**Comparing Iterative and Recursive Approaches in the**

**Floyd-Warshall Algorithm**

By

Ricardo Migliorini

Submitted to

The University of Liverpool

MASTER OF DATA SCIENCE AND ARTIFICIAL INTELLIGENCE

*Software Development*

Word Count: 1368

24/02/2024

**Comparing Iterative and Recursive Approaches in the**

**Floyd-Warshall Algorithm**

Submitted to

The University of Liverpool

Word Count: 1368

24/02/2024

TABLE OF CONTENTS

Page

[LIST OF FIGURES 2](#_Toc159684745)

[Chapter 1. Introduction 3](#_Toc159684746)

[Chapter 2. Background and Theorical Framework 4](#_Toc159684747)

[2.1 Floyd-Warshall Algorithm 4](#_Toc159684748)

[2.2 Recursive and Iterative Approachs 5](#_Toc159684749)

[2.3 Application Example 5](#_Toc159684750)

[Chapter 3. Methodology 6](#_Toc159684751)

[3.1 Implementation of the Floyd-Warshall Algorithm 6](#_Toc159684752)

[3.2 Test Design and Coverage 6](#_Toc159684753)

[3.3 Analytical Methods 7](#_Toc159684754)

[Chapter 4. Performance Analysis 8](#_Toc159684755)

[4.1 Detail Performance Metrics 8](#_Toc159684756)

[4.2 Detail Performance Metrics 8](#_Toc159684757)

[4.3 Detail Performance Metrics 9](#_Toc159684758)

[Chapter 5. Conclusion 10](#_Toc159684759)

[REFERENCES 11](#_Toc159684760)

[APPENDICES 12](#_Toc159684761)

LIST OF FIGURES

Page

[Figure 1. Illustration of the Floyd Warshall Algorithm Process 4](#_Toc159684737)

[Figure 2. Measurements Results of Iterative and Recursive Code Application 8](#_Toc159684738)

[Figure 3. Iterative Applications – Graph 1: Positive Edges 29](#_Toc159684739)

[Figure 4. Iterative Applications – Graph 2: Negative Edges 30](#_Toc159684740)

[Figure 5. Iterative Applications – Graph 3: Negative Cycles 31](#_Toc159684741)

[Figure 6. Recursive Applications – Graph 1: Positive Edges 32](#_Toc159684742)

[Figure 7. Recursive Applications – Graph 2: Negative Edges 33](#_Toc159684743)

[Figure 8. Recursive Applications – Graph 3: Negative Cycles 34](#_Toc159684744)

# Introduction

The Floyd-Warshall algorithm stands as a cornerstone in the realm of computer science, specifically within graph theory, for its ability to find the shortest paths between all pairs of vertices in a weighted graph.

Traditionally implemented in an iterative manner, this project explores a novel recursive approach. By leveraging recursion and memorization, this version aims not only to calculate shortest paths efficiently but also to offer insights into the comparative performance and conceptual distinctions between recursive and iterative methodologies. This exploration is crucial for understanding the adaptability and scalability of algorithmic solutions in addressing fundamental computational problems.

# Background and Theorical Framework

This chapter lays the foundation for understanding the Floyd-Warshall algorithm and the comparative study of its recursive and iterative implementations.

Floyd-Warshall Algorithm

Named after its creator Robert Floyd and Stephen Warshall, the Floyd-Warshall algorithm is a simple and widely used algorithm to compute shortest paths between all pairs of vertices in an edge weighted directed graph (Hougardy, 2010). The algorithm operates on a principle of dynamic programming, systematically updating distances between all pairs of vertices by considering each vertex as an intermediate waypoint.

Diagrama

Descrição gerada automaticamente

**Font: Adapted from "Floyd Warshall Algorithm | DP-16" at GeeksforGeeks (https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/).**

Figure 1. Illustration of the Floyd Warshall Algorithm Process

Due to its simplicity, the Floyd-Warshall algorithm is preferred over Dijkstra’s (applying it for every vertex as a source) or Johnson’s algorithms for solving all-pairs shortest paths problems most of the time. It can handle small-sized problems better than its alternatives, solving graphs with both positive and negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems (Toroslu, 2021).

The algorithm, however, is not applicable to graphs containing negative cycles, where the sum of the edges in a cycle is negative (Geeks for Geeks, n.d.). The Floyd-Warshall algorithm can detect negative cycles by observing if the distance from a vertex to itself becomes negative after the algorithm's execution (Hougardy, 2010). This is a critical feature as negative cycles can indicate the possibility of endlessly decreasing path lengths, which is problematic in practical applications.

