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

Laboratory Work 4

Study and empirical analysis of graph traversal algorithms
Analysis of Dijkstra, Floyd–Warshall

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 shortest path algorithms, specifically **Dijkstra's Algorithm** and the **Floyd–Warshall Algorithm**. The analysis will be conducted through empirical testing, evaluating the performance of these algorithms across various types of **weighted 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 Dijkstra and Floyd–Warshall algorithms in a programming language.
2. Establish the properties of the graph input data (e.g., directed/undirected, sparse/-dense, connected/disconnected, all **weighted**).
3. Choose appropriate metrics for comparing algorithm performance (e.g., execution time, memory usage).
4. Perform empirical analysis of the Dijkstra and Floyd–Warshall 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 derivations and there is no need for specific input.	The algorithm is analyzed by taking some sample of input and no mathematical derivation 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 Shortest Path Algorithms

Empirical Analysis Process

1. **Define the Objective:** Assess and compare the performance of Dijkstra's and Floyd–Warshall algorithms in computing shortest paths on weighted graphs.
2. **Select Performance Metrics:** Use execution time and memory usage as primary performance indicators.
3. **Determine Graph Characteristics:** Evaluate performance across graph variations such as:
   - Directed vs. Undirected
   - Sparse vs. Dense
   - Connected vs. Disconnected
   - 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 Shortest Path Algorithms

Dijkstra's Algorithm is a classic greedy algorithm for finding the shortest paths from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It performs efficiently with priority queues and is well-suited for sparse graphs.

On the other hand, the **Floyd–Warshall Algorithm** is a dynamic programming approach that computes shortest paths between all pairs of vertices. Though it has higher time complexity than Dijkstra's algorithm, it is very effective for dense graphs or when many shortest-path queries need to be answered.

Each algorithm is studied in terms of time and space complexity, and their behavior is empirically evaluated on graphs with varying properties. The aim is to understand their practical performance and how different graph characteristics impact their efficiency.

1.8 Comparison Metrics

To assess the performance of shortest path algorithms, we consider the following metrics:

- **Execution Time:** Measures the time required to compute shortest paths on graphs 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 shortest path algorithm scales with increasing graph sizes and densities.

1.9 Input Format

The evaluation explores how Dijkstra's and Floyd–Warshall algorithms perform across various **weighted** 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.
- **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.
- **Trees:** A special case of connected, acyclic graphs with exactly $n - 1$ edges for n nodes.
- **All graphs are weighted:** Each edge has a numerical weight representing cost, distance, or time.

These input variations ensure a thorough empirical analysis of shortest path algorithm behavior across diverse graph topologies.

Chapter 2

Implementation

This section outlines the implementation of the **Dijkstra** and **Floyd–Warshall** algorithms, along with the experimental setup designed to evaluate their performance across various weighted 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 weighted graphs with diverse characteristics—directed, undirected, sparse, dense, connected, disconnected, cyclic, acyclic, and tree-structured.
- **Performance Metrics:** Execution time and memory usage were recorded for each algorithm and graph configuration.

Shortest Path Algorithms Implementation

This section provides the core implementation of the two algorithms under analysis: Dijkstra's and Floyd–Warshall. Each algorithm is presented along with both a basic and optimized version, using adjacency list or matrix representations as appropriate.

Adjacency Matrix Representation

An **adjacency matrix** is a two-dimensional array used to represent a graph. Each cell indicates the presence and weight of an edge between two nodes.

- The entry at position `matrix[i][j]` contains the weight of the edge from node i to node j.
- If no edge exists, the entry is set to 0 or ∞ depending on the algorithm.
- This format is primarily used in the Floyd–Warshall algorithm.

Advantages

- **Fast Access:** Constant-time lookup for any edge.
- **Best for Dense Graphs:** Efficient storage and access when most edges exist.

Disadvantages

- **High Memory Usage:** Space complexity of $\mathcal{O}(n^2)$ even for sparse graphs.
- **Poor Scalability:** Inefficient for very large, sparse graphs.

Adjacency List Representation

An **adjacency list** represents a graph by listing, for each node, its directly connected neighbors along with corresponding edge weights.

- This format is used for the Dijkstra algorithm for space and time efficiency.
- It is implemented as a dictionary where each key maps to a list of $(neighbor, weight)$ tuples.

Advantages

- **Memory Efficient:** Suitable for sparse graphs with few edges per node.
- **Scalable:** Handles large graphs more efficiently in terms of memory.

Disadvantages

- **Slower Edge Lookup:** Requires iterating through a list to find a specific neighbor.
- **Less Ideal for Dense Graphs:** Matrix-based approaches can be faster in dense scenarios.

2.1 Dijkstra's Algorithm

Dijkstra's algorithm is a fundamental and widely used greedy algorithm for finding the shortest path from a single source node to all other nodes in a weighted graph with non-negative edge weights. Developed by Edsger W. Dijkstra in 1956, this algorithm guarantees optimal solutions for graphs without negative weights, making it suitable for a wide range of real-world applications such as navigation systems, network routing, and transportation planning.

The algorithm operates by continuously selecting the node with the smallest known distance from the source and updating the shortest paths to its adjacent nodes accordingly. This process continues until the shortest path to every reachable node has been determined.

Core Concepts

- A **distance table** is maintained, initially filled with ∞, except the source node which is set to 0.
- At each iteration, the algorithm selects the **unvisited node with the smallest distance**.
- For each unvisited neighbor, it checks whether the current path offers a shorter route and updates the distance accordingly.
- The algorithm repeats this process until all nodes are either visited or unreachable.

Advantages

- Provides optimal paths in graphs with non-negative weights.
- Efficient when implemented with a priority queue (heap).
- Performs well on sparse graphs.

Limitations

- Cannot handle graphs with negative edge weights.
- Less efficient for dense graphs without optimization.
- Needs modifications to work on dynamic graphs (e.g., real-time systems).

Basic Python Implementation

The basic implementation uses a linear search to select the next node with the smallest tentative distance.

```
def dijkstra(graph, start=None):
    adj_list = graph_to_adj_list_weighted(graph)
    if start is None:
        start = next((node for node in adj_list if adj_list[node
           ]), None)

    distances = {node: math.inf for node in adj_list}
    distances[start] = 0
    visited = set()

    while len(visited) < len(adj_list):
        u = min((node for node in adj_list if node not in visited
           ),
                key=lambda node: distances[node], default=None)
        if u is None or distances[u] == math.inf:
            break
        visited.add(u)

        for v, weight in adj_list[u]:
```

```
18                 if distances[u] + weight < distances[v]:
19                     distances[v] = distances[u] + weight
20
21     return distances
```

Listing 2.1: Basic Dijkstra Implementation

Optimized Version with Priority Queue

The optimized implementation of Dijkstra's algorithm uses a **min-heap** (also known as a priority queue), implemented via Python's `heapq` module, to improve the performance of selecting the next node with the smallest tentative distance.

In the basic implementation, this selection is done by scanning all unvisited nodes—an operation that takes $\mathcal{O}(n)$ time per iteration. In contrast, the heap-based approach reduces this operation to $\mathcal{O}(\log n)$, significantly enhancing efficiency, especially on larger graphs.

How the optimization works:

- A heap is initialized with the start node, prioritized by its current known shortest distance (initially 0).
- In each iteration, the node with the lowest distance is `heappop`'d from the queue.
- Its neighbors are evaluated, and if a shorter path to a neighbor is found, the neighbor is pushed back into the heap with the updated distance.
- The heap may contain multiple entries for the same node, but only the one with the smallest distance is processed, thanks to the distance check: `if current_distance > distances[u]: continue`.

Why it's better:

- Time complexity improves from $\mathcal{O}(n^2)$ (basic) to $\mathcal{O}((n+m)\log n)$, where m is the number of edges.
- This version is particularly well-suited for **sparse graphs**, where $m \ll n^2$.
- Priority queues are standard in many real-world implementations, including GPS navigation and real-time systems.

```
1  def dijkstra_optimized(graph, start=None):
2      adj_list = graph_to_adj_list_weighted(graph)
3      if start is None:
4          start = next((node for node in adj_list if adj_list[node
               ]), None)
5
6      distances = {node: math.inf for node in adj_list}
7      distances[start] = 0
8      heap = [(0, start)]
9
10     while heap:
11         current_distance, u = heapq.heappop(heap)
```

```
        if current_distance > distances[u]:
            continue

        for v, weight in adj_list.get(u, []):
            if v not in distances:
                continue
            distance = current_distance + weight
            if distance < distances[v]:
                distances[v] = distance
                heapq.heappush(heap, (distance, v))

    return distances
```

Listing 2.2: Optimized Dijkstra with Min-Heap

2.2 Floyd–Warshall Algorithm

The Floyd–Warshall algorithm is a dynamic programming-based solution for computing the shortest paths between **all pairs of nodes** in a weighted graph. Unlike Dijkstra's algorithm, which solves the single-source shortest path problem, Floyd–Warshall solves the *all-pairs* shortest path problem and is capable of handling graphs with both positive and negative edge weights—provided there are no negative cycles.

This algorithm is particularly well-suited for dense graphs or scenarios where multiple shortest path queries need to be answered efficiently after pre-processing.

Core Concepts

- The algorithm uses a **distance matrix** where `dist[i][j]` stores the current shortest path from node i to node j.
- At each iteration, an intermediate node k is considered. If a path from i to j through k is shorter than the current value, the matrix is updated.
- This process repeats for all combinations of i, j, and k, ensuring that the result reflects the shortest path considering any number of intermediate nodes.

Advantages

- Solves the all-pairs shortest path problem in one pass.
- Handles both positive and negative weights.
- Simple and elegant dynamic programming formulation.

Limitations

- Time complexity is $\mathcal{O}(n^3)$, making it inefficient for large sparse graphs.
- Cannot handle graphs with negative cycles.
- Requires an adjacency matrix, which consumes $\mathcal{O}(n^2)$ space.

Basic Python Implementation

This implementation directly applies the triple-nested loop strategy using a manually constructed distance matrix.

```
def floyd_warshall(graph):
    matrix, nodes = graph_to_adj_matrix(graph)
    n = len(matrix)
    dist = [[math.inf if i != j and matrix[i][j] == 0 else matrix
       [i][j]
             for j in range(n)] for i in range(n)]

    for k in range(n):
        for i in range(n):
            for j in range(n):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

Listing 2.3: Basic Floyd–Warshall Implementation

Optimized Version with NumPy

The optimized version of the Floyd–Warshall algorithm leverages the power of **NumPy**, a high-performance numerical computing library in Python, to replace the three nested loops with vectorized operations. This drastically reduces overhead and improves runtime performance by utilizing low-level, compiled C operations under the hood.

How the optimization works:

- The adjacency matrix is converted into a NumPy array with floating-point values to support `inf` and numerical comparisons.
- A vectorized distance matrix is initialized with:

$$\texttt{dist[i][j]} = \begin{cases} \texttt{matrix[i][j]}, & \text{if edge exists} \\ \infty, & \text{if no edge and } i \neq j \\ 0, & \text{if } i = j \end{cases}$$

- The innermost update loop is replaced by:

```
dist = np.minimum(dist, dist[:, k, None] + dist[k, :])
```

  This performs matrix broadcasting, updating all path combinations through node k in a single call.

Why it's better:

- **Broadcasting and vectorization** remove the explicit loop over i and j, replacing it with efficient native C-level operations.

- Though the algorithm's theoretical time complexity remains $\mathcal{O}(n^3)$, the practical performance improves significantly due to optimized memory access and SIMD instructions.
- This version is especially useful for **dense graphs** or **batch computation scenarios** in scientific applications where performance is critical.

