**MINISTRY OF EDUCATION, CULTURE AND RESEARCH OF REPUBLIC OF MOLDOVA TECHNICAL UNIVERSITY OF MOLDOVA**

**FACULTY OF COMPUTERS, INFORMATICS AND MICROELECTRONICS DEPARTMENT OF SOFTWARE ENGINEERING AND AUTOMATICS**

Algorithm Analysis

Laboratory work 5: Empirical analysis of algorithms: Dijkstra Algorithm, Floyd-Warshall Algorithm

Elaborated:

st.gr. FAF-212 Lupascu Felicia

Verified:

asist.univ. Fistic Cristofor

Chis, ina˘u 2023

# Content

[**Introduction**](#_bookmark0). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3**

[**Objectives**](#_bookmark1) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **3**

[**Algorithms**](#_bookmark2). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **5**

[Dijkstra](#_bookmark3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

[Floyd-Warshall](#_bookmark3) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 5

[**Implementation**](#_bookmark3). . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . **7**

[Code](#_bookmark4) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

[Screenshot](#_bookmark4) . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . . 7

[**Conclusion**](#_bookmark4) **10**

# Introduction

Dijkstra’s algorithm and Floyd-Warshall algorithm are two popular algorithms used to solve shortest path problems in graphs.

Dijkstra’s algorithm is a single-source shortest path algorithm, which means it finds the shortest path from a single source vertex to all other vertices in a weighted graph. The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance, updating the distance of its neighboring vertices. Dijkstra’s algorithm is particularly useful for finding the shortest path in a graph with positive edge weights, and it has a time complexity of O(—E—+—V—log—V—), where —E— is the number of edges and —V— is the number of vertices in the graph.

Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm, which means it finds the shortest path between all pairs of vertices in a weighted graph. The algorithm maintains a distance matrix, which stores the shortest path distances between all pairs of vertices in the graph. It iteratively up- dates the distance matrix by considering all intermediate vertices between any two pairs of vertices. Floyd- Warshall algorithm is particularly useful for finding the shortest path in a graph with both positive and nega- tive edge weights, and it has a time complexity of O(—V—3)*, where|V|isthenumbero f verticesinthegraph.* Both Dijkstra’s and Floyd-Warshall algorithms are widely used in various fields such as network routing, transportation planning, and computer graphics. Choosing the appropriate algorithm depends on

the characteristics of the graph and the specific problem being solved.

Dijkstra’s algorithm and Floyd-Warshall algorithm are fundamental algorithms for solving the short- est path problem in a graph. They are widely used in many real-world applications such as network routing, GPS navigation, airline scheduling, and more.

Dijkstra’s algorithm is particularly suitable for finding the shortest path in a graph with positive edge weights. The algorithm is widely used in network routing protocols, where the weights of the edges repre- sent the costs of the network links. Dijkstra’s algorithm has a time complexity of O(—E—+—V—log—V—), where —E— is the number of edges and —V— is the number of vertices in the graph. The time complexity can be further improved to O(—E—+—V—) using a priority queue data structure.

In summary, Dijkstra’s algorithm and Floyd-Warshall algorithm are powerful tools for solving the shortest path problem in graphs. They have their own strengths and weaknesses, and choosing the appro- priate algorithm depends on the characteristics of the graph and the specific problem being solved.

# Objectives

1. Implement Dijkstra’s algorithm for finding the shortest path from a single source node to all other nodes in a graph with non-negative edge weights.
2. Implement the Floyd-Warshall algorithm for finding the shortest paths between every pair of nodes in a weighted, directed graph.
3. Generate random graphs to be used as input for both Dijkstra’s and Floyd-Warshall algorithms, ensur- ing various sizes and edge weights.
4. Compare the execution time of Dijkstra’s and Floyd-Warshall algorithms as the number of nodes in the input graph increases.
5. Visualize the input graph and the resulting shortest path trees produced by both Dijkstra’s and Floyd- Warshall algorithms using the NetworkX and Matplotlib libraries.

