Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei
Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 5:

Empirical analysis of Dijkstra's and Floyd Warshall algorithms

Elaborated:
st. gr. FAF-213 Konjevic Alexandra

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

ALGORITHM ANALYSIS

Objective:

Study and analyze algorithms for finding the shortest paths in graph: Dijkstra's and Floyd-Warshall algorithms.

Tasks:

1. Implement the listed algorithms 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.

Introduction:

The Dijkstra and Floyd-Warshall algorithms are two of the most well-known and widely used algorithms in computer science. These algorithms are used to find the shortest paths in weighted graphs, which is a fundamental problem in many applications, such as network routing, transportation planning, and social network analysis. The Dijkstra algorithm is used to find the shortest path between two vertices in a graph, while the Floyd-Warshall algorithm is used to find the shortest paths between all pairs of vertices in a graph. In this report, I will explore these two algorithms in detail, including their theoretical background and implementation. I will also compare and contrast the strengths and weaknesses of these algorithms and provide insights into when to use each of them based on the specific problem

requirements. This report aims to provide a comprehensive overview of the Dijkstra and Floyd-Warshall algorithms and their practical relevance, as well as to highlight their importance in the field of computer science.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)).

Input Format:

I generated random graphs of lengths from 100 vertexes to 700, with the step 100. For each graph I measured the time of executing of functions `dijkstra` and `floyd-warshall`.

IMPLEMENTATION

First of all, I used python library `networkx` to generate random graphs with edges, that have a random weight from 0 to 100, via the method:

```python
def generate_random_graph(num_nodes):
    G = nx.Graph()
    for i in range(num_nodes):
        G.add_node(i)
    for i in range(num_nodes):
        for j in range(i+1, num_nodes):
            if random.random() < 0.5:  # Add an edge with probability 0.5
                weight = random.randint(1, 100)  # Generate a random weight
                G.add_weighted_edges_from([(i, j, weight)])
    return G
```

Figure 1. Method to generate random graphs in python

1. Dijkstra method implementation

```python
def dijkstra(graph, start):
    import heapq
    """
    Apply Dijkstra's algorithm to a networkx graph starting from a given node.
    Args:
        graph (networkx.Graph): The graph to apply Dijkstra's algorithm to.
        start: The starting node.
    Returns:
        dict: A dictionary where the keys are the nodes in the graph and the values are the shortest distances
        from the start node to each node.
    """
    distances = {node: float('inf') for node in graph.nodes(
    )}  # Initialize all distances as infinite
    distances[start] = 0  # Distance from start node to itself is 0
    # Priority queue of (distance, node) tuples, starting with the start node
    heap = [(0, start)]
    visited = set()  # Set of visited nodes

    while heap:
        # Get the node with the smallest distance
        (dist, current_node) = heapq.heappop(heap)
        if current_node in visited:
            continue  # Ignore nodes that have already been visited
        visited.add(current_node)
        for neighbor in graph.neighbors(current_node):
            tentative_distance = dist + graph[current_node][neighbor]['weight']
            if tentative_distance < distances[neighbor]:
                distances[neighbor] = tentative_distance
                # Add the neighbor to the queue
                heapq.heappush(heap, (tentative_distance, neighbor))

    return distances
```

Figure 2. Dijkstra implementation in python

Algorithm Description:

Dijkstra's algorithm is a well-known algorithm in computer science that is used to find the shortest path between two nodes in a weighted graph. The algorithm was named after its inventor, Dutch computer scientist Edsger W. Dijkstra.

The algorithm works by starting at the source node and calculating the shortest path to all other nodes in the graph. It maintains a set of nodes that have already been visited and a set of nodes that have not yet been visited. At each step,

it selects the node with the shortest distance from the source node that has not yet been visited and adds it to the set of visited nodes. It then updates the distance to all adjacent nodes that have not yet been visited by adding the distance from the current node to the adjacent node to the distance from the source node to the current node. If the new distance is shorter than the previous distance, the new distance is assigned to the adjacent node.

The algorithm repeats this process until all nodes have been visited or the destination node has been reached. Once the destination node is reached, the algorithm can be stopped and the shortest path can be determined by tracing back from the destination node to the source node using the information that has been stored in the algorithm.

Dijkstra's algorithm is guaranteed to find the shortest path between the source node and any other node in the graph if no negative edge weights are present in the graph. The time complexity of Dijkstra's algorithm is O(|V|^2) for a graph with |V| nodes using a simple implementation, but more efficient implementations exist that can reduce the time complexity to O(|E| + |V|log|V|) using a priority queue.

