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 : Dynamic programming

Elaborated:
st.gr. FAF-211 Grama Alexandru
Verified:
asist.univ. Fiștic Cristofor

Chișinău, 2023

Content

Introduction

Dijkstra's algorithm and Floyd-Warshall algorithm, using dynamic programming techniques. These algorithms are widely used for solving graph-related problems, particularly finding the shortest paths in a graph.

Dijkstra's algorithm is a single-source shortest path algorithm that operates on weighted graphs. It efficiently finds the shortest path from a source vertex to all other vertices in the graph. The algorithm works by maintaining a set of vertices whose shortest path from the source vertex is known, gradually expanding this set until all vertices have been included. Dijkstra's algorithm utilizes a priority queue or min-heap data structure to select the next vertex with the smallest distance.

Floyd-Warshall algorithm, on the other hand, is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. It is capable of handling both positive and negative edge weights, but it does not work with graphs containing negative cycles. The algorithm builds a matrix of shortest distances iteratively, considering all possible intermediate vertices along the paths. By updating the distances using the dynamic programming approach, it finds the shortest paths between all pairs of vertices efficiently.

To implement these algorithms, we will utilize the time module to measure the execution time of the algorithms. Additionally, we will use random to generate random graphs for testing purposes. The numpy library (imported as np) will assist in matrix manipulations required for the Floyd-Warshall algorithm. Furthermore, we will employ networkx and matplotlib.pyplot libraries to visualize the graphs and their resulting trees.

The code provides functions to generate random graphs, implement Dijkstra's algorithm, and Floyd-Warshall algorithm. It also includes functions to plot the execution time comparison between the two algorithms and to draw the graphs and resulting trees for visualization.

By running the main function, the code demonstrates the execution of both algorithms on random graphs, measures their execution time, and presents the resulting trees using visualization techniques. This allows us to analyze and compare the performance and outputs of Dijkstra's algorithm and Floyd-Warshall algorithm.

Overall, this code provides a practical implementation of Dijkstra's algorithm and Floyd-Warshall algorithm using dynamic programming, enabling us to explore their applications in solving shortest path problems in graphs.

Dijkstra's algorithm

Dijkstra's algorithm is a well-known algorithm for finding the shortest paths from a single source vertex to all other vertices in a weighted graph. It is widely used in various applications, such as route planning, network optimization, and graph analysis.

The algorithm operates on directed or undirected graphs with non-negative edge weights. It maintains a set of vertices for which the shortest distance from the source vertex is known. Initially, the distance to the source vertex is set to 0, and the distances to all other vertices are set to infinity.

Dijkstra's algorithm proceeds in iterations, repeatedly selecting the vertex with the smallest distance from the set of vertices whose shortest distances are not yet finalized. This selection is usually done using a priority queue or min-heap data structure to efficiently retrieve the vertex with the minimum distance.

In each iteration, the algorithm updates the distances of the neighboring vertices of the current vertex. It compares the current distance plus the weight of the edge to each neighbor with their current distances. If the calculated distance is smaller, it updates the neighbor's distance with the new, shorter distance. Additionally, it keeps track of the previous vertex that leads to the shortest path to each vertex.

The algorithm continues iterating until all vertices have been visited or until the target vertex is reached. Once the algorithm completes, the shortest path and its distance from the source vertex to all other vertices are determined.

Dijkstra's algorithm guarantees the correctness of the shortest paths it computes, given that the graph satisfies the conditions mentioned earlier. It also exhibits an efficient runtime complexity of O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph.

There are several variations and optimizations of Dijkstra's algorithm, such as using a Fibonacci heap data structure for efficient priority queue operations or terminating the algorithm early if the target vertex is reached. Additionally, the algorithm can be modified to handle graphs with negative edge weights by applying techniques like the Bellman-Ford algorithm.

