MINISTRY OF EDUCATION AND RESEARCH OF REPUBLIC OF MOLDOVA
TECHNICAL UNIVERSITY OF MOLDOVA
FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS
DEPARTMENT OF SOFTWARE AND AUTOMATION ENGINEERING

ANALYSIS OF ALGORITHMS

LABORATORY WORK #3

Study and empirical analysis of algorithms: Depth First Search (DFS), Breadth First Search (BFS)

Author:

Andrei Chicu

std. gr. FAF–233

Verified:

Fistic Cristofor

Department of SEA, FCIM UTM

Chișinău, 2025

1 Analysis of Algorithms

github url: `https://github.com/andyp1xe1/aa_labs/tree/main/lab3`

1.1 Objective

The objective of this laboratory work is to implement and analyze two fundamental graph traversal algorithms: Depth First Search (DFS) and Breadth First Search (BFS). The analysis aims to compare their performance across various graph types and sizes to understand their efficiency characteristics, time complexity behavior, and practical applications in different graph scenarios.

1.2 Tasks

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.

1.3 Theoretical Notes

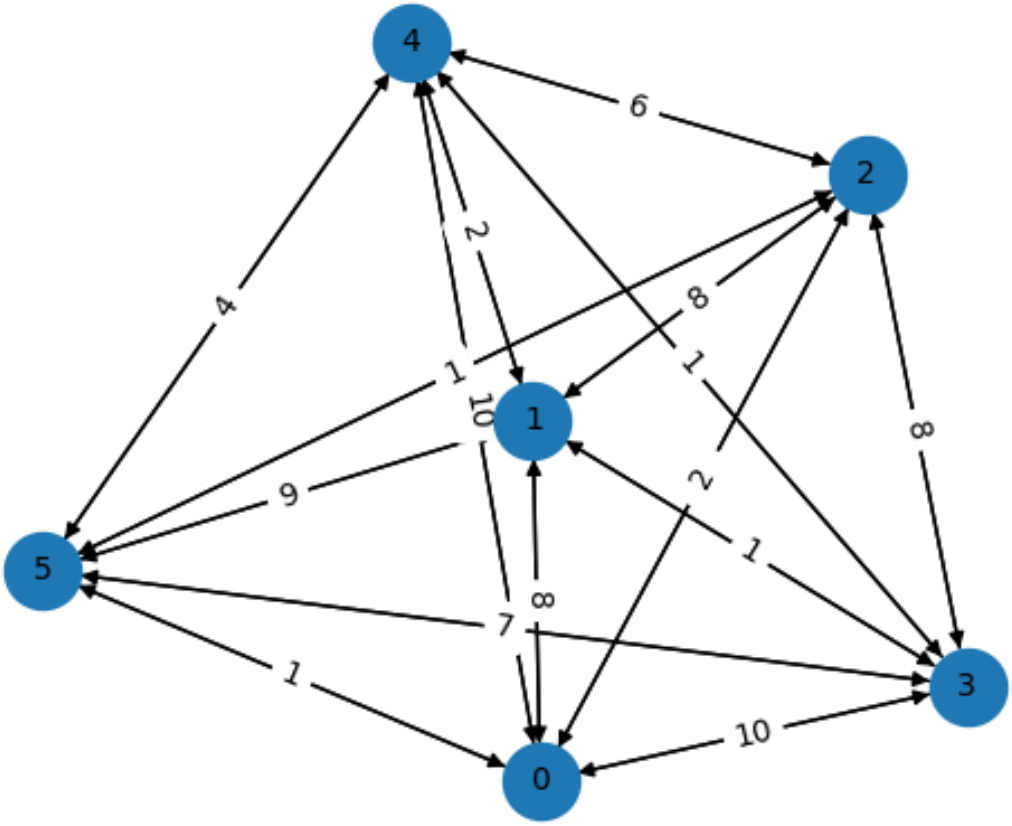
This lab explores various graph algorithms and their applications. Below, we discuss the different types of graphs used in our implementations and analyses.

A graph G is a pair (V, E) where V is a set of vertices (nodes) and E is a set of edges connecting these vertices. In this laboratory work, we focus on directed weighted graphs represented using adjacency lists.

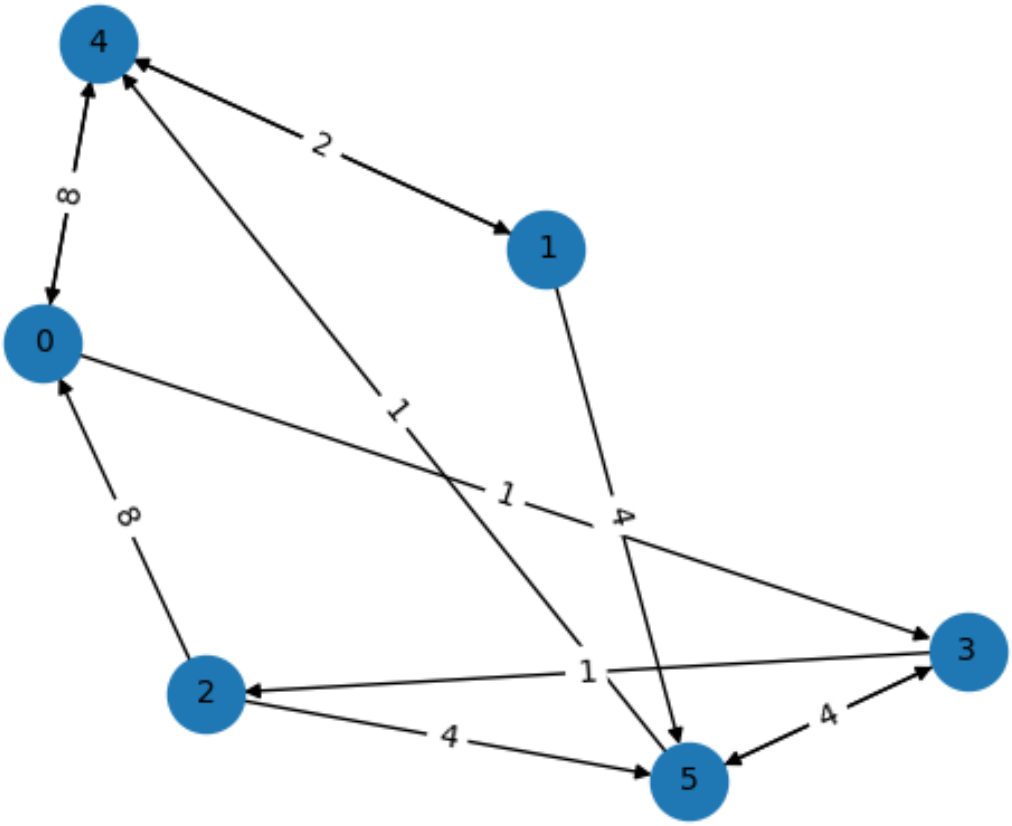
1.3.1 Types of Graphs

In this laboratory, we analyze the following types of graphs:

