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

# Laboratory work 4: Study and empirical analysis of dynamic programming method of designing algorithms. Analysis of Dijkstra and Floyd-Warshall algorithms.

Elaborated:
st. gr. FAF-222 Martiniuc Artiom

Verified:
asist. univ. Fiștic Cristofor

Chişinău - 2024

# TABLE OF CONTENTS

## Objective

ALGORITHM ANALYSIS

Study and analyze different designing algorithms of 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:**

An alternative to mathematical analysis of complexity is empirical analysis.

This may be useful for: obtaining preliminary information on the complexity class of an algorithm; comparing the efficiency of two (or more) algorithms for solving the same problems; comparing the efficiency of several implementations of the same algorithm; obtaining information on the efficiency of implementing an algorithm on a particular computer.

In the empirical analysis of an algorithm, the following steps are usually followed:

1. The purpose of the analysis is established.
2. Choose the efficiency metric to be used (number of executions of an operation (s) or time execution of all or part of the algorithm.
3. The properties of the input data in relation to which the analysis is performed are established (data size or specific properties).
4. The algorithm is implemented in a programming language.
5. Generating multiple sets of input data.
6. Run the program for each input data set.
7. The obtained data are analyzed.

The choice of the efficiency measure depends on the purpose of the analysis. If, for example, the aim is to obtain information on the complexity class or even checking the accuracy of a theoretical estimate then it is appropriate to use the number of operations performed. But if the goal is to assess the behavior of the implementation of an algorithm then execution time is appropriate.

After the execution of the program with the test data, the results are recorded and, for the purpose of the analysis, either synthetic quantities (mean, standard deviation, etc.) are calculated or a graph with appropriate pairs of points (i.e. problem size, efficiency measure) is plotted.

## Introduction:

In this laboratory exercise, we explore and analyze two different dynamic programming algorithms: Dijkstra and Floyd-Warshall. Both algorithms operate on the principle that a shortest path between two vertices in a graph can be determined by building up the solution incrementally, using the idea that the shortest path from point A to point B is a sum of the shortest paths from A to some intermediate point C and from C to B. The key difference is in their purposes and their scalability: Dijkstra is single-source, more scalable but cannot handle negative cycles, whereas Floyd-Warshall is all-pairs, less scalable but can handle graphs with negative cycles as long as there are no negative weight cycles.

As mentioned previously, the performance of an algorithm can be analyzed mathematically (derived through mathematical reasoning) or empirically (based on experimental observations).

Within this laboratory, we will be analyzing the 2 naïve algorithms empirically.

## Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)), The amount of memory used by the algorithm as the graph size increases.

### Input Format:

The input for both algorithms are sparse and dense graphs, with different sizes and densities.

## IMPLEMENTATION

Two algorithms will be implemented in their native 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 o memory of the device used.

The error margin determined will constitute 2.5 seconds as per experimental measurement.

**Dijkstra Algorithm**

The algorithm maintains a set of unvisited nodes and calculates tentative distances from the source node to all others, updating them as shorter paths are found. It uses a priority queue to select the node with the shortest tentative distance to visit next.