In summary, Dijkstra's algorithm is a powerful tool for finding the shortest paths in a weighted graph. Its simplicity, efficiency, and wide range of applications make it a fundamental algorithm in graph theory and network optimization.

```
for each vertex v in Graph.Vertices:
    dist[v] ← INFINITY
    prev[v] ← UNDEFINED
    add v to Q
dist[source] ← 0
```

```
while Q is not empty:
   u ← vertex in Q with min dist[u]
   remove u from Q

   for each neighbor v of u still in Q:
       alt ← dist[u] + Graph.Edges(u, v)
       if alt < dist[v]:
           dist[v] ← alt
           prev[v] ← u

return dist[], prev[]
```

Floyd-Warshall algorithm

The Floyd-Warshall algorithm is a dynamic programming algorithm used to find the shortest paths between all pairs of vertices in a weighted graph. It works with both directed and undirected graphs, and it can handle graphs with positive or negative edge weights (except for graphs containing negative cycles).

The algorithm utilizes a dynamic programming approach to build a matrix of shortest distances, known as the ”distance matrix.” The matrix is initialized with the weights of the edges between vertices. For vertices that are not directly connected, the matrix entries are set to infinity or a large value to represent an unreachable path.

The Floyd-Warshall algorithm then iterates over all possible intermediate vertices and considers whether using an intermediate vertex can result in a shorter path between two vertices. By gradually including more intermediate vertices, the algorithm finds the shortest paths between all pairs of vertices in the graph.

The algorithm updates the distance matrix by considering each vertex as a possible intermediate vertex. For each pair of vertices (i, j), it compares the distance between i and j directly (without any intermediate vertex) with the distance between i and j through the intermediate vertex k. If the latter distance is shorter, it updates the distance matrix accordingly.

The key idea behind the Floyd-Warshall algorithm is that if there exists a shorter path between two vertices by using an intermediate vertex, then there must be a sequence of shorter paths by using multiple intermediate vertices. By considering all possible intermediate vertices in a nested loop, the algorithm systematically finds the shortest paths between all pairs of vertices.

Once the algorithm completes, the distance matrix contains the shortest distances between all pairs

of vertices in the graph. It can be used to determine the shortest path between any two vertices by backtracking through the intermediate vertices.

The runtime complexity of the Floyd-Warshall algorithm is $O(V^3)$, *where V is the number of vertices in the graph. It is efficient for small to medium-sized graphs, but it may become impractical for large graphs due to its cubic time complexity.*

The Floyd-Warshall algorithm is commonly used in situations where the shortest paths between all pairs of vertices need to be computed in advance, such as in network routing protocols, graph analysis, and optimization problems involving multiple paths.

In summary, the Floyd-Warshall algorithm is a powerful tool for finding the shortest paths between all pairs of vertices in a weighted graph. Its dynamic programming approach allows it to efficiently handle both positive and negative edge weights (excluding negative cycles). By considering all possible intermediate vertices, the algorithm systematically constructs the distance matrix, enabling the determination of shortest paths for any pair of vertices in the graph.

```
let dist be a |V| × |V| array of minimum distances initialized to  (infinity)
    for each edge (u, v) do
        dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
    for each vertex v do
        dist[v][v] ← 0
    for k from 1 to |V|
        for i from 1 to |V|
            for j from 1 to |V|
                if dist[i][j] > dist[i][k] + dist[k][j]
                    dist[i][j] ← dist[i][k] + dist[k][j]
                end if
```

Implementation

Code in python:

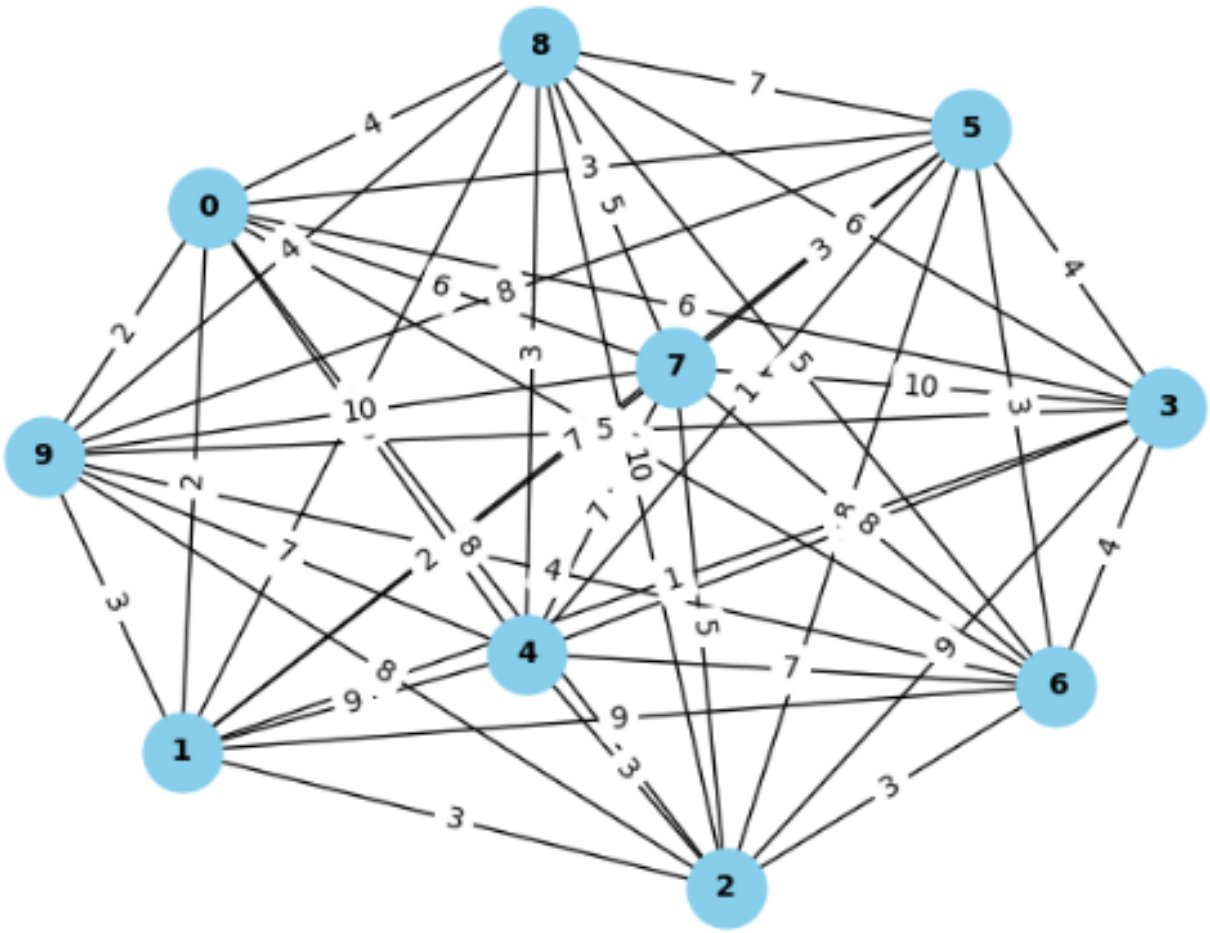
```
import time
import random
import numpy as np
import networkx as nx
import matplotlib.pyplot as plt

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

    while len(visited) != len(graph):
        min_node = None
        for node in dist:
            if node not in visited:
                if min_node is None or dist[node] < dist[min_node]:
                    min_node = node
        visited.add(min_node)

        for neighbor, weight in graph[min_node].items():
            new_dist = dist[min_node] + weight
            if new_dist < dist[neighbor]:
                dist[neighbor] = new_dist
    return dist

def floyd(graph):
    nodes = list(graph.keys())
    n = len(nodes)
    dist = np.full((n, n), np.inf)
```