```
def floyd_warshall_optimized(graph):
    matrix, nodes = graph_to_adj_matrix(graph)
    n = len(matrix)
    matrix = np.array(matrix, dtype=float)
    dist = np.where((matrix == 0) & (np.eye(n) == 0), np.inf,
       matrix)

    for k in range(n):
        dist = np.minimum(dist, dist[:, k, np.newaxis] + dist[k,
           :])

    return dist.tolist()
```

Listing 2.4: Optimized Floyd–Warshall with NumPy

2.3 Performance Evaluation

This section presents the empirical results of applying shortest path algorithms—**Dijkstra** and **Floyd–Warshall**—to various types of weighted graphs, including both directed and undirected configurations. The performance of each algorithm is assessed based on two primary metrics: **execution time** and **memory consumption**.

The results are presented graphically to facilitate comparison and interpretation of trends across different configurations. These visualizations provide insight into how algorithmic complexity and data structure choices affect real-world performance under diverse graph conditions.

2.3.1 Sparse vs Dense Weighted Graphs

This experiment focused on evaluating the performance of shortest path algorithms — Floyd–Warshall, Floyd–Warshall (NumPy-optimized), Dijkstra, and Optimized Dijkstra—across sparse and dense weighted graphs of increasing sizes (10 to 500 nodes). Both execution time and memory usage were recorded to compare scalability and efficiency under varying edge densities.

Empirical Results Table

Table 2.1: Execution Time and Memory Usage – Sparse vs Dense Weighted Graphs

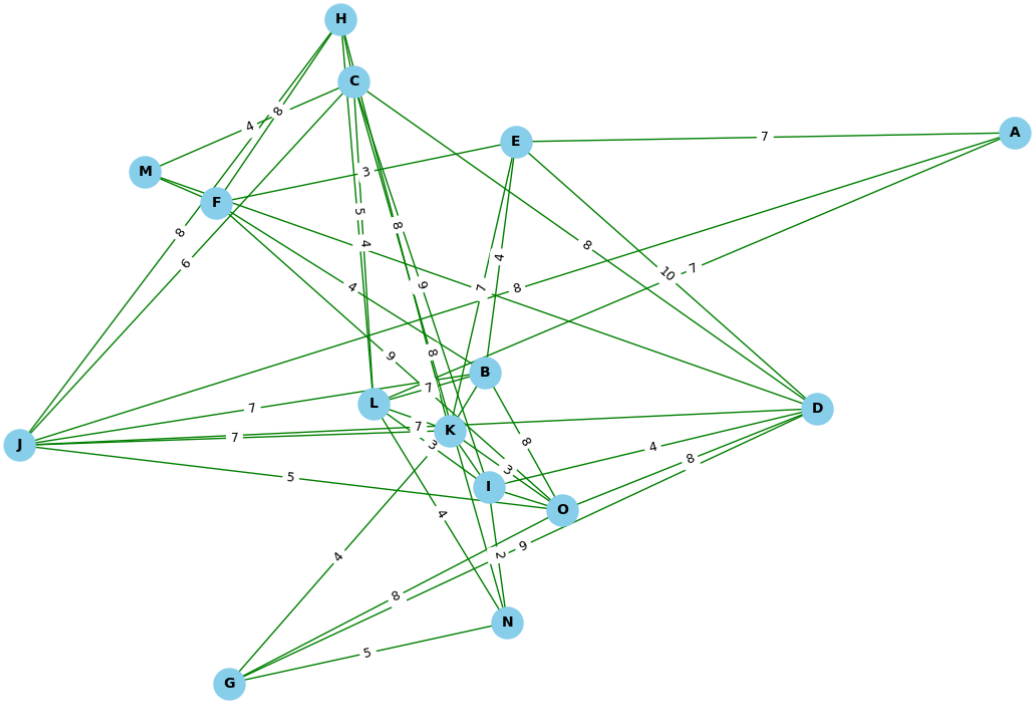
Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Floyd–Warshall	10	Sparse (Weighted)	0.000202	1,880	N/A
Floyd–Warshall	10	Dense (Weighted)	0.000433	3,176	N/A
Floyd–Warshall	15	Sparse (Weighted)	0.000973	4,712	N/A
Floyd–Warshall	15	Dense (Weighted)	0.001206	5,144	N/A
Floyd–Warshall	50	Sparse (Weighted)	0.016861	177,205	N/A
Floyd–Warshall	50	Dense (Weighted)	0.025003	193,983	N/A
Floyd–Warshall	75	Sparse (Weighted)	0.019728	43,768	N/A
Floyd–Warshall	75	Dense (Weighted)	0.067179	94,952	N/A
Floyd–Warshall	100	Sparse (Weighted)	0.064772	123,472	N/A
Floyd–Warshall	100	Dense (Weighted)	0.110377	175,960	N/A
Floyd–Warshall	250	Sparse (Weighted)	0.954164	811,600	N/A
Floyd–Warshall	250	Dense (Weighted)	1.643050	1,195,976	N/A
Floyd–Warshall	500	Sparse (Weighted)	18.382200	2,951,440	N/A
Floyd–Warshall	500	Dense (Weighted)	18.472000	4,283,128	N/A
Floyd–Warshall (NumPy)	10	Sparse (Weighted)	0.000163	6,984	N/A
Floyd–Warshall (NumPy)	10	Dense (Weighted)	0.000165	7,612	N/A
Floyd–Warshall (NumPy)	15	Sparse (Weighted)	0.000305	10,664	N/A
Floyd–Warshall (NumPy)	15	Dense (Weighted)	0.000312	12,136	N/A
Floyd–Warshall (NumPy)	50	Sparse (Weighted)	0.000774	79,928	N/A
Floyd–Warshall (NumPy)	50	Dense (Weighted)	0.001241	123,848	N/A
Floyd–Warshall (NumPy)	75	Sparse (Weighted)	0.001353	119,032	N/A
Floyd–Warshall (NumPy)	75	Dense (Weighted)	0.002492	268,424	N/A
Floyd–Warshall (NumPy)	100	Sparse (Weighted)	0.002708	336,960	N/A
Floyd–Warshall (NumPy)	100	Dense (Weighted)	0.004372	487,504	N/A
Floyd–Warshall (NumPy)	250	Sparse (Weighted)	0.020874	2,082,848	N/A
Floyd–Warshall (NumPy)	250	Dense (Weighted)	0.031186	3,094,168	N/A
Floyd–Warshall (NumPy)	500	Sparse (Weighted)	0.453363	7,747,692	N/A
Floyd–Warshall (NumPy)	500	Dense (Weighted)	0.861769	11,988,532	N/A
Dijkstra	10	Sparse (Weighted)	0.000085	2,168	N/A
Dijkstra	10	Dense (Weighted)	0.000130	2,520	N/A
Dijkstra	15	Sparse (Weighted)	0.000091	2,744	N/A
Dijkstra	15	Dense (Weighted)	0.000225	3,128	N/A
Dijkstra	50	Sparse (Weighted)	0.000750	6,648	N/A
Dijkstra	50	Dense (Weighted)	0.000834	10,344	N/A
Dijkstra	75	Sparse (Weighted)	0.000855	9,032	N/A
Dijkstra	75	Dense (Weighted)	0.001108	10,920	N/A
Dijkstra	100	Sparse (Weighted)	0.001505	17,960	N/A
Dijkstra	100	Dense (Weighted)	0.001997	26,480	N/A
Dijkstra	250	Sparse (Weighted)	0.006248	42,720	N/A
Dijkstra	250	Dense (Weighted)	0.010020	197,912	N/A
Dijkstra	500	Sparse (Weighted)	0.024877	109,832	N/A
Dijkstra	500	Dense (Weighted)	0.107380	159,632	N/A

Table 2.2: Execution Time and Memory Usage – Sparse vs Dense Weighted Graphs (Continuation)

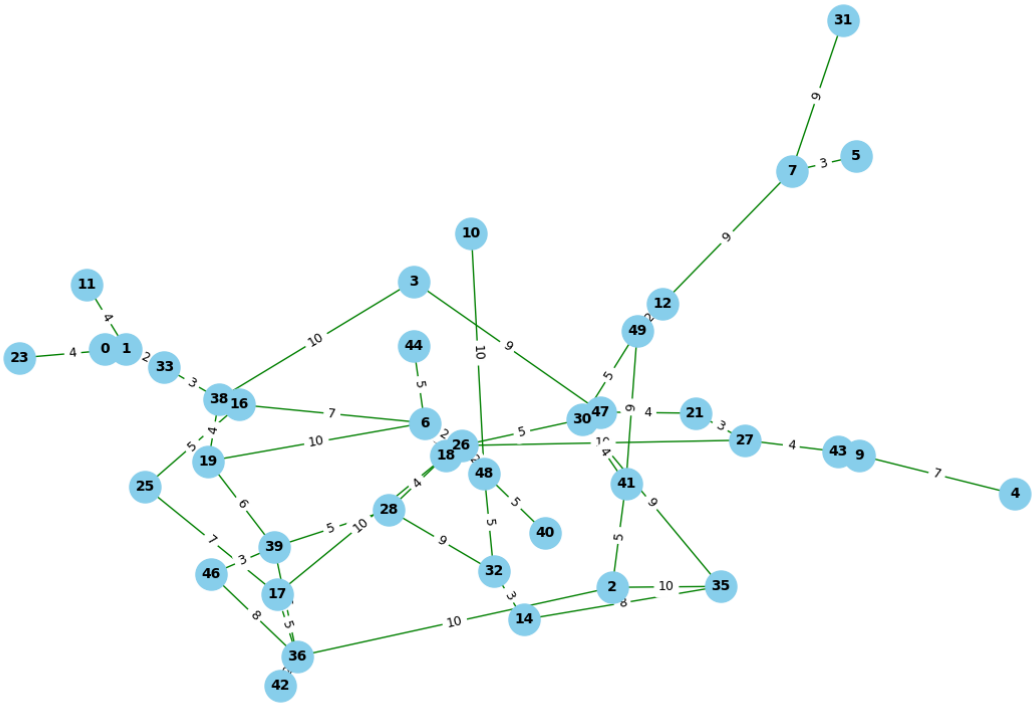
Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Dijkstra (Optimized)	10	Sparse (Weighted)	0.000040	1,480	N/A
Dijkstra (Optimized)	10	Dense (Weighted)	0.000108	1,832	N/A
Dijkstra (Optimized)	15	Sparse (Weighted)	0.000074	2,152	N/A
Dijkstra (Optimized)	15	Dense (Weighted)	0.000125	2,536	N/A
Dijkstra (Optimized)	50	Sparse (Weighted)	0.000095	4,720	N/A
Dijkstra (Optimized)	50	Dense (Weighted)	0.000367	9,048	N/A
Dijkstra (Optimized)	75	Sparse (Weighted)	0.000131	7,800	N/A
Dijkstra (Optimized)	75	Dense (Weighted)	0.000539	9,176	N/A
Dijkstra (Optimized)	100	Sparse (Weighted)	0.000336	8,736	N/A
Dijkstra (Optimized)	100	Dense (Weighted)	0.000763	18,672	N/A
