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 #4

Study and empirical analysis of algorithms: Dijkstra and Floyd-Warshall

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/lab345`

1.1 Objective

The objective of this laboratory work is to implement and analyze two fundamental shortest path algorithms: Dijkstra's algorithm and the Floyd-Warshall algorithm. 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, with a specific focus on dynamic programming concepts used in Floyd-Warshall.

1.2 Tasks

1 To study the dynamic programming method of designing algorithms. 2 To implement in a programming language algorithms Dijkstra and Floyd–Warshall using dynamic programming. 3 Do empirical analysis of these algorithms for a sparse graph and for a dense graph. 4 Increase the number of nodes in graphs and analyze how this influences the algorithms. Make a graphical presentation of the data obtained 5 To make a report.

1.3 Theoretical Notes

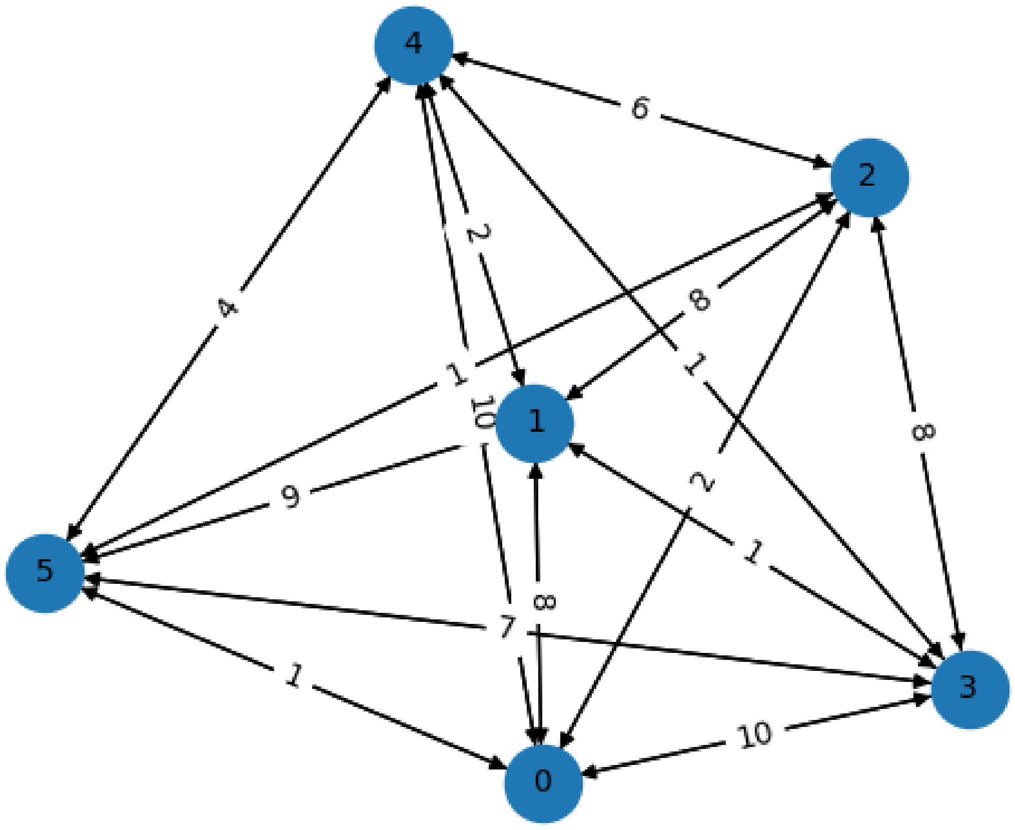
1.3.1 Graph Theory Basics

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 where each edge has an associated weight or cost.

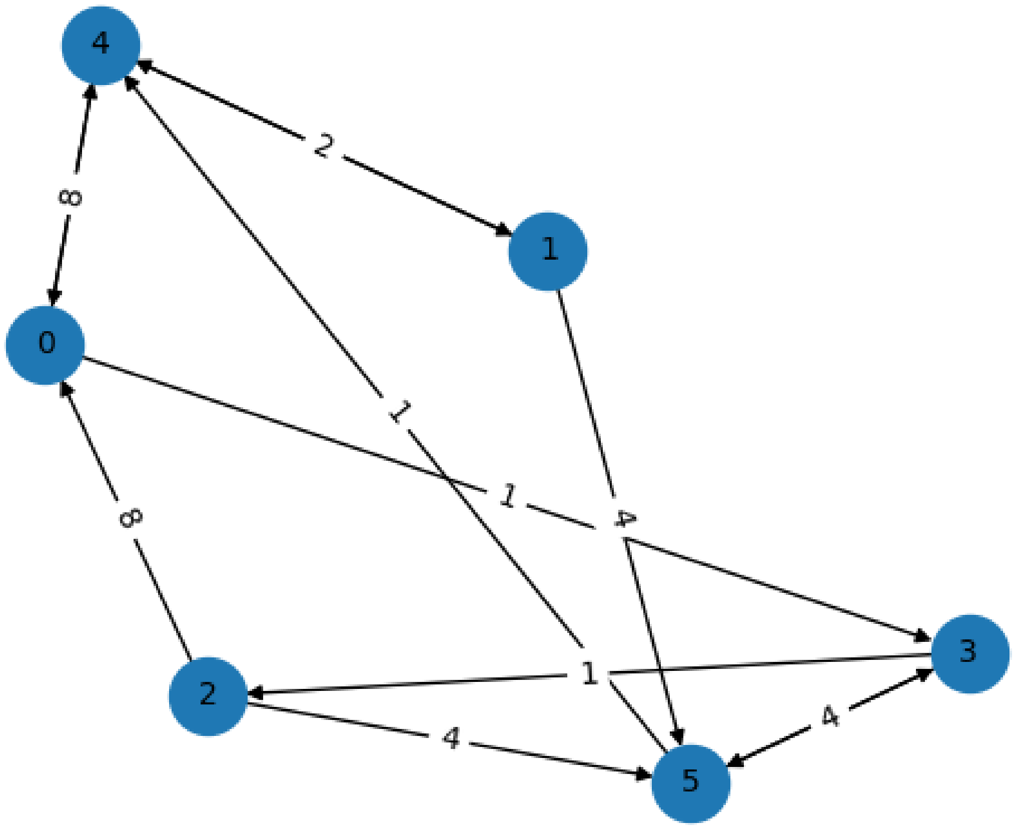
1.3.2 Types of Graphs

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

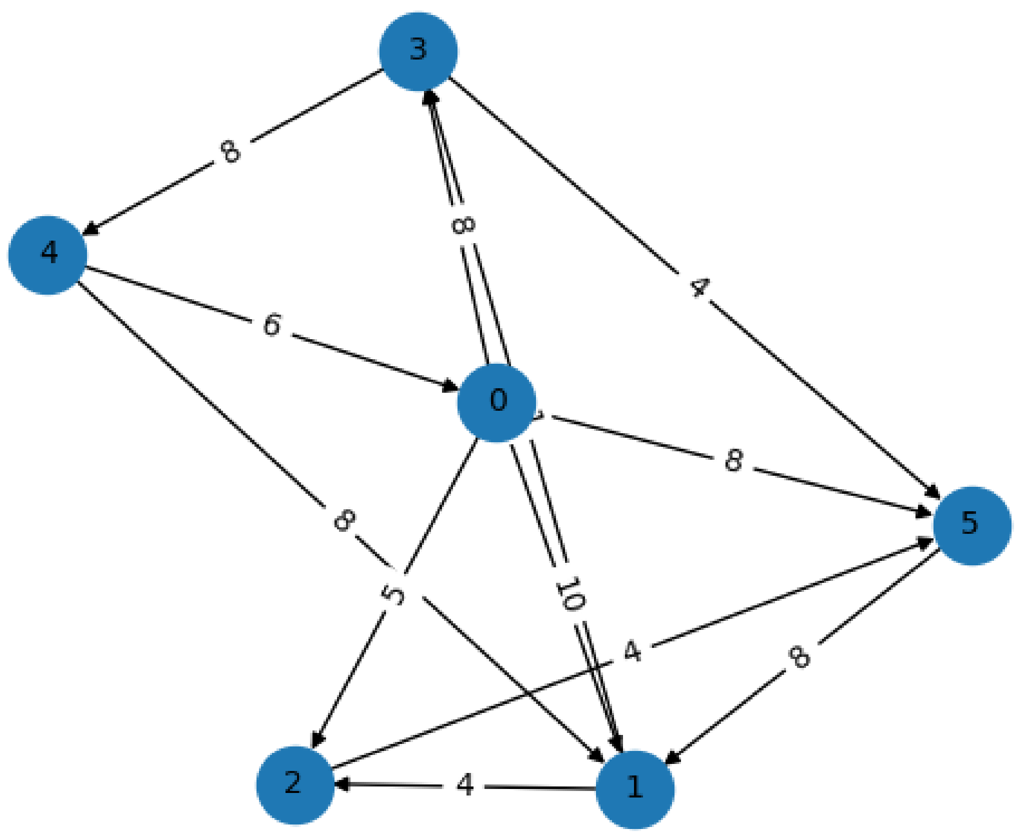
1. Complete Graph Every vertex is connected to every other vertex, resulting in |V|(|V|-1) edges in a directed graph.