Vertexes	Time
100	0.00200
200	0.00900
300	0.02100
400	0.04200
500	0.05800
600	0.09200
700	0.11700

Figure 3. Results for Dijkstra

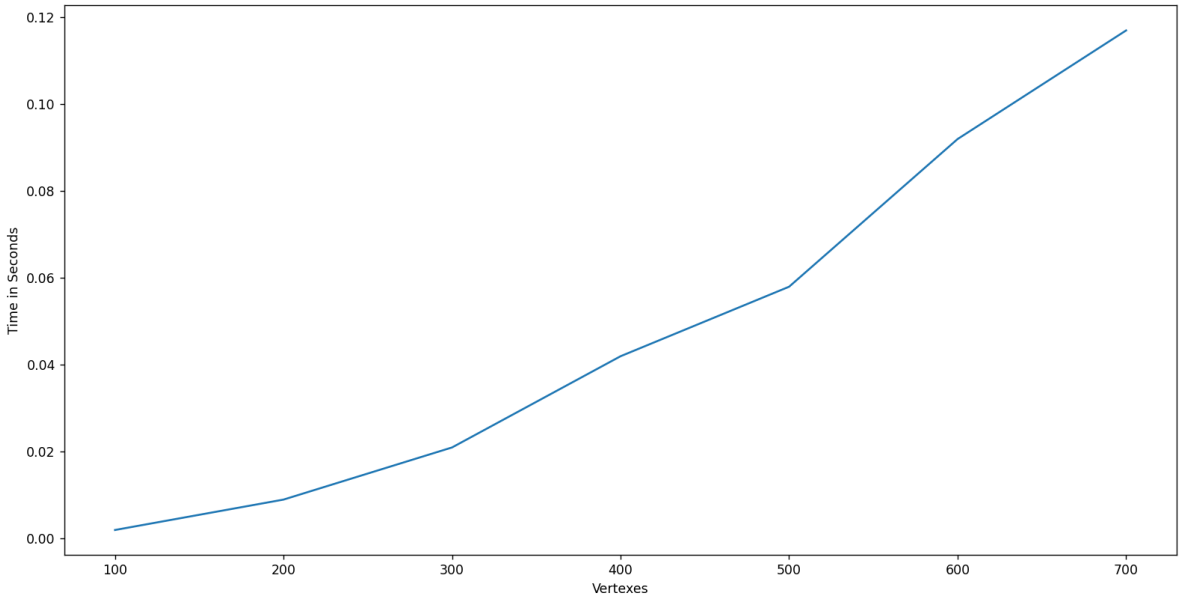

Figure 4. Graph for Dijkstra

2. Floyd-Warshall method implementation

```python
def floyd_warshall(graph, start):
    import numpy as np
    """
    Applies the Floyd-Warshall algorithm to the given graph.
    Args:
        graph (networkx.Graph): The graph to apply the algorithm to.
    Returns:
        numpy.ndarray: A 2D array where element i,j is the shortest path from node i to node j in the graph.
    """
    nodes = list(graph.nodes())
    dist = np.full((len(nodes), len(nodes)), np.inf)
    np.fill_diagonal(dist, 0)

    for node1, node2, weight in graph.edges(data='weight'):
        dist[node1-1, node2-1] = weight
        dist[node2-1, node1-1] = weight

    for k in range(len(nodes)):
        for i in range(len(nodes)):
            for j in range(len(nodes)):
                dist[i, j] = min(dist[i, j], dist[i, k] + dist[k, j])

    return dist
```

Figure 5. Floyd-Warshall implementation in python

Algorithm Description:

The Floyd-Warshall algorithm is a well-known algorithm in computer science that is used to find the shortest paths between all pairs of vertices in a weighted graph. The algorithm was named after its inventors, Robert Floyd and Stephen Warshall.

The algorithm works by maintaining a matrix of the shortest distances between every pair of vertices in the graph. Initially, the matrix contains the weights of the edges between adjacent vertices and infinity for all other pairs of vertices. The algorithm then iteratively updates the matrix by considering all possible intermediate vertices between each pair of vertices.

More specifically, for each intermediate vertex k, the algorithm updates the distance between vertices i and j as follows:

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

where distance[i][j] represents the shortest distance between vertices i and j, and distance[i][k] + distance[k][j] represents the distance between vertices i and j passing through intermediate vertex k.

The algorithm repeats this process for all possible intermediate vertices, which means that it considers all possible paths between each pair of vertices. After all iterations are complete, the matrix contains the shortest distances between all pairs of vertices in the graph.

The time complexity of the Floyd-Warshall algorithm is O(|V|^3), where |V| is the number of vertices in the graph. This makes it less efficient than other algorithms, such as Dijkstra's algorithm, for finding the shortest path between two vertices in large graphs. However, the Floyd-Warshall algorithm is useful when all pairwise distances are needed, as it avoids the need to run Dijkstra's algorithm multiple times.

