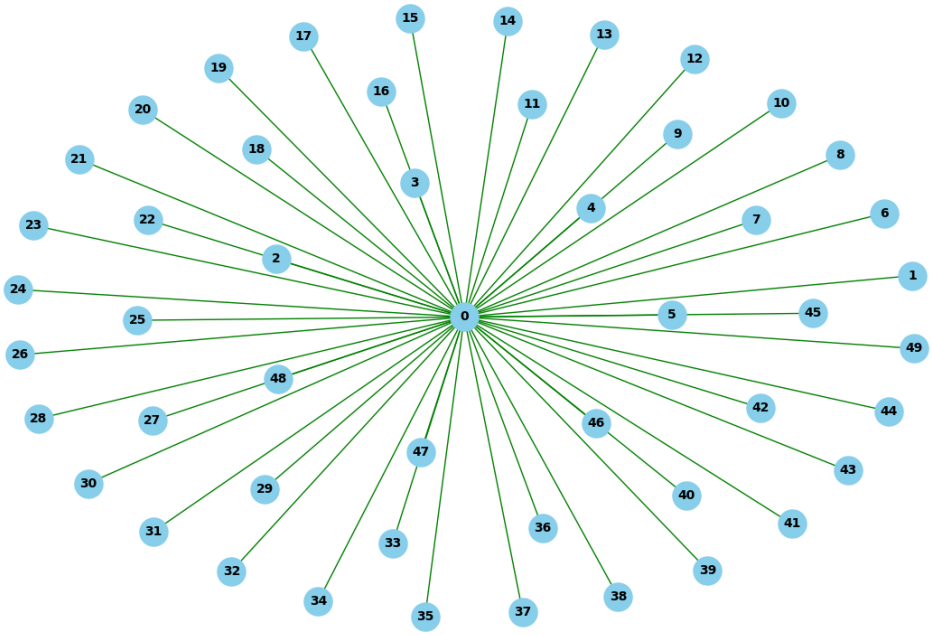


Figure 2.15: Star graph example with 50 nodes

Performance Plots

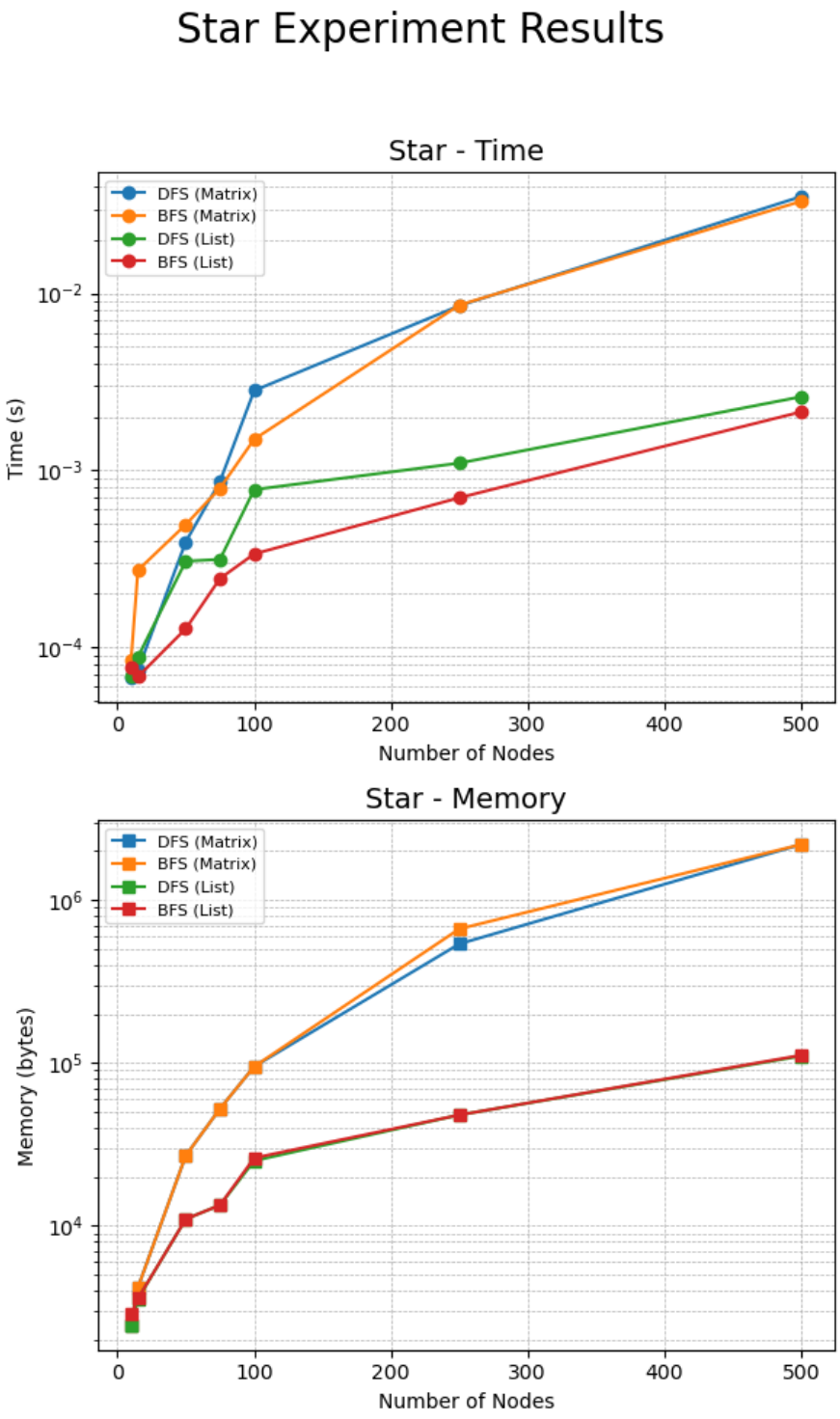


Figure 2.16: Star Graph: Time and Memory Performance for BFS and DFS

Analysis

- The star graph presents a simple but high-fanout structure where one central node connects to all others.

- **BFS performance** benefits slightly from this structure, traversing in a single layer, making it ideal for exploring shallow graphs.
- **DFS**, while still efficient, has no depth to explore beyond one hop from the center.
- **Matrix representations** consume much more memory than lists, especially as the node count increases.
- **List-based traversals** are more efficient for sparse graphs like stars, especially in BFS.

2.3.9 Ring Graphs

Ring graphs are a fundamental topology where each node is connected to exactly two others, forming a closed cycle. This structure is commonly encountered in network design, token ring protocols, and load-balancing systems. From a traversal perspective, rings pose minimal complexity in connectivity but introduce cyclical paths which can affect algorithmic behavior slightly differently compared to trees or stars.

Empirical Results Tables

The table below shows the execution time and memory usage for both matrix and list representations of BFS and DFS when applied to ring graphs of increasing size.