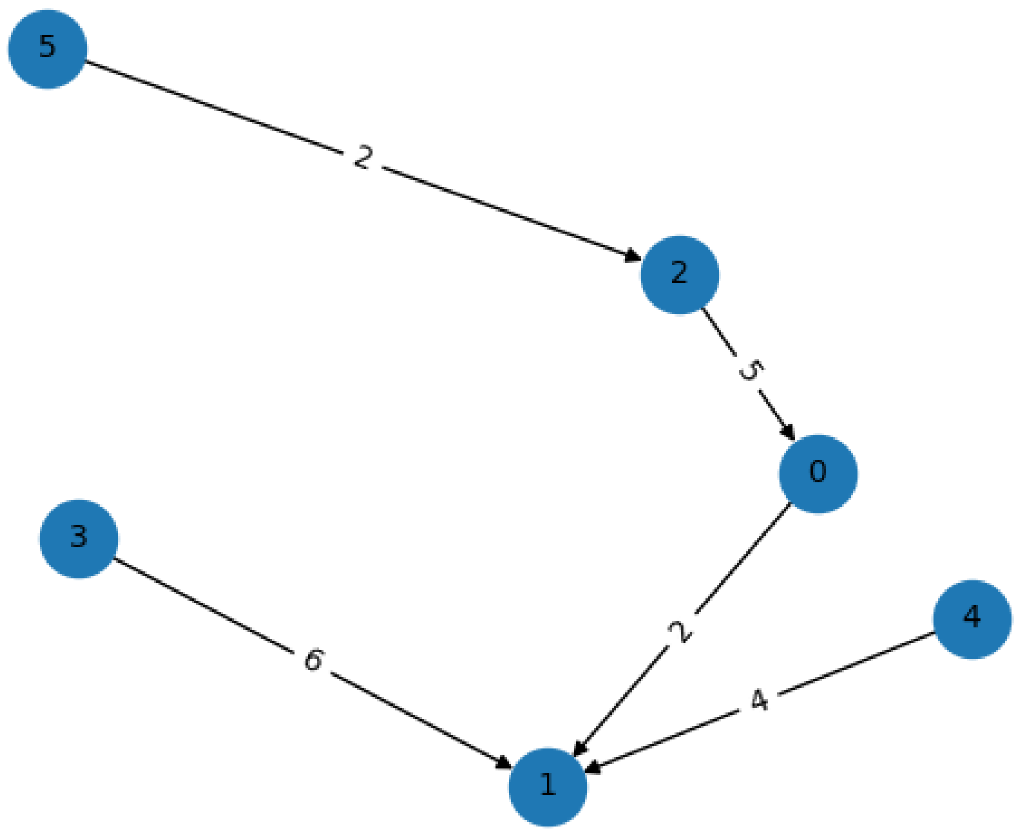
2. Dense Graph Has approximately 80% of the maximum possible edges.



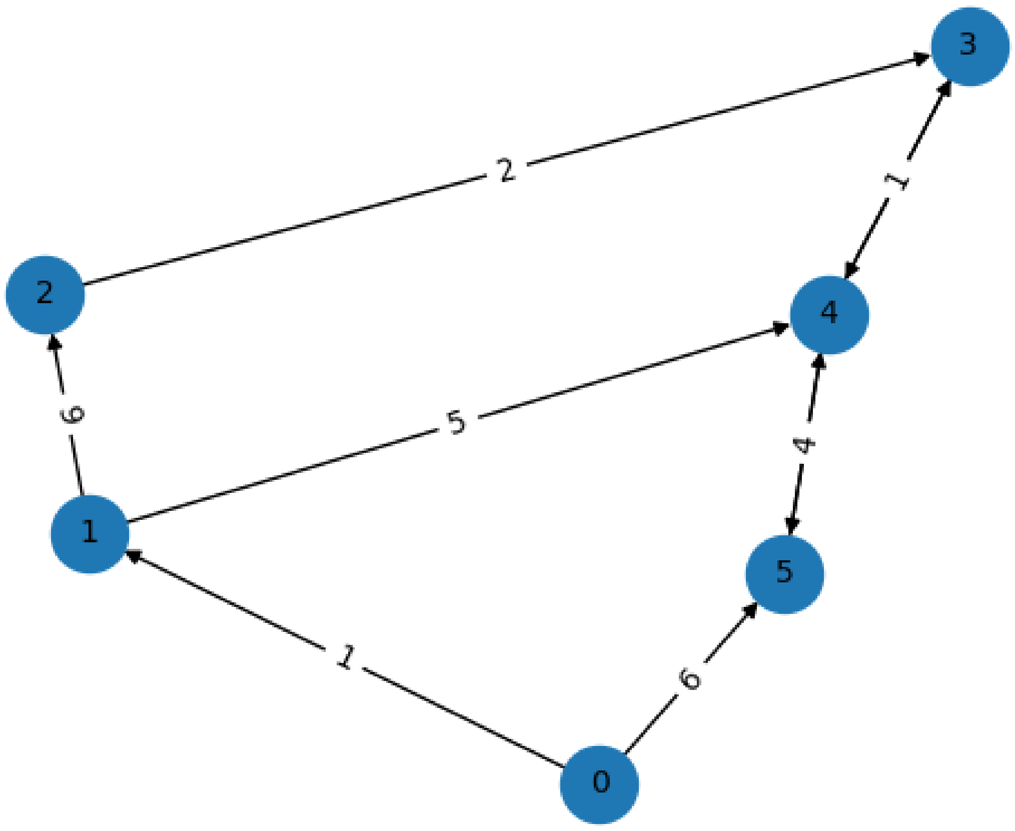
3. Sparse Graph Contains relatively few edges, approximately 2|V| edges.



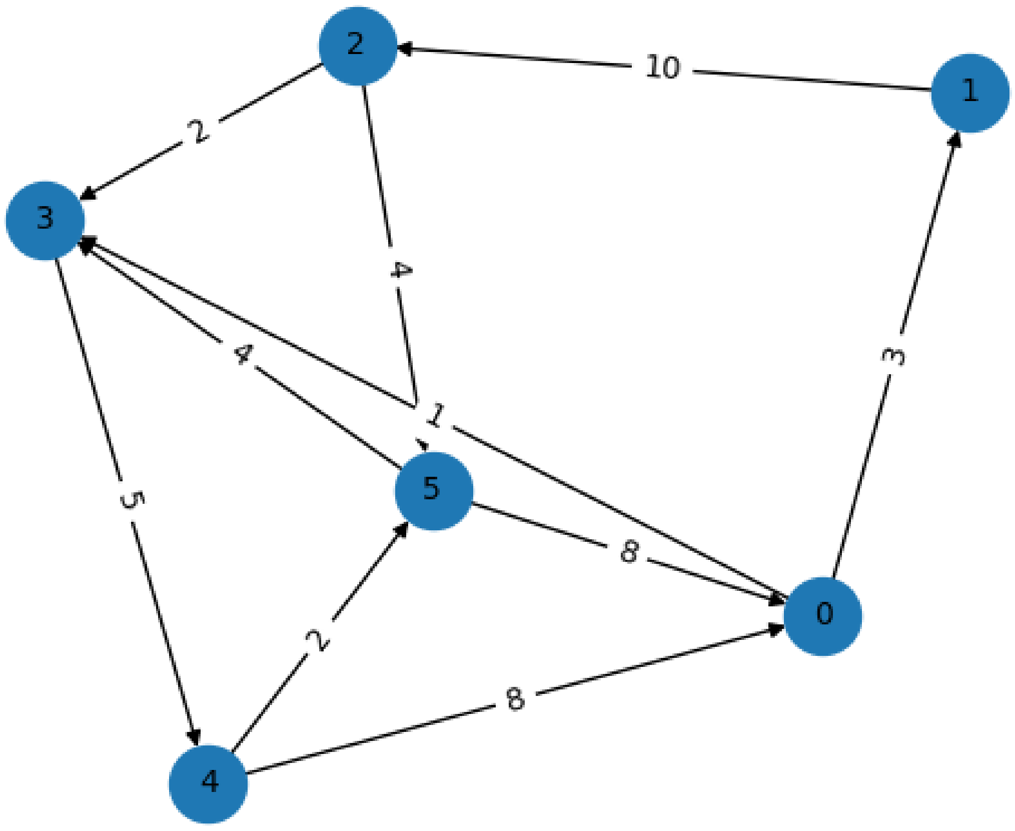
4. Tree Graph Connected acyclic graph with exactly |V|-1 edges.