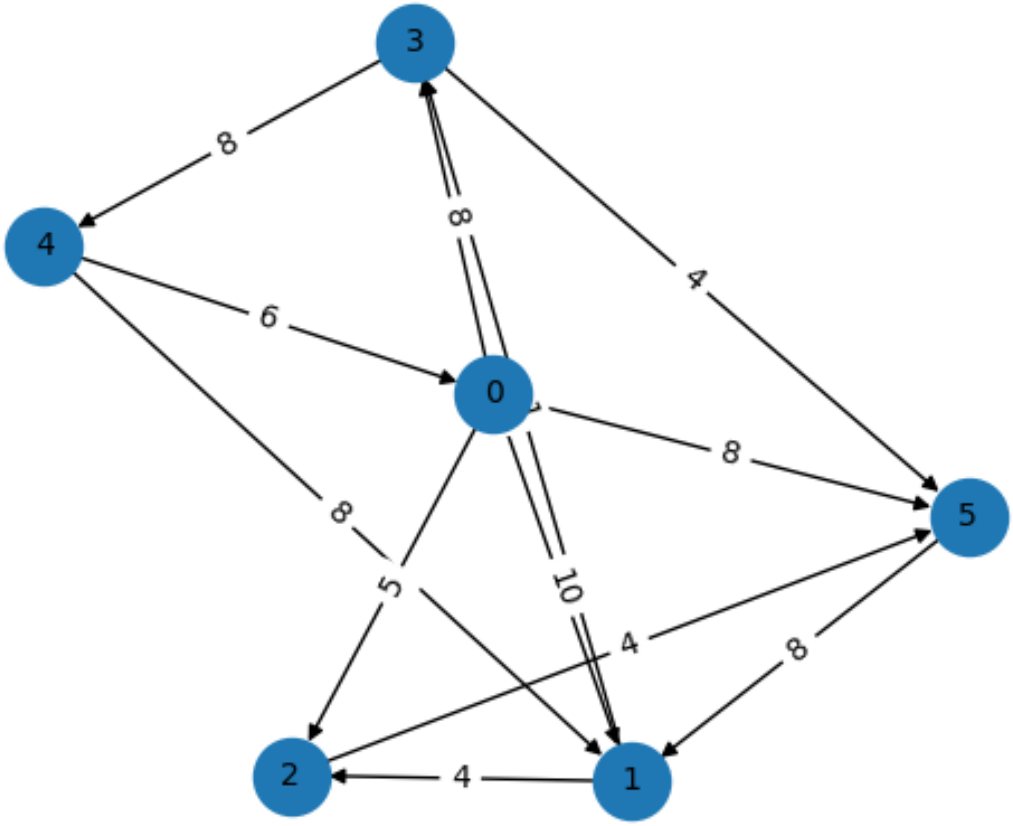
1. Complete Graph Every vertex is connected to every other vertex, resulting in |V|(|V|-1) edges in a directed graph.



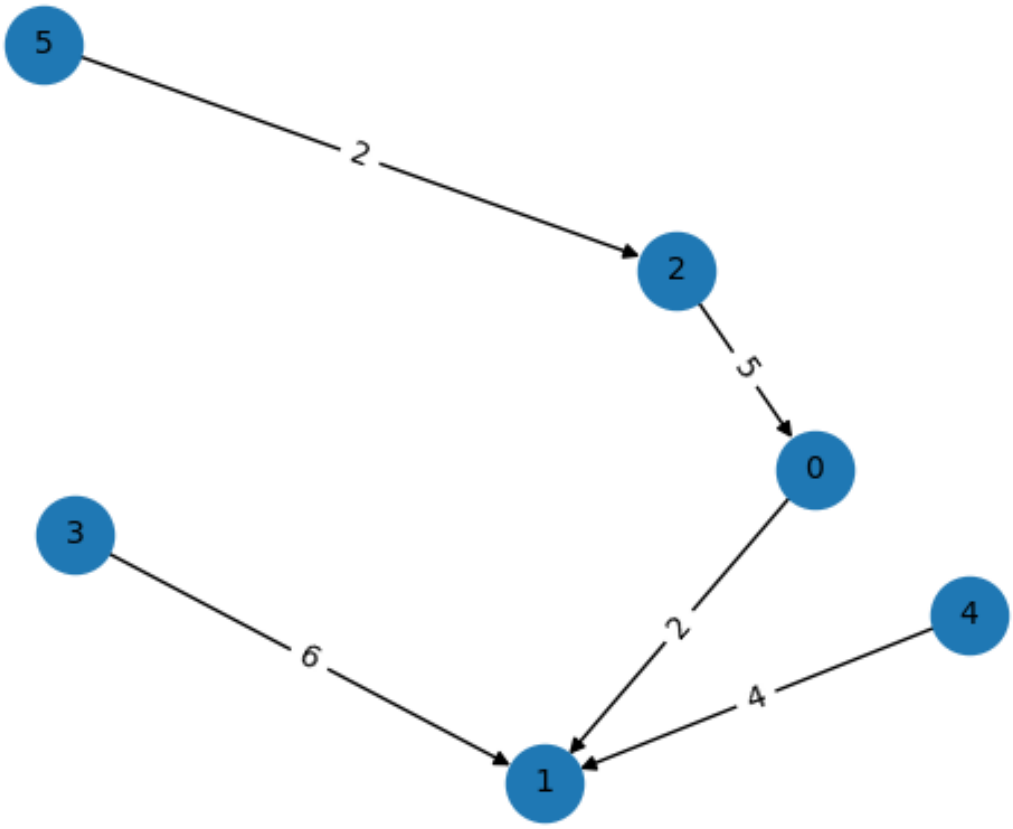
2. Dense Graph Has approximately 80% of the maximum possible edges.



