**Ministerul Educaţiei și Cercetării al Republicii Moldova**

**Universitatea Tehnică a Moldovei**

**Facultatea Calculatoare, Informatică și Microelectronică**

Laboratory work 4:

Study and Empirical Analysis of Algorithms: Dijkstra and Floyd–Warshall

Elaborated:

st. gr. FAF-233Iamandii Ion

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

# ALGORITHM ANALYSIS

## Objective

## Analysis of Dijkstra and Floyd–Warshall using dynamic programming

## Tasks:

## 1 Implement the algorithms listed above 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 these algorithms for a sparse graph and for a dense graph.

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

## 6 Make a conclusion on the work done.

## 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:

## Dynamic programming approach is similar to divide and conquer in breaking down the problem into smaller and yet smaller possible sub-problems. But unlike divide and conquer, these sub-problems are not solved independently. Rather, results of these smaller sub-problems are remembered and used for similar or overlapping sub-problems.

## Mostly, dynamic programming algorithms are used for solving optimization problems. Before solving the in-hand sub-problem, dynamic algorithm will try to examine the results of the previously solved sub-problems. The solutions of sub-problems are combined in order to achieve the best optimal final solution. This paradigm is thus said to be using Bottom-up approach.

## 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 the shortest path algorithms consists of directed, weighted graphs represented as adjacency lists (for Dijkstra) and adjacency matrices (for Floyd–Warshall). Each graph contains a variable number of nodes (e.g., 10, 50, 100, up to 500), with randomly generated edge weights ranging between 1 and 10. For each node count, two graph types are constructed: sparse graphs (20% edge probability) and dense graphs (80% edge probability), simulating low- and high-connectivity environments. Each graph assumes unique node identifiers from 0 to n − 1, and edge weights are strictly positive to ensure compatibility with Dijkstra’s assumptions. Dijkstra's algorithm is run from a fixed starting node (node 0) to compute single-source shortest paths, while Floyd–Warshall computes all-pairs shortest paths. This structured variation in graph density and size enables a detailed empirical comparison of algorithmic behavior under different computational loads.

# IMPLEMENTATION

Both Dijkstra’s algorithm (using a priority queue with adjacency list representation) and the Floyd–Warshall algorithm (using dynamic programming with a 2D matrix) are implemented in Python and tested empirically. Dijkstra is evaluated on a per-source basis, while Floyd–Warshall exhaustively computes all pairwise shortest paths. Execution time is measured for each algorithm on both sparse and dense graphs across multiple input sizes. The results are plotted for comparative analysis. Due to Floyd–Warshall’s cubic time complexity, it exhibits significant slowdowns on large graphs, particularly in dense configurations. In contrast, Dijkstra demonstrates better scalability with sparse graphs but may still experience performance degradation on dense graphs due to increased edge processing. The observed timings reflect both algorithmic efficiency and system-dependent runtime behavior, allowing for meaningful conclusions regarding their practical applicability.

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

Github repo: <https://github.com/ion190/aa-labs/tree/main/lab4>

## Dijkstra Algorithm:

Dijkstra’s Algorithm is used to resolve the Single Source Shortest Path issue. In other words, we want to identify the shortest route between a particular source node and a specific destination node. This algorithm is used effectively in the link-state routing protocol, where each node applies it to build an internal representation of the network.

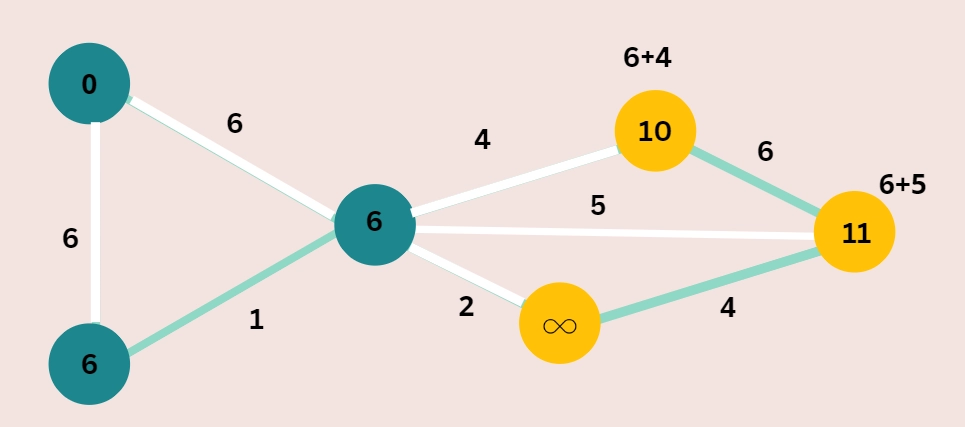


Figure 1. Dijkstra example

*Algorithm Description:*

The Dijkstra algorithm follows the algorithm as shown in the next pseudocode:

function Dijkstra(Graph, source):

for each vertex v in Graph.Vertices:

dist[v] ← INFINITY

prev[v] ← UNDEFINED

add v to Q

dist[source] ← 0

while Q is not empty:

u ← vertex in Q with minimum dist[u]

Q.remove(u)

for each arc (u, v) in Q:

alt ← dist[u] + Graph.Edges(u, v)