Recursive and Iterative Approachs

Dynamic programming, a method for solving complex problems by breaking them down into simpler subproblems, can be implemented using either recursive or iterative approaches.

The Recursive Approach focuses on solving problems by calling a function within itself (Geeks for Geeks, n.d.), using previous results to simplify subsequent calculations. However, it can lead to higher memory usage due to the call stack and potentially slower execution times.

Alternatively, iterative solutions leverage loops to repetitively solve subproblems (Geeks for Geeks, n.d.). This approach is generally more memory-efficient and can offer faster execution times.

Application Example

A practical application of the Floyd-Warshall algorithm is seen in optimizing garbage transport routes in West Medan, Indonesia. The city's high population density results in significant waste generation, necessitating efficient route optimization for waste collection vehicles.

By applying the Floyd-Warshall algorithm alongside the Clarke & Wright Savings method, the study achieved a reduction in travel distances, thereby enhancing operational efficiency and reducing environmental impact. This example underscores the algorithm's utility in urban planning and public service management, demonstrating its capacity to contribute to more sustainable city logistics. (K Syahputri et al 2020).

# Methodology

This chapter delineates the methodologies adopted for the implementation, testing, and analysis of the Floyd-Warshall algorithm. It focuses on a comparative evaluation of recursive and iterative strategies, aiming to uncover their effectiveness in solving the all-pairs shortest path problem in weighted graphs.

Implementation of the Floyd-Warshall Algorithm

The Floyd-Warshall algorithm was implemented in two distinct approaches: recursive and iterative. Both versions were developed in Python due to its readability and support for both programming paradigms.

Recursive version utilizes self-calling functions, complemented by memoization, to efficiently compute shortest paths. This approach is designed to minimize redundant calculations, enhancing the algorithm's performance. Iterative implementation employs loop-based logic to systematically update the distance matrix, accurately determining the shortest paths between all vertex pairs. This method avoids the complexity of recursive calls, opting for a more straightforward computational process.

Test Design and Coverage

The tests were conducted using an 8x8 matrix showcased, serving as a uniform benchmark across tests to demonstrate Θ(N^3) complexity effectiveness. The tests covered three essential graph configurations scenarios of the algorithm, englobing graphs with positive and negative edges only, along with the detection of negative cycles.

To ensure the replicability of our study, detailed setup instructions and dependency requirements are provided in the accompanying README and REQUIREMENTS files within the project's code repository.

Analytical Methods

A comparative analysis was conducted to evaluate the performance differences between the recursive and iterative approaches. This analysis considered the execution time, measured by Python's “*time”* module capturing the duration of each algorithm's execution. The memory usage, assessed through the “*psutil”* library, identifying patterns particularly in relation to the graph's complexity and the implementation approach.

Furthermore, we measured the number of iterations and recursive calls to assess the computational overhead and to understand the impact of memorization on performance. Lastly, the scalability, was examined by observing how each approach adapted to increasing sizes of graphs, providing insight into their potential for handling larger, more complex datasets effectively.

# Performance Analysis

This chapter delves into a detailed examination of the Floyd-Warshall algorithm's recur-sive and iterative implementations, scrutinizing their efficiency, scalability, and applicabil-ity in real-world contexts through key metrics such as execution time, memory usage, and computational load. The results can be found in the figure below.

Tela azul com letras brancas

Descrição gerada automaticamente

Figure 2. Measurements Results of Iterative and Recursive Code Application

Detail Performance Metrics

The study's empirical data paint a nuanced picture of each approach's performance. In the Iterative Approach showed consistent and efficient execution times, with the longest being 0.0012 seconds for Graph 3. Memory usage for Graph 1 was 20,480 bytes, which, while significant, was less than half of the recursive method for the same graph. The consistent iteration count of 512 for all graphs reinforces the iterative method's stable performance.

In contrast, Recursive Approach demonstrated variable execution times, with the quickest being 0.00099 seconds for Graph 2 and the slowest at 0.0065 seconds for Graph 1. Memory usage peaked at 45,056 bytes for Graph 1, suggesting a significant resource requirement for complex calculations. Recursive calls ranged from 1,600 to 4,800, indicating an increasing computational load with graph complexity.

Detail Performance Metrics

The iterative approach consistently offered faster execution times and lower memory usage compared to the recursive approach, especially in complex graph scenarios. For instance, Graph 1 processed nearly twice as fast iteratively, while also utilizing less memory. This consistent efficiency is reinforced by a stable iteration count across all graph complexities, highlighting the iterative method’s adaptability.