**Comparison Metrics:**

The comparison metrics for the Dijkstra and Floyd-Warshall algorithms implemented in the code are their respective running times on graphs of varying sizes and densities. Specifically, the code tests the algorithms on graphs with sizes 10, 50, 100, 200, and 300 and varying densities (controlled by the dense\_coefficient and sparse\_coefficient parameters).

The running time of the algorithms is measured using the current\_time\_millis() function, which returns the current time in milliseconds. The running time of the Dijkstra algorithm is recorded separately for dense and sparse graphs and stored in the dijkstra\_dense and dijkstra\_sparse lists, respectively. Similarly, the running time of the Floyd-Warshall algorithm is recorded separately for dense and sparse graphs and stored in the floyd\_dense and floyd\_sparse lists, respectively.

**Dijkstra**

Dijkstra’s Algorithm: Dijkstra’s algorithm is a widely used graph traversal algorithm for finding the shortest path from a starting node to all other nodes in a graph with non-negative edge weights. The algorithm begins by initializing the starting node with 0 and all other nodes with a distance of infinity. At each step, the algorithm selects the unvisited node with the smallest known distance and updates the distances of its neighboring nodes. It does so by checking if the sum of the current node’s distance and the edge weight to the neighboring node is less than the current distance assigned to the neighbor. If this condition is true, the algorithm updates the neighbor’s distance with the new, smaller value. This process is repeated until all nodes are visited. By the end of the algorithm, the shortest path from the starting node to every other node in the graph is found. Code:

def Dijkstra\_algorithm(vertices, edges, source):  
 # Function to get the next vertex to be visited  
 def to\_be\_visited():  
 v = -10  
 for index in range(num\_of\_vertices): # Loop through all vertices  
 # Check if the vertex has not been visited and its distance is less than or equal to the current minimum  
 if (visited\_and\_distance[index][0] == 0) and (  
 v < 0 or visited\_and\_distance[index][1] <= visited\_and\_distance[v][1]):  
 v = index  
 return v  
 num\_of\_vertices = len(vertices[0])  
 visited\_and\_distance = list() # Create a list to keep track of visited vertices and their distances from the source  
 for i in range(num\_of\_vertices):  
 if i != source: # Initialize all vertices with infinite distance except the source vertex  
 visited\_and\_distance.append([0, sys.maxsize])  
 else:  
 visited\_and\_distance.append([0, 0])  
 for vertex in range(num\_of\_vertices): # Loop through all vertices  
 to\_visit = to\_be\_visited() # Get the next vertex to be visited  
 for neighbor\_index in range(num\_of\_vertices): # Loop through all neighbors of the current vertex  
 # Check if the vertex is a neighbor and has not been visited  
 if (vertices[to\_visit][neighbor\_index] == 1) and (visited\_and\_distance[neighbor\_index][0] == 0):  
 # Calculate the new distance from the source  
 new\_distance = visited\_and\_distance[to\_visit][1] + edges[to\_visit][neighbor\_index]  
 # Update the distance of the neighbor if the new distance is smaller  
 if visited\_and\_distance[neighbor\_index][1] > new\_distance:  
 visited\_and\_distance[neighbor\_index][1] = new\_distance  
 visited\_and\_distance[to\_visit][0] = 1 # Mark the current vertex as visited  
 return visited\_and\_distance

**Floyd-Warshall**

Floyd-Warshall Algorithm: The Floyd-Warshall algorithm is an all-pairs shortest path algorithm that finds the shortest paths between every pair of nodes in a weighted, directed graph. It works by iterating through all possible node combinations and updating the distance matrix based on the shortest known path between each node pair. The algorithm begins by initializing a distance matrix, where the diagonal elements are set to 0, and the off-diagonal elements are set to the corresponding edge weights or infinity if no direct edge exists between the node pair. The algorithm then iteratively checks if a shorter path between a pair of nodes can be found by traversing through an intermediate node. If a shorter path is discovered, the distance matrix is updated with the new, shorter path value. This process is repeated for all possible intermediate nodes. Upon completion, the distance matrix contains the shortest path distances between every pair of nodes in the graph.