Dijkstra (Optimized)	250	Sparse (Weighted)	0.000740	37,464	N/A
Dijkstra (Optimized)	250	Dense (Weighted)	0.001602	45,176	N/A
Dijkstra (Optimized)	500	Sparse (Weighted)	0.001457	78,376	N/A
Dijkstra (Optimized)	500	Dense (Weighted)	0.008562	288,427	N/A

Visual Graph Examples

Figure 2.1 shows two representative weighted graphs used during the experiment: one sparse and one dense.



(a) Dense Weighted Graph with 15 Nodes



(b) Sparse Weighted Graph with 50 Nodes

Figure 2.1: Example of Sparse vs Dense Weighted Graphs

Graphical Analysis of Algorithm Performance

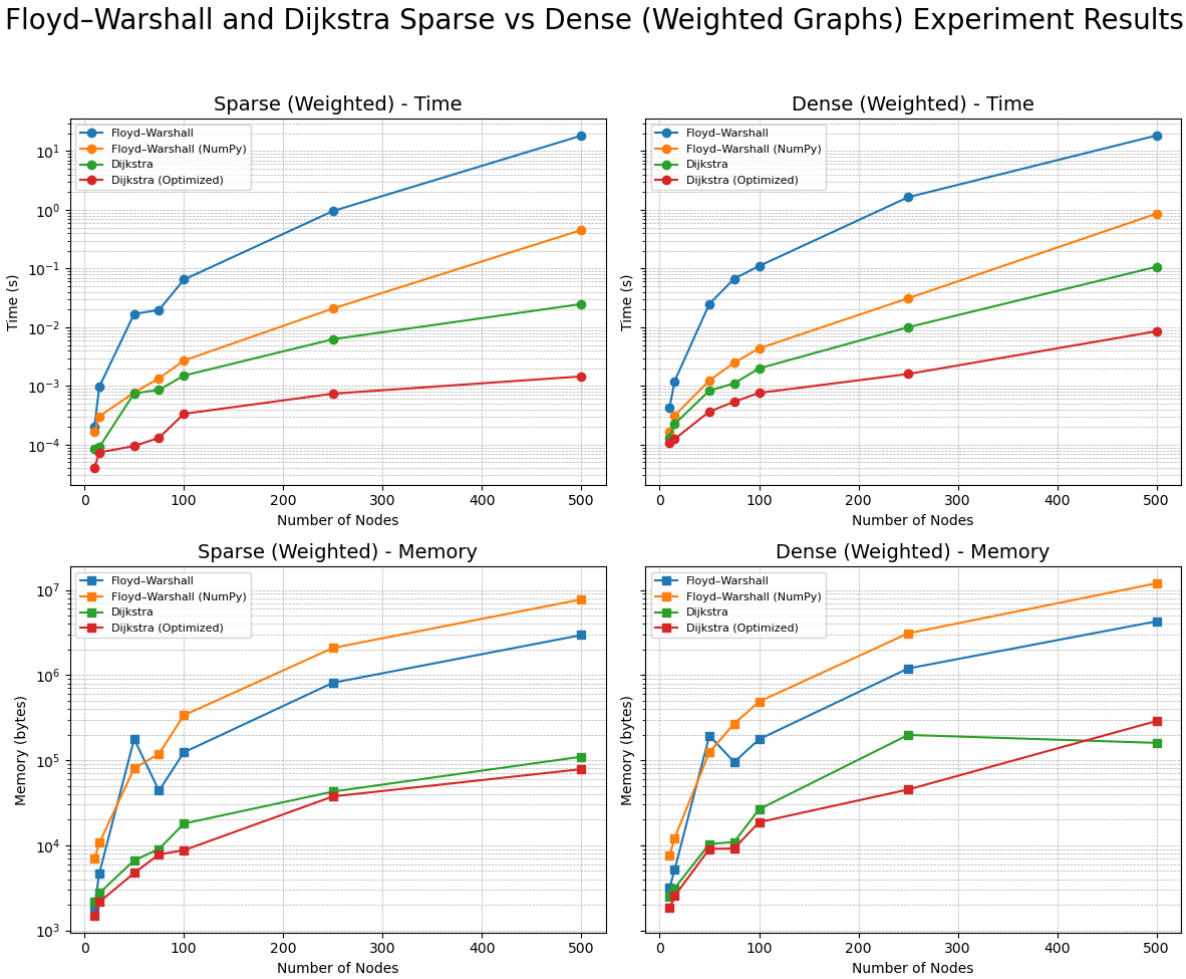


Figure 2.2: Sparse vs Dense: Time and Memory Analysis of Floyd–Warshall and Dijkstra Variants

Graph Density Impact (Sparse vs Dense): Graph density significantly impacts both execution time and memory usage. Dense graphs introduce more edges, increasing the complexity of shortest path computations—especially for all-pairs algorithms like Floyd–Warshall. In contrast, sparse graphs generally benefit Dijkstra's algorithm due to fewer edge relaxations. Notably, optimized Dijkstra consistently outperforms others in both speed and memory across all densities.

Key Observations

- Floyd–Warshall algorithms scale poorly in dense graphs due to cubic time complexity, despite NumPy optimizations.
- Dijkstra (Optimized) was consistently the most efficient in both time and memory, particularly in sparse graphs.
- Memory consumption grows significantly faster in dense graphs, particularly for matrix-based implementations.

- NumPy-based Floyd–Warshall reduced time but at the cost of high memory usage.

2.3.2 Directed vs Undirected Weighted Graphs

This experiment evaluates how graph orientation affects the performance of two shortest path algorithms: Floyd–Warshall (including a NumPy-optimized variant) and Dijkstra (including an optimized implementation). We analyze execution time and memory usage on both directed and undirected **weighted graphs** of increasing size, from 10 to 500 nodes.

Empirical Results Table

Table 2.3: Execution Time and Memory Usage – Directed vs Undirected Weighted Graphs

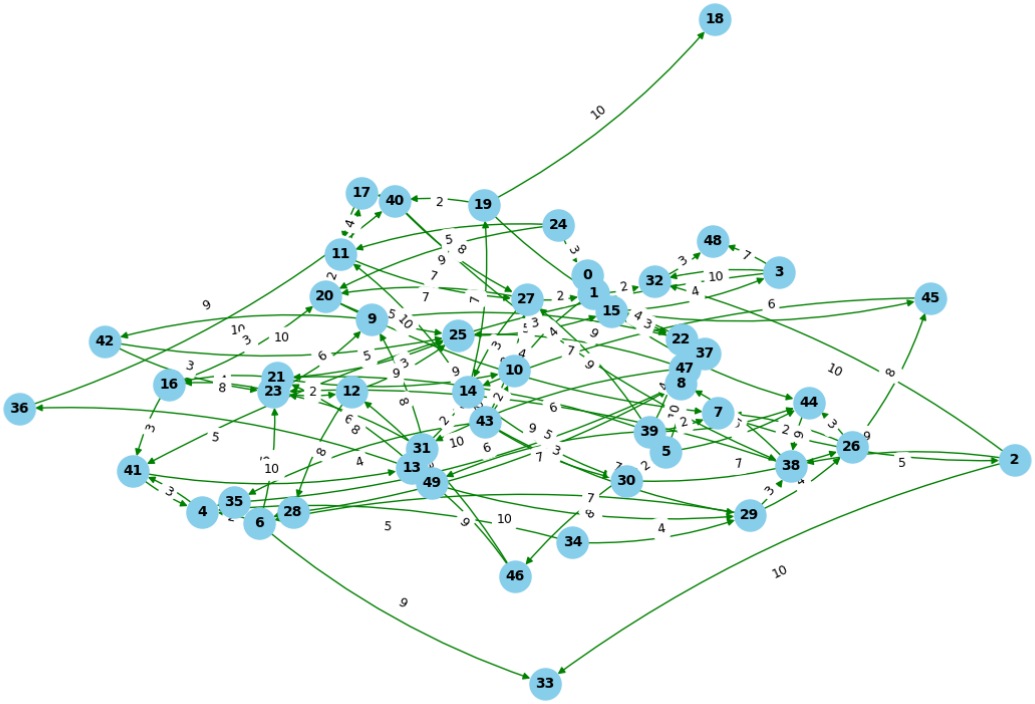
Algorithm	**Nodes**	**Graph Type**	**Time (s)**	**Memory (B)**	**Result**
Floyd–Warshall	10	Undirected	0.00013	1,880	N/A
Floyd–Warshall	10	Directed	0.00038	3,176	N/A
Floyd–Warshall	15	Undirected	0.000637	4,712	N/A
Floyd–Warshall	15	Directed	0.000725	5,144	N/A
Floyd–Warshall	50	Undirected	0.008305	28,664	N/A
Floyd–Warshall	50	Directed	0.013564	45,032	N/A
Floyd–Warshall	75	Undirected	0.017533	192,094	N/A
Floyd–Warshall	75	Directed	0.039749	241,184	N/A
Floyd–Warshall	100	Undirected	0.042886	123,472	N/A
Floyd–Warshall	100	Directed	0.078794	175,960	N/A
Floyd–Warshall	250	Undirected	0.838781	665,044	N/A
Floyd–Warshall	250	Directed	1.49569	1,024,684	N/A
Floyd–Warshall	500	Undirected	6.80327	2,951,184	N/A
Floyd–Warshall	500	Directed	26.7981	4,173,272	N/A
Floyd–Warshall (NumPy)	10	Undirected	0.000191	6,984	N/A
Floyd–Warshall (NumPy)	10	Directed	0.000196	7,612	N/A
Floyd–Warshall (NumPy)	15	Undirected	0.000307	10,664	N/A
Floyd–Warshall (NumPy)	15	Directed	0.000264	12,136	N/A
Floyd–Warshall (NumPy)	50	Undirected	0.001276	79,928	N/A
Floyd–Warshall (NumPy)	50	Directed	0.001317	123,848	N/A
Floyd–Warshall (NumPy)	75	Undirected	0.000952	119,032	N/A
Floyd–Warshall (NumPy)	75	Directed	0.001937	261,304	N/A
Floyd–Warshall (NumPy)	100	Undirected	0.003122	336,960	N/A
Floyd–Warshall (NumPy)	100	Directed	0.004106	487,504	N/A
Floyd–Warshall (NumPy)	250	Undirected	0.017993	2,082,872	N/A
Floyd–Warshall (NumPy)	250	Directed	0.036267	2,999,800	N/A
Floyd–Warshall (NumPy)	500	Undirected	0.472447	7,747,692	N/A
Floyd–Warshall (NumPy)	500	Directed	0.858460	11,560,628	N/A

Table 2.4: Execution Time and Memory Usage – Directed vs Undirected Weighted Graphs (Continuation)

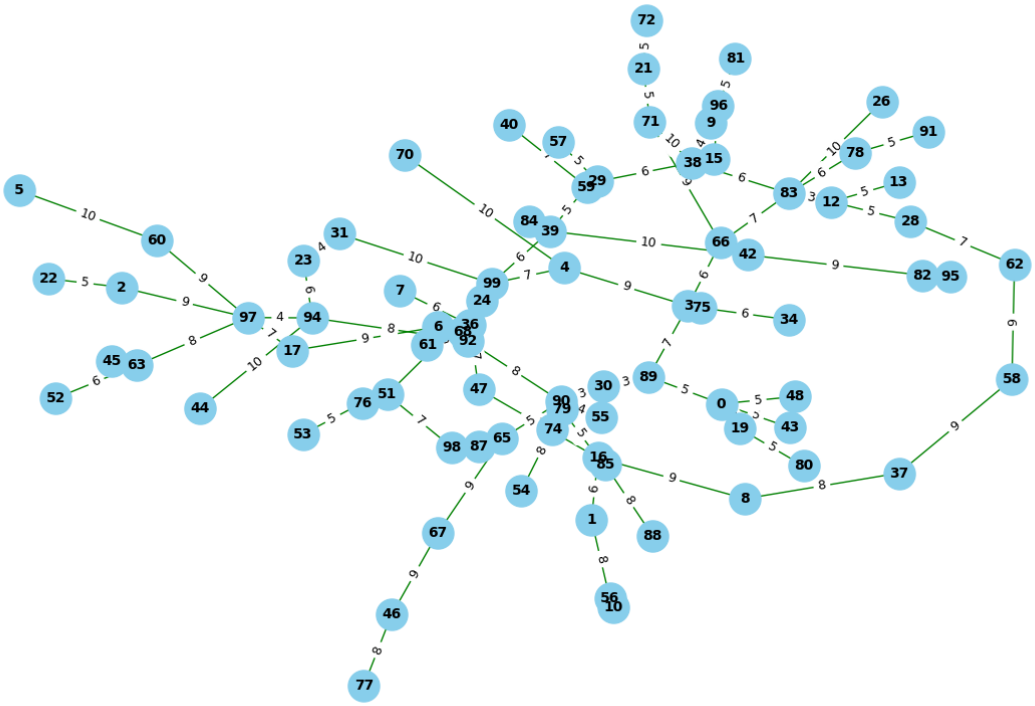
Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Dijkstra	10	Undirected	0.00006	2,168	N/A
Dijkstra	10	Directed	0.000107	2,264	N/A
Dijkstra	15	Undirected	0.000113	2,744	N/A
Dijkstra	15	Directed	0.000066	2,200	N/A
Dijkstra	50	Undirected	0.000556	6,648	N/A
Dijkstra	50	Directed	0.001041	8,968	N/A
Dijkstra	75	Undirected	0.000741	9,032	N/A
Dijkstra	75	Directed	0.001700	9,416	N/A
Dijkstra	100	Undirected	0.001313	17,960	N/A
Dijkstra	100	Directed	0.001843	24,048	N/A
Dijkstra	250	Undirected	0.007918	42,720	N/A
Dijkstra	250	Directed	0.010390	192,694	N/A
Dijkstra	500	Undirected	0.036178	258,947	N/A
Dijkstra	500	Directed	0.040839	263,696	N/A
Dijkstra (Optimized)	10	Undirected	0.00005	1,480	N/A
Dijkstra (Optimized)	10	Directed	0.000074	1,536	N/A
Dijkstra (Optimized)	15	Undirected	0.000075	2,152	N/A
Dijkstra (Optimized)	15	Directed	0.000065	1,824	N/A
Dijkstra (Optimized)	50	Undirected	0.000184	4,720	N/A
Dijkstra (Optimized)	50	Directed	0.000181	7,152	N/A
Dijkstra (Optimized)	75	Undirected	0.000205	7,800	N/A