Conversely, the recursive approach demonstrated variability in performance, with the slowest execution time and the highest memory usage occurring in the most complex graph. The increasing number of recursive calls from Graph 1 to Graph 3, ranging from 1,600 to 4,800, underscores a growing computational demand, which could impede scalability and efficiency in extensive graph analyses.

Detail Performance Metrics

The performance analysis indicates the iterative method's advantages in terms of scalability and efficiency. While the recursive approach may be beneficial for smaller graphs or educational purposes, the iterative method's consistent performance and lower resource consumption make it a more suitable choice for practical applications involving complex or large-scale graph analyses.

# Conclusion

The exploration of the Floyd-Warshall algorithm's recursive and iterative approaches concludes that theoretical expectations are mirrored in empirical results. Iterative methods, in line with dynamic programming principles, display consistent execution time and memory usage, indicative of their robustness for complex graph computations. These findings correlate with the case study on West Medan's garbage transport optimization, where the algorithm's practical application resulted in notable efficiencies in route planning.

The methodology applied, involving a meticulous evaluation using an 8x8 matrix, substantiates the algorithm's Θ(N^3) complexity and provides a tangible context for assessing scalability and performance. Such an approach not only demonstrates the iterative method's efficacy but also underscores the recursive method's limitations in handling increased complexity, despite its conceptual value in illustrating dynamic programming techniques.

Future research directions could delve into optimizing recursive algorithms to extend their utility or experiment with these approaches in different programming environments. The potential for these methods in addressing a broader spectrum of algorithmic problems is a promising avenue for future investigation.

In synthesizing the study's theoretical and practical insights, the report emphasizes the importance of selecting appropriate algorithmic strategies that align with specific computational demands. The iterative approach stands out for real-world applications requiring efficiency at scale, as demonstrated in the improvement of waste management systems. This study not only advances the academic dialogue on algorithm selection but also offers a framework for developing solutions attuned to the nuanced challenges of modern computational problems.

REFERENCES

1. Hougardy, S. (2010) 'The Floyd–Warshall algorithm on graphs with negative cycles'.
2. Syahputri, K. et al. (2020) 'Optimization of garbage transport routes in West Medan using Clarke & Weigh saving and Floyd Warshall algorithm', IOP Conference Series: Materials Science and Engineering, 801, p. 012029.
3. Toroslu, I.H. (2021) 'Improving The Floyd-Warshall All Pairs Shortest Paths Algorithm'.
4. Toroslu, I.H. (2021) 'The Floyd-Warshall all-pairs shortest paths algorithm for disconnected and very sparse graphs'.
5. 'Floyd Warshall Algorithm | DP-16' at GeeksforGeeks. Available at: <https://www.geeksforgeeks.org/floyd-warshall-algorithm-dp-16/> (Accessed: 22 February 2024).
6. 'Difference Between Recursion and Iteration' at GeeksforGeeks. Available at: [https://www.geeksforgeeks.org/difference-between-recursion-and-iteration/](https://www.geeksforgeeks.org/difference-between-recursion-and-iteration/%20) (Accessed: 22 February 2024).

APPENDICES

###### Iterative Python Code Application of Floyd-Warshall Algorithm

This code exemplifies the use of an iterative approach to execute the Floyd-Warshall algorithm in python. Python Code:

import matplotlib.pyplot as plt

import numpy as np

import sys

import psutil

import time

# Constants

NO\_PATH = sys.maxsize # Use sys.maxsize to represent no path in the graph

# Initialize counters for performance measurements

iterations = 0

def floyd\_warshall\_iterative(graph):

"""

Applies the Floyd-Warshall algorithm in an iterative way to find the shortest paths

between all pairs of nodes.

Args:

graph (list): The adjacency matrix of the graph.

Returns:

tuple: The distance matrix, a boolean indicating if a negative cycle was detected,

execution time, memory usage, and number of iterations.