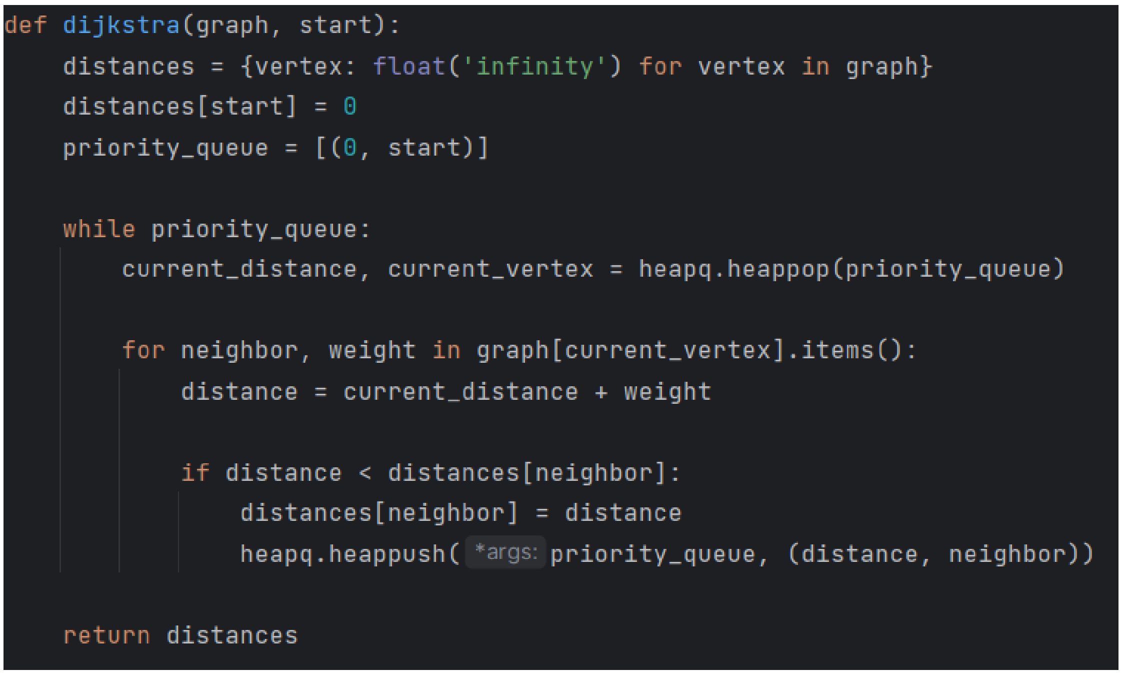
Does not work with graphs containing negative weight edges.

The time complexity can vary depending on the implementation:

O((V + E) log V) with a binary heap or priority queue

O($V^2$) without a priority queue, suitable when the graph is stored as an adjacency matrix.

*Implementation:*



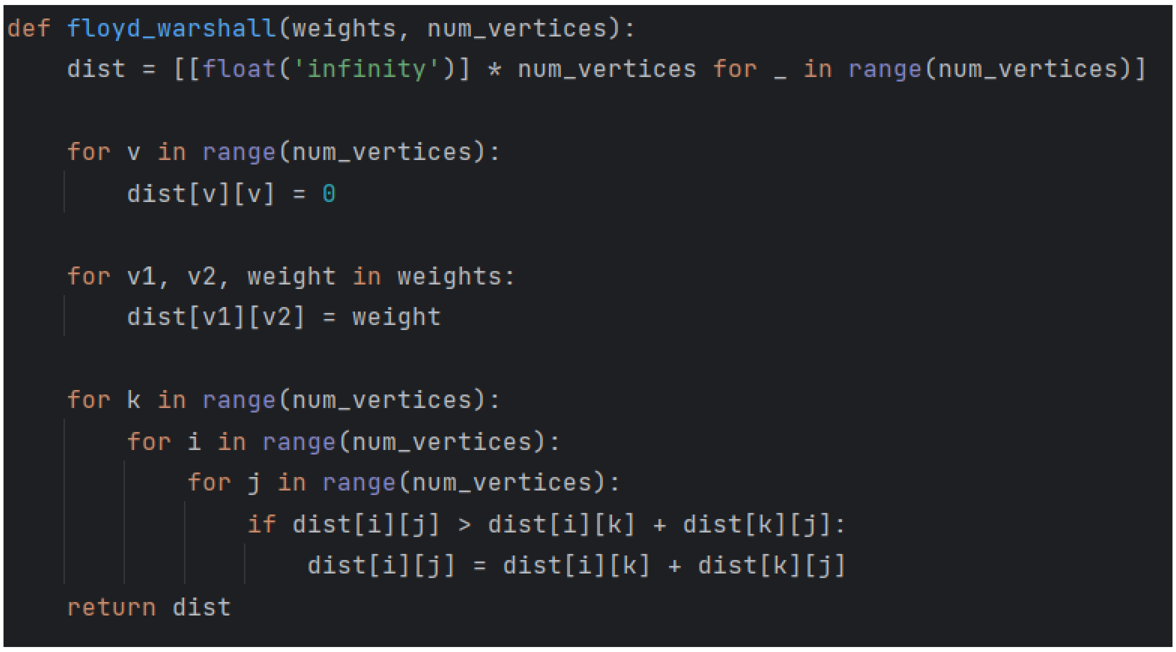
```python
def dijkstra(graph, start):
    distances = {vertex: float('infinity') for vertex in graph}
    distances[start] = 0
    priority_queue = [(0, start)]

    while priority_queue:
        current_distance, current_vertex = heapq.heappop(priority_queue)

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush( *args: priority_queue, (distance, neighbor))

    return distances
```