Dijkstra (Optimized)	75	Directed	0.000329	7,600	N/A
Dijkstra (Optimized)	100	Undirected	0.000295	8,736	N/A
Dijkstra (Optimized)	100	Directed	0.000454	15,784	N/A
Dijkstra (Optimized)	250	Undirected	0.000707	181,244	N/A
Dijkstra (Optimized)	250	Directed	0.001201	39,672	N/A
Dijkstra (Optimized)	500	Undirected	0.002166	79,312	N/A
Dijkstra (Optimized)	500	Directed	0.003233	83,768	N/A

Visual Graph Examples

Figure 2.3 presents example graphs from this experiment.



(a) Directed Weighted Graph with 50 Nodes



(b) Undirected Weighted Graph with 100 Nodes

Figure 2.3: Example of Directed vs Undirected Weighted Graph Structures

Graphical Analysis of Performance

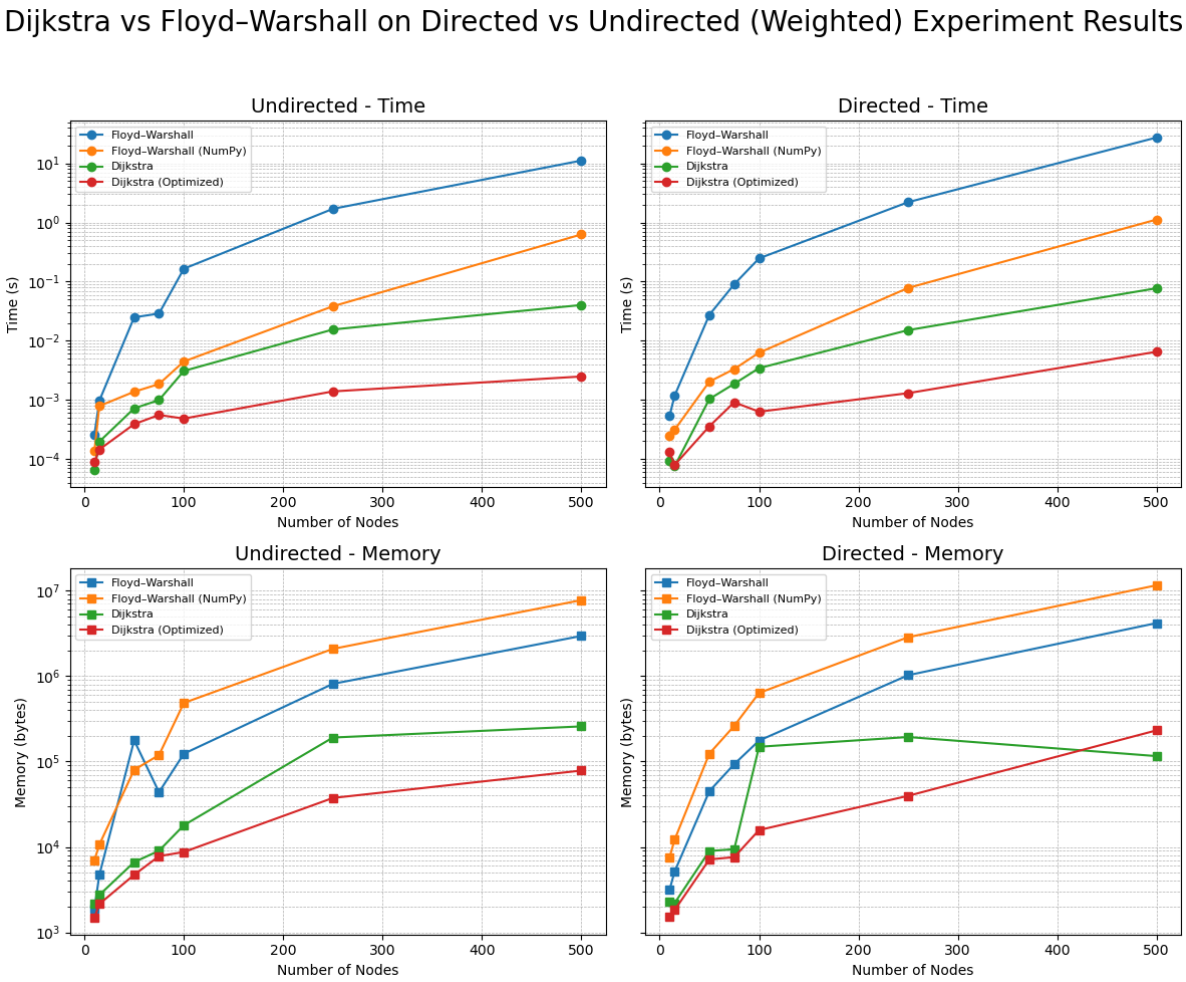


Figure 2.4: Performance Comparison on Directed vs Undirected Weighted Graphs

Graph Orientation Impact: The choice between directed and undirected graphs affects both time and memory usage across algorithms. Directed graphs often lead to longer paths or unreachable nodes, increasing computational cost. This is particularly evident in Floyd–Warshall (both versions). Dijkstra variants, especially the optimized one, maintain strong performance on both orientations.

Key Observations

- Floyd–Warshall (standard and NumPy) incurs high cost on directed graphs due to exhaustive all-pairs comparisons.
- NumPy-based Floyd–Warshall slightly reduces time but consumes significantly more memory.
- Dijkstra (Optimized) consistently delivers the best time/memory efficiency, regardless of orientation.
- Undirected graphs offer simpler connectivity, enabling slightly faster traversal and less memory overhead.

2.3.3 Cyclic vs Acyclic Weighted Graphs

This experiment compares the performance of Floyd–Warshall and Dijkstra (including NumPy and optimized variants) on **cyclic vs acyclic weighted graphs**. The goal is to evaluate how the presence or absence of cycles influences algorithm efficiency in terms of execution time and memory usage as graph sizes increase.

Empirical Results Table

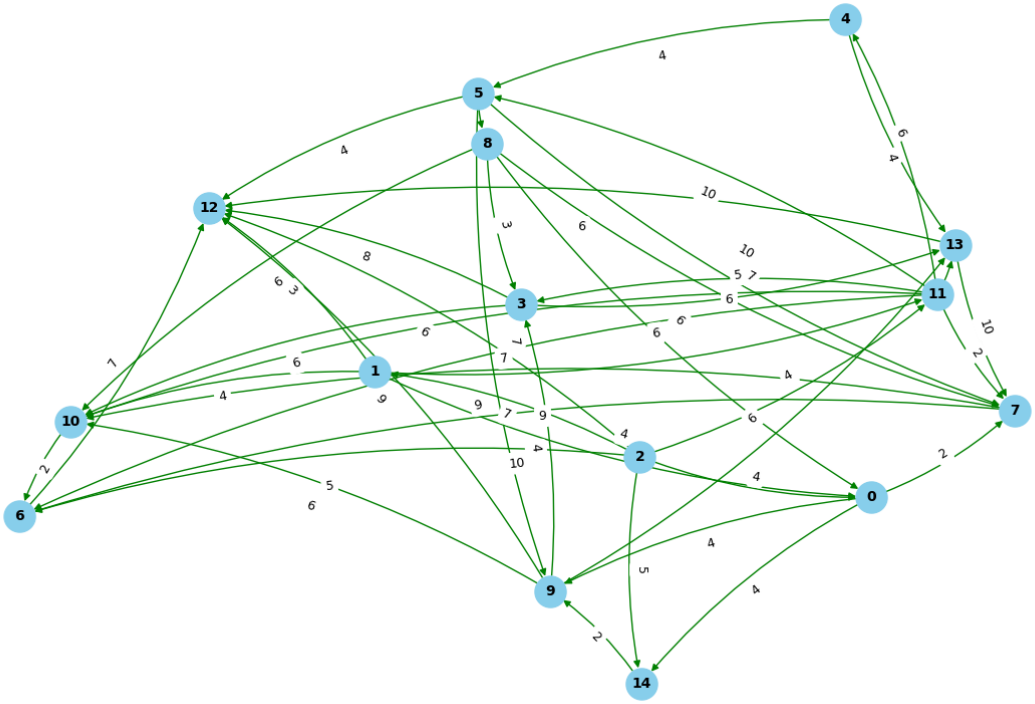
Table 2.5: Execution Time and Memory Usage – Cyclic vs Acyclic Weighted Graphs

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Floyd–Warshall	10	Acyclic (Weighted)	0.000246	3,176	N/A
Floyd–Warshall	10	Cyclic (Weighted)	0.000268	2,896	N/A
Floyd–Warshall	15	Acyclic (Weighted)	0.000630	5,144	N/A
Floyd–Warshall	15	Cyclic (Weighted)	0.000692	5,144	N/A
Floyd–Warshall	50	Acyclic (Weighted)	0.012935	45,032	N/A
Floyd–Warshall	50	Cyclic (Weighted)	0.012829	193,152	N/A
Floyd–Warshall	75	Acyclic (Weighted)	0.046940	96,656	N/A
Floyd–Warshall	75	Cyclic (Weighted)	0.053421	92,832	N/A
Floyd–Warshall	100	Acyclic (Weighted)	0.111871	177,988	N/A
Floyd–Warshall	100	Cyclic (Weighted)	0.080778	175,960	N/A
Floyd–Warshall	250	Acyclic (Weighted)	1.506730	1,068,132	N/A
Floyd–Warshall	250	Cyclic (Weighted)	1.325260	1,022,672	N/A
Floyd–Warshall	500	Acyclic (Weighted)	13.277200	4,295,248	N/A
Floyd–Warshall	500	Cyclic (Weighted)	13.553800	4,173,448	N/A
Floyd–Warshall (NumPy)	10	Acyclic (Weighted)	0.000212	7,612	N/A
Floyd–Warshall (NumPy)	10	Cyclic (Weighted)	0.000245	7,612	N/A
Floyd–Warshall (NumPy)	15	Acyclic (Weighted)	0.000336	12,136	N/A
Floyd–Warshall (NumPy)	15	Cyclic (Weighted)	0.000302	12,136	N/A
Floyd–Warshall (NumPy)	50	Acyclic (Weighted)	0.001447	123,848	N/A
Floyd–Warshall (NumPy)	50	Cyclic (Weighted)	0.001471	123,848	N/A
Floyd–Warshall (NumPy)	75	Acyclic (Weighted)	0.002754	275,880	N/A
Floyd–Warshall (NumPy)	75	Cyclic (Weighted)	0.005043	261,304	N/A
Floyd–Warshall (NumPy)	100	Acyclic (Weighted)	0.005308	635,232	N/A
Floyd–Warshall (NumPy)	100	Cyclic (Weighted)	0.005127	487,504	N/A
Floyd–Warshall (NumPy)	250	Acyclic (Weighted)	0.047777	3,018,304	N/A
Floyd–Warshall (NumPy)	250	Cyclic (Weighted)	0.045032	2,852,160	N/A
Floyd–Warshall (NumPy)	500	Acyclic (Weighted)	1.299670	12,036,548	N/A
Floyd–Warshall (NumPy)	500	Cyclic (Weighted)	1.145650	11,560,444	N/A

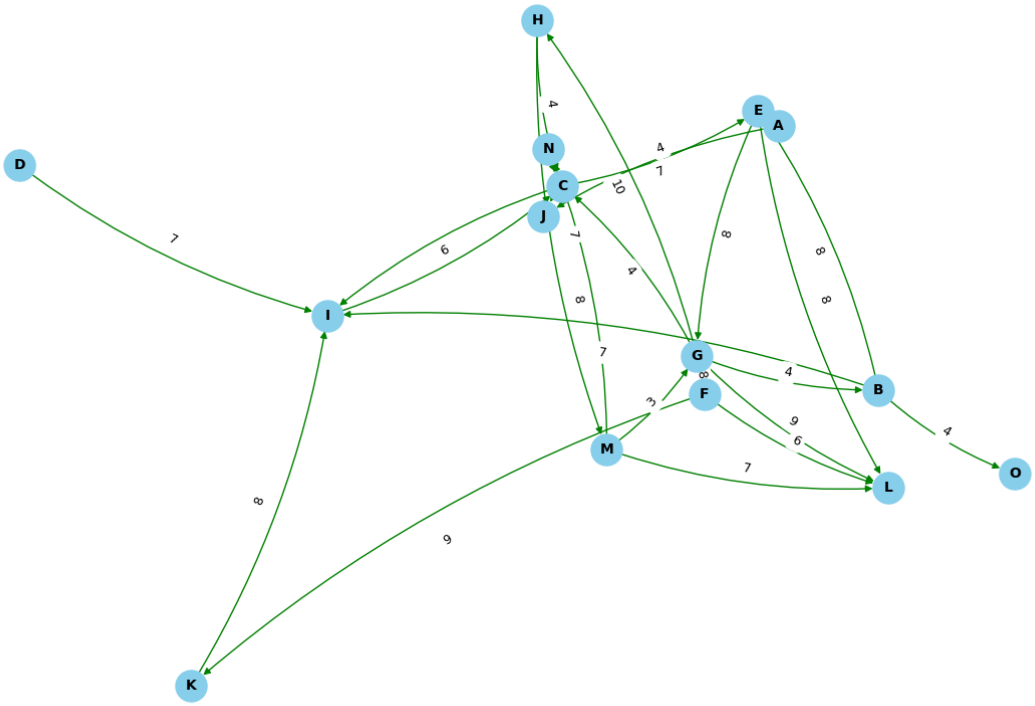
Table 2.6: Execution Time and Memory Usage – Cyclic vs Acyclic Weighted Graphs (Continuation)

Algorithm	**Nodes**	**Graph Type**	**Time (s)**	**Memory (B)**	**Result**
Dijkstra	10	Acyclic (Weighted)	0.000189	2,424	N/A
Dijkstra	10	Cyclic (Weighted)	0.000191	2,264	N/A
Dijkstra	15	Acyclic (Weighted)	0.000138	3,144	N/A
Dijkstra	15	Cyclic (Weighted)	0.000148	2,200	N/A
Dijkstra	50	Acyclic (Weighted)	0.000863	9,352	N/A
Dijkstra	50	Cyclic (Weighted)	0.000931	8,968	N/A
Dijkstra	75	Acyclic (Weighted)	0.000349	8,768	N/A
Dijkstra	75	Cyclic (Weighted)	0.001403	9,416	N/A
Dijkstra	100	Acyclic (Weighted)	0.002297	17,208	N/A
Dijkstra	100	Cyclic (Weighted)	0.002494	24,048	N/A
Dijkstra	250	Acyclic (Weighted)	0.004975	42,608	N/A
Dijkstra	250	Cyclic (Weighted)	0.010009	45,512	N/A
Dijkstra	500	Acyclic (Weighted)	0.003248	89,208	N/A
Dijkstra	500	Cyclic (Weighted)	0.057157	115,712	N/A
Dijkstra (Optimized)	10	Acyclic (Weighted)	0.000031	1,648	N/A
