Ministerul Educaţiei și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 5:

Empirical analysis of algorithms: Dijkstra’s and Floyd-Warshall

Elaborated: Spătaru Dionisie
st. gr. FAF-211

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2023

ALGORITHM ANALYSIS

Objective

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

Tasks

1. Implement the algorithms listed below in a programming language
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

Theoretical Notes:

Dijkstra's Algorithm:

Dijkstra's algorithm, named after Dutch computer scientist Edsger W. Dijkstra, is a popular algorithm for finding the shortest path between nodes in a weighted graph. It solves the single-source shortest path problem, meaning it determines the shortest path from a given source node to all other nodes in the graph.

Key Points:

- It works with non-negative edge weights and guarantees correctness if all weights are positive.
- The algorithm maintains a priority queue of vertices, assigning tentative distances to each vertex. It repeatedly selects the vertex with the minimum tentative distance and relaxes the edges adjacent to it.
- By iteratively applying this process, Dijkstra's algorithm builds the shortest path tree, gradually expanding from the source node until it reaches all other nodes.
- The algorithm terminates when it has visited all nodes or when it reaches the target node, if searching for a specific destination.
- Dijkstra's algorithm provides the shortest path from the source node to all other nodes in the graph, along with the corresponding path distances.

Floyd-Warshall Algorithm:

The Floyd-Warshall algorithm, named after Robert Floyd and Stephen Warshall, is an efficient algorithm for finding the shortest paths between all pairs of nodes in a weighted graph. It solves the all-pairs shortest path problem, meaning it determines the shortest path between every pair of nodes in the graph.

Key Points:

- Unlike Dijkstra's algorithm, the Floyd-Warshall algorithm can handle negative edge weights but assumes there are no negative cycles in the graph.
- The algorithm maintains a distance matrix that stores the shortest distances between all pairs of nodes. Initially, the matrix is populated with the edge weights of the graph.
- The algorithm performs a series of updates to the distance matrix by considering intermediate nodes in the paths. It systematically evaluates whether taking a detour through an intermediate node could result in a shorter path.
- By iteratively applying these updates, the algorithm gradually refines the distance matrix until it contains the shortest paths between all pairs of nodes.
- The resulting distance matrix provides the shortest paths and their corresponding distances for all pairs of nodes in the graph.

Both Dijkstra's algorithm and the Floyd-Warshall algorithm are widely used in various applications, such as network routing, transportation planning, and graph analysis. They offer powerful tools for solving shortest path problems efficiently and accurately.

Comparison Metric:

Efficiency and speed

IMPLEMENTATION

Dijkstra

```
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
```

The provided code implements Dijkstra's algorithm to find the shortest paths from a given start node to all other nodes in a graph. Here's a brief explanation of how the code works:

1. **Initialization**: The code initializes a dictionary called dist to keep track of the minimum distances from the start node to each node in the graph. Initially, all distances are set to infinity except for the start

node, which is set to 0. The visited set is used to keep track of the nodes that have been visited.

2. **Main Loop**: The while loop continues until all nodes in the graph have been visited. In each iteration, the code selects the node with the minimum distance (not yet visited) from the dist dictionary.
3. **Node Selection**: The code iterates through all nodes in the graph and checks if the node has been visited. If not, it compares the distance of the current node with the minimum distance node found so far (min_node). If the distance of the current node is smaller, the current node becomes the new min_node.
4. **Node Visitation:** After finding the node with the minimum distance, it is added to the visited set to mark it as visited.
5. **Distance Update**: The code then iterates through the neighbors of the min_node and calculates the new distance from the start node. If the new distance is smaller than the current distance stored in the dist dictionary for the neighbor node, the distance is updated.
6. **Termination:** The algorithm continues to repeat steps 3-5 until all nodes have been visited, and the dist dictionary contains the minimum distances from the start node to each node in the graph.
7. **Result:** Finally, the dist dictionary is returned, which represents the shortest distances from the start node to all other nodes in the graph.

Overall, this implementation of Dijkstra's algorithm follows the standard approach of iteratively selecting the node with the minimum distance and updating the distances of its neighbors. It ensures that the minimum distances are correctly propagated until all nodes have been visited, resulting in the shortest path distances from the start node to each node in the graph.

