**Ministerul Educaţiei și Cercetării al Republicii Moldova Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 4:

Dynamic programming

Elaborated:

st. gr. FAF-22 Gavriliuc Tudor

Verified:

asist. univ. Fiștic Cristofor

prof. univ. Andrievschi-Bagrin Veronica

Chişinău - 2024

**Objective:**

Study and analyze different graph traversing algorithms.

**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

Theoretical Notes:

Empirical analysis provides an alternative approach to understanding the efficiency of algorithms when mathematical complexity analysis is impractical or insufficient. This method proves beneficial in various scenarios:

1. Initial Insights: It offers preliminary insights into an algorithm's complexity class, aiding in the understanding of its efficiency characteristics.
2. Comparative Analysis: It facilitates the comparison of multiple algorithms tackling the same problem, allowing for informed decisions regarding efficiency.
3. Implementation Comparison: Empirical analysis enables the comparison of different implementations of the same algorithm, providing insights into which may perform better in practice.
4. Hardware-specific Evaluation: It helps in assessing an algorithm's efficiency on a particular computing platform, taking into account hardware constraints and capabilities.

The empirical analysis of an algorithm typically involves the following steps:

Establishing Analysis Goals: Clearly define the objectives and scope of the analysis.

1. Choosing Efficiency Metrics: Select appropriate metrics, such as the number of operations executed or the execution time, based on the analysis goals.
2. Defining Input Data Properties: Determine the characteristics of the input data relevant to the analysis, including data size or specific attributes.
3. Implementation: Develop the algorithm in a programming language, ensuring it accurately reflects the intended logic.
4. Generating Input Data Sets: Create multiple sets of input data to cover a range of scenarios and edge cases.
5. Execution and Data Collection: Execute the program for each input data set, recording relevant performance metrics.
6. Data Analysis: Analyze the collected data, either by computing synthetic quantities like mean and standard deviation or by plotting graphs to visualize the relationship between problem size and efficiency metrics.
7. The choice of efficiency measure depends on the analysis's objectives. For instance, if assessing complexity class or verifying theoretical estimates, counting the number of operations may be suitable. Conversely, if evaluating algorithm implementation behavior, measuring execution time becomes more relevant.

8. Post-execution, recorded results undergo analysis. This involves computing statistical measures or plotting graphs to visualize the algorithm's performance characteristics in terms of problem size and efficiency metrics. Such analyses aid in making informed decisions regarding algorithm selection and optimization strategies.

Introduction:

Dijkstra's Algorithm:

Dijkstra's algorithm, named after Dutch computer scientist Edsger W. Dijkstra, is a widely used algorithm for finding the shortest paths between nodes in a graph, particularly for graphs with non-negative edge weights. It is commonly employed in various applications such as network routing protocols and GPS navigation systems.

The algorithm works by iteratively selecting the node with the smallest tentative distance from a set of unvisited nodes and updating the distances to its neighbors accordingly. It maintains a priority queue or a min heap to efficiently select the next node to visit.

Dijkstra's algorithm guarantees the shortest path from a single source node to all other nodes in the graph. However, it does not handle negative edge weights and requires non-negative weights for its correctness.

Time Complexity: O(V^2) with adjacency matrix representation and O((V + E)logV) with adjacency list representation, where V is the number of vertices and E is the number of edges in the graph.

Space Complexity: O(V) for storing distances and predecessors.

Floyd-Warshall Algorithm:

The Floyd-Warshall algorithm is a dynamic programming-based algorithm used to find the shortest paths between all pairs of vertices in a weighted graph, including graphs with negative edge weights (but with no negative cycles). It was developed independently by Bernard Roy in 1959 and later by Stephen Warshall in 1962.

The algorithm works by iteratively considering all pairs of vertices as intermediate vertices and updating the shortest path distances between them. It maintains a two-dimensional array to store the shortest distances between all pairs of vertices.

Floyd-Warshall algorithm provides a convenient way to find the shortest paths between all pairs of vertices in a graph, making it suitable for applications such as network topology analysis and traffic routing.

Time Complexity: O(V^3), where V is the number of vertices in the graph.

Space Complexity: O(V^2) for storing the distance matrix.

## **Comparison Metric:**

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

## **Input Format:**

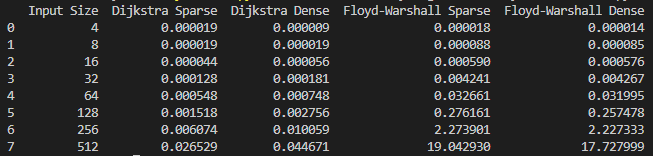
As input, each algorithm will receive 8 series of numbers of nodes 4, 8, 16, 32, 64, 128, 256, ,512.

Next, using this numbers of nodes, it will be generated randomly graphs with that amount of nodes.