*Figure 1 Dijkstra in Python*

**Floyd-Warshall Algorithm**

This algorithm uses a dynamic programming approach. It iteratively updates an adjacency matrix to reflect the shortest distance between each pair of nodes. Each iteration allows the algorithm to "consider" intermediate vertices, effectively finding the shortest paths using an increasing number of intermediate steps.

Can handle negative weights and provides the shortest path distance for all pairs of nodes.
Complexity: O($V^3$) where V is the number of vertices in the graph.
*Implementation:*



```python
def floyd_warshall(weights, num_vertices):
    dist = [[float('infinity')] * num_vertices for _ in range(num_vertices)]

    for v in range(num_vertices):
        dist[v][v] = 0

    for v1, v2, weight in weights:
        dist[v1][v2] = weight

    for k in range(num_vertices):
        for i in range(num_vertices):
            for j in range(num_vertices):
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
    return dist
```

*Figure 2 Floyd-Warshall in Python*

## RESULTS

The provided graphs display the execution time of Dijkstra's and Floyd-Warshall algorithms applied to sparse and dense graphs with an increasing number of nodes. The observations are as follows: As you can see, both algorithms exhibit a trend where the time increases with the number of nodes. However, the exact performance and efficiency depend on the structure of the graph and how nodes are connected.

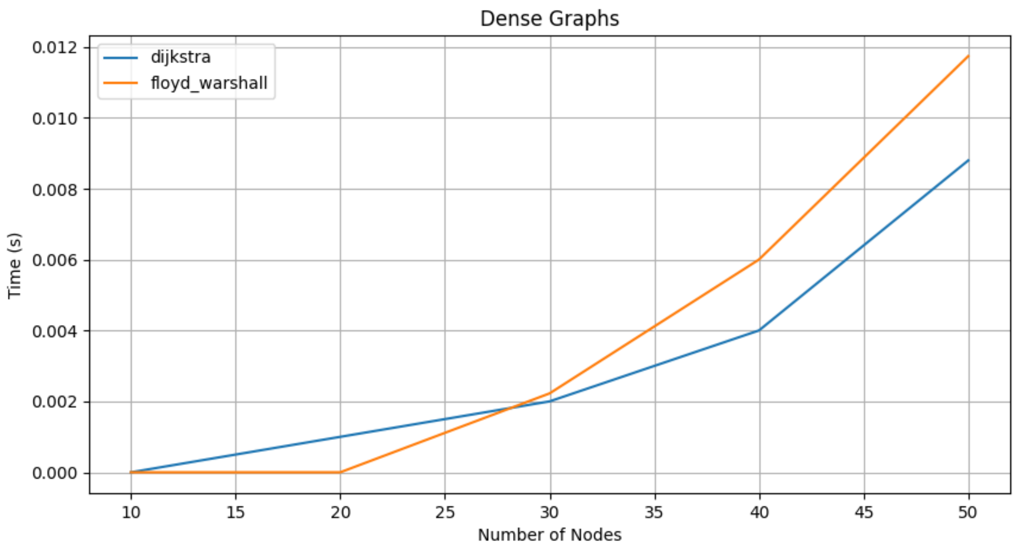


*Figure 3 Graph of Algorithms Performance in Dense Graphs*

Both algorithms exhibit an increase in execution time as the number of nodes increases.