Vertexes	Time
100	0.44600
200	3.50454
300	11.43787
400	27.46393
500	53.84710
600	93.24701
700	145.14857

Figure 6. Results for Floyd-Warshall

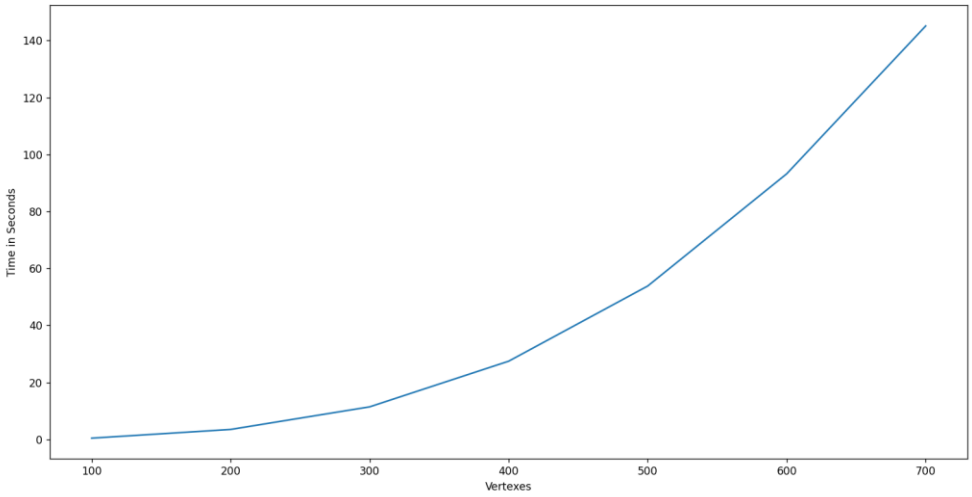

Figure 9. Graph for Floyd-Warshall

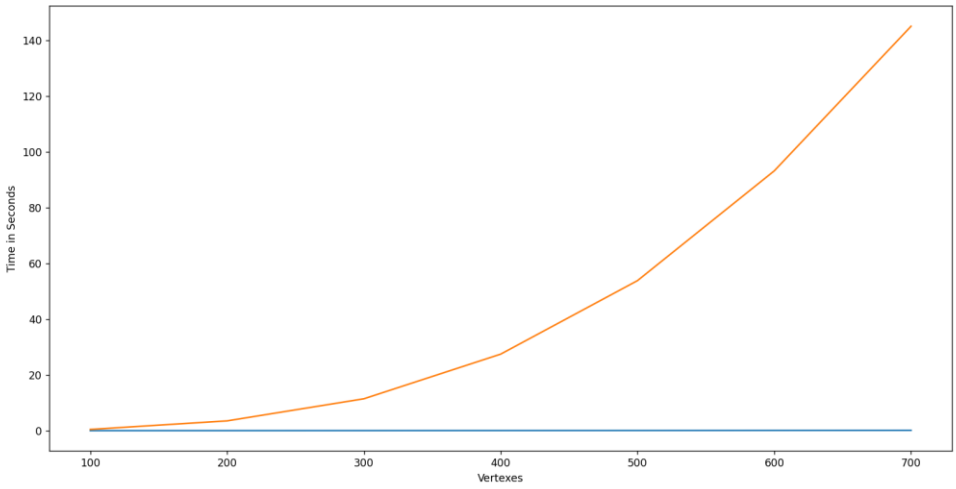

Figure 7. Graph with both Dijkstra and Floyd-Warshall

● – Floyd-Warshall ● –Dijkstra

Conclusion

In conclusion, after comparing the Dijkstra and Floyd-Warshall algorithms and implementing them in Python, we can see that both algorithms have their strengths and weaknesses. The Dijkstra algorithm is faster and more efficient for finding the shortest path between two points, but it cannot handle negative edge weights. On the other hand, the Floyd-Warshall algorithm can handle negative edge weights and can find the shortest path between all pairs of nodes in a graph, but it is slower than Dijkstra's algorithm for finding the shortest path between two points.

When choosing which algorithm to use, it's important to consider the characteristics of the problem at hand. If the graph has no negative edge weights and we need to find the shortest path between two points, then Dijkstra's algorithm is a better choice. If the graph has negative edge weights and we need to find the shortest path between all pairs of nodes, then the Floyd-Warshall algorithm is the way to go.

The significant difference in execution time in this laboratory work, is due to the fact that Dijkstra’s algorithm was applied to find the shortest paths only form a single starting point to all the other vertices, whereas the Floyd-Warshall algorithm, found the shortest paths between each pair of vertices from the graph.

Overall, both algorithms are powerful tools for solving graph problems, and understanding their strengths and weaknesses can help us choose the best approach for a given problem. By implementing these algorithms in Python, we have gained a deeper understanding of their inner workings and learned valuable skills in algorithmic problem solving.

Github repository: https://github.com/alya1007/Labs-semester-4/tree/master/AA