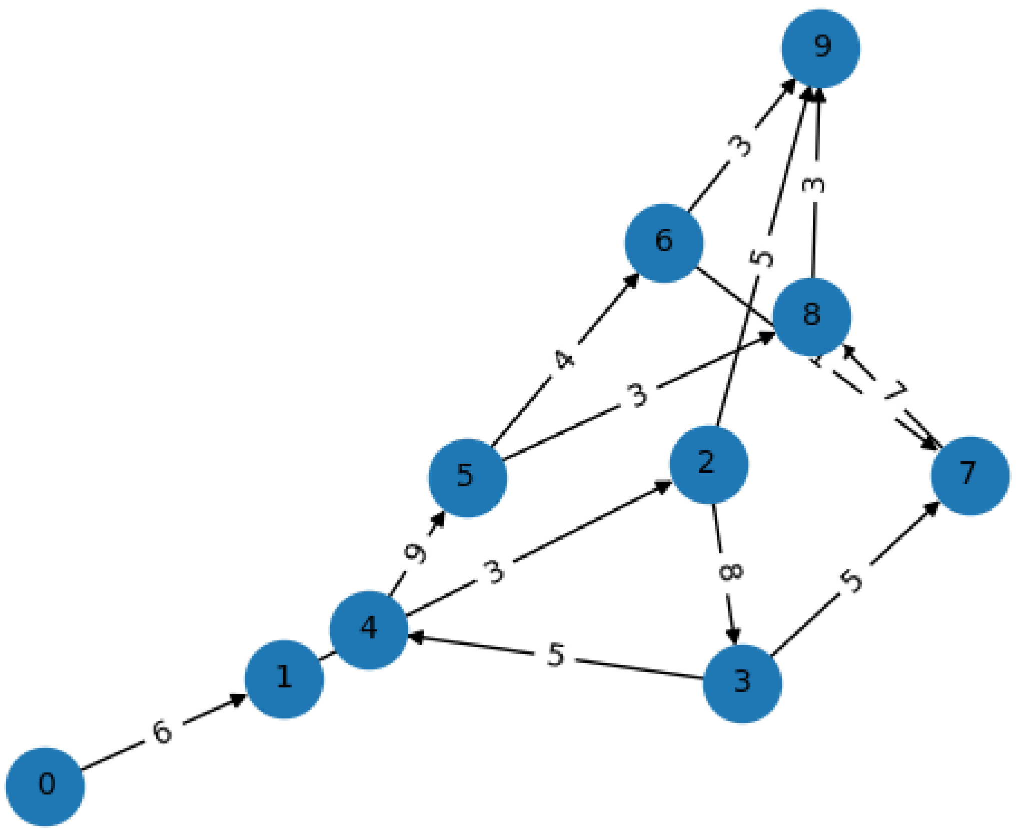
5. Connected Graph There exists a path between any pair of vertices.



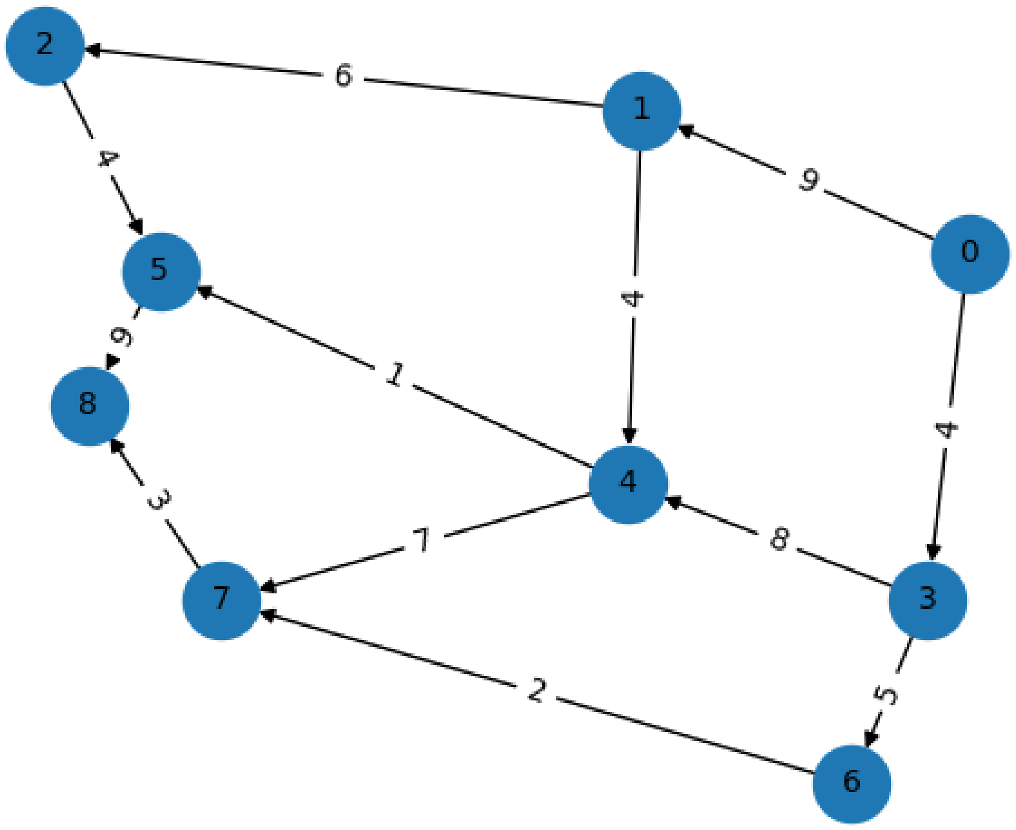
6. Cyclic Graph Contains at least one cycle (a path that starts and ends at the same vertex).



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.



1.3.3 Dynamic Programming

Dynamic programming is an algorithmic paradigm that solves complex problems by breaking them down into simpler subproblems and storing the solutions to avoid redundant computations. Key elements of dynamic programming include:

1. **Optimal Substructure**: An optimal solution to the problem contains optimal solutions to its subproblems.
2. **Overlapping Subproblems**: The same subproblems are solved multiple times, making caching (memoization) beneficial.

The Floyd-Warshall algorithm is a classic example of dynamic programming applied to the all-pairs shortest path problem.

1.3.4 Dijkstra's Algorithm

Dijkstra's algorithm finds the shortest path from a source vertex to all other vertices in a weighted graph with non-negative edge weights. The algorithm works as follows:

1. Initialize distances to all vertices as infinity and distance to source as 0.
2. Create a priority queue and add the source vertex with distance 0.
3. While the priority queue is not empty: a. Extract the vertex with minimum distance value. b. For each adjacent vertex, if the distance through current vertex is less than its current distance, update the distance and add to priority queue.
4. After the algorithm completes, the distances array contains the shortest distance from source to all vertices.

While Dijkstra's algorithm doesn't use dynamic programming in its pure form, it shares the principle of optimal substructure.

1.3.5 Floyd-Warshall Algorithm

Floyd-Warshall is a dynamic programming algorithm that finds shortest paths between all pairs of vertices in a weighted graph. The algorithm works as follows:

1. Initialize the distance matrix with direct edge weights (or infinity if there's no edge).

2. For each vertex k as an intermediate vertex: a. For each pair of vertices (i,j): i. If distance[i][j] > distance[i][k] + distance[k][j], update distance[i][j]

3. After the algorithm completes, distance[i][j] contains the shortest path from vertex i to vertex j.

This algorithm clearly demonstrates dynamic programming principles by systematically building solutions for all pairs using previously computed results.

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. **Memory Usage**: Indirectly measured by observing the maximum size of the data structures used.