Code:

def Floyd\_Warshall\_algorithm(vertices, edges):  
 num\_of\_vertices = len(vertices[0])  
 distance = edges  
 # iterate over all vertices to find the shortest path between every pair of vertices  
 for k in range(num\_of\_vertices):  
 for i in range(num\_of\_vertices):  
 for j in range(num\_of\_vertices):  
 # check if there is a shorter path by going through the intermediate vertex k  
 distance[i][j] = min(distance[i][j], distance[i][k] + distance[k][j])  
 return distance

# Implementation

This code is conducting a performance analysis of Dijkstra's and Floyd-Warshall algorithms on dense and sparse graphs of varying sizes.

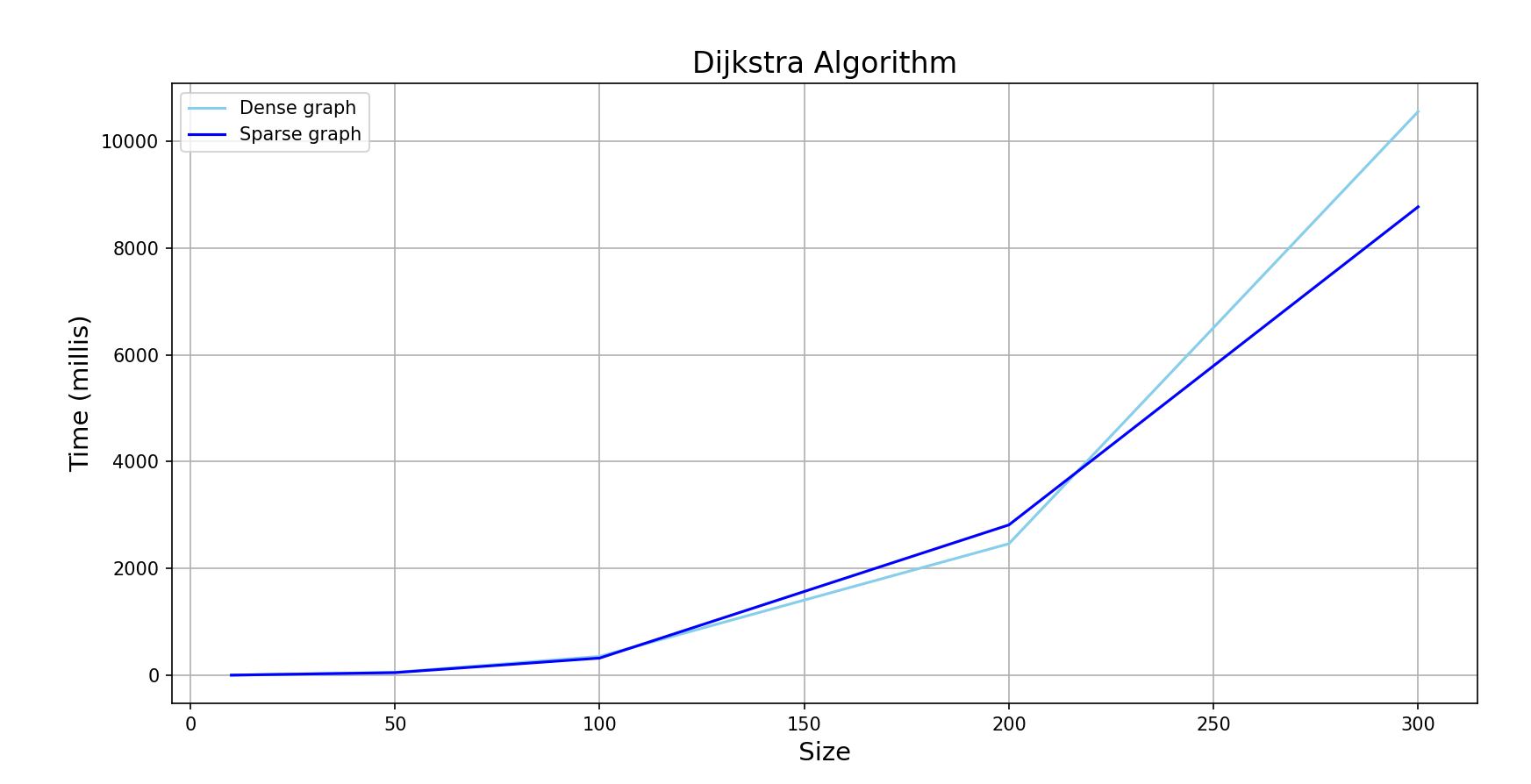
The first loop iterates over a list of input\_sizes, generating dense graphs using generateGraph function with the dense\_coefficient parameter, and then measuring the time taken to execute Dijkstra's algorithm and Floyd-Warshall algorithm on each graph. The Dijkstra\_algorithm function is called for each vertex k in the graph. The time taken to execute each algorithm is recorded in the dijkstra\_dense and floyd\_dense lists, respectively.