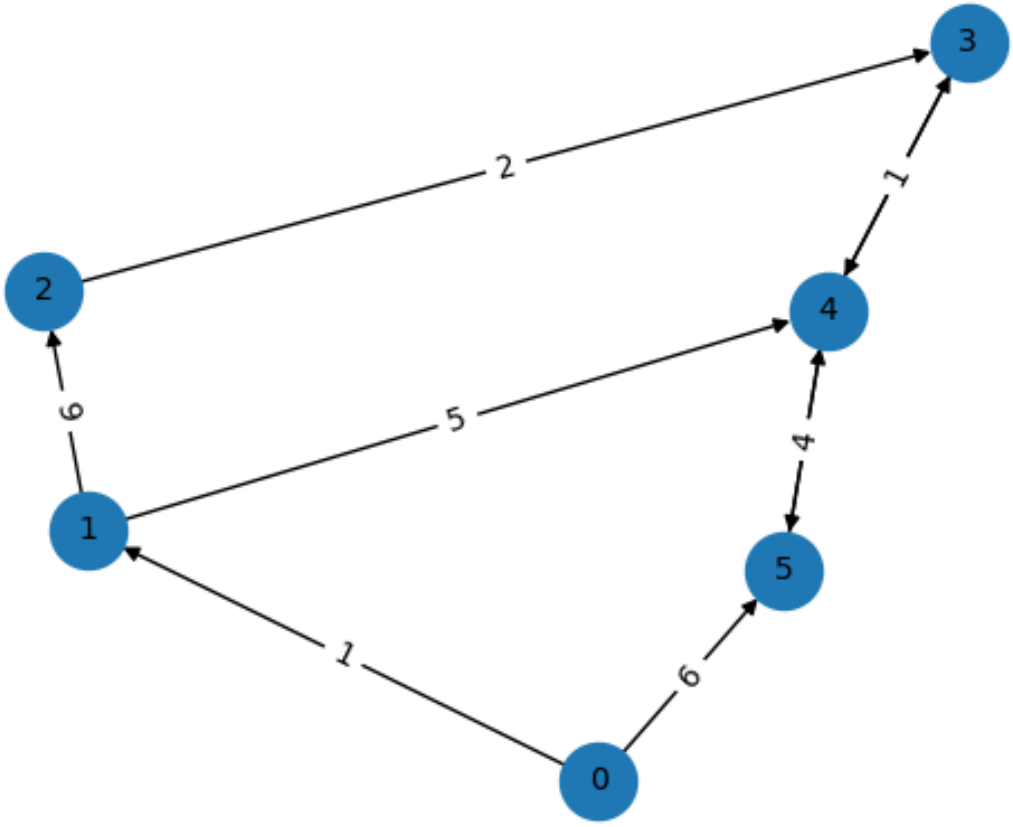
3. Sparse Graph Contains relatively few edges, approximately 2|V| edges.



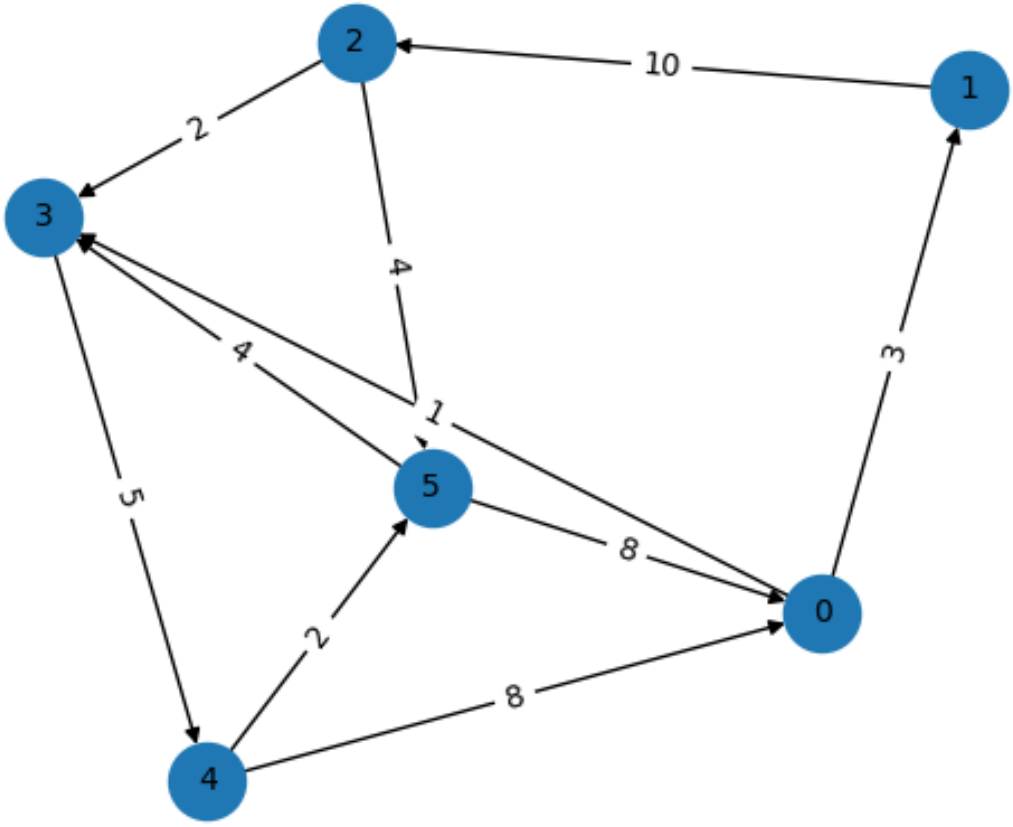
4. Tree Graph Connected acyclic graph with exactly |V|-1 edges.