Table 2.17: Execution Time and Memory Usage – Ring Graphs

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
DFS (Matrix)	10	Ring	7.6e-05	2,456	N/A
DFS (Matrix)	15	Ring	0.000122	4,192	N/A
DFS (Matrix)	50	Ring	0.000422	27,032	N/A
DFS (Matrix)	75	Ring	0.000876	52,432	N/A
DFS (Matrix)	100	Ring	0.00124	95,056	N/A
DFS (Matrix)	250	Ring	0.008642	537,592	N/A
DFS (Matrix)	500	Ring	0.044265	2,085,472	N/A
BFS (Matrix)	10	Ring	5.9e-05	2,456	N/A
BFS (Matrix)	15	Ring	6e-05	4,192	N/A
BFS (Matrix)	50	Ring	0.000345	27,032	N/A
BFS (Matrix)	75	Ring	0.000615	52,432	N/A
BFS (Matrix)	100	Ring	0.00152	95,568	N/A
BFS (Matrix)	250	Ring	0.008111	537,592	N/A
BFS (Matrix)	500	Ring	0.028928	2,181,324	N/A

Table 2.18: Execution Time and Memory Usage – Ring Graphs (Continuation)

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
DFS (List)	10	Ring	4.9e-05	2,296	N/A
DFS (List)	15	Ring	4.7e-05	3,296	N/A
DFS (List)	50	Ring	0.000246	10,192	N/A
DFS (List)	75	Ring	0.000224	11,840	N/A
DFS (List)	100	Ring	0.00051	23,976	N/A
DFS (List)	250	Ring	0.001043	189,871	N/A
DFS (List)	500	Ring	0.001764	103,720	N/A
BFS (List)	10	Ring	5.1e-05	2,768	N/A
BFS (List)	15	Ring	0.000113	3,488	N/A
BFS (List)	50	Ring	0.000198	10,248	N/A
BFS (List)	75	Ring	0.000363	12,464	N/A
BFS (List)	100	Ring	0.0003	25,280	N/A
BFS (List)	250	Ring	0.000874	43,096	N/A
BFS (List)	500	Ring	0.001458	253,103	N/A