Dijkstra (Optimized)	10	Cyclic (Weighted)	0.000073	1,536	N/A
Dijkstra (Optimized)	15	Acyclic (Weighted)	0.000050	2,056	N/A
Dijkstra (Optimized)	15	Cyclic (Weighted)	0.000039	1,824	N/A
Dijkstra (Optimized)	50	Acyclic (Weighted)	0.000209	7,536	N/A
Dijkstra (Optimized)	50	Cyclic (Weighted)	0.000213	7,152	N/A
Dijkstra (Optimized)	75	Acyclic (Weighted)	0.000409	8,688	N/A
Dijkstra (Optimized)	75	Cyclic (Weighted)	0.000269	7,600	N/A
Dijkstra (Optimized)	100	Acyclic (Weighted)	0.000491	16,488	N/A
Dijkstra (Optimized)	100	Cyclic (Weighted)	0.000347	15,784	N/A
Dijkstra (Optimized)	250	Acyclic (Weighted)	0.001020	42,528	N/A
Dijkstra (Optimized)	250	Cyclic (Weighted)	0.000691	39,672	N/A
Dijkstra (Optimized)	500	Acyclic (Weighted)	0.002453	89,128	N/A
Dijkstra (Optimized)	500	Cyclic (Weighted)	0.001724	83,768	N/A

Visual Graph Examples



(a) Acyclic Weighted Graph with 15 Nodes



(b) Cyclic Weighted Graph with 15 Nodes

Figure 2.5: Example of Acyclic vs Cyclic Weighted Graph Structures

Graphical Performance Comparison

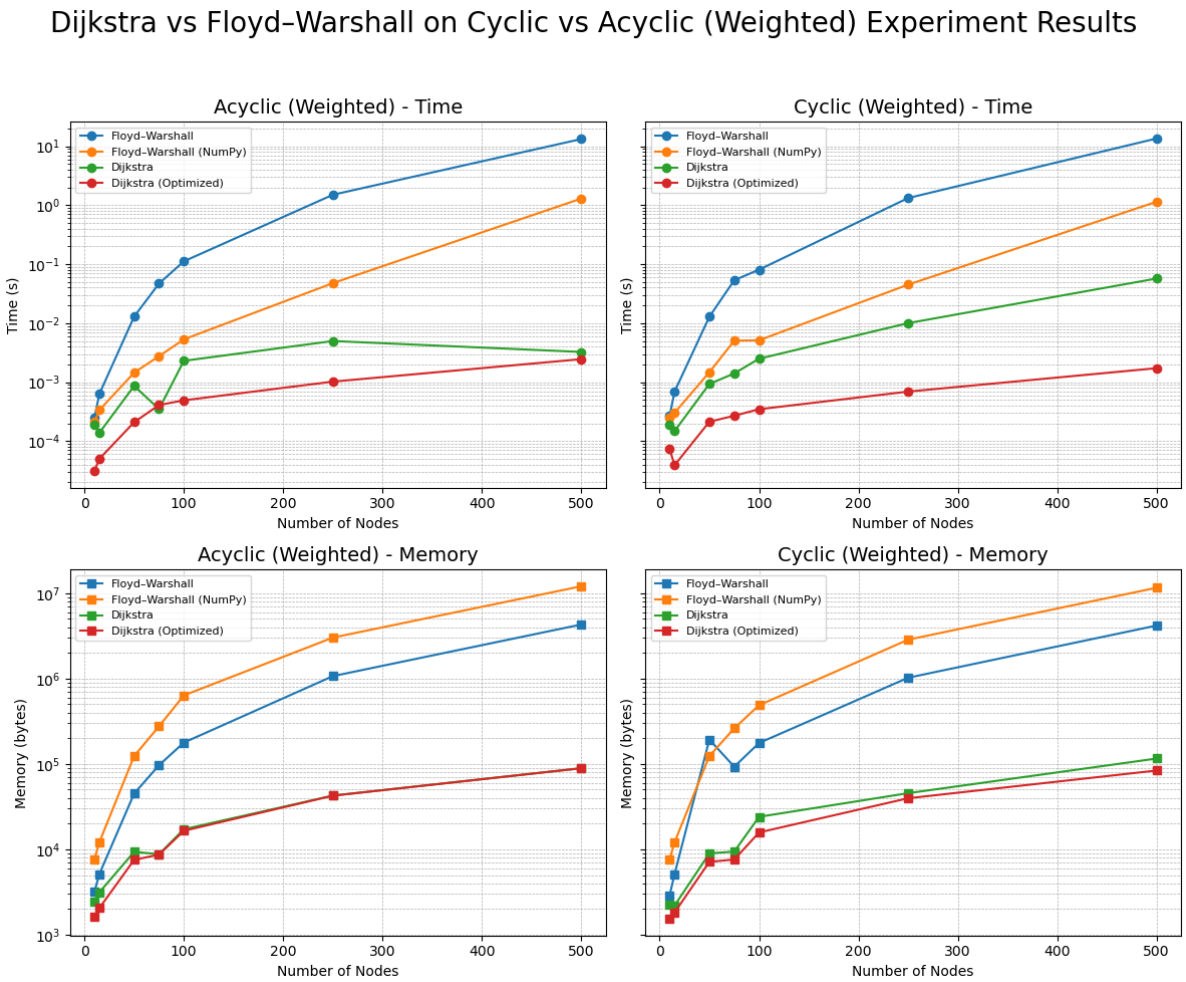


Figure 2.6: Performance Comparison on Cyclic vs Acyclic Weighted Graphs

Effect of Cycles in Graph Structure: Cycles generally introduce more potential paths and complexity in traversal. However, for dense enough graphs, the performance impact becomes marginal. Dijkstra's performance is slightly better on acyclic graphs due to fewer redundant paths. In contrast, Floyd–Warshall performs consistently across both structures due to its exhaustive all-pairs comparison nature.

Key Observations

- Cycle presence has a more noticeable effect on Dijkstra (especially non-optimized).
- Floyd–Warshall's memory usage remained similar across both cyclic and acyclic graphs.
- Optimized Dijkstra again proved the most efficient regardless of structure.
- Acyclic graphs yielded slightly better performance due to simpler topologies.

2.3.4 Connected vs Disconnected Weighted Graphs

In this experiment, we analyze the impact of graph connectivity on the performance of Dijkstra's and Floyd–Warshall's algorithms (including optimized and NumPy-based variants) on weighted graphs. The comparison is based on execution time and memory usage over graph sizes ranging from 10 to 500 nodes.

Empirical Results Table

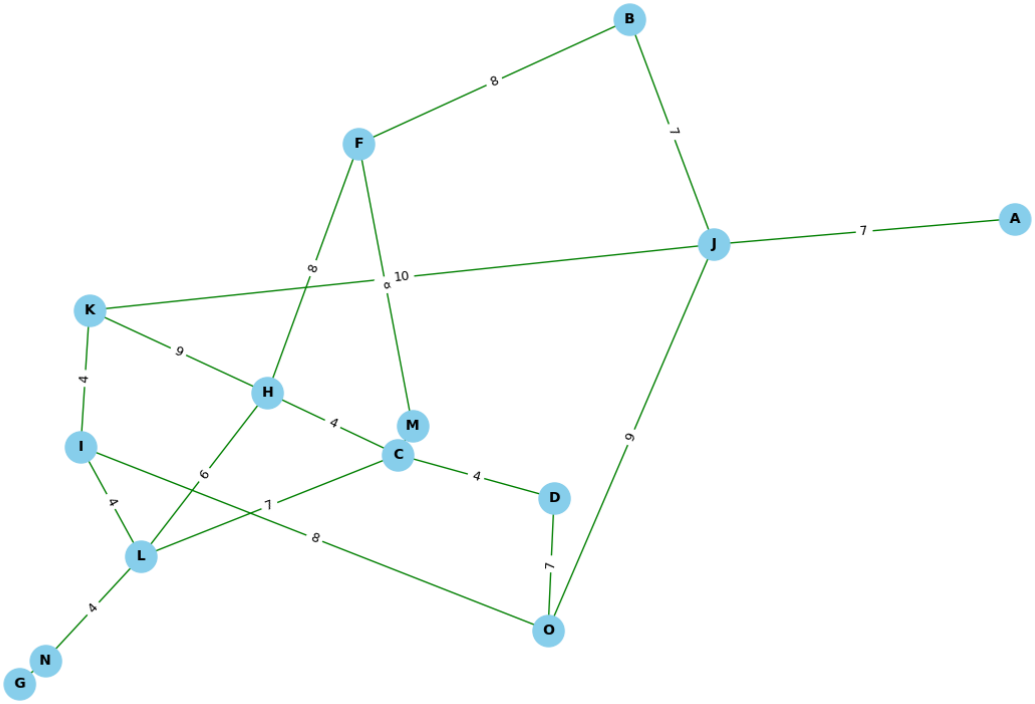
Table 2.7: Execution Time and Memory Usage – Connected vs Disconnected Weighted Graphs

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Floyd–Warshall	10	Connected	0.000208	1,880	N/A
Floyd–Warshall	10	Disconnected	0.000390	3,176	N/A
Floyd–Warshall	15	Connected	0.001049	4,712	N/A
Floyd–Warshall	15	Disconnected	0.000775	5,144	N/A
Floyd–Warshall	50	Connected	0.010971	29,136	N/A
Floyd–Warshall	50	Disconnected	0.014016	193,200	N/A
Floyd–Warshall	75	Connected	0.015272	191,904	N/A
Floyd–Warshall	75	Disconnected	0.054114	96,656	N/A
Floyd–Warshall	100	Connected	0.070524	123,472	N/A
Floyd–Warshall	100	Disconnected	0.122480	175,960	N/A
Floyd–Warshall	250	Connected	0.972929	663,032	N/A
Floyd–Warshall	250	Disconnected	1.854440	1,068,132	N/A
Floyd–Warshall	500	Connected	8.061010	2,951,128	N/A
Floyd–Warshall	500	Disconnected	16.299400	4,295,392	N/A
Floyd–Warshall (NumPy)	10	Connected	0.000176	6,984	N/A
Floyd–Warshall (NumPy)	10	Disconnected	0.000210	7,612	N/A
Floyd–Warshall (NumPy)	15	Connected	0.000248	10,664	N/A
Floyd–Warshall (NumPy)	15	Disconnected	0.000283	12,136	N/A
Floyd–Warshall (NumPy)	50	Connected	0.001008	79,928	N/A
Floyd–Warshall (NumPy)	50	Disconnected	0.001615	123,848	N/A
Floyd–Warshall (NumPy)	75	Connected	0.001097	119,032	N/A
Floyd–Warshall (NumPy)	75	Disconnected	0.002516	275,320	N/A
Floyd–Warshall (NumPy)	100	Connected	0.003455	336,960	N/A
Floyd–Warshall (NumPy)	100	Disconnected	0.006157	635,016	N/A
Floyd–Warshall (NumPy)	250	Connected	0.036444	2,082,288	N/A
Floyd–Warshall (NumPy)	250	Disconnected	0.062978	3,018,304	N/A
Floyd–Warshall (NumPy)	500	Connected	0.818622	7,747,692	N/A
Floyd–Warshall (NumPy)	500	Disconnected	1.635220	12,185,004	N/A

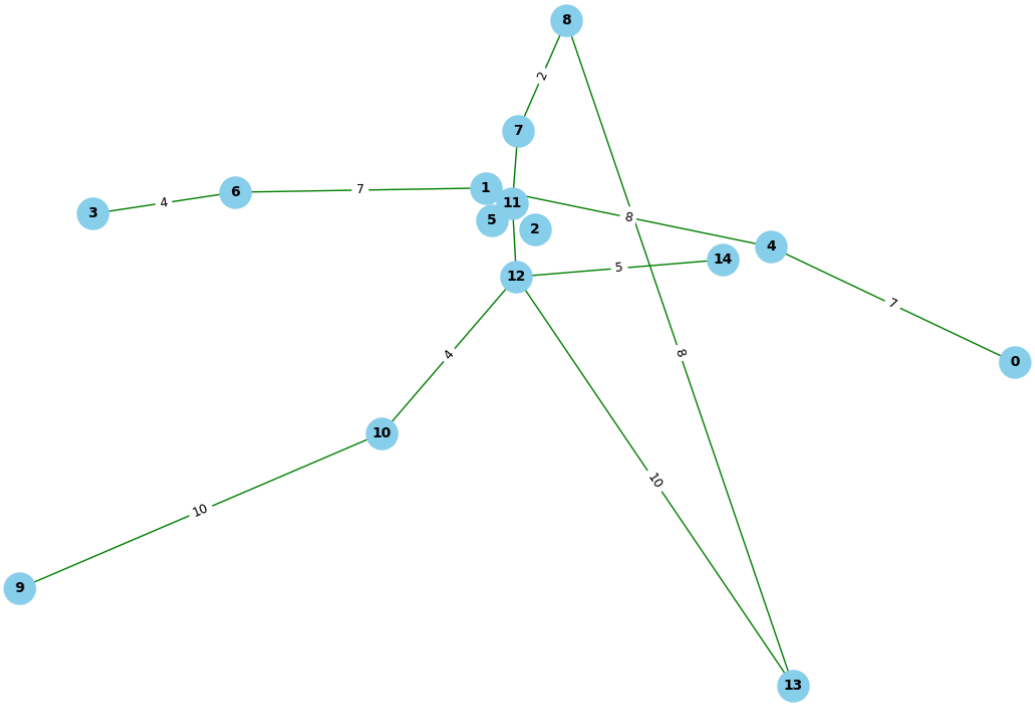
Table 2.8: Execution Time and Memory Usage – Connected vs Disconnected Graphs (Continuation)

Algorithm	**Nodes**	**Graph Type**	**Time (s)**	**Memory (B)**	**Result**
Dijkstra	10	Connected	0.000092	2,168	N/A
Dijkstra	10	Disconnected	0.000100	2,392	N/A
Dijkstra	15	Connected	0.000156	2,744	N/A
Dijkstra	15	Disconnected	0.000169	3,048	N/A
Dijkstra	50	Connected	0.000935	6,648	N/A
Dijkstra	50	Disconnected	0.000218	7,848	N/A
Dijkstra	75	Connected	0.000932	9,032	N/A
Dijkstra	75	Disconnected	0.001097	9,544	N/A
Dijkstra	100	Connected	0.002382	165,909	N/A
Dijkstra	100	Disconnected	0.002639	16,536	N/A
Dijkstra	250	Connected	0.013257	42,720	N/A
Dijkstra	250	Disconnected	0.009829	45,392	N/A
Dijkstra	500	Connected	0.044583	109,832	N/A
Dijkstra	500	Disconnected	0.042755	85,680	N/A
Dijkstra (Optimized)	10	Connected	0.000064	1,480	N/A
Dijkstra (Optimized)	10	Disconnected	0.000060	1,704	N/A