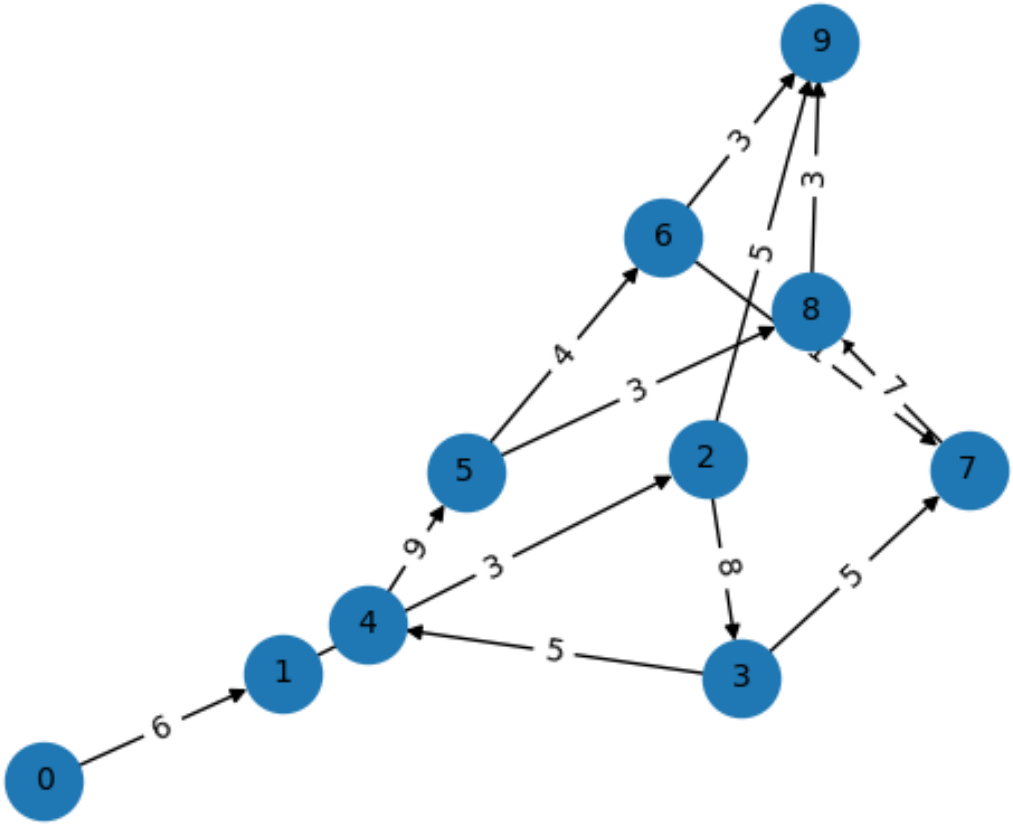
5. Connected Graph There exists a path between any pair of vertices.



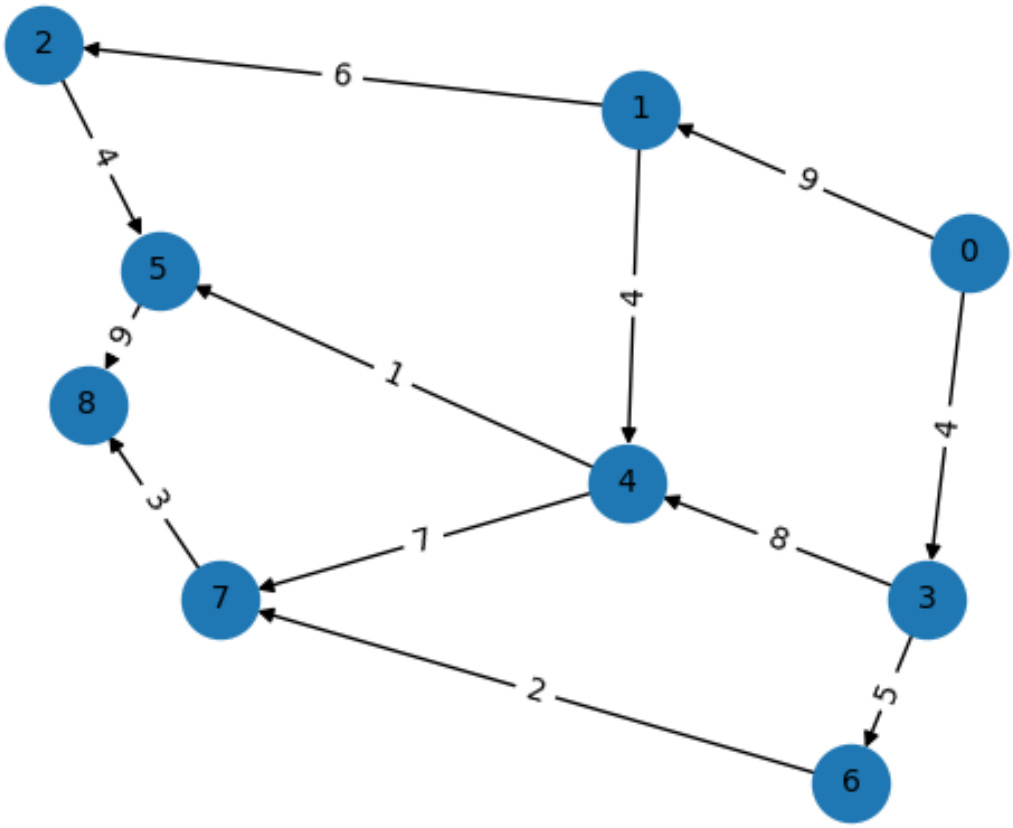
6. Cyclic Graph Contains at least one cycle (a path that starts and ends at the same vertex).



7. Acyclic Graph Contains no cycles (DAG - Directed Acyclic Graph).



8. Grid Graph Vertices arranged in a grid-like structure with connections primarily to adjacent nodes.



1.3.2 Depth First Search (DFS)

DFS is a graph traversal algorithm that explores as far as possible along each branch before backtracking. The algorithm works as follows:

1. Start at a designated source vertex
2. Explore the first neighbor of the current vertex
3. Continue this process recursively for each unexplored neighbor
4. Backtrack when a vertex has no unexplored neighbors
5. Continue until all reachable vertices have been visited

DFS uses a stack data structure (implicitly through recursion in many implementations) to keep track of vertices.

1.3.3 Breadth First Search (BFS)

BFS explores all neighbors at the current depth level before moving to vertices at the next depth level:

1. Start at a designated source vertex
2. Explore all immediate neighbors of the source
3. For each of those neighbors, explore their unexplored neighbors
4. Continue level by level until all reachable vertices have been visited

BFS uses a queue data structure to keep track of vertices to be explored.

1.4 Comparison Metric

For this empirical analysis, we measure the following metrics:

1. **Execution Time**: The primary metric is the actual execution time in seconds for each algorithm across different graph types and sizes. This provides an empirical measure of the algorithm's efficiency.
2. **Visited Nodes**: The total number of nodes visited during the traversal. In most cases, this will be all reachable nodes from the starting vertex.
3. **Memory Usage**: Indirectly measured by observing the maximum size of the data structures used (stack for DFS, queue for BFS).