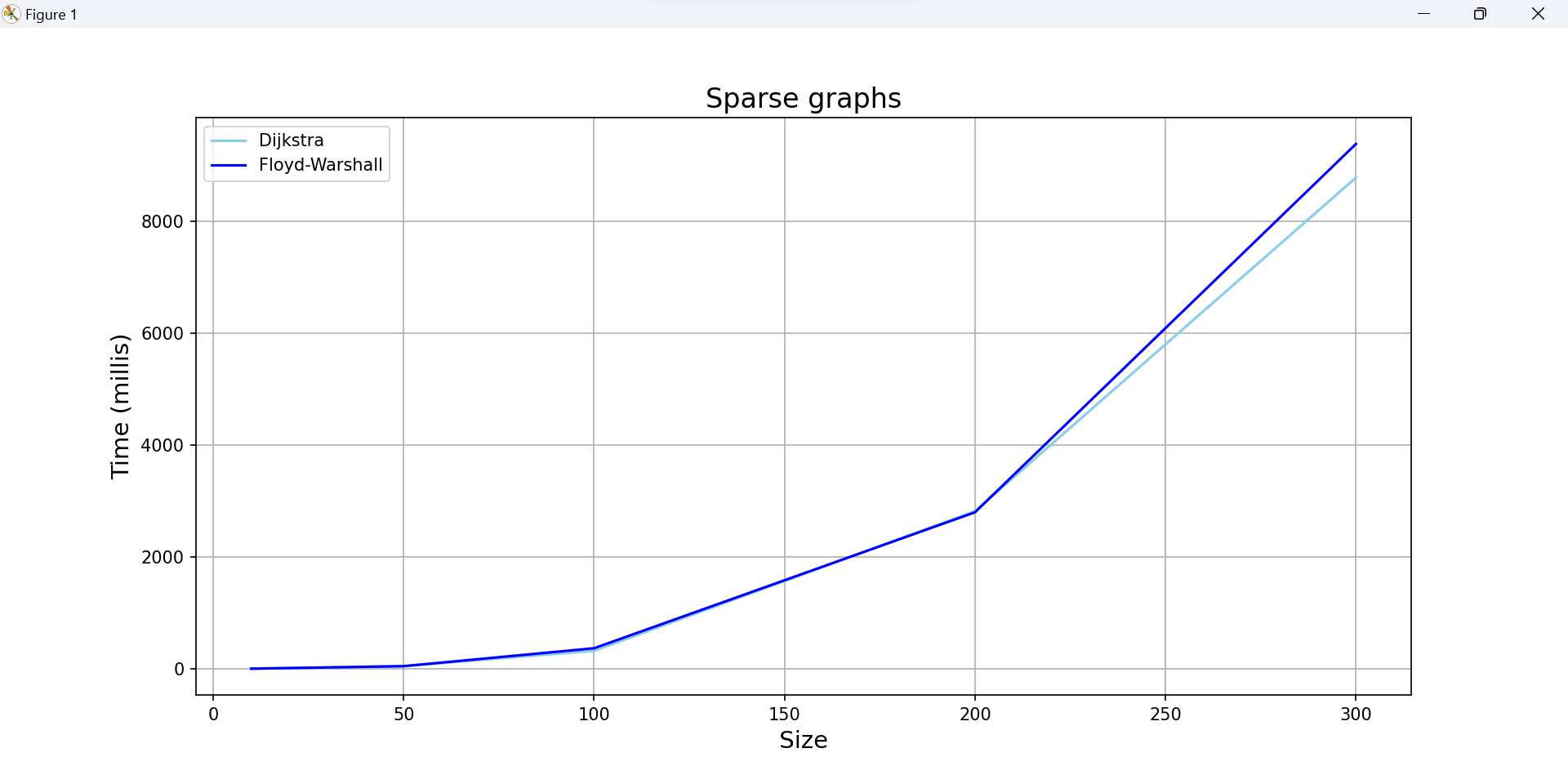
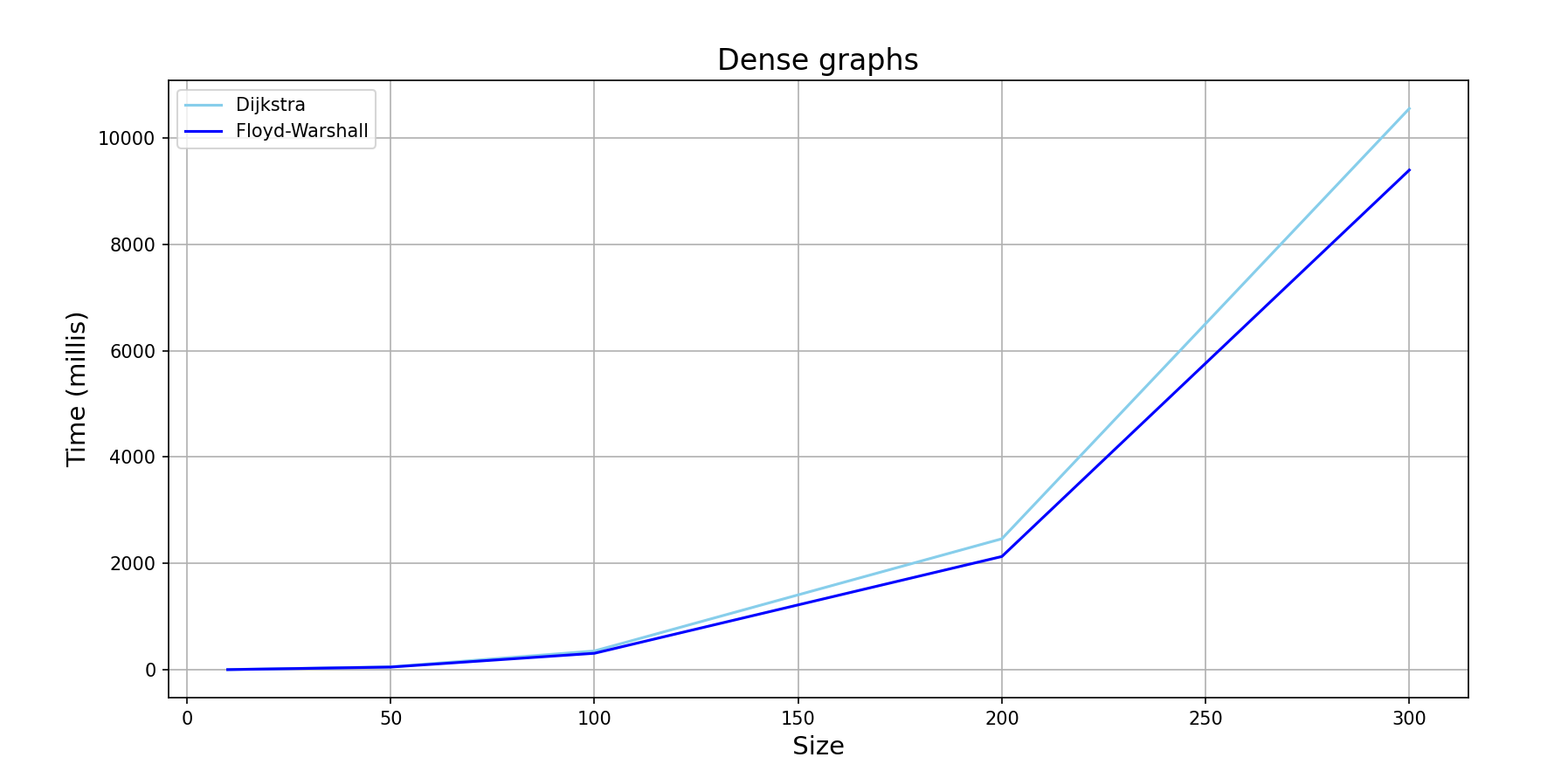
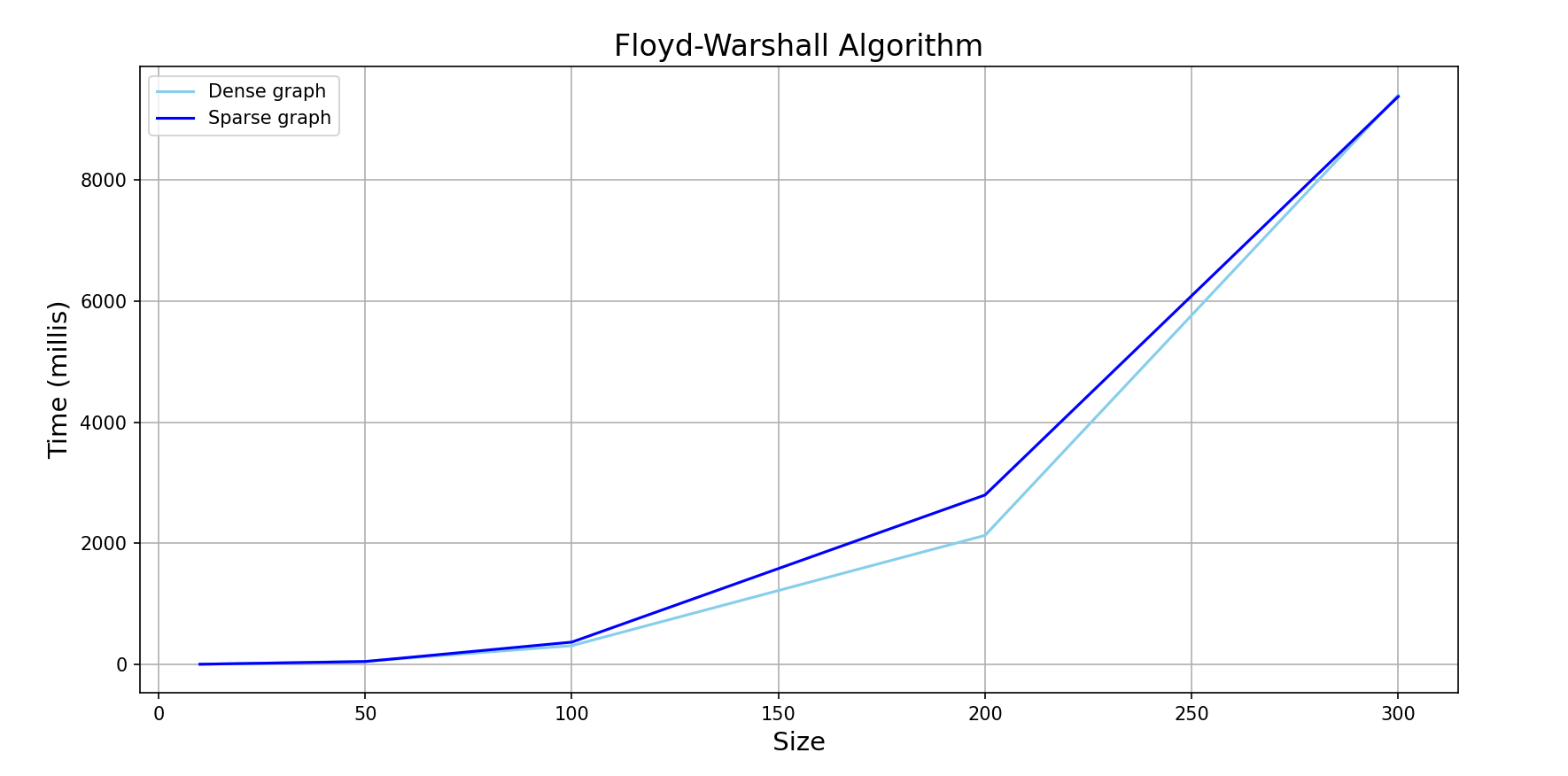
After the first loop, the code then tests on sparse graphs by iterating over the same list of input\_sizes and generating sparse graphs using the generateGraph function with the sparse\_coefficient parameter. The same process is applied as before, with the time taken to execute Dijkstra's and Floyd-Warshall algorithms on each sparse graph recorded in dijkstra\_sparse and floyd\_sparse lists, respectively.