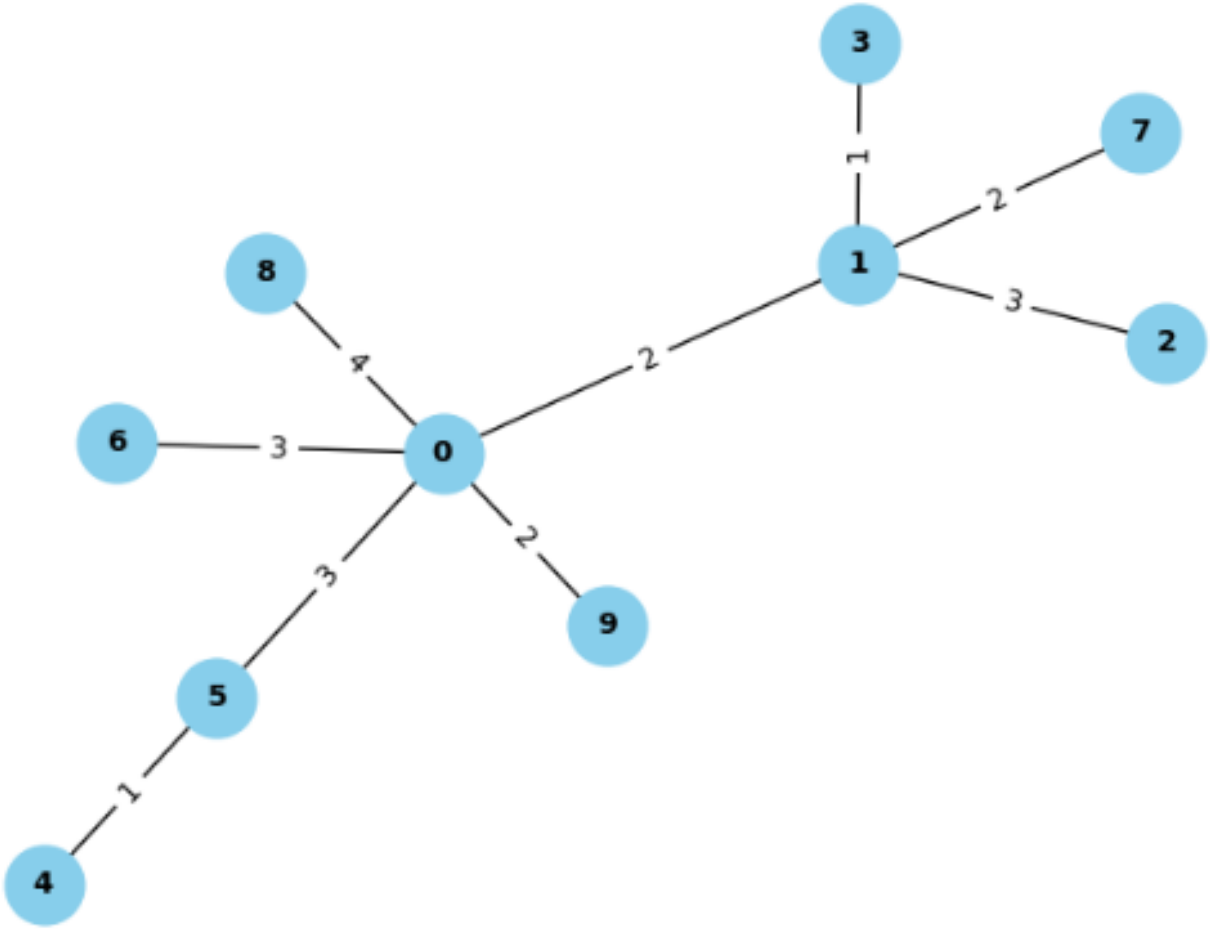
```
    for i, node in enumerate(nodes):
        dist[i, i] = 0
        for neighbor, weight in graph[node].items():
            j = nodes.index(neighbor)
            dist[i, j] = weight

    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 {nodes[i]: {nodes[j]: dist[i, j] for j in range(n)} for i in range(n)}

def generate_random_graph(num_nodes, max_weight=10):
    graph = {i: {} for i in range(num_nodes)}
    for i in range(num_nodes):
        for j in range(i+1, num_nodes):
            weight = random.randint(1, max_weight)
            graph[i][j] = weight
            graph[j][i] = weight
    return graph

def plot_comparison(nodes, dijkstra_times, floyd_times):
    plt.plot(nodes, dijkstra_times, label="Dijkstra")
    plt.plot(nodes, floyd_times, label="Floyd")
    plt.xlabel("Number of Nodes")
    plt.ylabel("Execution Time (s)")
    plt.legend()
    plt.title("Dijkstra vs. Floyd Execution Time Comparison")
    plt.show()

def draw_graph_tree(graph, title):
    G = nx.Graph(graph)
```