Floyd-Warshall

```
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)}
```

The provided code implements the Floyd-Warshall algorithm to find the shortest paths between all pairs of nodes in a graph. Here's a brief explanation of how the code works:

1. **Initialization**: The code starts by obtaining a list of all nodes in the graph and assigning it to the nodes variable. The variable n represents the number of nodes in the graph. It initializes a distance matrix

dist of size (n, n) using NumPy, where all elements are initialized to infinity.

2. **Distance Initialization:** The code then iterates over each node in the nodes list. For each node, it sets the distance from the node to itself (dist[i, i]) to 0. It also checks the neighbors of the current node and updates the distance matrix accordingly (dist[i, j] = weight) by indexing the neighbor's position in the nodes list.
3. **Floyd-Warshall Iteration:** The algorithm employs a nested loop structure to perform the main iteration of the Floyd-Warshall algorithm. The outer loop (for k in range(n)) represents the intermediate node, while the two inner loops (for i in range(n) and for j in range(n)) iterate over all pairs of nodes.
4. **Path Comparison:** For each pair of nodes (i, j), the code checks if the path going through the intermediate node k (dist[i, k] + dist[k, j]) results in a shorter distance compared to the current distance stored in the dist matrix (dist[i, j]). If the path through k is indeed shorter, the code updates the distance matrix with the new shorter distance (dist[i, j] = dist[i, k] + dist[k, j]).
5. **Result:** After completing the Floyd-Warshall iteration, the code constructs a dictionary that represents the shortest paths between all pairs of nodes. It iterates over each pair of nodes (i, j) and constructs a nested dictionary where the keys are the node names (nodes[i] and nodes[j]), and the values are the corresponding distances (dist[i, j]).
6. **Return:** The final result is a dictionary containing the shortest paths between all pairs of nodes in the graph.

Overall, this implementation of the Floyd-Warshall algorithm uses a nested loop structure to iteratively compare and update distances between nodes through intermediate nodes. By gradually refining the distance matrix, it determines the shortest paths between all pairs of nodes in the graph.