Visual Graph Example

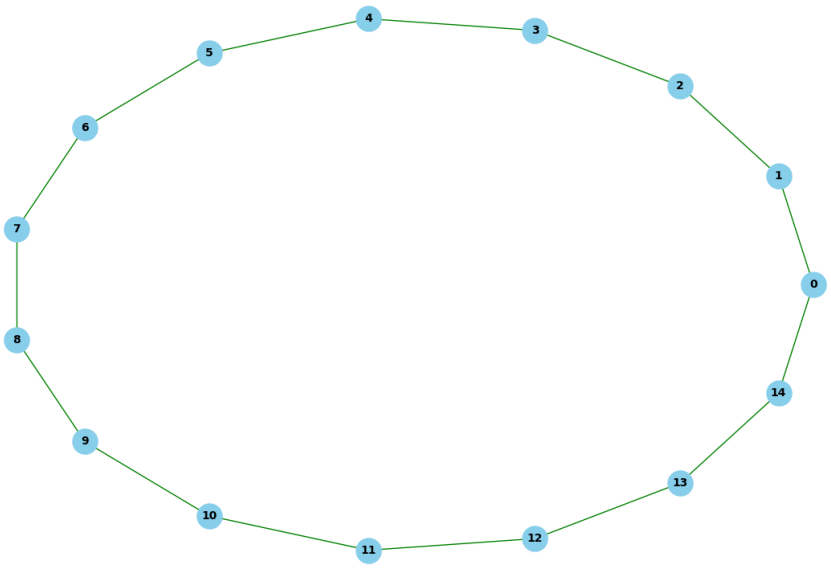


Figure 2.17: Example of a Ring Graph with 15 Nodes

Graphical Analysis of Traversal Performance

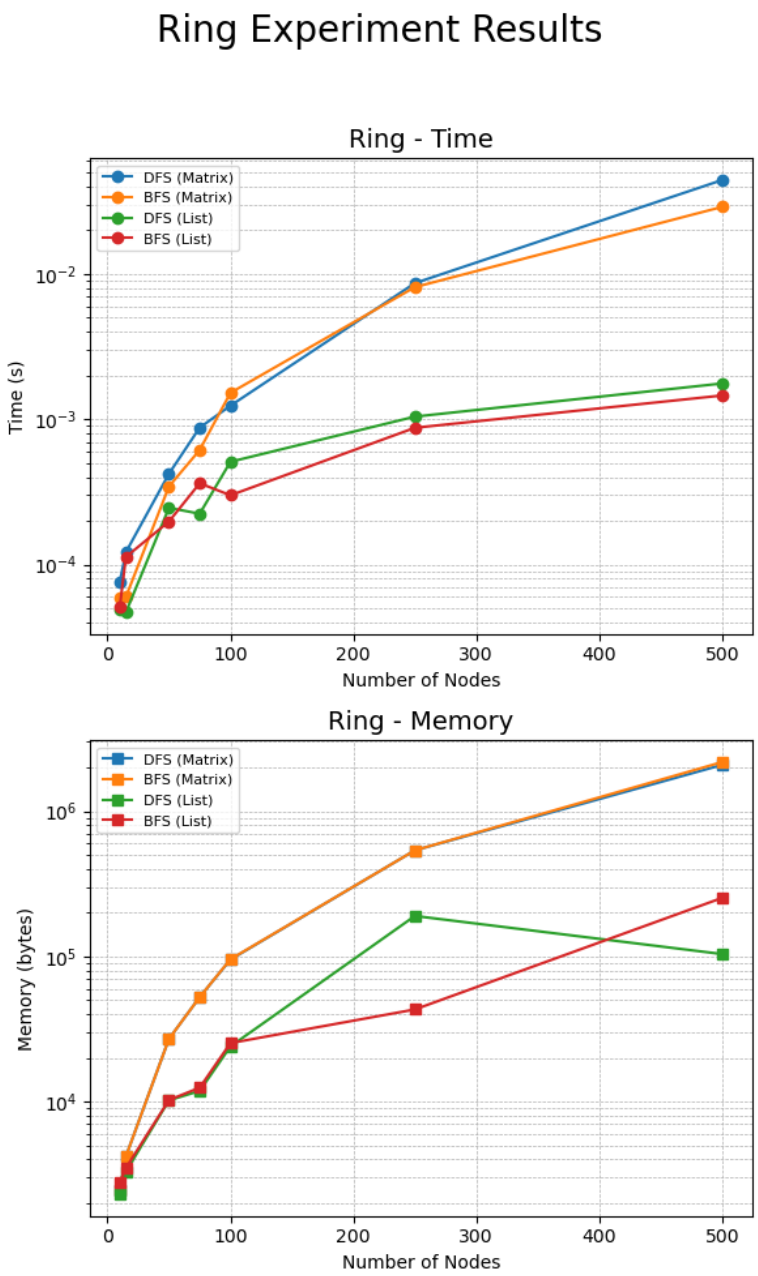


Figure 2.18: Ring Graph: BFS and DFS Time and Memory Analysis

Behavioral Characteristics of Traversal on Ring Graphs:

Ring graphs exhibit cyclical behavior without hierarchical structure. As such:

- **DFS** may revisit nodes unnecessarily if not properly marked, which emphasizes the need for visited tracking even in simple cycles.
- **BFS** performs efficiently, traversing neighbors level-wise with minimal branching.
- The closed loop ensures all nodes are connected linearly, but cyclic redundancy does not contribute to traversal performance.

- **Matrix representations** remain costly in memory as the number of nodes increases, regardless of edge count.
- **List representations** are far more memory-efficient, especially in sparse structures like rings.

Analysis

- Traversal in ring graphs is predictable and stable, resulting in moderate time complexity for both BFS and DFS.
- BFS slightly outperforms DFS in time for larger rings, likely due to its linear queue expansion vs DFS's potential stack overhead in cyclic structures.
- Ring graphs introduce uniform behavior across all nodes, minimizing variance in traversal paths.
- As expected, memory usage remains consistent across experiments, with matrix representations consuming substantially more space.

Chapter 3

Conclusion

This laboratory report presents a detailed empirical evaluation of the Depth-First Search (DFS) and Breadth-First Search (BFS) algorithms across a wide variety of graph structures and configurations. The experiments were executed on an **Asus ZenBook 14** with an **Intel Core i7 8th Gen processor**, **16GB RAM**, and **512GB SSD**, using **Python** in **Jupyter Notebook** locally.

The analysis focused on comparing algorithm performance in terms of **execution time** and **memory usage**, with both **adjacency matrix** and **adjacency list** implementations evaluated. Graphs of increasing size (from 10 to 500 nodes) were generated using various topologies, including trees, stars, rings, sparse/dense, connected/disconnected, cyclic/acyclic, and more.

Key Insights