The algorithms are evaluated on various graph types with sizes ranging from 100 to 2500/10,000 vertices to observe how they scale with input size.

```
sizes = [100, 200, 500, 1000, 2000, 2500]
big_sizes = [200, 500, 1000, 2000, 2500, 3000, 3500, 4000, 5000, 6000, 7000,
↪  8000, 9000, 10_000]
```

For each graph type and size, algorithms were executed with 5 repetitions to account for system variability and provide statistical robustness. This approach allows us to:

- Calculate mean execution times for more reliable performance comparison
- Determine standard deviation to assess result consistency and reliability
- Identify potential outliers or anomalous behavior in specific test cases

The error bars in the plots represent the standard deviation across these repetitions, providing a visual indication of measurement variability.

1.5 Input Format

In this laboratory work, graphs are represented using adjacency lists implemented as nested Python dictionaries. Each vertex is a key in the outer dictionary, and its value is another dictionary mapping neighboring vertex IDs to edge weights.

The testing framework generates graphs of various types and sizes, then executes both DFS and BFS algorithms on them, measuring the execution time.

Example input graph structure:

```
graph = {
    '0': {'1': 5, '2': 3},
    '1': {'3': 2, '4': 4},
    '2': {},
    '3': {},
    '4': {'2': 1}
}
```

In this representation:

- Vertex ’0’ has edges to vertices ’1’ (weight 5) and ’2’ (weight 3)
- Vertex ’1’ has edges to vertices ’3’ (weight 2) and ’4’ (weight 4)

- Vertices ’2’ and ’3’ have no outgoing edges
- Vertex ’4’ has an edge to vertex ’2’ (weight 1)

All graphs are generated with the functions provided in the uploaded code, with vertices labeled as strings from ’0’ to ’n-1’ and edge weights ranging from 1 to 10.

2 Implementation

2.1 Depth First Search

Depth First Search (DFS) is implemented recursively, exploring one path as deeply as possible before backtracking. The algorithm maintains a set of visited vertices to avoid cycles and infinite recursion.

```
def dfs(graph, start, visited=None, proc=None):
    """
        Depth-First Search algorithm implementation.
        Args:
        graph (dict): Adjacency list representation of the graph.
        start (str): Starting vertex.
        Returns:
        list: Order of vertices visited.
        """
    if visited is None:
        visited = set()
    if proc is None:
        proc = []

    visited.add(start)

    for neighbor, weight in graph[start].items():
        if neighbor not in visited:
            proc.append((start, neighbor, weight))
            dfs(graph, neighbor, visited, proc)
    return proc
```

Listing 1: Implementation of Depth First Search

2.2 Breadth First Search

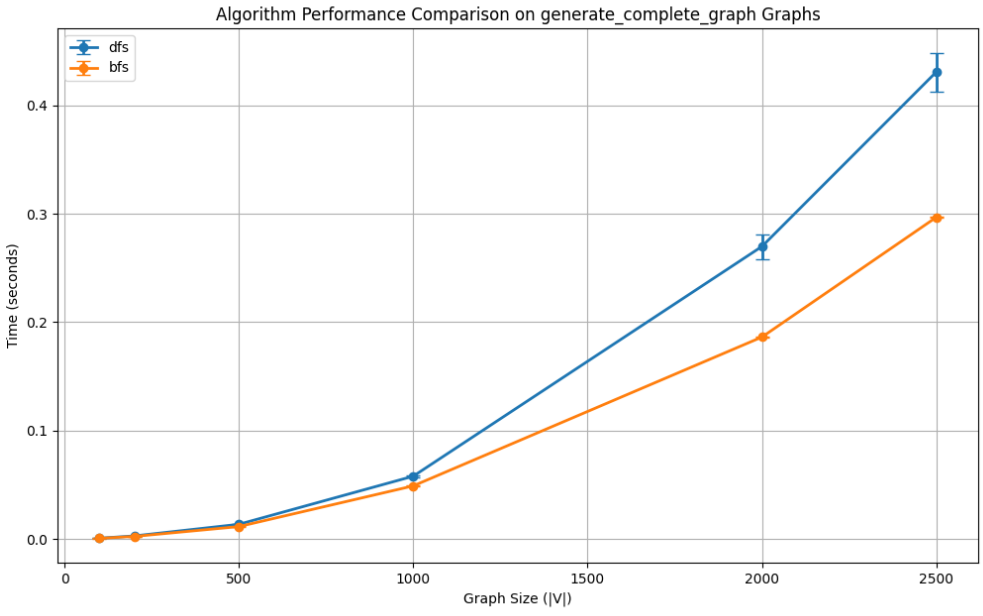
Breadth First Search (BFS) uses a queue to visit vertices in level order, exploring all neighbors of the current vertex before moving to the next level. Similar to DFS, it maintains a visited set to avoid revisiting vertices.

```python
import collections

def bfs(graph, root):
    """
        Breadth-First Search algorithm implementation.
        Args:
        graph (dict): Adjacency list representation of the graph.
        start (str): Starting vertex.
        Returns:
        list: Order of vertices visited.
        """
    visited = set()
    queue = collections.deque([root])
    visited.add(root)
    proc = []

    while queue:
        vertex = queue.popleft()

        for neighbor, weight in graph[vertex].items():
            if neighbor not in visited:
                visited.add(neighbor)
                queue.append(neighbor)
                proc.append((vertex, neighbor, weight))
    return proc
```

Listing 2: Implementation of Breadth First Search

3 Results

3.1 Complete Graph



Both algorithms show a clear polynomial growth pattern as the graph size increases, consistent with