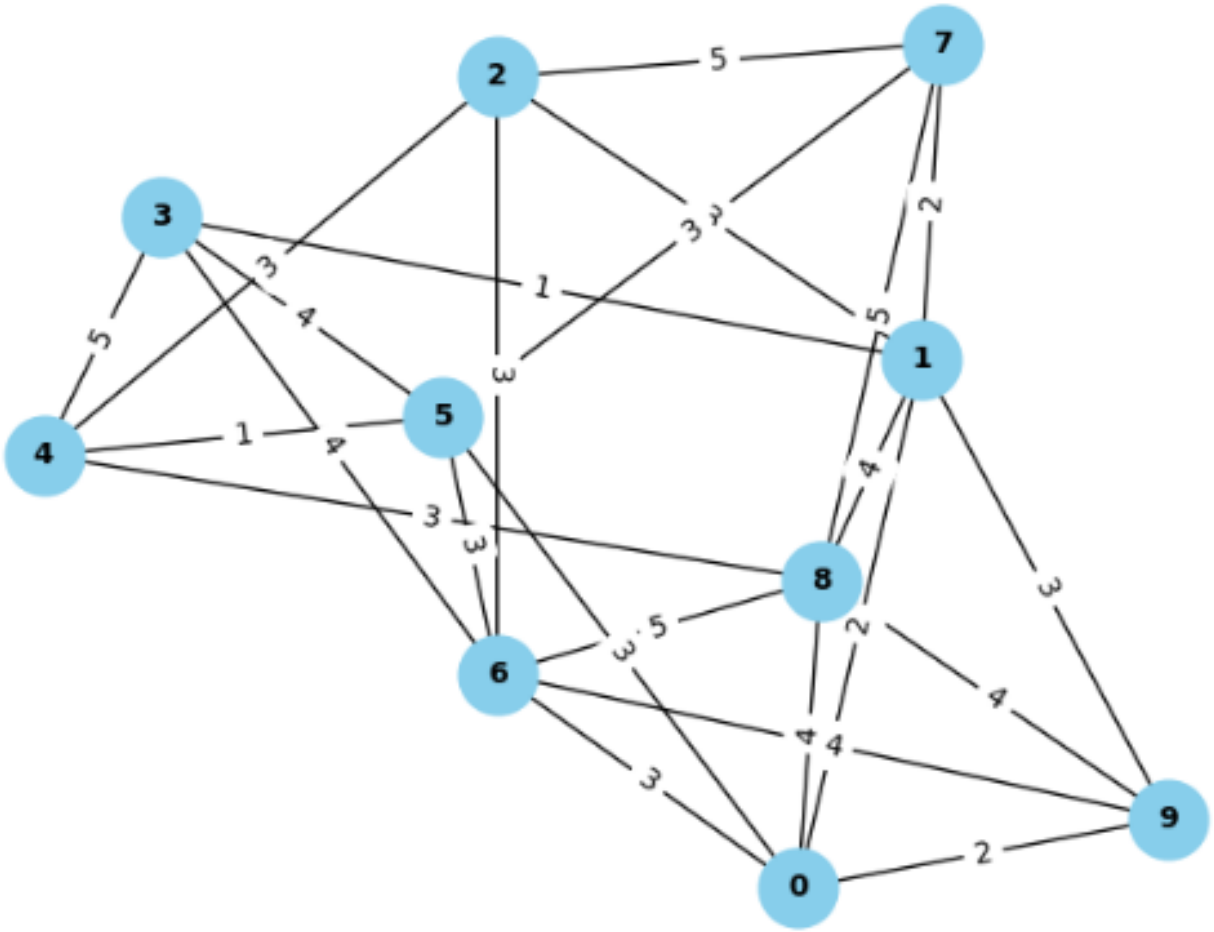
```
    pos = nx.spring_layout(G)
    nx.draw(G, pos, with_labels=True, node_color='skyblue', font_size=10, font_weight='bol
    nx.draw_networkx_edge_labels(G, pos, edge_labels={(i, j): graph[i][j] for i in graph f
    plt.title(title)
    plt.show()

def main():
    # Prompt the user to enter the number of vertices
    num_nodes = input("Please enter the number of vertices: ")
    try:
        num_nodes = [int(num_nodes)]
    except ValueError:
        print("Invalid input. Please enter a number.")
        return

    dijkstra_times = []
    floyd_times = []

    for n in num_nodes:
        graph = generate_random_graph(n)
        start_time = time.time()
        dijkstra(graph, 0)
        dijkstra_times.append(time.time() - start_time)

        start_time = time.time()
        floyd(graph)
        floyd_times.append(time.time() - start_time)

    plot_comparison(num_nodes, dijkstra_times, floyd_times)

    sample_graph = generate_random_graph(num_nodes[0])
    draw_graph_tree(sample_graph, "Sample Graph for Dijkstra and Floyd Algorithms")

    dijkstra_tree = dijkstra(sample_graph, 0)
```