- **Matrix vs. List Representations:** Across all graph types, adjacency lists demonstrated superior memory efficiency, particularly on sparse and tree-like graphs. Matrix representations exhibited consistent but higher memory consumption due to their $O(n^2)$ space complexity.
- **DFS vs. BFS Behavior:** While both algorithms maintained linear time performance on acyclic and sparse graphs, DFS (List) generally exhibited the best runtime efficiency. BFS (List) consistently showed the lowest memory usage.
- **Effect of Graph Structure:**
  - *Trees:* Provided optimal conditions for both algorithms, with linear traversal paths and minimal memory requirements. DFS (List) achieved the fastest execution times.
  - *Star Graphs:* BFS benefitted from the shallow high-fanout layout, reaching all nodes in one layer. DFS handled the layout well but lacked opportunity for deep recursion.
  - *Ring Graphs:* Introduced cyclicity and uniform node degree, leading to stable but slightly elevated memory usage. DFS showed modestly longer execution times due to cycle-handling overhead.
- **Traversal Depth and Structure Impact:** DFS suffered more in deeply nested or cyclic structures, particularly in matrix form due to stack depth and lookup

overhead. BFS scaled more gracefully in such conditions, especially with list representations.

- **Scalability Observations:** Execution time increased linearly with node count in list-based implementations, while matrix-based versions scaled worse due to memory and access overhead. This confirmed the practical value of space-efficient structures for large-scale graphs.
- **Graph Connectivity and Loops:** Disconnected and loop-enhanced graphs marginally increased traversal time and memory, primarily in matrix form. However, list-based implementations showed resilience to such variations.

Final Remarks

This empirical study validated theoretical expectations while revealing subtle performance characteristics influenced by graph structure, implementation choices, and traversal strategies. List-based representations are recommended for scalable, efficient BFS and DFS operations across real-world graphs. Moreover, the findings emphasize that careful selection of data structures and awareness of graph topology are crucial for optimizing graph traversal algorithms in practice.

Source Code

GitHub Repository: https://github.com/PatriciaMoraru/AA_Laboratory_Works