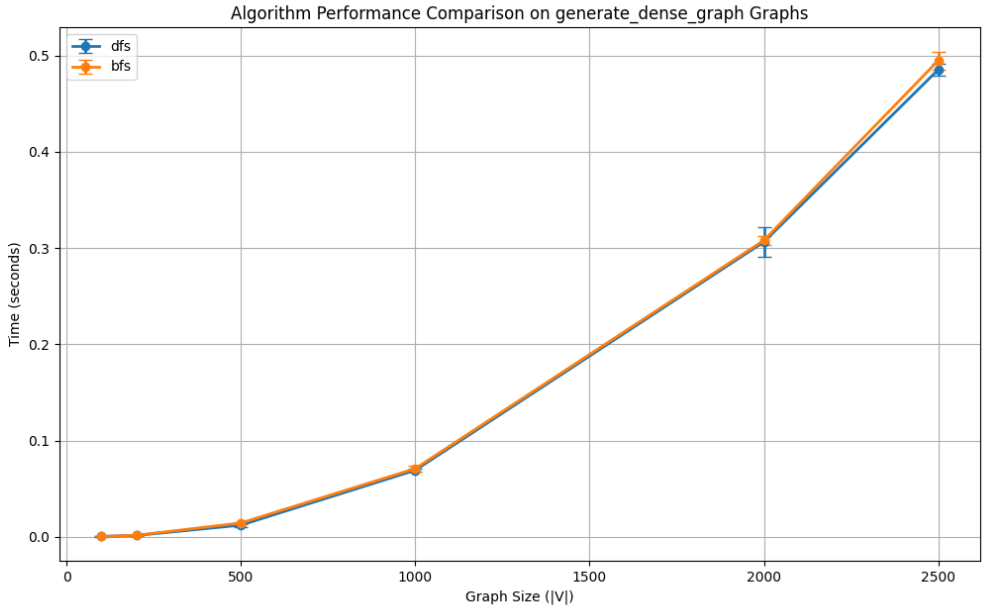
their O(V+E) complexity where E = O(V^2) for complete graphs.

DFS consistently demonstrates higher execution times compared to BFS, with the gap widening as graph size increases. At 2500 vertices, DFS takes approximately 0.43 seconds while BFS requires around 0.30 seconds.

The performance difference is likely due to the recursive implementation of DFS versus the iterative implementation of BFS, which incurs additional function call overhead.

Standard deviation is relatively small for both algorithms, indicating consistent performance across repetitions.

3.2 Dense Graph



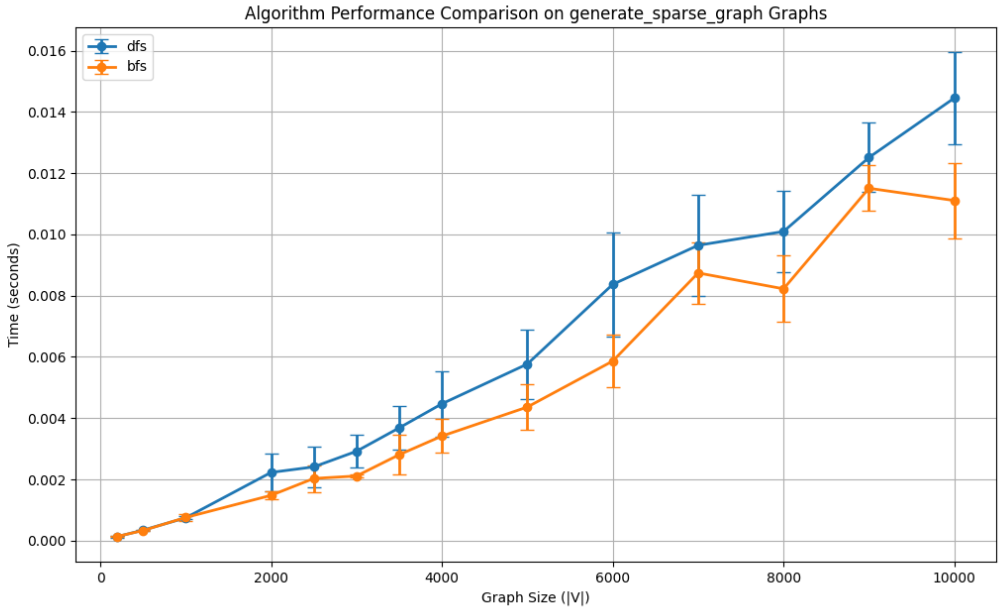
In dense graphs (approximately 80% of maximum possible edges), both algorithms show nearly identical performance profiles.

The execution time growth is polynomial, matching the theoretical expectation for graphs where E O(V^2).

Unlike in complete graphs, the performance difference between DFS and BFS is minimal, suggesting that the overhead of recursion in DFS is offset by other factors in dense but not complete graphs.

The execution times reach approximately 0.5 seconds at 2500 vertices for both algorithms. Low standard deviation values indicate high consistency in measurement.

3.3 Sparse Graph



For sparse graphs (with approximately 2|V| edges), both algorithms exhibit a more linear growth pattern, consistent with their theoretical O(V+E) complexity where E = O(V).

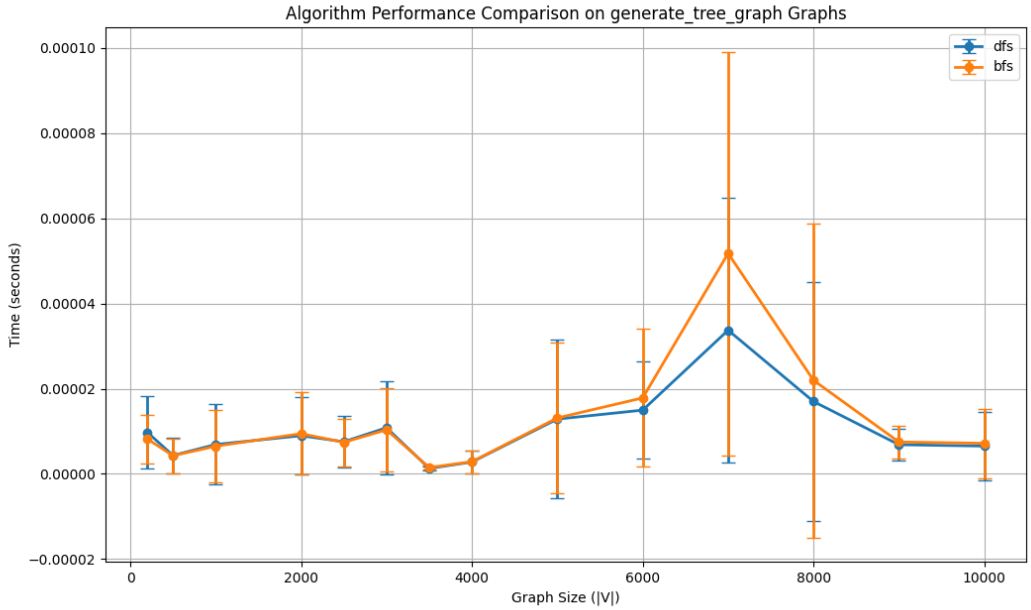
DFS consistently performs worse than BFS across all graph sizes, with the gap becoming more pronounced as the graph size increases.