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 = [10, 50, 60, 70, 100]
mid_sizes = [100, 200, 300, 500, 700, 1000]
dj_sizes = [100, 500, 1000, 1500, 2000, 2500, 3000]
```

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 Dijkstra and Floyd-Warshall 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 Dijkstra’s Algorithm

Dijkstra’s algorithm is implemented using a priority queue to efficiently select the vertex with the minimum distance in each iteration. The implementation maintains distances and visited status for all vertices.

2.2 Floyd-Warshall Algorithm

The Floyd-Warshall algorithm is implemented using dynamic programming principles, with a three-dimensional loop structure that considers each vertex as a potential intermediate node in shortest paths.

```python
import heapq

def dijkstra(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    prev = {}  # to track how we got to each node
    explored_edges = []

    pq = [(0, start)]

    while pq:
        current_distance, current_node = heapq.heappop(pq)

        # skip outdated queue entries
        if current_distance > distances[current_node]:
            continue

        for neighbor, weight in graph[current_node].items():
            explored_edges.append((current_node, neighbor, weight))
            distance = current_distance + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                prev[neighbor] = (current_node, weight)
                heapq.heappush(pq, (distance, neighbor))

    # build the final shortest-path tree from prev
    tree_edges = [(src, dst, weight) for dst, (src, weight) in prev.items()]

    return tree_edges, explored_edges, distances
```

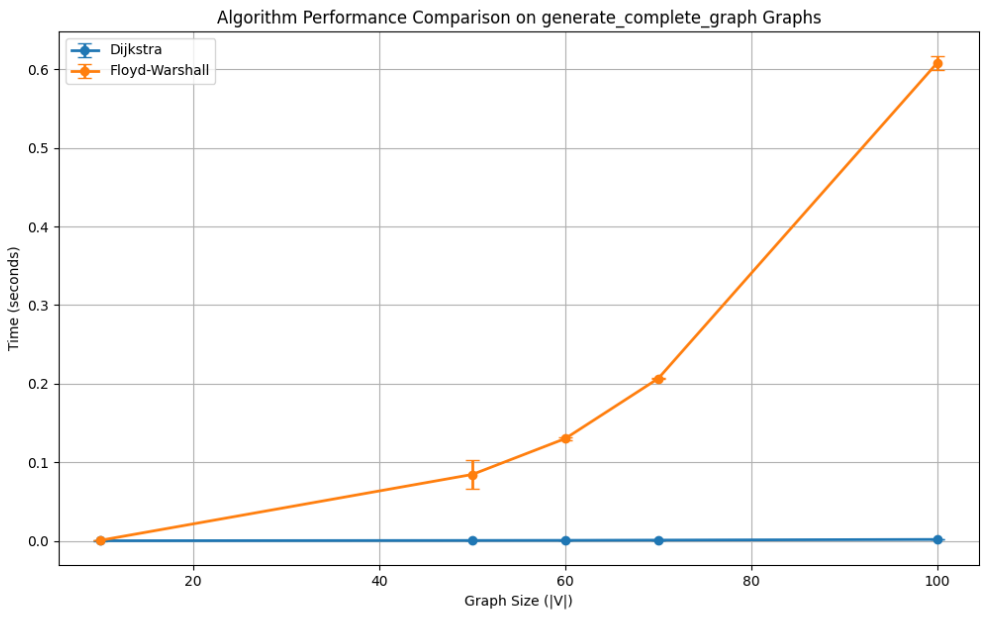
Listing 1: Implementation of Dijkstra's Algorithm

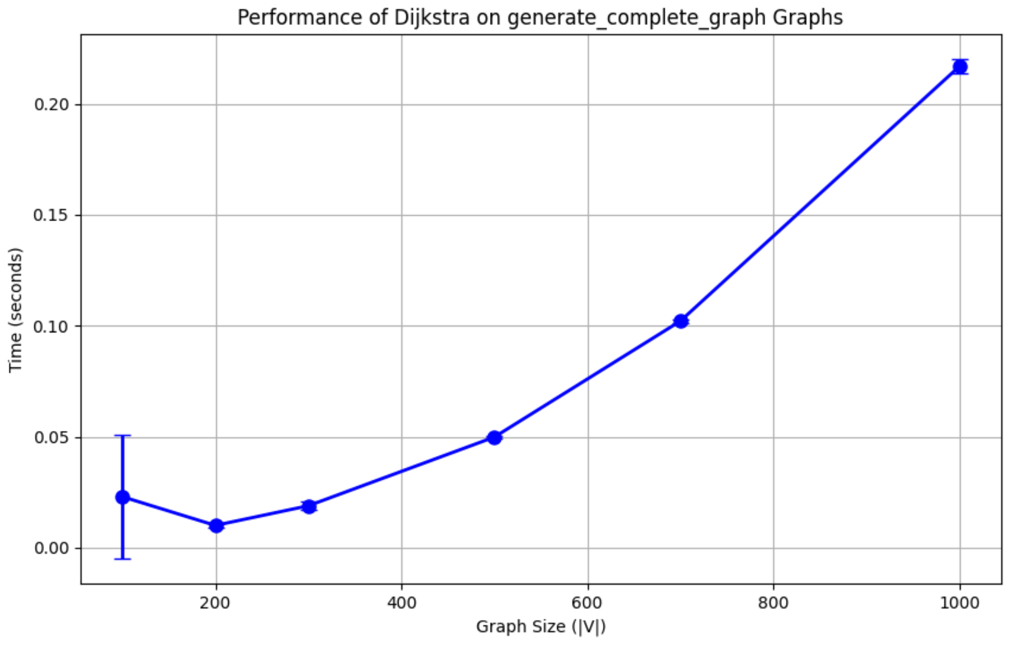
```
def floyd_warshall(graph, start=None):





    """
        Finds shortest paths between all pairs of vertices using Floyd-Warshall
        algorithm.
    # Extract all vertices
    vertices = list(graph.keys())

    # Initialize distances dictionary
    distances = {u: {v: float('inf') for v in vertices} for u in vertices}

    # Initialize predecessor dictionary for path reconstruction
    prev = {}

    # Set distance to self as 0, and direct edges from the graph
    for u in vertices:
        distances[u][u] = 0
        for v, weight in graph[u].items():
            distances[u][v] = weight
            if v not in prev:
                prev[v] = {}
            prev[v][u] = (u, weight)  # Store tuple (predecessor, weight) like in
            ↪  Dijkstra

    # Track all explored edges
    explored_edges = []

    # Floyd-Warshall dynamic programming algorithm
    for k in vertices:
        for i in vertices:
            for j in vertices:
                # Record this exploration
                explored_edges.append((i, k, j, distances[i][j]))

                # If path through k is shorter than current path from i to j
                if distances[i][k] != float('inf') and distances[k][j] !=
                ↪  float('inf'):
                    candidate_dist = distances[i][k] + distances[k][j]
                    if candidate_dist < distances[i][j]:
                        distances[i][j] = candidate_dist
                        if j not in prev:
                            prev[j] = {}
                        prev[j][i] = prev[j][k]  # Update predecessor

    # Build the final shortest-path tree from prev (similar to Dijkstra)
    tree_edges = []
    for j in vertices:
        if j in prev:
            for i in prev[j]:
                if isinstance(prev[j][i], tuple):
                    src, weight = prev[j][i]
                    tree_edges.append((src, j, weight))