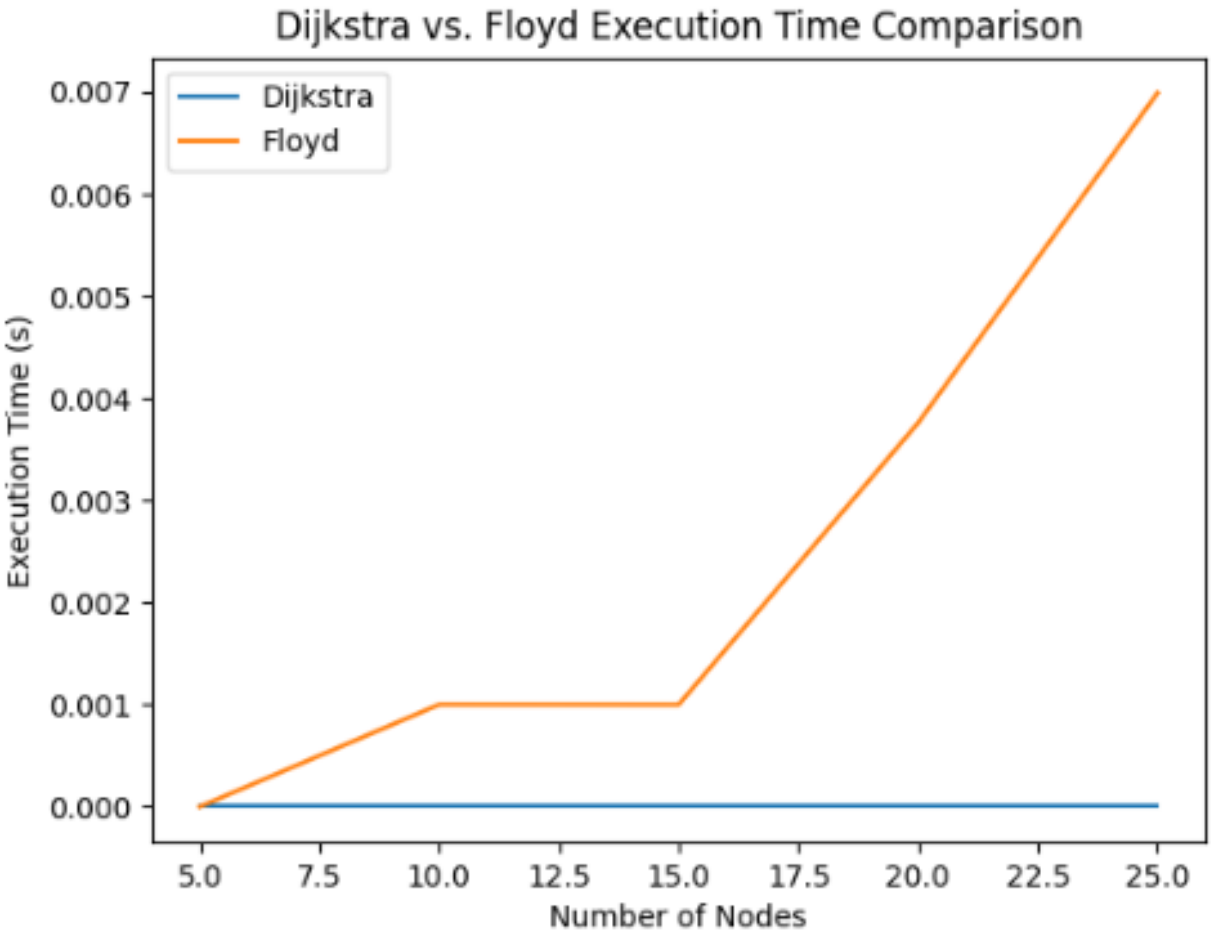
```
    draw_graph_tree({node: {neighbor: sample_graph[node][neighbor] for neighbor in sample_
                             dijkstra_tree[neighbor] == dijkstra_tree[node] + sample_graph[
                     sample_graph}, "Dijkstra's Algorithm Graph Tree")

    floyd_tree = floyd(sample_graph)
    draw_graph_tree({node: {neighbor: sample_graph[node][neighbor] for neighbor in sample_
                             floyd_tree[node][neighbor] == sample_graph[node][neighbor]} fo
                   "Floyd's Algorithm Graph Tree")


if __name__ == "__main__":
    main()
```

