Ministry of Education, Culture and Research of the Republic of Moldova

Technical University of Moldova

Department of Software and Automation Engineering

**REPORT**

Laboratory work No. 5

Discipline: Algorithms’ Analysis

Topic: Dynamic programming

Elaborated: Tudor Sclifos,

student group FAF-211

Verified: Fiștic Cristofor,

university assistant

Chișinău – 2023

Algorithm analysis

**Objective:**

Implement algorithms Dijkstra and Warshall-Floyd using dynamic programming**.**

**Tasks:**

1 To study the dynamic programming method of designing algorithms.

2 To implement in a programming language algorithms Dijkstra and Floyd–Warshall using dynamic programming.

3 Do empirical analysis of these algorithms for a sparse graph and for a dense graph.

4 Increase the number of nodes in graphs and analyze how this influences the algorithms. Make a graphical presentation of the data obtained

5 To make a report.

**Theoretical Notes:**

Dijkstra's algorithm is a popular algorithm used to find the shortest path between a starting vertex and all other vertices in a weighted graph. It works by maintaining a priority queue or heap of vertices and iteratively selecting the vertex with the minimum distance from the starting vertex.

The algorithm starts by assigning a tentative distance of 0 to the starting vertex and infinity to all other vertices. It then repeatedly selects the vertex with the minimum tentative distance from the priority queue and explores its adjacent vertices. For each adjacent vertex, the algorithm updates its tentative distance if a shorter path is found. This process continues until all vertices have been processed or the target vertex is reached.

Kruskal's algorithm is another widely used algorithm for finding the minimum spanning tree (MST) of a weighted undirected graph. Similar to Prim's algorithm, Kruskal's algorithm seeks to connect all vertices of the graph while minimizing the total weight of the edges.

One of the advantages of Kruskal's algorithm is its simplicity and efficiency. It has a time complexity of O(E log E), where E is the number of edges in the graph. This makes it suitable for large-scale problems. Additionally, Kruskal's algorithm can handle disconnected graphs and graphs with weighted edges that are not necessarily positive.

**Introduction:**

In this laboratory work I have to implement Dijkstra and Floyd algorithms and apply them on a sparse and a dense graph. Also I need to analyse the outputs.

**Comparison metric:**

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

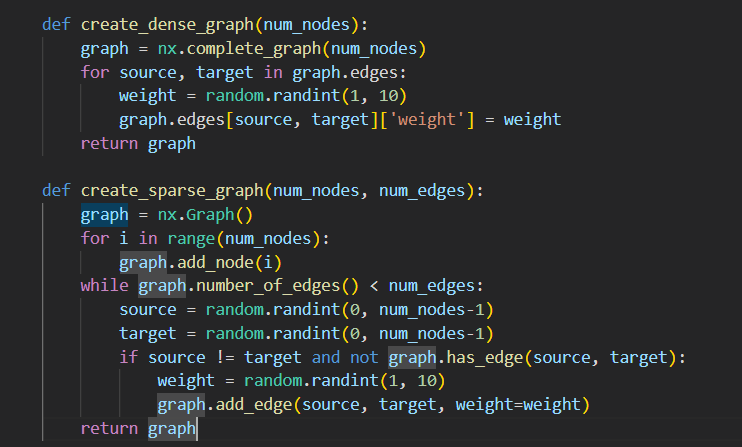
**Input format:**

The input for this laboratory work are the list of nodes for graphs, one dense and another sparse.

**IMPLEMENTATION**

**Graph Generator**

In summary, this function generates a matrix representation of a weighted graph by randomly assigning edge weights based on a given density. The higher the density value, the more likely it is to have edges between vertices, resulting in a denser graph.



**Figure 1. Implementation Graph**

**Dijkstra**

**Implementation:**

The function **dijkstra** takes a graph as input, represented as a dictionary of dictionaries. The outer dictionary represents the vertices, and each inner dictionary represents the neighboring vertices and their edge weights.

The variable **all\_distances** is initialized as an empty dictionary. This dictionary will store the shortest distances from the starting vertex to all other vertices.

The function iterates over each vertex in the graph using the **for** loop. This sets the starting vertex for each iteration.

Inside the loop, a new **distances** dictionary is created, where the initial distances from the starting vertex to all other vertices are set to a maximum value (**sys.maxsize**), indicating infinity. The distance from the starting vertex to itself is set to 0.

The variable **queue** is initialized as a list containing a tuple **(0, start)**, representing the starting vertex and its distance from itself (which is 0). The **heapq** module is used to maintain the queue in a way that the vertex with the smallest distance is always at the front.