    return tree_edges, explored_edges, distances
```

Listing 2: Implementation of Floyd-Warshall Algorithm

3 Results

3.1 Complete Graph





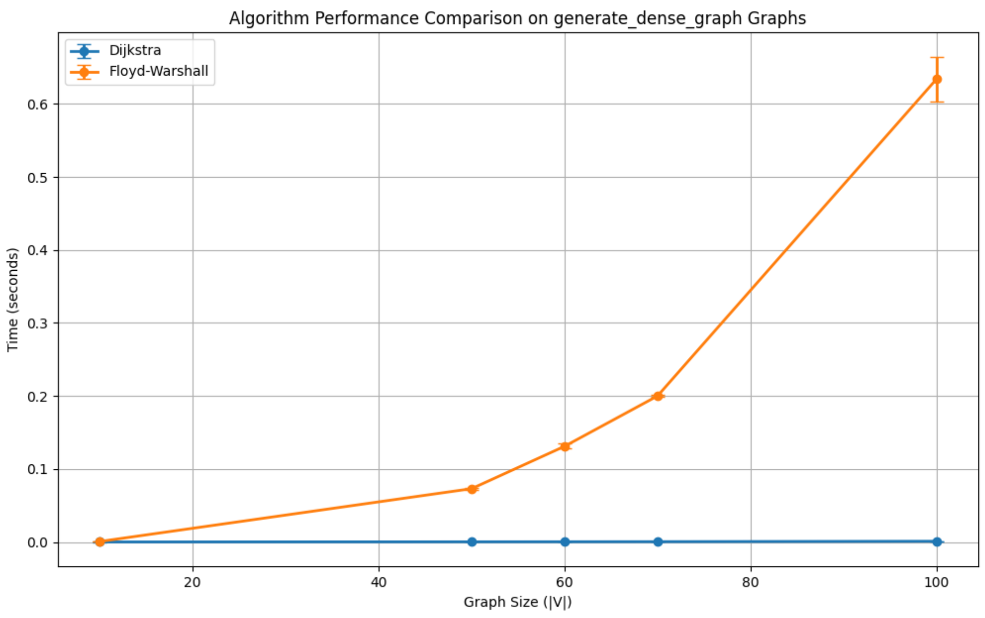
Both algorithms show a clear polynomial growth pattern as the graph size increases. Floyd-Warshall exhibits $O(V^3)$ complexity while Dijkstra (even with priority queue optimization) approaches $O(V^2)$ for complete graphs where $E = O(V^2)$.

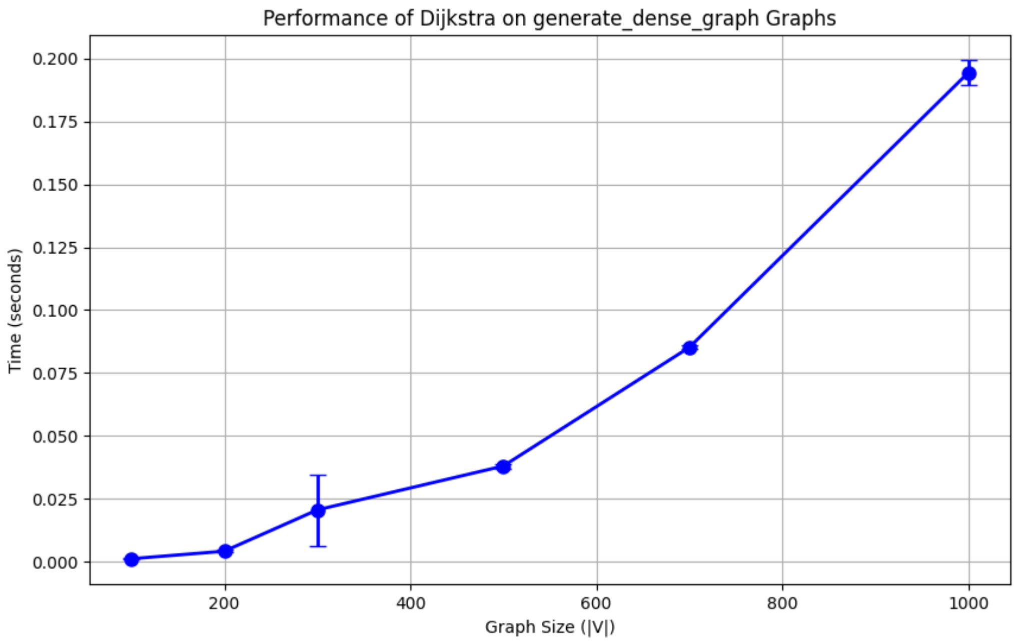
Floyd-Warshall consistently demonstrates higher execution times compared to Dijkstra, with the gap widening as graph size increases. This aligns with theoretical expectations given Floyd-Warshall's cubic complexity versus Dijkstra's near-quadratic behavior on complete graphs.

For larger graph sizes (mid$_{\text{sizes}}$ plot), Dijkstra maintains reasonable performance, showing the expected quadratic growth pattern as the number of vertices increases.

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

3.2 Dense Graph





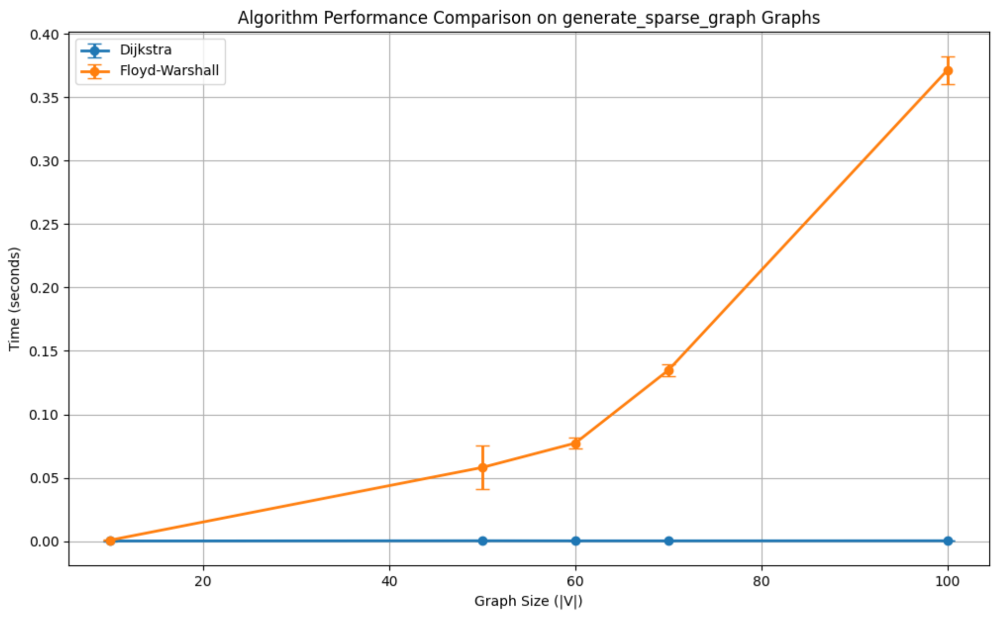
In dense graphs (approximately 80% of maximum possible edges), both algorithms show similar performance patterns to complete graphs, with Floyd-Warshall requiring significantly more execution time.

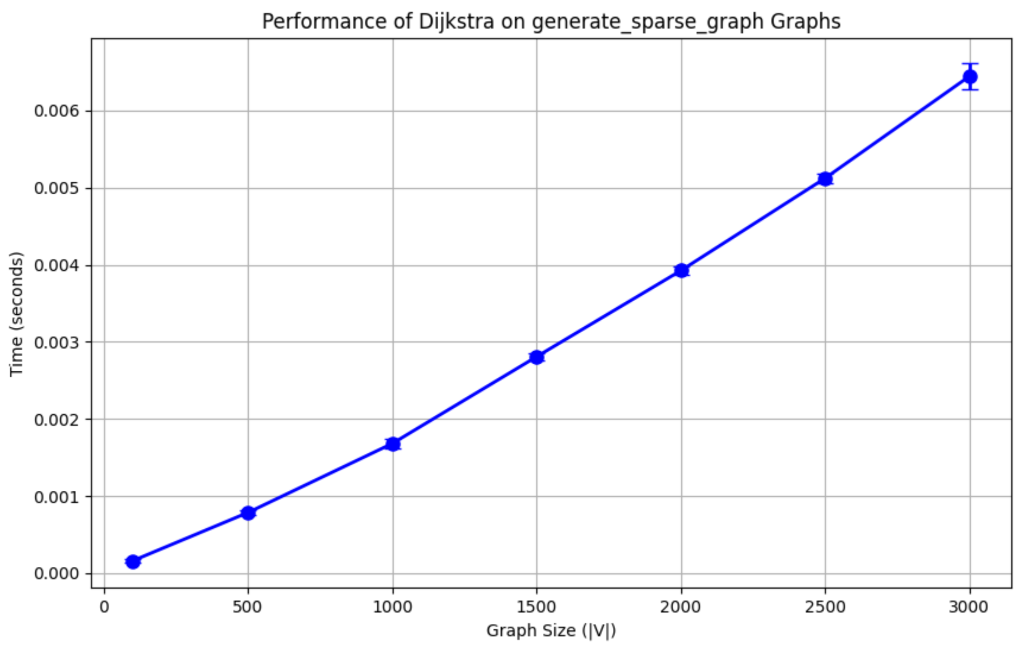
The execution time growth is polynomial, with Floyd-Warshall showing cubic growth O(V^3) while Dijkstra demonstrates more efficient scaling with its priority queue implementation.

For larger graph sizes, Dijkstra maintains reasonable scalability, although the execution time increases noticeably with the graph size due to the high edge density.

The standard deviation values are low, indicating high consistency in measurement across test repetitions.

3.3 Sparse Graph





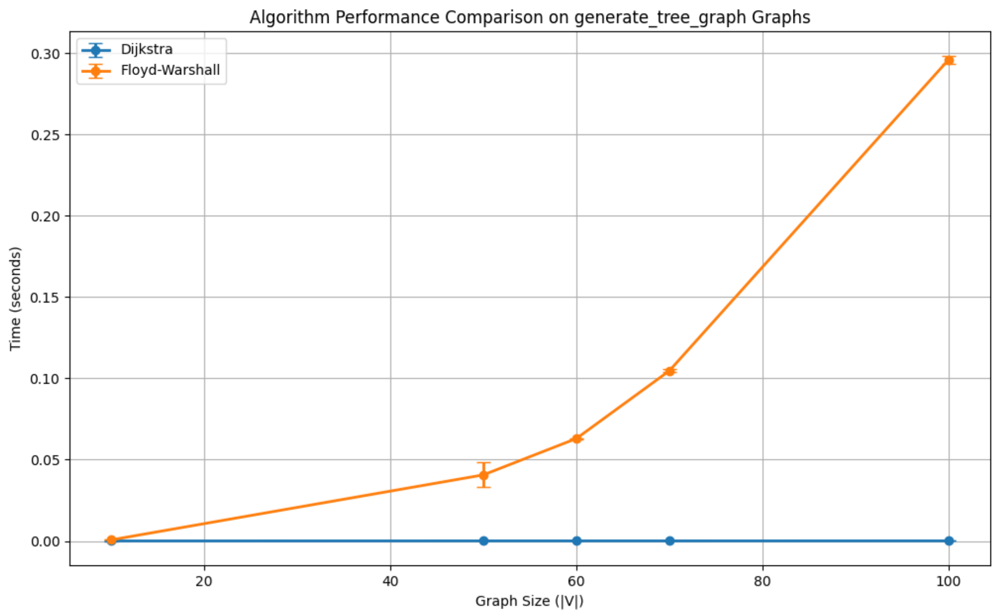