Dijkstra (Optimized)	15	Connected	0.000092	2,152	N/A
Dijkstra (Optimized)	15	Disconnected	0.000063	2,504	N/A
Dijkstra (Optimized)	50	Connected	0.000224	4,720	N/A
Dijkstra (Optimized)	50	Disconnected	0.000115	7,768	N/A
Dijkstra (Optimized)	75	Connected	0.000180	7,800	N/A
Dijkstra (Optimized)	75	Disconnected	0.000262	7,728	N/A
Dijkstra (Optimized)	100	Connected	0.000448	8,736	N/A
Dijkstra (Optimized)	100	Disconnected	0.000479	16,400	N/A
Dijkstra (Optimized)	250	Connected	0.001417	38,447	N/A
Dijkstra (Optimized)	250	Disconnected	0.000886	39,552	N/A
Dijkstra (Optimized)	500	Connected	0.002739	78,376	N/A
Dijkstra (Optimized)	500	Disconnected	0.001808	84,456	N/A

Visual Graph Examples



(a) Connected Weighted Graph with 15 Nodes



(b) Disconnected Weighted Graph with 15 Nodes

Figure 2.7: Example of Connected vs Disconnected Graph Structures

Performance Comparison Plot

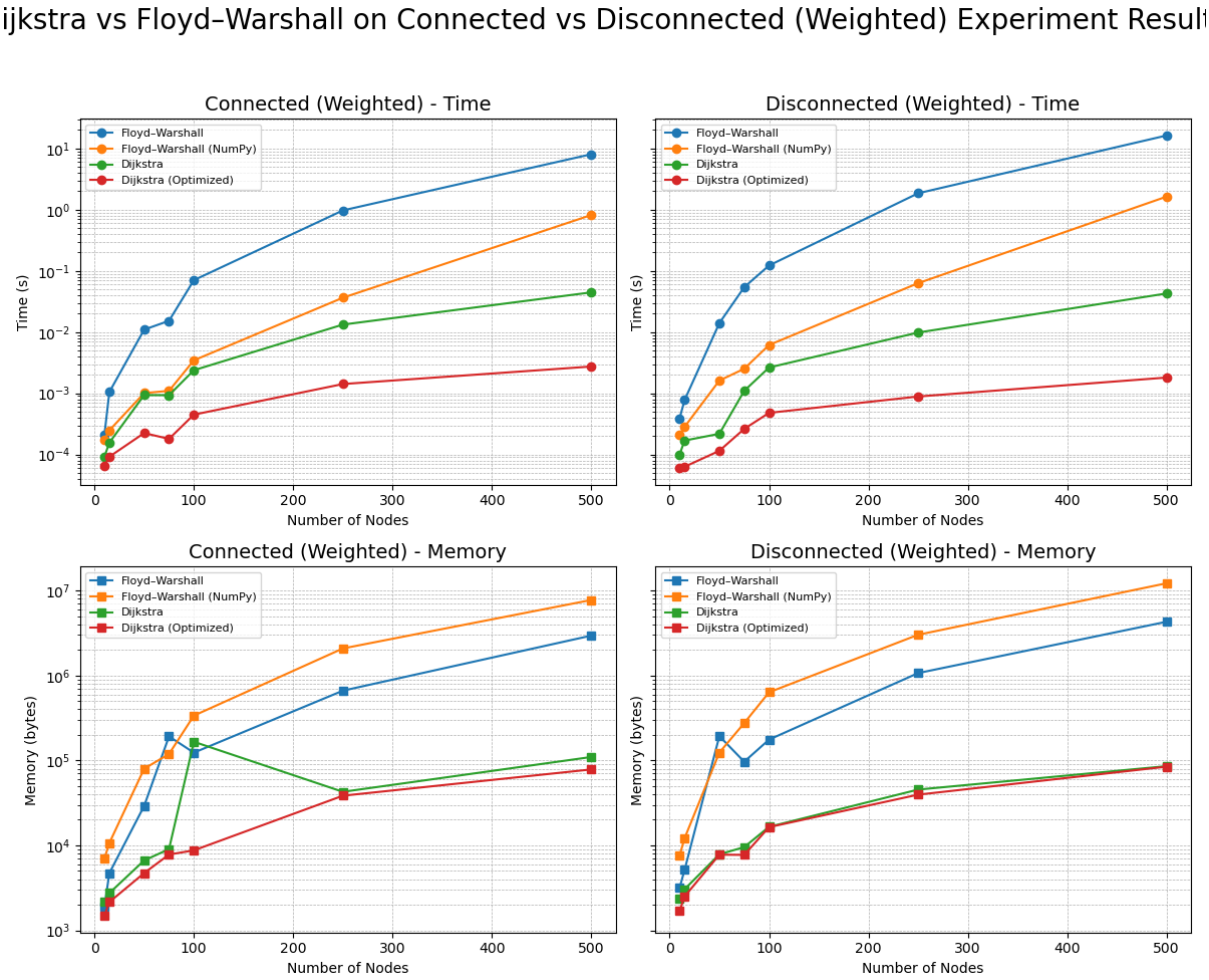


Figure 2.8: Performance Comparison on Connected vs Disconnected Weighted Graphs

Effect of Connectivity: Disconnected graphs tend to increase execution time and memory consumption for algorithms that attempt full traversal or pairwise comparison. This is particularly noticeable with Floyd–Warshall variants, which process all possible node pairs. Optimized Dijkstra is largely unaffected, gracefully handling unreachable nodes with early exits.

Key Observations

- Floyd–Warshall (and NumPy) are sensitive to disconnected graphs, showing increased time and memory usage.
- Dijkstra variants, especially the optimized one, are resilient even on disconnected topologies.
- Connected graphs allow smoother propagation across nodes, slightly reducing traversal time.
- Disconnected graphs result in wasted computation for unreachable pairs, especially for all-pairs algorithms.

2.3.5 Loops vs No Loops in Graphs

This section analyzes the performance of Floyd–Warshall and Dijkstra algorithms on graphs with and without self-loops. The experiment covered increasing graph sizes and recorded time and memory metrics for both configurations.

Empirical Results Table

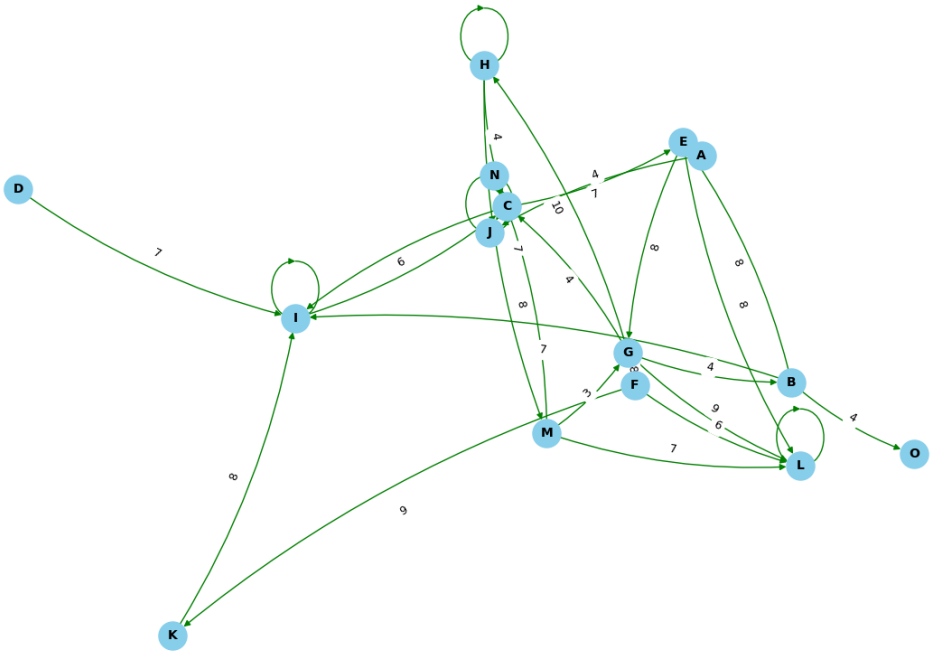
Table 2.9: Execution Time and Memory Usage – Loops vs No Loops (Part 1)

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Floyd–Warshall	10	No Loops	0.000364	3,176	N/A
Floyd–Warshall	10	With Loops	0.000338	2,784	N/A
Floyd–Warshall	15	No Loops	0.000596	5,144	N/A
Floyd–Warshall	15	With Loops	0.000601	5,144	N/A
Floyd–Warshall	50	No Loops	0.019507	193,207	N/A
Floyd–Warshall	50	With Loops	0.018039	193,734	N/A
Floyd–Warshall	75	No Loops	0.048568	92,832	N/A
Floyd–Warshall	75	With Loops	0.043743	92,832	N/A
Floyd–Warshall	100	No Loops	0.119201	175,960	N/A
Floyd–Warshall	100	With Loops	0.135772	175,960	N/A
Floyd–Warshall	250	No Loops	1.81718	1,024,700	N/A
Floyd–Warshall	250	With Loops	1.58836	1,022,672	N/A
Floyd–Warshall	500	No Loops	13.5428	4,173,640	N/A
Floyd–Warshall	500	With Loops	24.1817	4,173,584	N/A
Floyd–Warshall (NumPy)	10	No Loops	0.000291	7,612	N/A
Floyd–Warshall (NumPy)	10	With Loops	0.00019	7,612	N/A
Floyd–Warshall (NumPy)	15	No Loops	0.000354	12,136	N/A
Floyd–Warshall (NumPy)	15	With Loops	0.000437	12,136	N/A
Floyd–Warshall (NumPy)	50	No Loops	0.001761	123,848	N/A
Floyd–Warshall (NumPy)	50	With Loops	0.001933	123,848	N/A
Floyd–Warshall (NumPy)	75	No Loops	0.004035	261,304	N/A
Floyd–Warshall (NumPy)	75	With Loops	0.00354	261,304	N/A
Floyd–Warshall (NumPy)	100	No Loops	0.00752	635,232	N/A
Floyd–Warshall (NumPy)	100	With Loops	0.006715	487,504	N/A
Floyd–Warshall (NumPy)	250	No Loops	0.059542	2,852,160	N/A
Floyd–Warshall (NumPy)	250	With Loops	0.06293	2,852,160	N/A
Floyd–Warshall (NumPy)	500	No Loops	1.86332	11,560,444	N/A
Floyd–Warshall (NumPy)	500	With Loops	1.74177	11,709,084	N/A

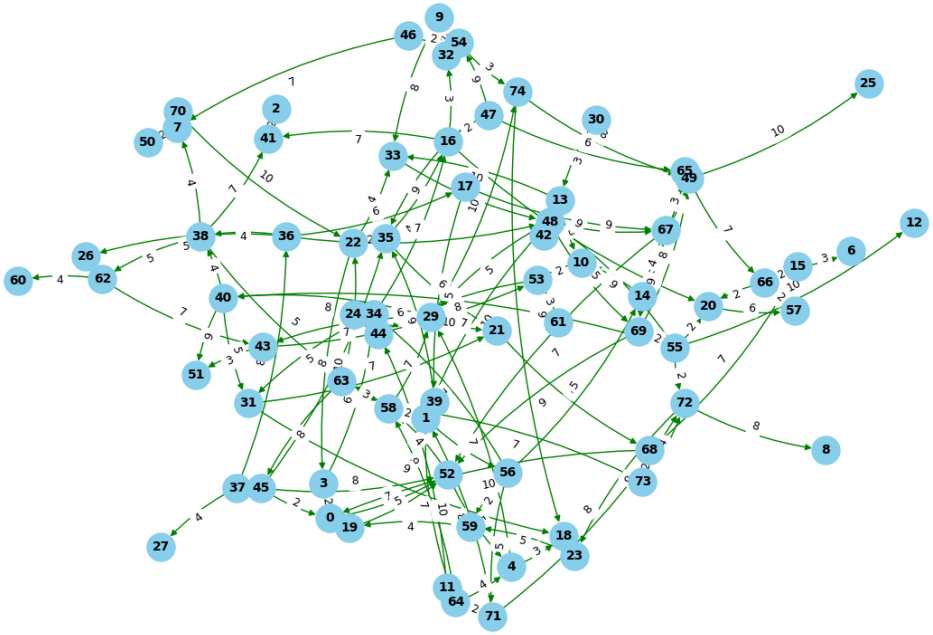
Table 2.10: Execution Time and Memory Usage – Loops vs No Loops (Part 2)

Algorithm	**Nodes**	**Graph Type**	**Time (s)**	**Memory (B)**	**Result**
Dijkstra	10	No Loops	0.00023	2,264	N/A
Dijkstra	10	With Loops	0.000215	2,264	N/A
Dijkstra	15	No Loops	0.000125	2,200	N/A
Dijkstra	15	With Loops	0.000171	2,264	N/A
Dijkstra	50	No Loops	0.001401	8,968	N/A
Dijkstra	50	With Loops	0.00121	9,000	N/A
Dijkstra	75	No Loops	0.002521	9,416	N/A
Dijkstra	75	With Loops	0.001774	9,512	N/A
Dijkstra	100	No Loops	0.003505	24,048	N/A
Dijkstra	100	With Loops	0.002703	24,240	N/A
Dijkstra	250	No Loops	0.013557	45,512	N/A
Dijkstra	250	With Loops	0.01713	46,120	N/A
Dijkstra	500	No Loops	0.065124	115,712	N/A
Dijkstra	500	With Loops	0.06308	116,672	N/A
Dijkstra (Optimized)	10	No Loops	0.000104	1,536	N/A
Dijkstra (Optimized)	10	With Loops	0.000141	1,536	N/A
Dijkstra (Optimized)	15	No Loops	0.000097	1,824	N/A
Dijkstra (Optimized)	15	With Loops	0.000101	1,888	N/A
Dijkstra (Optimized)	50	No Loops	0.000339	7,152	N/A
Dijkstra (Optimized)	50	With Loops	0.000413	7,184	N/A
Dijkstra (Optimized)	75	No Loops	0.000553	7,600	N/A