**Implementation**

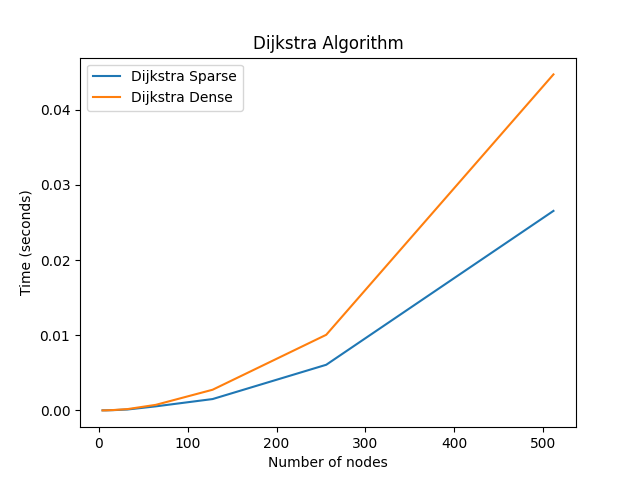
All algorithms will be implemented in their naïve form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on memory of the device used.

Dijkstra Algorithm:

def dijkstra(graph, start):  
 # Initialize distances from start to all other nodes as infinite, except for the start node itself  
 distances = {node: float('inf') for node in graph}  
 distances[start] = 0  
 visited = set()  
  
 # Iterate until all nodes are visited  
 while len(visited) < len(graph):  
 # Select the unvisited node with the smallest distance  
 current\_node = min((node for node in graph if node not in visited), key=lambda x: distances[x])  
 visited.add(current\_node)  
  
 # Update the distance for each adjacent node  
 for neighbor, weight in graph[current\_node].items():  
 new\_distance = distances[current\_node] + weight  
 if new\_distance < distances[neighbor]:  
 distances[neighbor] = new\_distance  
  
 return distances

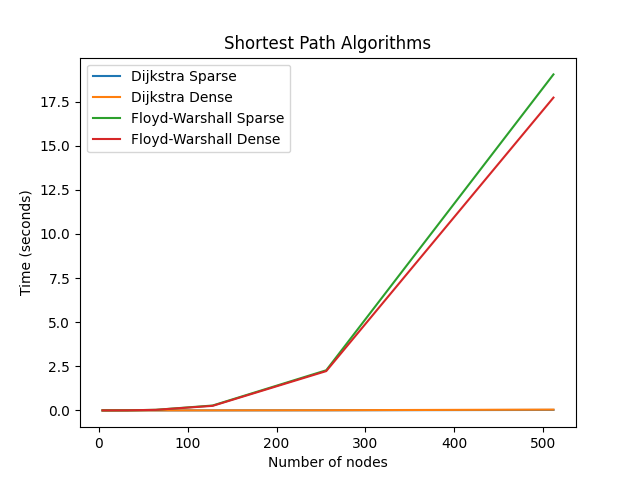
**Figure 1. Time table for algorithms**

As observed in the table, Dijkstra algorithm is faster than Floyd-Warshall in all cases. For a small number of nodes, either the graph is sparse of dense, the algorithms are executed almost in the same time. But as the number of nodes increase, the time also increase considerably for Floyd-Warshall.

**Figure 2. Graph for Dijkstra algorithm**

Floyd-Warshall:

def floyd\_warshall(graph):  
 # Initialize distance matrix from every node to every other node  
 dist = {i: {j: float('inf') for j in graph} for i in graph}  
 for i in graph:  
 dist[i][i] = 0 # Distance to itself is zero  
 for j, w in graph[i].items():  
 dist[i][j] = w # Distance to direct neighbors  
  
 # Update distances across every pair of nodes through another node  
 for k in graph:  
 for i in graph:  
 for j in graph:  
 if dist[i][j] > dist[i][k] + dist[k][j]:  
 dist[i][j] = dist[i][k] + dist[k][j]  
  
 return dist



**Conclusion**

This review highlights the features and performance of Dijkstra's and Floyd-Warshall algorithms for shortest path determination in graphs. Both provide effective methods to compute shortest paths between vertices, but they differ in their operational principles and display varying computational complexities under different conditions. Dijkstra's algorithm uses a greedy method to explore paths from a source node to all other nodes based on the shortest accumulated distances. Its time complexity varies based on its implementation, typically ranging from O(V^2) to O((V + E)logV). This makes it adaptable for both sparse and dense graphs. However, it is limited to non-negative edge weights and cannot process negative cycles. Conversely, the Floyd-Warshall algorithm uses dynamic programming to find shortest paths between all vertex pairs, effectively handling graphs with negative edge weights, provided there are no negative cycles. Although its cubic time complexity of O(V^3) makes it less suitable for large graphs, it is quite effective for moderate sizes. Ultimately, the selection between Dijkstra's and Floyd-Warshall algorithms depends on the graph's specific attributes and the problem's requirements. Dijkstra's is particularly useful where there are non-negative edge weights and the task is to find shortest paths from one source to all vertices.