For sparse graphs (with approximately 2|V| edges), both algorithms show different scaling patterns. Floyd-Warshall exhibits its characteristic cubic growth $O(V^3)$ regardless of edge density, while Dijkstra performs significantly better, leveraging its more efficient $O((V+E)\log V)$ complexity.

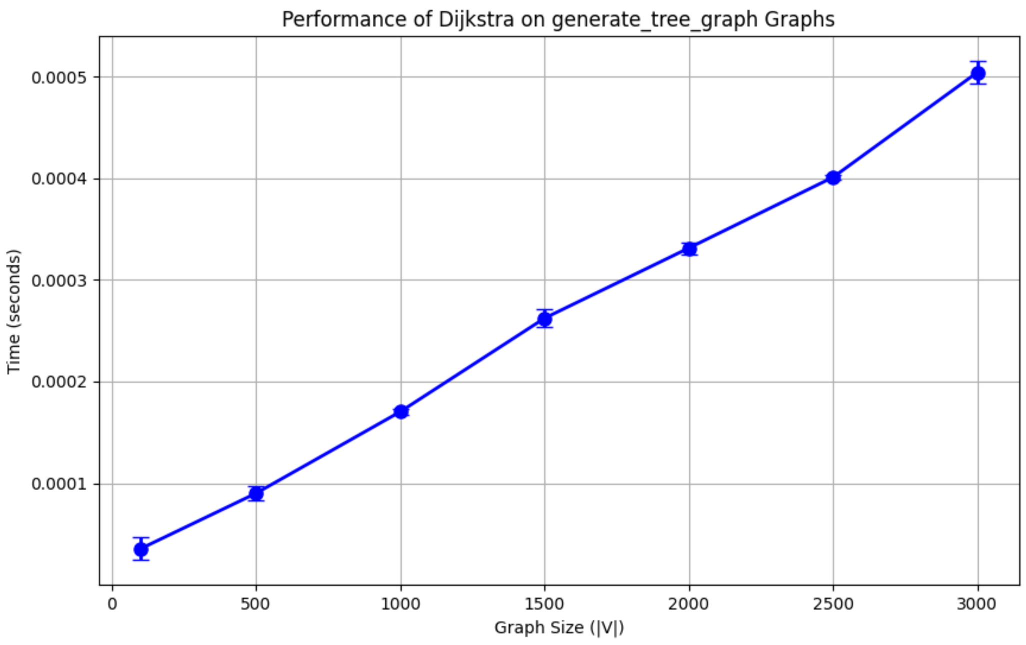
Dijkstra's algorithm consistently outperforms Floyd-Warshall across all tested graph sizes, with the performance gap widening as the graph size increases.

For larger graph sizes (in the dj_{sizes} plot), Dijkstra maintains excellent scalability even up to 3000 vertices, demonstrating its efficiency for single-source shortest path problems in sparse networks.

The standard deviation is relatively small, indicating consistent performance across test repetitions.

3.4 Tree Graph





Tree graphs show a distinctive performance pattern between the two algorithms.

Floyd-Warshall's execution time increases cubically with graph size, while Dijkstra shows near-linear growth due to the tree's minimal edge count (|V|-1 edges).

In the larger graph size tests, Dijkstra processes tree structures extremely efficiently, reflecting its optimal performance characteristics when there are no alternative paths to consider.

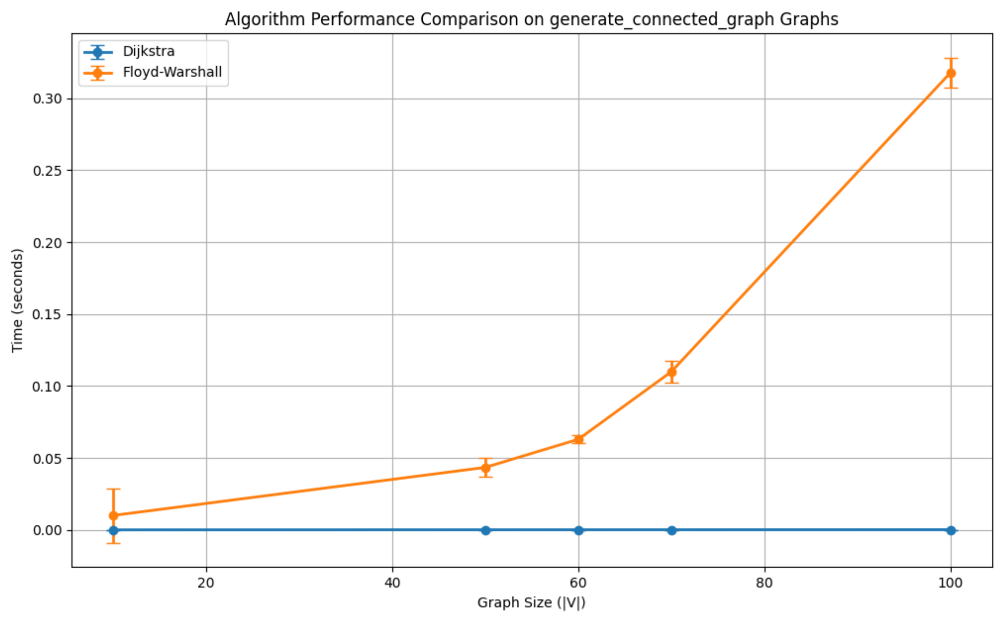
The standard deviation remains small relative to the mean values, showing measurement consistency.

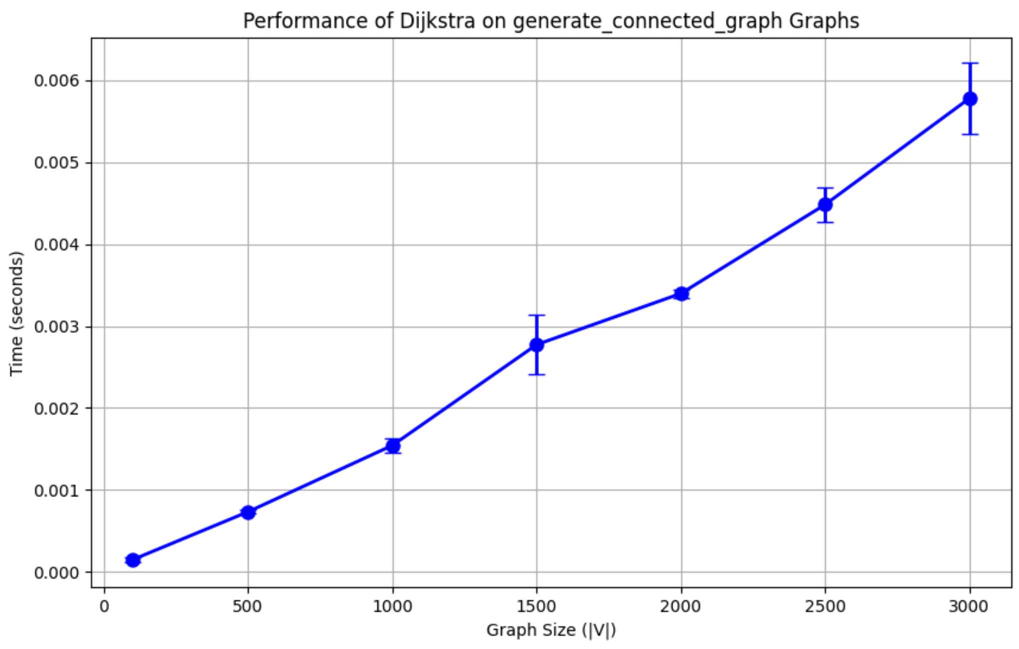
Dijkstra's algorithm particularly excels with tree structures because:

- Each vertex is visited exactly once
- No path recalculations are needed
- The priority queue operations are minimized in this optimal scenario

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

3.5 Connected Graph



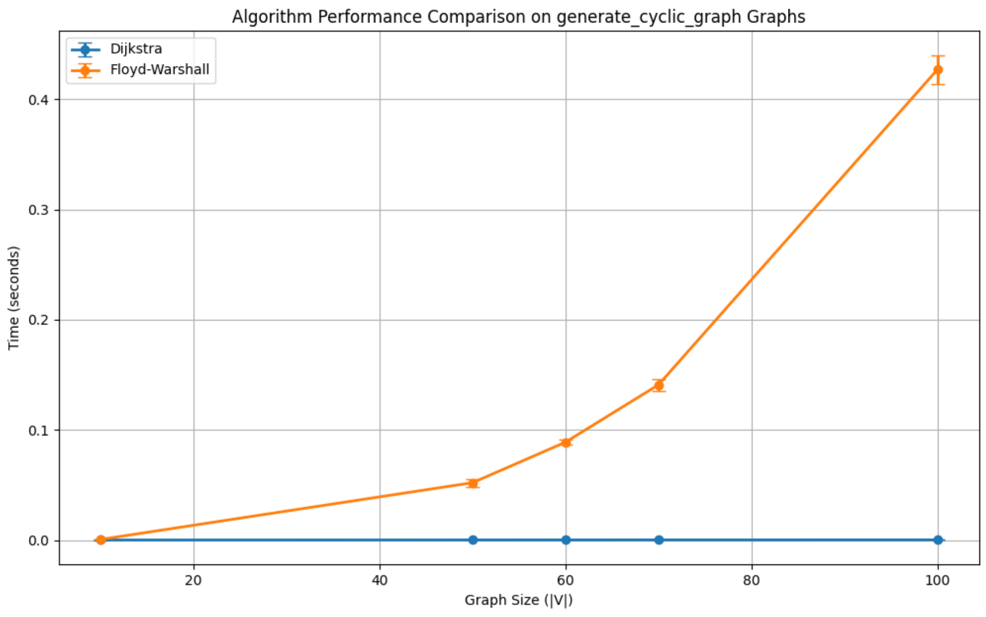


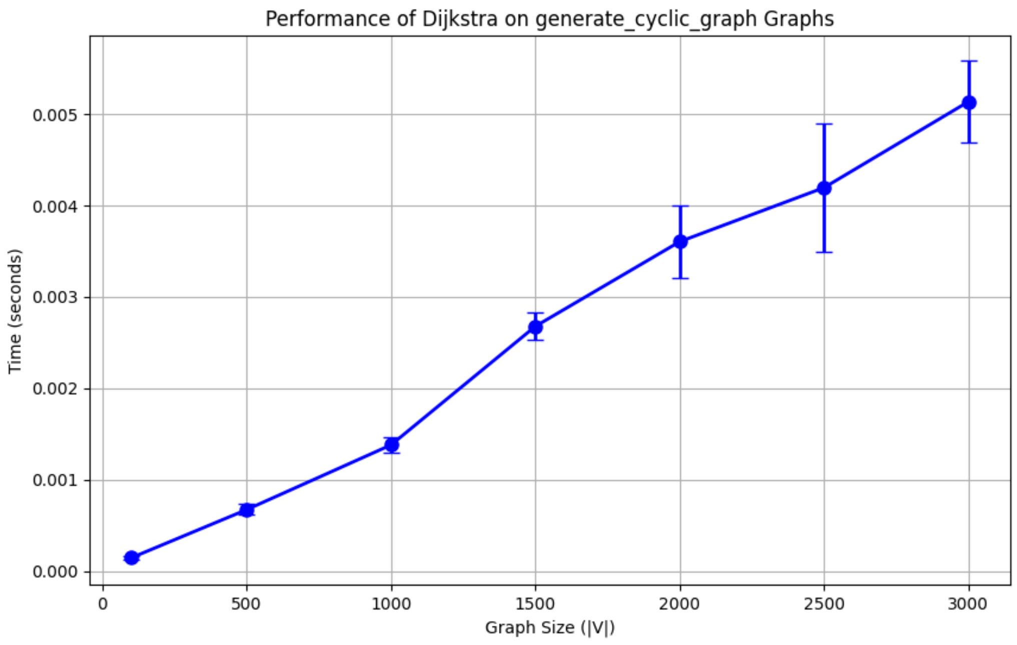
In connected graphs, Floyd-Warshall demonstrates cubic time complexity while Dijkstra shows more efficient scaling behavior.

For larger graph sizes, Dijkstra maintains good performance with execution time growing roughly proportionally to the product of edges and the logarithm of vertices.