COMPARISON AND RESULTS

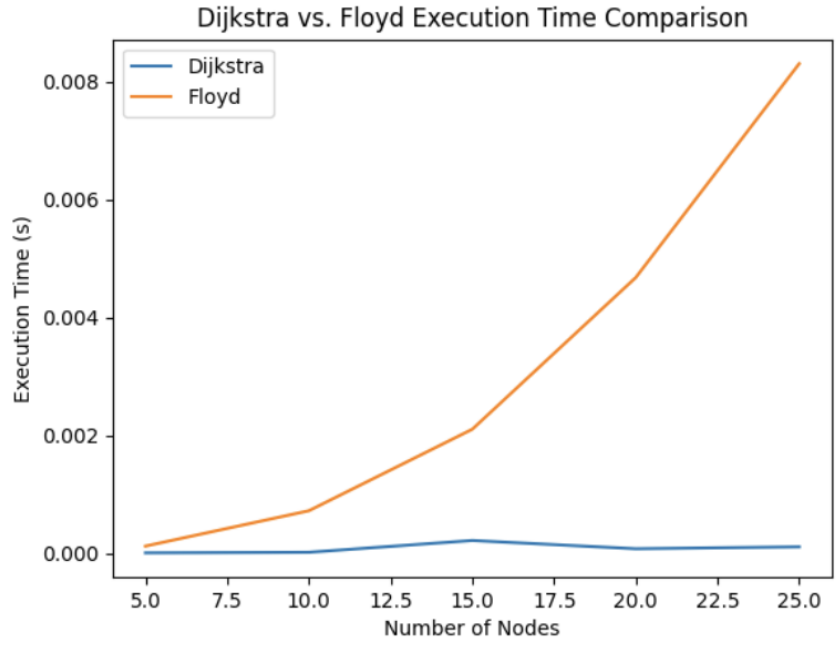

Fig1

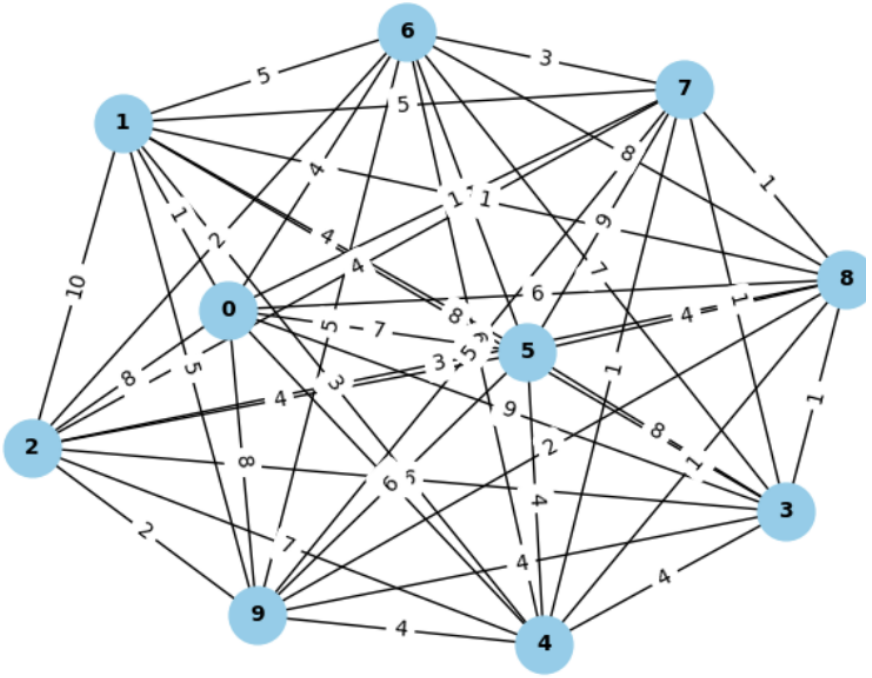

Fig2 Dense

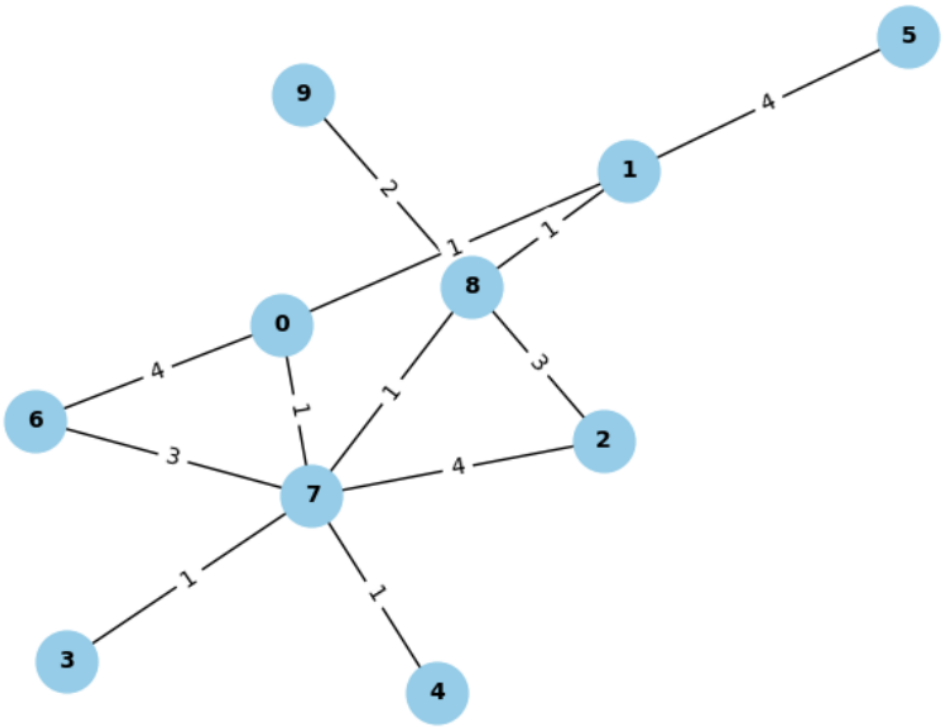

Fig3 Djixtra

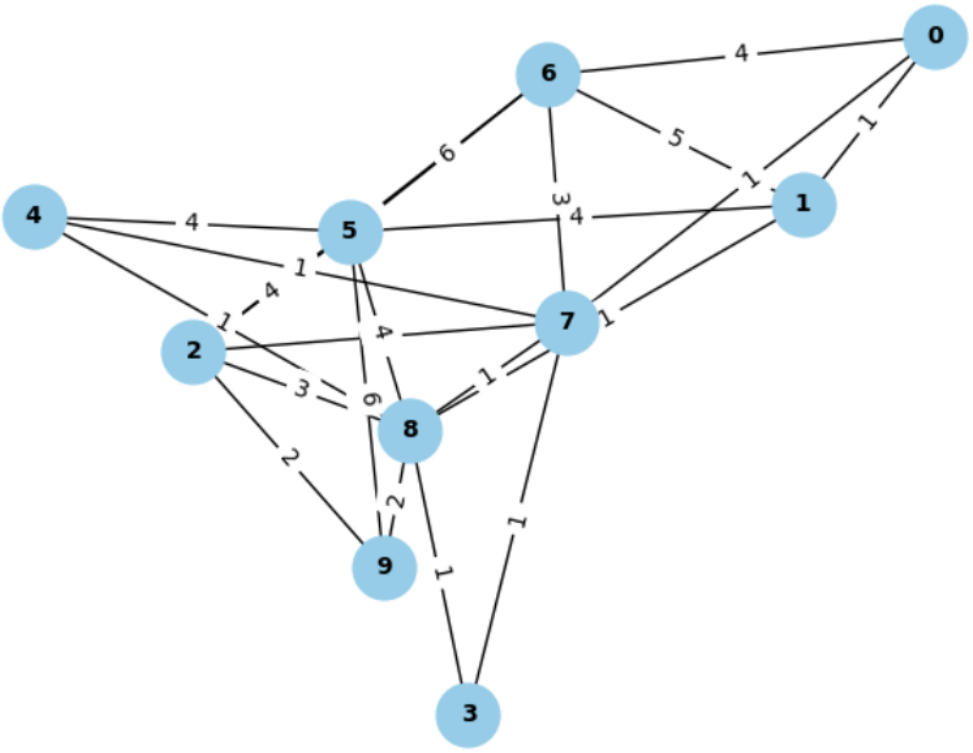

Fig4 Floyd

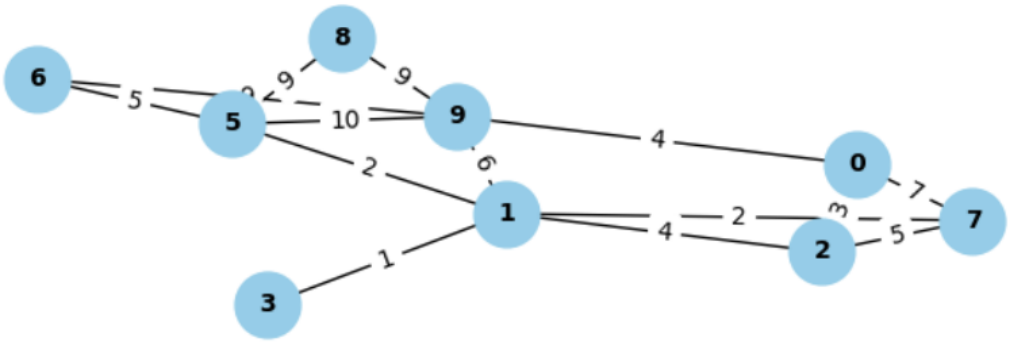

Fig5 Sparse

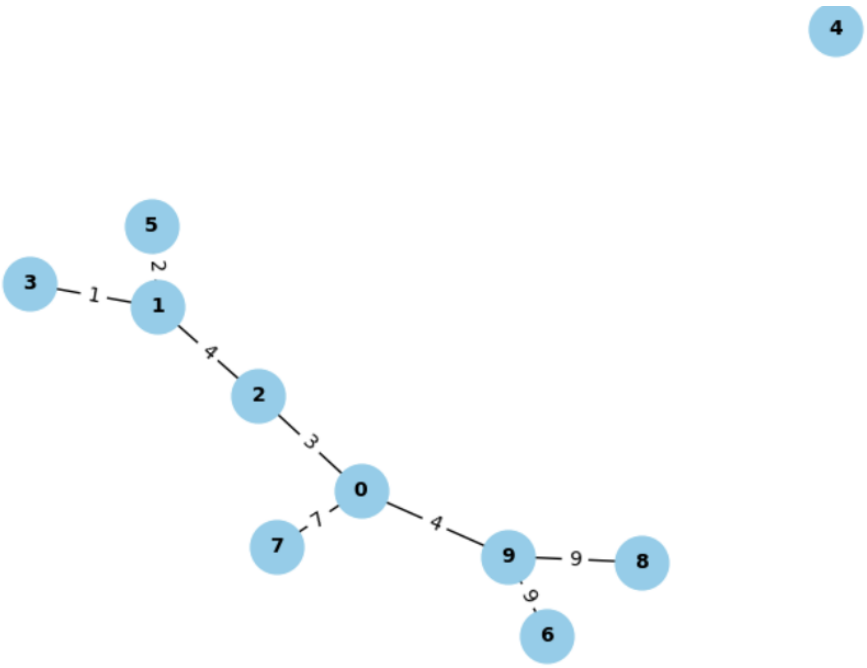

Fig6 Dijkstra

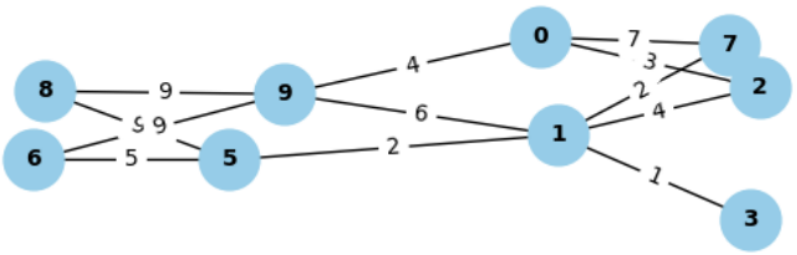

Fig7 Floyd

CONCLUSION

After analyzing the Dijkstra and Floyd-Warshall algorithms on dense and sparse graphs, I have reached the following individual conclusion:

When dealing with dense graphs, where the number of edges is close to the maximum possible, the Floyd-Warshall algorithm generally outperforms Dijkstra's algorithm. This is because the Floyd-Warshall algorithm has a