"""

global iterations

iterations = 0 # Reset iteration count for this run

n = len(graph) # Number of vertices in the graph

# Initialize the distance matrix

dist = [

[NO\_PATH if graph[i][j] == 0 else graph[i][j] for j in range(n)]

for i in range(n)

]

for i in range(n):

dist[i][i] = 0 # Distance from a vertex to itself is 0

# Measure execution time and memory usage

start\_time = time.time()

memory\_before = psutil.Process().memory\_info().rss

# Main loop of the Floyd-Warshall algorithm

for k in range(n):

for i in range(n):

for j in range(n):

iterations += 1 # Count iterations (similar to recursive calls)

# Update distance matrix

dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

# Calculate execution time and memory usage

end\_time = time.time()

memory\_after = psutil.Process().memory\_info().rss

execution\_time = end\_time - start\_time

memory\_usage = memory\_after - memory\_before

# Check for negative cycles

for i in range(n):

if dist[i][i] < 0:

return dist, True, execution\_time, memory\_usage, iterations

return dist, False, execution\_time, memory\_usage, iterations

def visualize\_matrix(matrix, title):

"""

Visualizes a matrix with enhancements for infinity values and adds a color scale.

Args:

matrix (list): The matrix to visualize.

title (str): The title for the plot.

"""

# Replace NO\_PATH values with infinity for visualization

matrix = np.array(matrix, dtype=float)

matrix[matrix == NO\_PATH] = np.inf

# Create the plot

fig, ax = plt.subplots()

cax = ax.matshow(matrix, interpolation='nearest', cmap='viridis',

extent=[-0.5, matrix.shape[1]-0.5, matrix.shape[0]-0.5, -0.5])

# Add text labels for each cell

for (i, j), val in np.ndenumerate(matrix):

ax.text(j, i, '∞' if np.isinf(val) else f'{val:.0f}', ha='center', va='center',

color='black' if np.isinf(val) else 'white', fontsize=10)

# Add a color bar and set the title

fig.colorbar(cax)

ax.set\_title(title)

plt.xlabel('Destination Vertex')

plt.ylabel('Source Vertex')

# Setup ticks and labels

ax.set\_xticks(np.arange(-0.5, len(matrix), 1), minor=True)

ax.set\_yticks(np.arange(-0.5, len(matrix), 1), minor=True)

ax.set\_xticklabels(range(len(matrix)))

ax.set\_yticklabels(range(len(matrix)))

# Add note for infinity representation

plt.figtext(0.99, 0.01, '∞ indicates no path (infinity)', horizontalalignment='right')

ax.grid(which='minor', color='black', linestyle='-', linewidth=2)

plt.tight\_layout()

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

# Define and visualize test graphs for the Floyd-Warshall algorithm

# Test 1: Graph with Positive Weights

graph\_positive = [

[0, 4, NO\_PATH, NO\_PATH, NO\_PATH, 10, NO\_PATH, 8],

[NO\_PATH, 0, 2, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 11],

[NO\_PATH, NO\_PATH, 0, 6, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 0, 3, NO\_PATH, NO\_PATH, NO\_PATH],

[7, NO\_PATH, NO\_PATH, NO\_PATH, 0, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0, 1, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 5, NO\_PATH, 8, 0, NO\_PATH],

[NO\_PATH, NO\_PATH, 4, NO\_PATH, 2, NO\_PATH, NO\_PATH, 0]

]

print("Test 1: Graph with Positive Weights")

visualize\_matrix(graph\_positive, "Initial Distance Matrix - Test 1")

result, has\_negative\_cycle, execution\_time, memory\_usage, iterations = floyd\_warshall\_iterative(graph\_positive)

if not has\_negative\_cycle:

visualize\_matrix(result, "Final Distance Matrix - Test 1")

print(f"Execution Time: {execution\_time} seconds")

print(f"Memory Usage: {memory\_usage} bytes")

print(f"Iterations: {iterations}")

# Test 2: Graph with Negative Weights (No Negative Cycle)

graph\_negative\_weights = [

[0, 2, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 1],

[NO\_PATH, 0, 3, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, 4, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 0, 5, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0, -1, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0, -1, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0, -1],

[-1, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0]

]

print("\nTest 2: Graph with Negative Weights")

visualize\_matrix(graph\_negative\_weights, "Initial Distance Matrix - Test 2")

result, has\_negative\_cycle, execution\_time, memory\_usage, iterations = floyd\_warshall\_iterative(graph\_negative\_weights)

if not has\_negative\_cycle:

visualize\_matrix(result, "Final Distance Matrix - Test 2")

print(f"Execution Time: {execution\_time} seconds")

print(f"Memory Usage: {memory\_usage} bytes")

print(f"Iterations: {iterations}")

# Test 3: Graph with a Negative Cycle

graph\_negative\_cycle = [

[0, 1, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 2],

[2, 0, 3, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, -4, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 0, -1, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, 1, NO\_PATH, 0, 3, NO\_PATH, NO\_PATH],