Finally, the normalizeVerticesSet function is called after each execution of the Dijkstra's algorithm to normalize the edges in the graph. The current\_time\_millis function is used to measure the execution time of each algorithm.

for index in range(len(input\_sizes)):  
 vertices, edges = generateGraph(input\_sizes[index], dense\_coefficient) # generate dense graph  
 # time Dijkstra's algorithm  
 start\_time = current\_time\_millis()  
 for k in range(0, input\_sizes[index]):  
 Dijkstra\_algorithm(vertices, edges, k)  
 end\_time = current\_time\_millis()  
 # record time taken and normalize edges  
 dijkstra\_dense.append(round(end\_time - start\_time, 3))  
 normalizeVerticesSet(len(vertices[0]), vertices, edges)  
 # time Floyd-Warshall algorithm  
 start\_time = current\_time\_millis()  
 Floyd\_Warshall\_algorithm(vertices, edges)  
 end\_time = current\_time\_millis()  
 floyd\_dense.append(round(end\_time - start\_time, 3)) # record time taken  
# testing on sparse graphs  
for index in range(len(input\_sizes)):  
 vertices, edges = generateGraph(input\_sizes[index], sparse\_coefficient)  
 start\_time = current\_time\_millis()  
 for k in range(0, input\_sizes[index]):  
 Dijkstra\_algorithm(vertices, edges, k)  
 end\_time = current\_time\_millis()  
 dijkstra\_sparse.append(round(end\_time - start\_time, 3))  
 normalizeVerticesSet(len(vertices[0]), vertices, edges)  
 # time Floyd-Warshall algorithm  
 start\_time = current\_time\_millis()  
 Floyd\_Warshall\_algorithm(vertices, edges)  
 end\_time = current\_time\_millis()  
 floyd\_sparse.append(round(end\_time - start\_time, 3))