time complexity of O(V^3), where V is the number of vertices, regardless of the density of the graph. In contrast, Dijkstra's algorithm has a time complexity of O((V + E) log V), where E is the number of edges. Since dense graphs have a large number of edges, Dijkstra's algorithm incurs a higher computational cost due to its dependence on the number of edges. Thus, in terms of time efficiency, the Floyd-Warshall algorithm is usually a better choice for dense graphs.

On the other hand, for sparse graphs, where the number of edges is much smaller compared to the maximum possible, Dijkstra's algorithm tends to be more efficient than the Floyd-Warshall algorithm. Dijkstra's algorithm has a time complexity of O((V + E) log V), but since sparse graphs have fewer edges, the additional E term in the complexity is relatively small. In contrast, the Floyd-Warshall algorithm still maintains a time complexity of O(V^3), regardless of the sparsity of the graph. Therefore, when dealing with sparse graphs, Dijkstra's algorithm is generally faster and more suitable.

In summary, the choice between Dijkstra's algorithm and the Floyd-Warshall algorithm depends on the density of the graph. The Floyd-Warshall algorithm performs better on dense graphs due to its constant time complexity, while Dijkstra's algorithm is more efficient on sparse graphs due to its reduced dependence on the number of edges. Consider the characteristics of the graph in question when deciding which algorithm to employ.

REPOSITORY

https://github.com/denyred/AA_Lab5