Dijkstra (Optimized)	75	With Loops	0.000596	7,696	N/A
Dijkstra (Optimized)	100	No Loops	0.000625	15,784	N/A
Dijkstra (Optimized)	100	With Loops	0.001034	15,976	N/A
Dijkstra (Optimized)	250	No Loops	0.001563	40,103	N/A
Dijkstra (Optimized)	250	With Loops	0.001295	188,359	N/A
Dijkstra (Optimized)	500	No Loops	0.002974	83,768	N/A
Dijkstra (Optimized)	500	With Loops	0.002553	84,728	N/A

Visual Graph Examples



(a) Graph with Loops (15 nodes)



(b) Graph without Loops (75 nodes)

Figure 2.9: Graph Structures with and without Self-Loops

Graphical Analysis

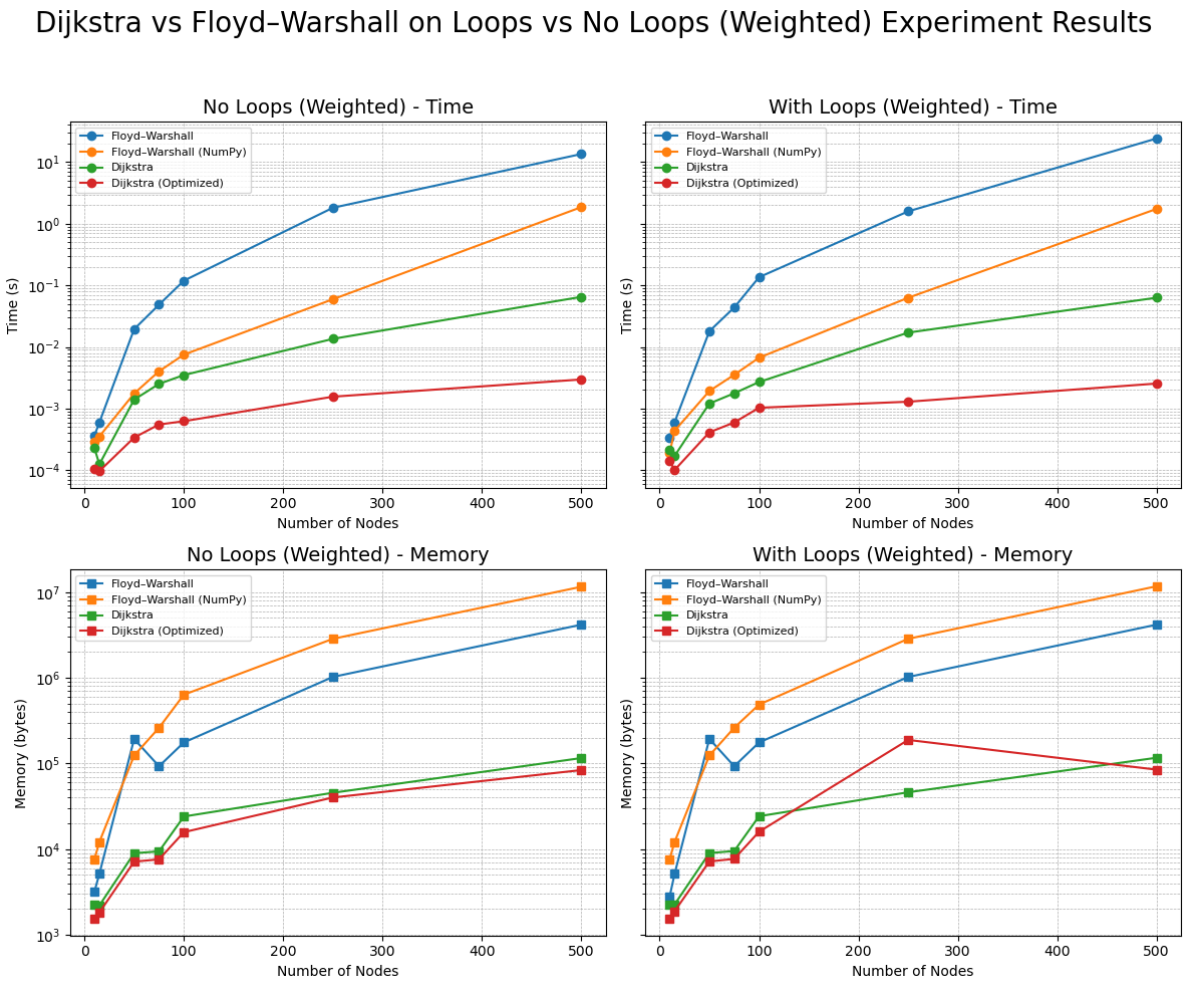


Figure 2.10: Time and Memory Performance – Loops vs No Loops

Observations:

- **Loop Presence Impact:** The inclusion of self-loops in graphs had a subtle yet noticeable effect on algorithmic performance, primarily impacting execution time rather than memory. This is because loops introduce redundant edges that require additional checks but do not significantly increase the overall graph structure or memory footprint.

- **Dijkstra (Optimized) Stability:** Across all tested node sizes, Dijkstra (Optimized) consistently demonstrated the most stable performance regardless of loop presence. The heap-based priority queue structure allowed it to efficiently bypass unnecessary loop traversals, minimizing any time penalties.

- **Floyd–Warshall Sensitivity:** Both standard and NumPy-based Floyd–Warshall implementations were more affected by the presence of loops. Since Floyd–Warshall evaluates every pair of nodes and considers all direct edges in its triple-nested iteration, self-loops added unnecessary evaluations that translated into longer execution times, particularly for graphs with higher node counts.

- **Scalability Trends:** As the graph size increased, loop effects became more pronounced in Floyd–Warshall time performance curves (especially for 250 and 500 nodes), while Dijkstra's line remained largely unaffected. This demonstrates the importance of loop handling when designing all-pairs shortest path algorithms for large graphs.
- **Algorithmic Robustness:** Despite the theoretical possibility of loops causing inefficiencies, most modern implementations of Dijkstra (especially the optimized one) proved resilient, effectively pruning or skipping self-loops during traversal. This suggests that Dijkstra-based solutions are more adaptable in real-world graph scenarios, where loops may naturally occur.
- **Practical Implication:** In applications where loop generation cannot be prevented (e.g., biological networks, transport systems, social graphs), using Dijkstra (Optimized) is highly recommended due to its loop-resilience and better scalability with minimal resource usage.

2.3.6 Shortest Path Algorithms on Tree Structures

Trees are a special class of graphs characterized by being connected and acyclic, with exactly $n-1$ edges for n nodes. These properties ensure that there exists only one unique path between any two nodes. In this section, we examine the empirical performance of four algorithms—Floyd–Warshall, Floyd–Warshall (NumPy), Dijkstra, and Dijkstra (Optimized)—on randomly generated trees of increasing size.

Empirical Results Table

Table 2.11: Execution Time and Memory Usage – Trees

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Floyd–Warshall	10	Tree	0.000882	3,176	N/A
Floyd–Warshall	15	Tree	0.001565	5,144	N/A
Floyd–Warshall	50	Tree	0.034067	193,136	N/A
Floyd–Warshall	75	Tree	0.099756	96,824	N/A
Floyd–Warshall	100	Tree	0.230083	175,960	N/A
Floyd–Warshall	250	Tree	3.611660	1,297,724	N/A
Floyd–Warshall	500	Tree	24.745900	4,426,756	N/A
Floyd–Warshall (NumPy)	10	Tree	0.000336	7,612	N/A
Floyd–Warshall (NumPy)	15	Tree	0.000302	12,136	N/A
Floyd–Warshall (NumPy)	50	Tree	0.002281	123,848	N/A
Floyd–Warshall (NumPy)	75	Tree	0.004267	275,656	N/A
Floyd–Warshall (NumPy)	100	Tree	0.009971	635,568	N/A
Floyd–Warshall (NumPy)	250	Tree	0.128860	3,018,304	N/A
Floyd–Warshall (NumPy)	500	Tree	1.867580	12,038,448	N/A

Table 2.12: Execution Time and Memory Usage – Trees

Algorithm	Nodes	Graph Type	Time (s)	Memory (B)	Result
Dijkstra	10	Tree	0.001085	2,392	N/A
Dijkstra	15	Tree	0.000866	3,144	N/A
Dijkstra	50	Tree	0.003583	9,032	N/A
Dijkstra	75	Tree	0.004190	9,832	N/A
Dijkstra	100	Tree	0.005651	24,176	N/A
Dijkstra	250	Tree	0.021665	195,262	N/A
Dijkstra	500	Tree	0.125095	118,032	N/A
Dijkstra (Optimized)	10	Tree	0.000416	1,752	N/A
Dijkstra (Optimized)	15	Tree	0.000313	2,632	N/A
Dijkstra (Optimized)	50	Tree	0.000762	7,800	N/A
Dijkstra (Optimized)	75	Tree	0.001032	8,408	N/A
Dijkstra (Optimized)	100	Tree	0.001004	164,413	N/A
Dijkstra (Optimized)	250	Tree	0.002883	40,928	N/A
Dijkstra (Optimized)	500	Tree	0.006486	225,240	N/A

Tree Graph Example

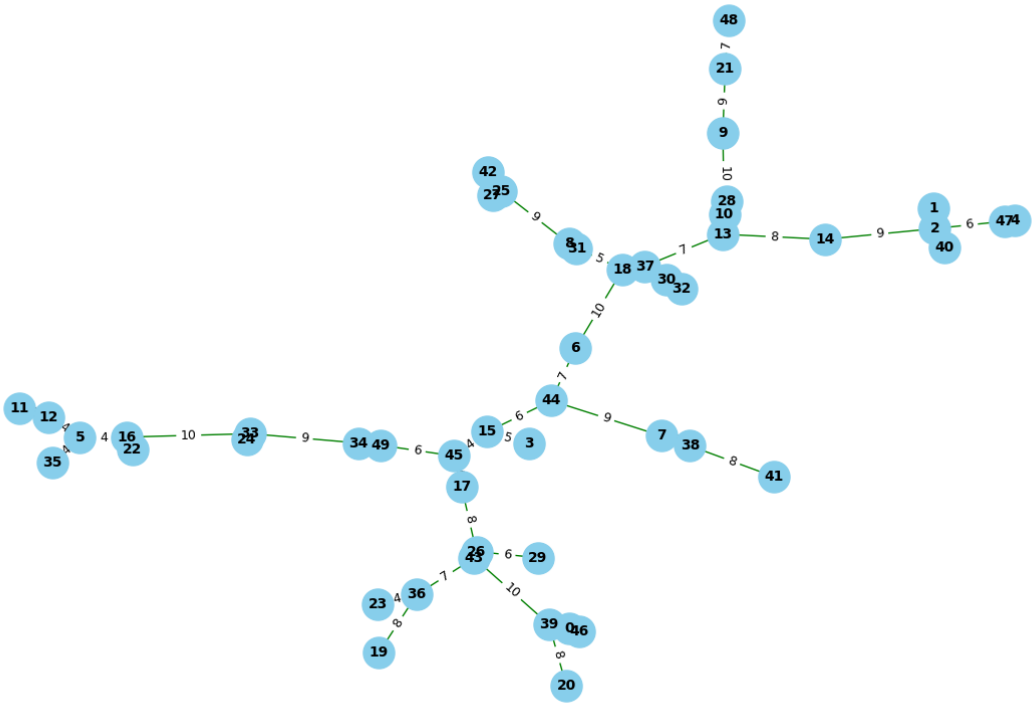


Figure 2.11: Example of a randomly generated tree with 50 nodes

Time and Memory Performance