The standard deviation increases somewhat with graph size but remains proportionally small, indicating reliable measurements across test repetitions.

3.6 Cyclic Graph





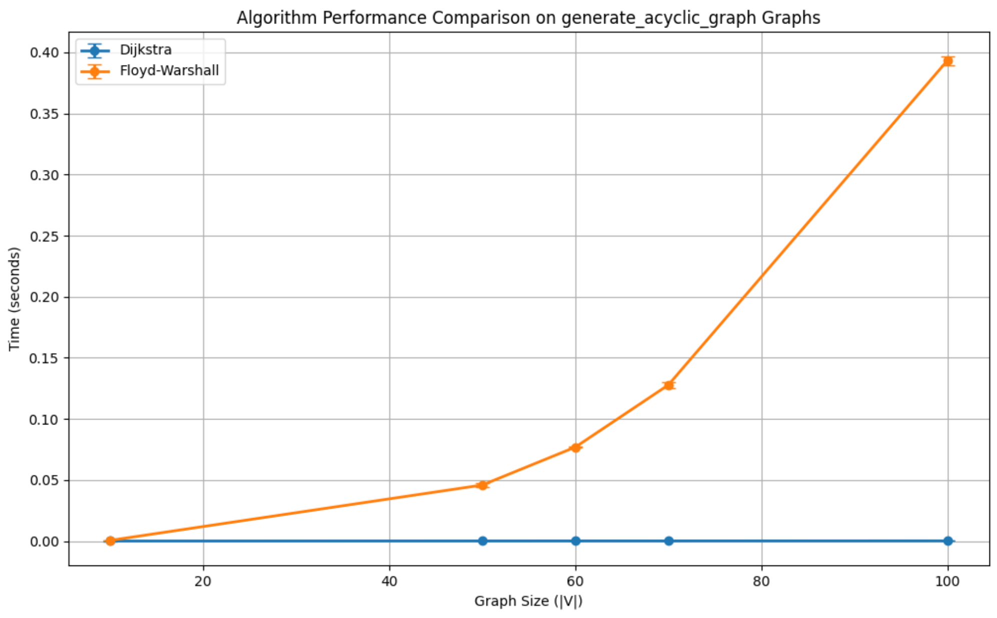
For cyclic graphs, Floyd-Warshall shows its characteristic O(V³) growth pattern while Dijkstra maintains more efficient scaling.

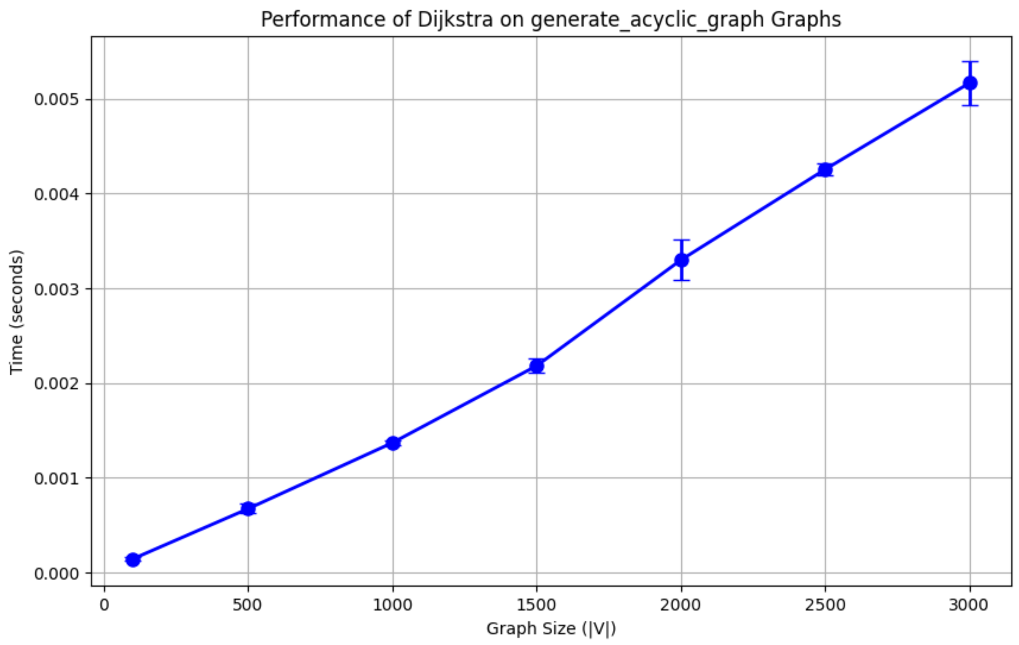
The comparison plot clearly demonstrates the increasing performance gap between the two algorithms as graph size grows, with Floyd-Warshall's execution time rising more steeply.

For larger graph sizes, Dijkstra continues to perform efficiently, handling cycles well due to its distance-tracking mechanism that prevents inefficient revisiting of vertices.

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

3.7 Acyclic Graph





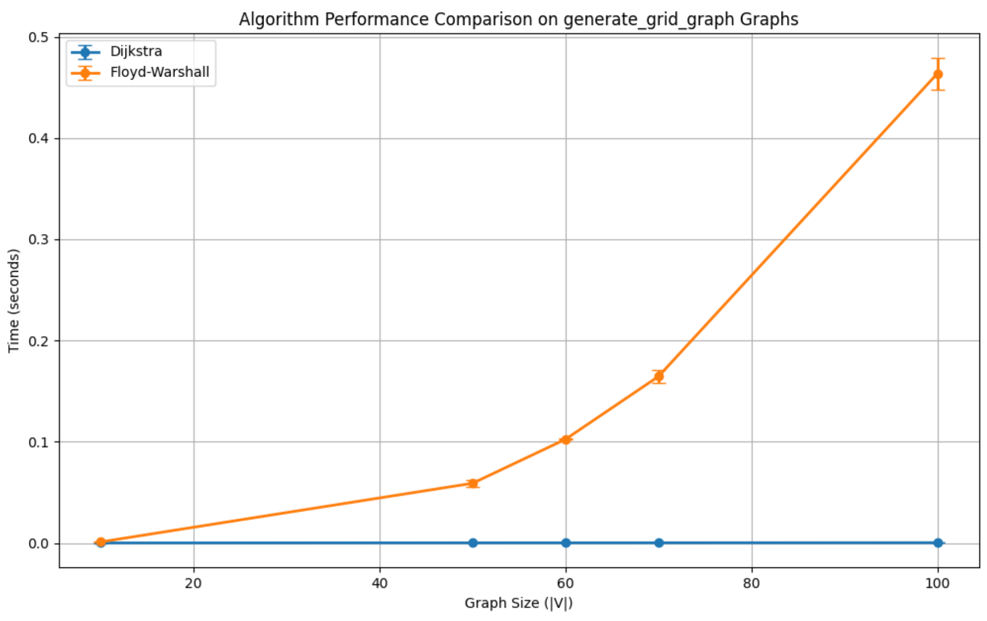
The acyclic graph tests show that Floyd-Warshall maintains its cubic complexity regardless of graph structure, while Dijkstra performs particularly well.

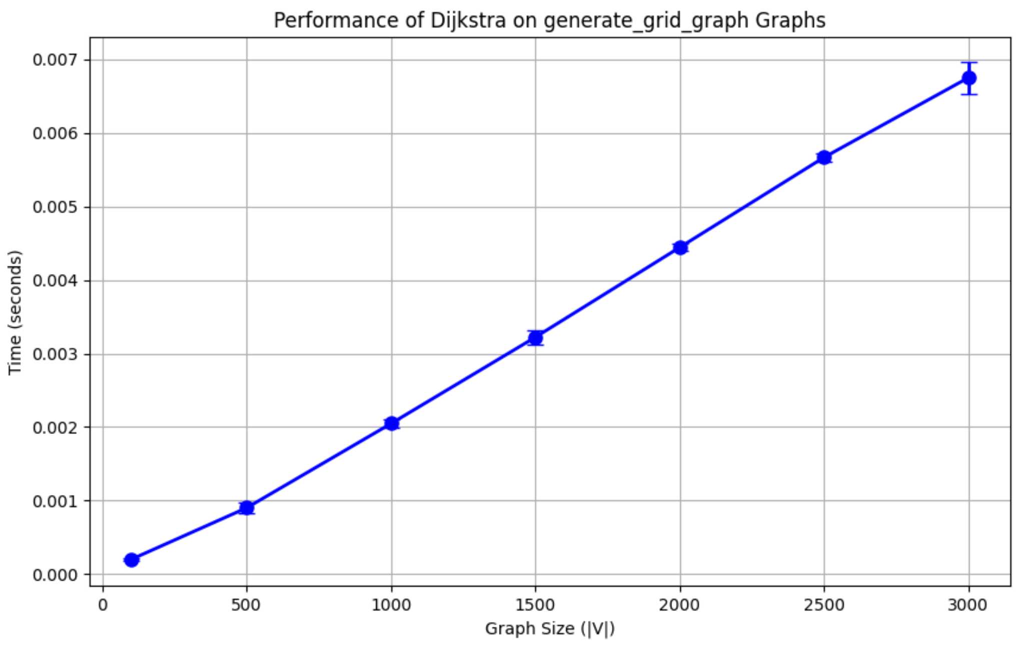
Despite some measurement fluctuations, there is a clear trend showing Dijkstra's superior performance across all graph sizes.

In the larger graph size tests, Dijkstra demonstrates excellent scalability with acyclic graphs since each vertex and edge needs to be processed at most once.

The error bars indicate some variability across repetitions, which may be attributed to specific characteristics of the generated acyclic graphs or system scheduling effects.

3.8 Grid Graph





Grid graphs show a clear performance difference between Dijkstra and Floyd-Warshall algorithms.

Both algorithms exhibit growth patterns consistent with their theoretical complexity, but Floyd-Warshall's cubic growth results in significantly higher execution times as graph size increases.

For larger grid sizes, Dijkstra maintains efficient performance due to its ability to prioritize promising paths through the priority queue implementation.