The **while** loop continues as long as the **queue** is not empty. This loop iteratively selects vertices and updates their distances.

Inside the loop, the vertex with the smallest distance (**current\_vertex**) is extracted from the front of the **queue** using **heapq.heappop()**. If the extracted distance is greater than the distance stored in **distances** for that vertex, it means a shorter path to that vertex has already been found, so the loop moves on to the next iteration using **continue**.

For each neighboring vertex (**neighbor**) of the **current\_vertex**, the weight of the edge connecting them is retrieved from the graph using **graph[current\_vertex][neighbor]['weight']**. The distance from the starting vertex to the neighbor via the current vertex is calculated as **current\_distance + weight**.

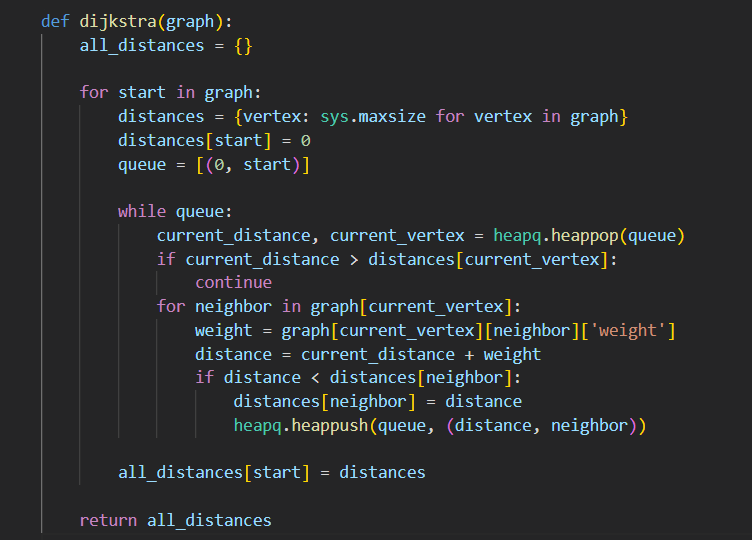
If this newly calculated distance is smaller than the distance stored in **distances** for the neighbor, it means a shorter path has been found. The distance is updated, and the neighbor is added to the **queue** with its new distance using **heapq.heappush()**.

After the **while** loop finishes, the **distances** dictionary for the current starting vertex is complete, representing the shortest distances from the starting vertex to all other vertices.

The **distances** dictionary is added to the **all\_distances** dictionary, with the starting vertex as the key.

The process continues for each vertex in the graph until all vertices have been processed.

Finally, the **all\_distances** dictionary containing the shortest distances from each starting vertex to all other vertices is returned.



**Figure 2. Implementation Dijkstra**

**Floyd**

**Implementation:**

The function **floyd** takes a graph as input.

The variable **num\_nodes** is initialized to the number of nodes in the graph.

The **distances** matrix is initialized as a 2D list of size **num\_nodes x num\_nodes**, with all elements initially set to a large value (**sys.maxsize**). This matrix will store the shortest distances between each pair of vertices.

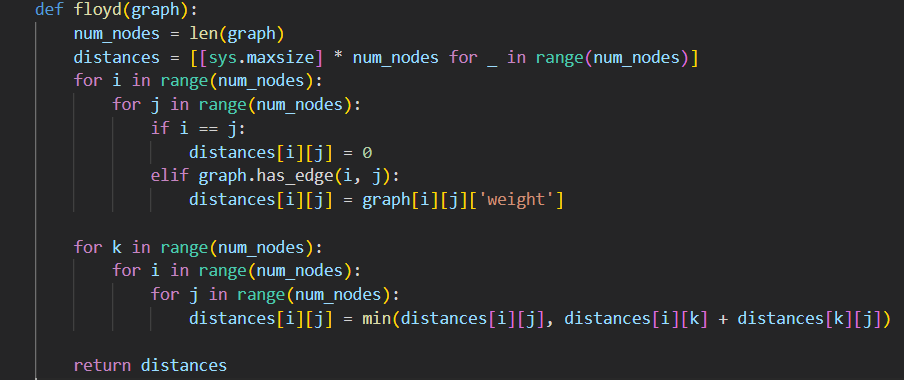
The first nested **for** loop iterates over each row and column of the **distances** matrix. It checks if the current row index **i** is equal to the column index **j**. If true, it means the current indices represent the same vertex, so the distance is set to 0. Otherwise, it checks if the graph has an edge between vertices **i** and **j**. If an edge exists, the weight of the edge is assigned as the distance between the vertices.