[NO\_PATH, -2, NO\_PATH, NO\_PATH, NO\_PATH, 0, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 2, NO\_PATH, NO\_PATH, 0, 1],

[-1, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, -3, 0]

]

print("\nTest 3: Graph with a Negative Cycle")

visualize\_matrix(graph\_negative\_cycle, "Initial Distance Matrix - Test 3")

result, has\_negative\_cycle, execution\_time, memory\_usage, iterations = floyd\_warshall\_iterative(graph\_negative\_cycle)

if has\_negative\_cycle:

print("Negative cycle detected. No shortest path matrix available.")

print(f"Execution Time: {execution\_time} seconds")

print(f"Memory Usage: {memory\_usage} bytes")

print(f"Iterations: {iterations}")

###### Recursive Python Code Application of Floyd-Warshall Algorithm

This code exemplifies the use of a recursive approach to execute the Floyd-Warshall algorithm in python. Python Code:

import matplotlib.pyplot as plt

import numpy as np

import sys

import psutil

import time

from matplotlib.ticker import FixedLocator

# Constants

NO\_PATH = sys.maxsize # Represents an infinite distance indicating no path between nodes

# Global variables for measuring execution time, memory usage, and counters

recursive\_calls = 0

iterations = 0

def recursive\_update(dist, i, j, k, memo):

"""

Recursively updates the distance matrix for the Floyd-Warshall algorithm.

Args:

dist (list): The distance matrix.

i (int): The source node index.

j (int): The target node index.

k (int): The intermediate node index.

memo (dict): Memorization dictionary to store intermediate results.

Returns:

int: The shortest path from i to j considering nodes up to k.

"""

global recursive\_calls

recursive\_calls += 1

if (i, j, k) in memo:

return memo[(i, j, k)]

if k == -1:

return dist[i][j]

without\_k = recursive\_update(dist, i, j, k - 1, memo)

with\_k = recursive\_update(dist, i, k, k - 1, memo) + recursive\_update(dist, k, j, k - 1, memo)

memo[(i, j, k)] = min(without\_k, with\_k)

return memo[(i, j, k)]

def floyd\_warshall\_recursive(graph):

"""

Applies the Floyd-Warshall algorithm recursively to find the shortest paths between all pairs of nodes.

Args:

graph (list): The adjacency matrix of the graph.

Returns:

tuple: The distance matrix, a boolean indicating if a negative cycle was detected,

execution time, and memory usage.

"""

n = len(graph)

dist = [[NO\_PATH if graph[i][j] == 0 else graph[i][j] for j in range(n)] for i in range(n)]

for i in range(n):

dist[i][i] = 0

memo = {}

global iterations

iterations = n\*\*3 # Total iterations in Floyd-Warshall

start\_time = time.time()

memory\_before = psutil.Process().memory\_info().rss

for i in range(n):

for j in range(n):

dist[i][j] = recursive\_update(dist, i, j, n-1, memo)

end\_time = time.time()

memory\_after = psutil.Process().memory\_info().rss

execution\_time = end\_time - start\_time

memory\_usage = memory\_after - memory\_before

for i in range(n):

if dist[i][i] < 0:

print("Negative cycle detected.")

return None, True, execution\_time, memory\_usage

return dist, False, execution\_time, memory\_usage

def visualize\_matrix(matrix, title):

"""

Visualizes a matrix with enhancements for infinity values and adds a color scale.

Args:

matrix (list): The matrix to visualize.

title (str): The title for the plot.

"""

matrix = np.array(matrix, dtype=float) # Convert matrix to float type for visualization

matrix[matrix == NO\_PATH] = np.inf # Replace NO\_PATH values with infinity

fig, ax = plt.subplots()

cax = ax.matshow(matrix, interpolation='nearest', cmap='viridis',

extent=[-0.5, matrix.shape[1]-0.5, matrix.shape[0]-0.5, -0.5])

for (i, j), val in np.ndenumerate(matrix):

ax.text(j, i, '∞' if np.isinf(val) else f'{val:.0f}', ha='center', va='center',

color='black' if np.isinf(val) else 'white', fontsize=10)

fig.colorbar(cax)

ax.set\_title(title)

plt.xlabel('Destination Vertex')

plt.ylabel('Source Vertex')

ax.set\_xticks(np.arange(-0.5, len(matrix), 1), minor=True)

ax.set\_yticks(np.arange(-0.5, len(matrix), 1), minor=True)

ax.set\_xticklabels(range(len(matrix)))

ax.set\_yticklabels(range(len(matrix)))

plt.figtext(0.99, 0.01, '∞ indicates no path (infinity)', horizontalalignment='right')

ax.grid(which='minor', color='black', linestyle='-', linewidth=2)

plt.tight\_layout()

plt.show()

if \_\_name\_\_ == "\_\_main\_\_":