At 10,000 vertices, DFS requires approximately 0.014 seconds while BFS takes around 0.011 seconds.

The standard deviation is relatively higher compared to other graph types, particularly for larger graph sizes, indicating some variability in performance measurements.

The performance advantage of BFS may be attributed to its level-by-level exploration, which could be more efficient in sparse graphs where most paths are relatively long.

3.4 Tree Graph



Tree graphs show the most unusual performance pattern among all tested graph types.

The execution times are extremely small (in the order of 10 seconds) even for large graphs with up to 10,000 vertices.

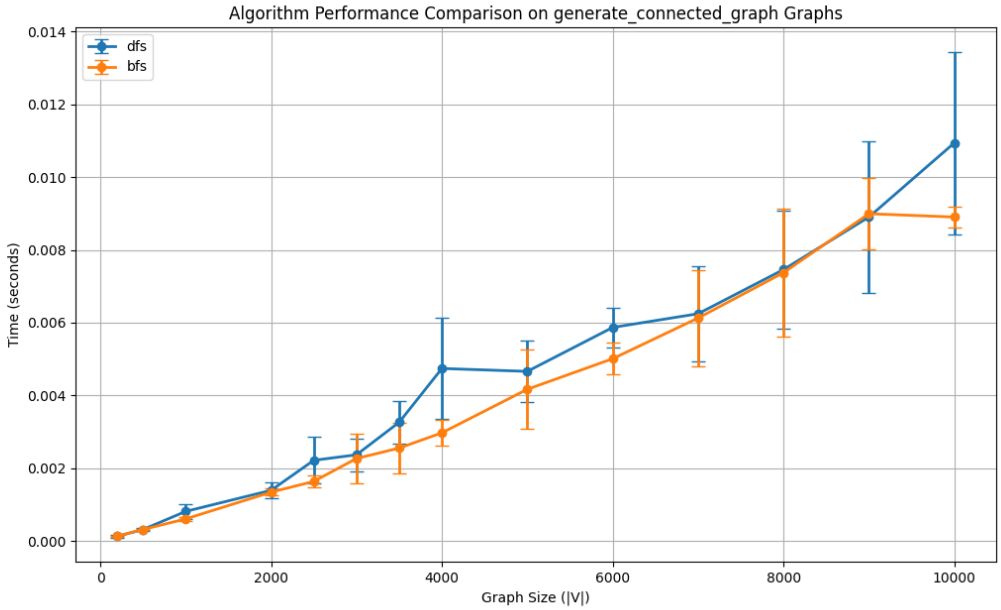
There is no clear trend in performance as graph size increases, with some larger graphs actually showing faster execution times than smaller ones.

The standard deviation is proportionally very large, often comparable to or exceeding the mean execution time. The unusual behavior might be attributed to:

- Highly efficient traversal of tree structures
- Caching effects becoming significant at these small time scales
- System timing resolution limitations at microsecond scales

The negative values on the y-axis are likely visualization artifacts due to error bars extending below zero.

3.5 Connected Graph



Both algorithms show a linear growth pattern, consistent with O(V+E) complexity when E is proportional to V.

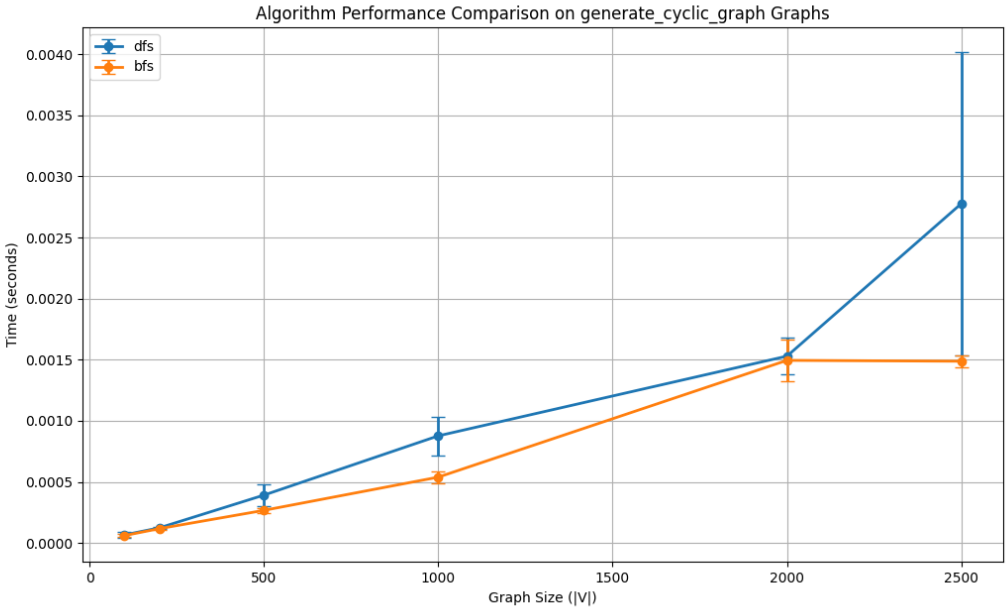
DFS performs slightly worse than BFS at larger graph sizes, but the difference is minimal compared to other graph types.

At 10,000 vertices, DFS takes approximately 0.011 seconds while BFS requires around 0.009 seconds.

The standard deviation increases with graph size but remains relatively consistent between the two algorithms.

The minimal performance difference suggests that both algorithms are well-suited for connected graphs with moderate edge density.