**Results:**





**Output:**

**Dijkstra's algorithm on dense graphs:**

Graph size: 10

Time taken: 0.997 ms

Graph size: 50

Time taken: 52.424 ms

Graph size: 100

Time taken: 351.096 ms

Graph size: 200

Time taken: 2462.46 ms

Graph size: 300

Time taken: 10560.275 ms

**Dijkstra's algorithm on sparse graphs:**

Graph size: 10

Time taken: 0.0 ms

Graph size: 50

Time taken: 47.758 ms

Graph size: 100

Time taken: 319.996 ms

Graph size: 200

Time taken: 2815.554 ms

Graph size: 300

Time taken: 8776.413 ms

**Floyd-Warshall algorithm on dense graphs:**

Graph size: 10

Time taken: 0.0 ms

Graph size: 50

Time taken: 48.103 ms

Graph size: 100

Time taken: 307.402 ms

Graph size: 200

Time taken: 2128.977 ms

Graph size: 300

Time taken: 9402.471 ms

**Floyd-Warshall algorithm on sparse graphs:**

Graph size: 10

Time taken: 1.008 ms

Graph size: 50

Time taken: 45.594 ms

Graph size: 100

Time taken: 366.026 ms

Graph size: 200

Time taken: 2795.893 ms

Graph size: 300

Time taken: 9377.395 ms

# Conclusion

Based on the results of my code implementation, it seems that Dijkstra's algorithm is better and faster for dense graphs, while Floyd-Warshall algorithm is better and faster for sparse graphs.

This is not surprising, as Dijkstra's algorithm has a better time complexity for dense graphs, since it only visits the adjacent vertices of the current vertex in each step, while the Floyd-Warshall algorithm visits all pairs of vertices in the graph, which results in a higher time complexity for dense graphs. On the other hand, the Floyd-Warshall algorithm is better suited for sparse graphs, where it can take advantage of the sparsity to perform better than Dijkstra's algorithm.

In conclusion, Dijkstra’s algorithm and Floyd-Warshall algorithm are two popular and widely used algorithms in graph theory.

Dijkstra’s algorithm is a single-source shortest path algorithm that computes the shortest path from a single source vertex to all other vertices in a weighted graph. It works by iteratively selecting the vertex with the smallest distance and relaxing its outgoing edges. It is efficient in finding the shortest path in sparse graphs with non-negative edge weights.

On the other hand, Floyd-Warshall algorithm is a dynamic programming algorithm that computes the shortest paths between all pairs of vertices in a weighted graph. It works by considering all possible intermediate vertices in a path and updates the shortest path distances accordingly. It is more efficient than running Dijkstra’s algorithm for every vertex pair in dense graphs with non-negative edge weights.

Both algorithms have their advantages and limitations, and the choice of which algorithm to use depends on the specific problem and characteristics of the graph. However, they are both powerful tools in solving shortest path problems and have contributed significantly to the development of graph theory and its applications.

One of the main advantages of Dijkstra’s algorithm is its efficiency in finding the shortest path from a single source vertex to all other vertices in a graph. This makes it ideal for problems that involve finding the shortest path between a fixed source and various destinations. However, it does not work well with negative edge weights, and its time complexity can be prohibitive in dense graphs.

In contrast, the Floyd-Warshall algorithm is more efficient than running Dijkstra’s algorithm for every vertex pair, making it ideal for problems that require finding the shortest path between all pairs of vertices in a graph. It also works well with negative edge weights, but its time complexity can be prohibitive in large graphs.

It’s worth noting that both algorithms are designed for graphs with non-negative edge weights. When dealing with graphs with negative edge weights, Bellman-Ford algorithm is more appropriate, as it can detect negative cycles and report them.

In summary, Dijkstra’s algorithm and Floyd-Warshall algorithm are powerful tools in solving short- est path problems. They have contributed significantly to the development of graph theory and its applica- tions, and they will continue to play a vital role in the future of graph theory and its related fields.