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

Dijkstra's advantage in grid graphs is particularly notable because the regular structure allows for efficient frontier expansion from the source vertex.

4 Conclusions

This lab work shows clear differences between Dijkstra's algorithm and the Floyd-Warshall algorithm when used on different types of graphs. Both solve shortest path problems but work very differently as graphs get bigger.

Floyd-Warshall always shows $O(V^3)$ time complexity no matter what the graph looks like. This matches what we expect from theory. The algorithm works by checking all possible paths through every vertex. It works well for finding all shortest paths at once, but becomes too slow for large graphs.

Dijkstra's algorithm works much better for large graphs, especially sparse ones and tree structures where it's almost linear in performance. The priority queue helps it process vertices efficiently by their current distance. For finding paths from a single starting point, Dijkstra is always faster than Floyd-Warshall in our tests.

As graphs get bigger, the speed difference between these algorithms grows very large. Dijkstra can handle thousands of vertices, while Floyd-Warshall becomes too slow after just a few hundred vertices.

Grid and tree graphs show interesting results, with Dijkstra working very well because it can focus on the most promising paths first. Complete and dense graphs are harder for both algorithms, but Dijkstra still performs better.

These results show why choosing the right algorithm matters. When you only need paths from one starting point, Dijkstra is better. When you need all possible paths and have small graphs, Floyd-Warshall is a good choice despite being slower.

This lab work shows that we need to study both the theoretical complexity and actual performance to choose the best algorithm for real problems.