3.6 Cyclic Graph



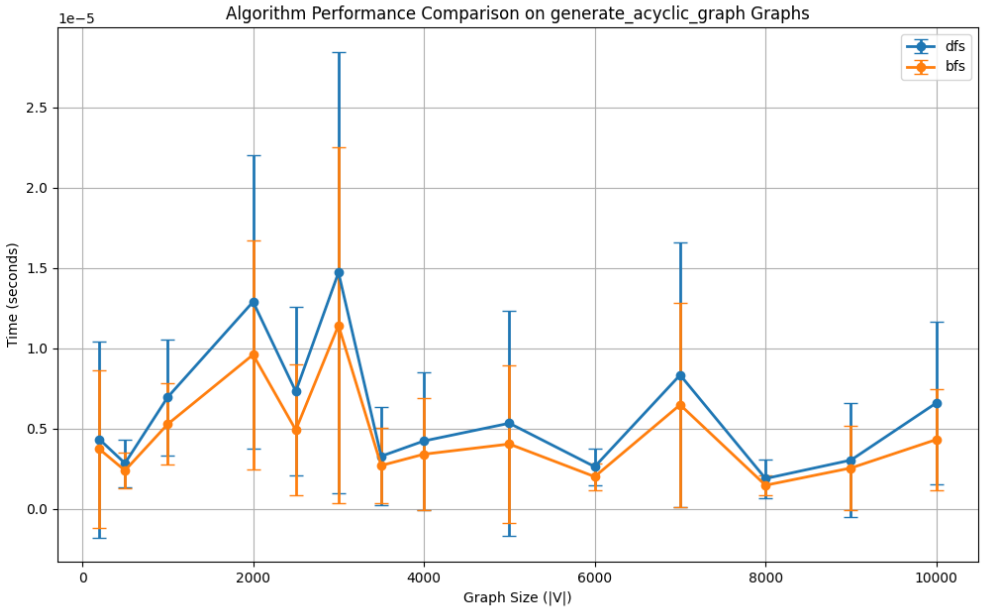
Both algorithms show a linear growth pattern up to 2000 vertices, after which BFS maintains linear growth while DFS begins to exhibit slightly faster growth.

At 2500 vertices, DFS takes approximately 0.0027 seconds while BFS requires around 0.0015 seconds.

BFS appears to handle cycles more efficiently than DFS, possibly because it avoids the deep recursion that can occur when DFS encounters cycles.

The standard deviation is moderate and increases with graph size, indicating reasonable measurement consistency.

3.7 Acyclic Graph



The acyclic graph tests show the most erratic behavior among all graph types, with significant fluctuations in execution time across different graph sizes.

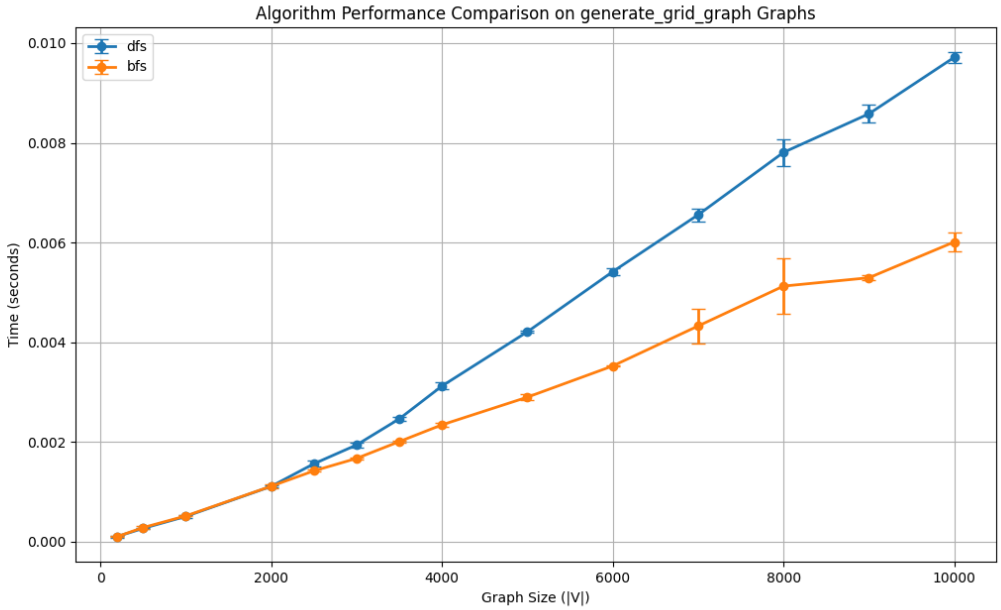
Despite the fluctuations, there is a general upward trend in execution time as graph size increases.

BFS generally outperforms DFS, particularly at larger graph sizes. The error bars are exceptionally large, indicating high variability across repetitions.

The erratic behavior might be attributed to:

- Specific topological characteristics of the generated acyclic graphs
- Memory access patterns that vary significantly between runs
- System scheduling effects becoming more pronounced

3.8 Grid Graph



Grid graphs show one of the clearest performance differences between DFS and BFS.

Both algorithms exhibit linear growth with graph size, but DFS shows a steeper slope.

At 10,000 vertices, DFS requires approximately 0.0097 seconds while BFS takes around 0.006 seconds.

The performance gap widens consistently as graph size increases, with BFS maintaining a roughly 40% advantage at the largest sizes.

The standard deviation is small relative to the mean values, indicating consistent measurements.

BFS's advantage in grid graphs is likely due to its level-by-level exploration pattern, which aligns well with the regular structure of grid graphs.

4 Conclusions

This laboratory work demonstrates that while DFS and BFS have the same theoretical time complexity (O(V+E)), their practical performance can vary based on graph structure and implementation details. The choice between these algorithms should consider not only execution time but also the specific requirements of the application, such as path characteristics, memory constraints, and the expected graph structure.

The empirical analysis confirms that while DFS and BFS share the same theoretical time complexity of O(V+E), their practical performance characteristics differ significantly depending on graph structure

BFS is generally more efficient across most graph types, particularly for grid structures and large

graphs. The performance advantage of BFS becomes more pronounced as graph size increases. For complete graphs and some specialized applications where depth-priority is beneficial, DFS may still be appropriate.

The recursive implementation of DFS introduces overhead that impacts real-world performance. An iterative implementation of DFS might reduce this gap. The choice between recursive and iterative implementations represents a trade-off between code clarity and performance.

Most graph types show consistent performance across repetitions, as evidenced by relatively small standard deviations. Tree graphs and acyclic graphs show higher variability, suggesting that performance for these specific structures may be more sensitive to system conditions.

Both algorithms scale according to theoretical predictions based on the relationship between vertices and edges. The practical growth rate is highly dependent on graph structure and edge density

These findings demonstrate the importance of empirical analysis to complement theoretical complexity analysis, as implementation details and structural characteristics can significantly impact real-world algorithm performance.