The next set of nested **for** loops implements the core logic of the Floyd-Warshall algorithm. It iterates over **k**, which represents an intermediate vertex that could potentially reduce the distance between two other vertices **i** and **j**.

Inside the **k** loop, it iterates over all pairs of vertices **i** and **j**. For each pair, it updates the distance by considering the possibility of going through the intermediate vertex **k**. It calculates the distance as the minimum of the current distance (**distances[i][j]**) and the sum of the distances from **i** to **k** and from **k** to **j** (**distances[i][k] + distances[k][j]**).

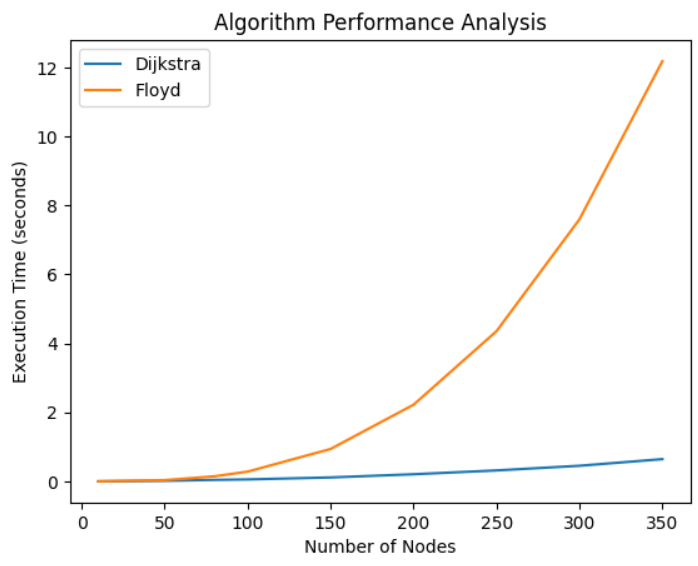
After the **k** loop finishes, the **distances** matrix contains the shortest distances between all pairs of vertices.

The **distances** matrix is returned as the output of the function, representing the shortest paths between all pairs of vertices in the graph.

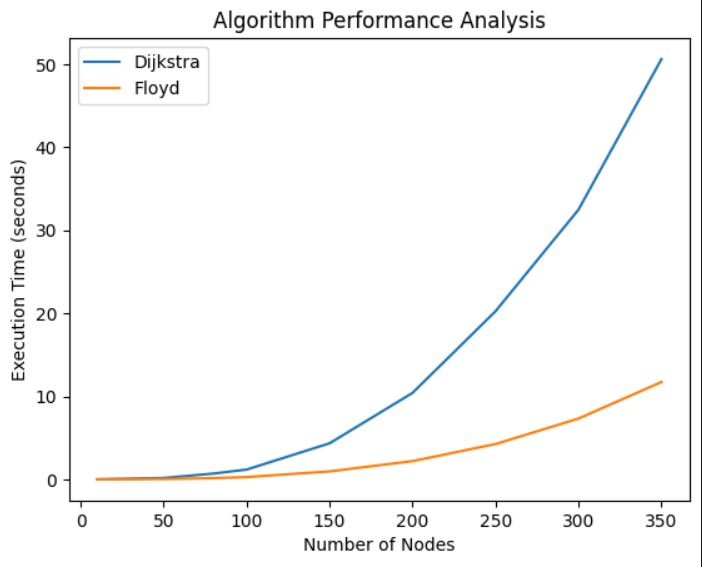


**Figure 3. Implementation Floyd**

**Results for the sparse tree:**



**Figure 4. Plot with results for the sparse tree.**



**Figure 5. Plot with results for the dense tree.**

**Conclusion:**

In summary, Dijkstra's algorithm and Floyd-Warshall algorithm are powerful tools for finding shortest paths in weighted graphs. They differ in their objectives, approach, time complexity, and handling of edge weights. The choice between the two algorithms depends on the specific requirements of the problem, the characteristics of the graph, and the desired output.

Dijkstra's algorithm is relatively easier to implement compared to Floyd-Warshall algorithm. It requires maintaining a priority queue and updating the distances for each vertex. Floyd-Warshall algorithm involves nested loops and requires initializing and updating a matrix of distances for all pairs of vertices.

Dijkstra's algorithm has a time complexity of O((V + E) log V), where V is the number of vertices and E is the number of edges in the graph. It performs well for sparse graphs with a limited number of edges. Floyd-Warshall algorithm has a time complexity of O(V^3), making it suitable for graphs with a moderate number of vertices but denser edge connections.

**Link to GitHub:** <https://github.com/SexomQ/AlgorithmsAnalysis-labs>