Screenshot:




3
1
7
2
8
1
4
2
3
6
3
0
2
3
9
5
1
4


7
2
5
3
3
2
1
5
4
3
4
5
1
5
4
1
4
3
3
8
3
5
2
6
4
4
4
3
9
2
0


Dijkstra vs. Floyd Execution Time Comparison
Dijkstra
Floyd
Execution Time (s)
0.007
0.006
0.005
0.004
0.003
0.002
0.001
0.000
5.0
7.5
10.0
12.5
15.0
17.5
20.0
22.5
25.0
Number of Nodes

Conclusion

Dijkstra's algorithm and Floyd-Warshall algorithm are two fundamental graph algorithms used for finding shortest paths in weighted graphs. While both algorithms aim to solve the same problem, they employ different strategies and have distinct use cases.

Dijkstra's algorithm is a single-source shortest path algorithm that finds the shortest path from a single source vertex to all other vertices in the graph. It operates efficiently on graphs with non-negative edge weights and is particularly useful when the goal is to find the shortest paths from a specific source vertex. Dijkstra's algorithm follows a greedy approach, iteratively selecting the vertex with the smallest distance and updating the distances of its neighboring vertices. It guarantees correctness and has a time complexity of $O((V+E)logV)$, making it suitable for scenarios where finding the shortest paths from a single source is the primary objective.

On the other hand, Floyd-Warshall algorithm is a dynamic programming algorithm that computes the shortest paths between all pairs of vertices in a graph. It works with directed or undirected graphs and can handle graphs with positive or negative edge weights (excluding negative cycles). Floyd-Warshall algorithm constructs a distance matrix by considering all possible intermediate vertices, gradually updating the shortest distances. It guarantees correctness and has a time complexity of $O(V^3)$, making it suitable for small to medium-sized graphs where finding the shortest paths between all pairs of vertices is required.

The choice between Dijkstra's algorithm and Floyd-Warshall algorithm depends on the specific problem and graph characteristics. Dijkstra's algorithm is preferable when the goal is to find shortest paths from a single source to multiple destinations, while Floyd-Warshall algorithm is suitable for finding shortest paths between all pairs of vertices in the graph. Additionally, Dijkstra's algorithm is more efficient for sparse graphs, whereas Floyd-Warshall algorithm is more efficient for dense graphs.

In summary, Dijkstra's algorithm and Floyd-Warshall algorithm are powerful tools for solving shortest path problems in graphs. They provide different approaches and cater to different requirements. Understanding the strengths and limitations of each algorithm is crucial for selecting the appropriate approach based on the problem's constraints, graph properties, and specific objectives.