Initially, Dijkstra's algorithm performs better than Floyd-Warshall, suggesting that despite the graph's density, the priority queue optimization is effective.

However, as the number of nodes grows, Floyd-Warshall begins to catch up with Dijkstra's algorithm. This trend might continue, and Floyd-Warshall could potentially outperform Dijkstra as the graph becomes even denser or the number of nodes continues to increase.

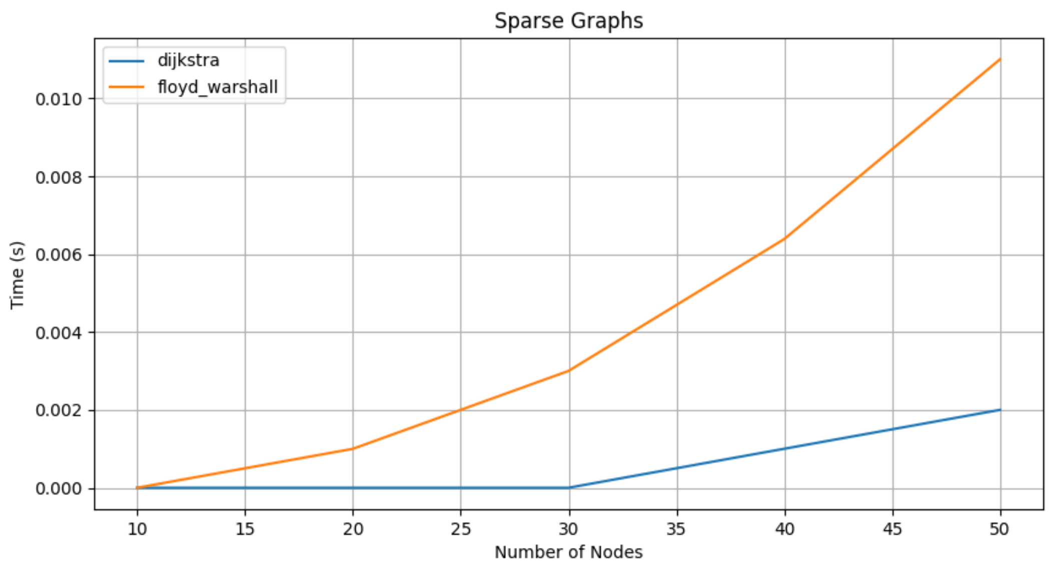


*Figure 4 Graph of Algorithms Performance in Sparse Graphs*

The performance gap between the two algorithms is more pronounced. Dijkstra's algorithm significantly outperforms Floyd-Warshall, taking much less time to complete.

The execution time for Dijkstra's algorithm grows linearly with the number of nodes, which aligns with expectations for sparse graphs where the number of edges does not increase quadratically with the number of nodes.

Floyd-Warshall's performance suffers due to its cubic time complexity, which does not benefit from the sparsity of the graph.

Both algorithms show increased computation times as the number of nodes increases, but the increase is more dramatic with the Floyd-Warshall algorithm due to its cubic time complexity. Dijkstra’s algorithm scales better, especially for sparse graphs.

# CONCLUSION

In practice, choosing the appropriate algorithm depends on the specific characteristics of the graph. For large sparse graphs, Dijkstra's algorithm is clearly more efficient. For smaller graphs or those where all-pairs shortest paths are frequently needed, Floyd-Warshall may be more appropriate despite the higher computational cost.

As the data suggests, for dense graphs and large numbers of nodes, there may be a tipping point where the Floyd-Warshall algorithm becomes more suitable. However, for most large-scale applications involving sparse graphs, Dijkstra's algorithm remains the preferred choice due to its scalability and lower execution times.