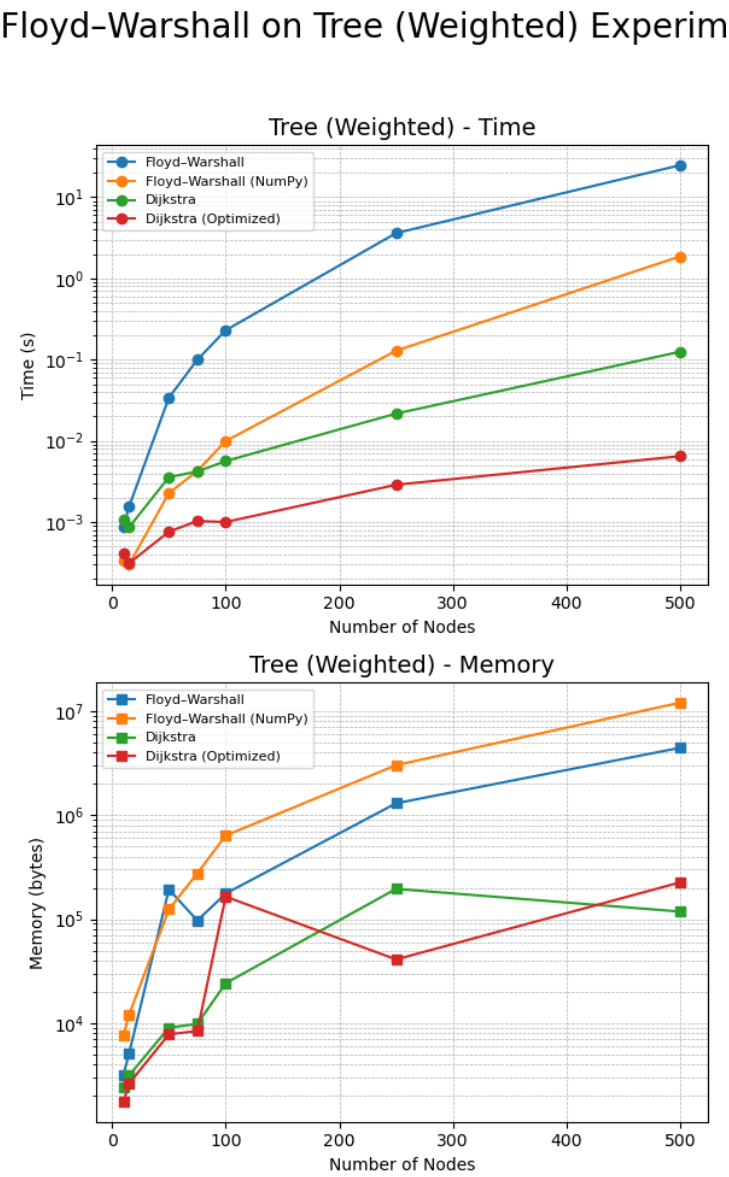


Figure 2.12: Performance of Shortest Path Algorithms on Trees

Analysis:

- All algorithms exhibit better performance on trees due to the absence of cycles and the guarantee of unique paths.
- **Floyd–Warshall (classic and NumPy)** remain less memory-efficient due to full-pairwise computation despite the tree's sparsity.
- **Dijkstra (Optimized)** clearly outperforms all others in both execution time and memory usage, especially as node count increases.
- **Standard Dijkstra** performs decently but begins to scale poorly in memory compared to its optimized variant.

- These results confirm that tree structures favor algorithms with path-specific focus rather than full-matrix approaches.
- Memory usage in Floyd–Warshall grows quadratically, which becomes unnecessary overhead in sparse trees where most node pairs are not directly connected.
- Execution times for Floyd–Warshall methods, although consistent, become impractical on trees with more than 250 nodes.
- Dijkstra's edge-by-edge strategy capitalizes on the minimal connectivity of trees, traversing only what's needed, making it ideal for such structures.

Chapter 3

Conclusion

The empirical analysis of shortest path algorithms presented in this report was conducted on an **Asus ZenBook 14**, equipped with an **Intel Core i7 8th Gen processor (1.8 GHz, 4 cores, 8 threads)**, **16GB RAM**, and **512GB SSD storage**. The experiments were implemented in **Python** using **Visual Studio Code** and executed in a **Jupyter Notebook** environment. Additional scalability tests were carried out on **Google Colab** to validate trends in a cloud-based setting.

The study focused on two key algorithms—**Dijkstra's Algorithm** and the **Floyd – Warshall Algorithm**—including their optimized versions. Each was evaluated across a range of graph types: sparse vs. dense, directed vs. undirected, cyclic vs. acyclic, connected vs. disconnected, with and without loops, and trees. Performance metrics such as **execution time** and **memory usage** were recorded using the `timeit` and `tracemalloc` modules.

Key Insights

- **Dijkstra (Optimized):** This version consistently offered the best performance in both execution time and memory usage, particularly on sparse graphs, tree structures, and disconnected graphs. Its use of a priority queue (heap) made it highly scalable.
- **Floyd–Warshall (Standard):** Performed reliably but scaled poorly on dense graphs due to its cubic time complexity and quadratic memory usage. Its exhaustive all-pairs approach proved costly for large graphs.
- **Floyd–Warshall (NumPy):** Provided performance improvements in time over the standard variant thanks to vectorized operations, but at the expense of significantly higher memory consumption.
- **Graph Topology Effects:**
  - *Sparse vs Dense:* Dense graphs negatively impacted all algorithms, especially Floyd–Warshall.
  - *Directed vs Undirected:* Directed graphs introduced longer or unreachable paths, increasing execution cost.
  - *Cyclic vs Acyclic:* Cycles had more effect on Dijkstra's variants; Floyd–Warshall remained unaffected due to its structure.

  - *Connected vs Disconnected:* Disconnected graphs led to higher computation and memory usage, particularly for Floyd–Warshall.
  - *Loops vs No Loops:* Self-loops had marginal impact on Dijkstra but added overhead in Floyd–Warshall due to unnecessary iterations.
  - *Trees:* All algorithms performed better on tree graphs, with Dijkstra (Optimized) again being the most efficient.

- **Optimized Implementations:** Across all tests, optimized versions of both algorithms demonstrated significant improvements over their standard counterparts, especially in scalability and runtime.

Complexity Overview

The empirical results closely align with the theoretical time complexities of the analyzed algorithms. Dijkstra's algorithm (when implemented with a binary heap) maintained a time complexity of $O((n+m)\log n)$, making it well-suited for sparse graphs. In contrast, Floyd–Warshall's $O(n^3)$ time complexity and $O(n^2)$ space complexity became limiting factors on larger or denser graphs. These results affirm the practical implications of asymptotic complexity in real-world performance.

Final Remarks

This laboratory work demonstrated the value of empirical analysis for understanding real-world performance of shortest path algorithms under varying graph conditions. While theoretical complexity offers upper bounds, practical performance is often influenced by graph structure, implementation strategies, and memory access patterns. Dijkstra's algorithm (particularly the heap-based variant) proved to be the most efficient overall, while Floyd–Warshall remains relevant for dense graphs or use cases requiring all-pairs shortest paths.

Source Code

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