# Define and visualize test graphs for the Floyd-Warshall algorithm

# Test 1: Graph with Positive Weights

graph\_positive = [

[0, 4, NO\_PATH, NO\_PATH, NO\_PATH, 10, NO\_PATH, 8],

[NO\_PATH, 0, 2, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 11],

[NO\_PATH, NO\_PATH, 0, 6, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 0, 3, NO\_PATH, NO\_PATH, NO\_PATH],

[7, NO\_PATH, NO\_PATH, NO\_PATH, 0, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0, 1, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 5, NO\_PATH, 8, 0, NO\_PATH],

[NO\_PATH, NO\_PATH, 4, NO\_PATH, 2, NO\_PATH, NO\_PATH, 0]

]

print("Test 1: Graph with Positive Weights")

visualize\_matrix(graph\_positive, "Initial Distance Matrix - Test 1")

result, has\_negative\_cycle, execution\_time, memory\_usage = floyd\_warshall\_recursive(graph\_positive)

if not has\_negative\_cycle:

visualize\_matrix(result, "Final Distance Matrix - Test 1")

print("Execution Time:", execution\_time, "seconds")

print("Memory Usage:", memory\_usage, "bytes")

print("Recursive Calls:", recursive\_calls)

print("Iterations:", iterations)

# Test 2: Graph with Negative Weights (No Negative Cycle)

graph\_negative\_weights = [

[0, 2, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 1],

[NO\_PATH, 0, 3, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, 4, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 0, 5, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0, -1, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0, -1, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0, -1],

[-1, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 0]

]

print("\nTest 2: Graph with Negative Weights")

visualize\_matrix(graph\_negative\_weights, "Initial Distance Matrix - Test 2")

result, has\_negative\_cycle, execution\_time, memory\_usage = floyd\_warshall\_recursive(graph\_negative\_weights)

if not has\_negative\_cycle:

visualize\_matrix(result, "Final Distance Matrix - Test 2")

print("Execution Time:", execution\_time, "seconds")

print("Memory Usage:", memory\_usage, "bytes")

print("Recursive Calls:", recursive\_calls)

print("Iterations:", iterations)

# Test 3: Graph with a Negative Cycle

graph\_negative\_cycle = [

[0, 1, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, 2],

[2, 0, 3, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, 0, -4, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 0, -1, NO\_PATH, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, 1, NO\_PATH, 0, 3, NO\_PATH, NO\_PATH],

[NO\_PATH, -2, NO\_PATH, NO\_PATH, NO\_PATH, 0, NO\_PATH, NO\_PATH],

[NO\_PATH, NO\_PATH, NO\_PATH, 2, NO\_PATH, NO\_PATH, 0, 1],

[-1, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, NO\_PATH, -3, 0]

]

print("\nTest 3: Graph with a Negative Cycle")

visualize\_matrix(graph\_negative\_cycle, "Initial Distance Matrix - Test 3")

result, has\_negative\_cycle, execution\_time, memory\_usage = floyd\_warshall\_recursive(graph\_negative\_cycle)

if has\_negative\_cycle:

print("Negative cycle detected. No shortest path matrix available.")

print("Execution Time:", execution\_time, "seconds")

print("Memory Usage:", memory\_usage, "bytes")

print("Recursive Calls:", recursive\_calls)

print("Iterations:", iterations)

###### Graphs and Measurements of Iterative Approach

Aplicativo, Tabela

Descrição gerada automaticamente com confiança média

Figure 3. Iterative Applications – Graph 1: Positive Edges

Tabela

Descrição gerada automaticamente

Figure 4. Iterative Applications – Graph 2: Negative Edges

Tabela

Descrição gerada automaticamente

Figure 5. Iterative Applications – Graph 3: Negative Cycles

###### Graphs and Measurements of Recursive Approach

Tabela

Descrição gerada automaticamente com confiança média

Figure 6. Recursive Applications – Graph 1: Positive Edges

Tabela, Calendário

Descrição gerada automaticamente

Figure 7. Recursive Applications – Graph 2: Negative Edges

Tabela

Descrição gerada automaticamente

Figure 8. Recursive Applications – Graph 3: Negative Cycles