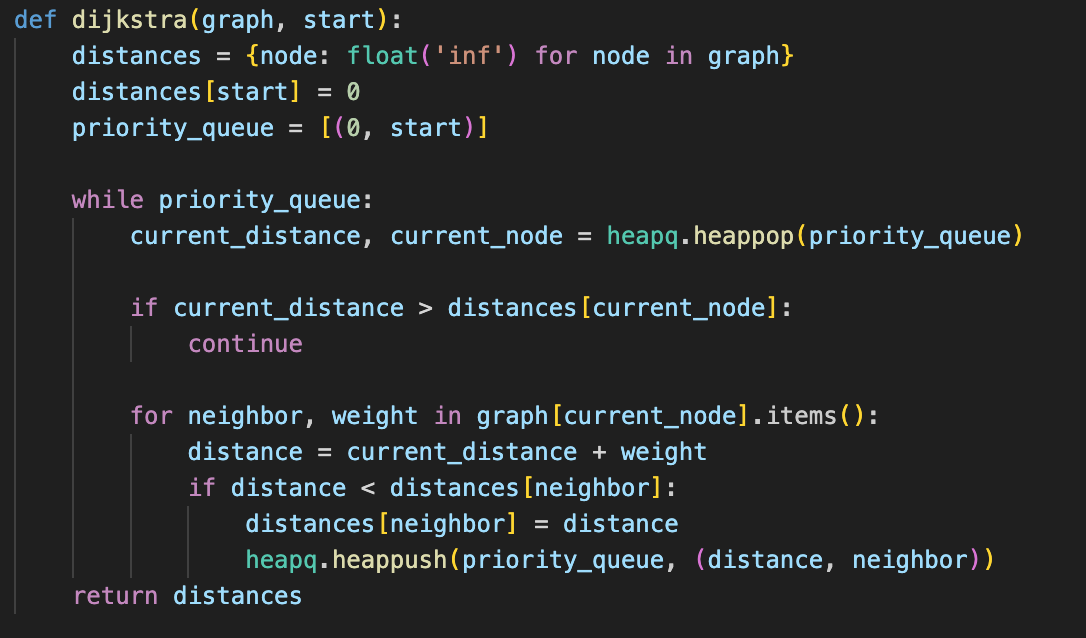
if alt < dist[v]:

dist[v] ← alt

prev[v] ← u

return dist[], prev[]

*Implementation:*



*Figure 2 Dijkstra algorithm in Python*

## Floyd–Warshall Algorithm:

The Floyd–Warshall’s Algorithm is used to find the All-Pairs Shortest Paths solution. We focus on determining the graph's shortest paths—a more time-consuming computing task—between each pair of nodes. Both the storage space and processing time needed for graph data are examples of how this computational cost is visible. However, because of how easily it can be implemented, the Floyd–Warshall’s Algorithm is still valuable.

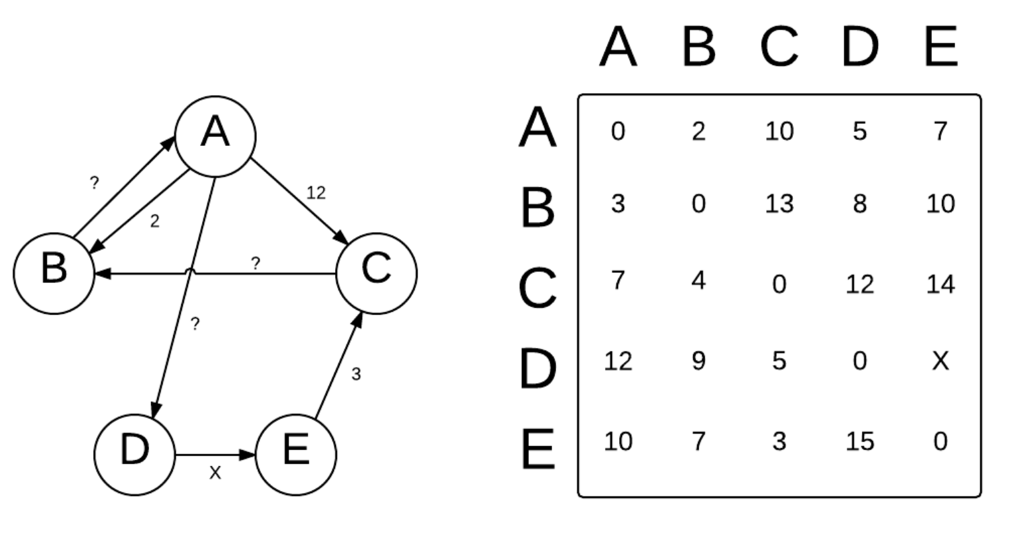


Figure 3. Floyd–Warshall’s Algorithm example

*Algorithm Description:*

The Floyd–Warshall’s Algorithm follows the algorithm as shown in the next pseudocode:

let dist be a |V| × |V| array of minimum distances initialized to ∞ (infinity)

for each edge (u, v) do

dist[u][v] = w(u, v) // The weight of the edge (u, v)

for each vertex v do

dist[v][v] = 0

for k from 1 to |V|

for i from 1 to |V|

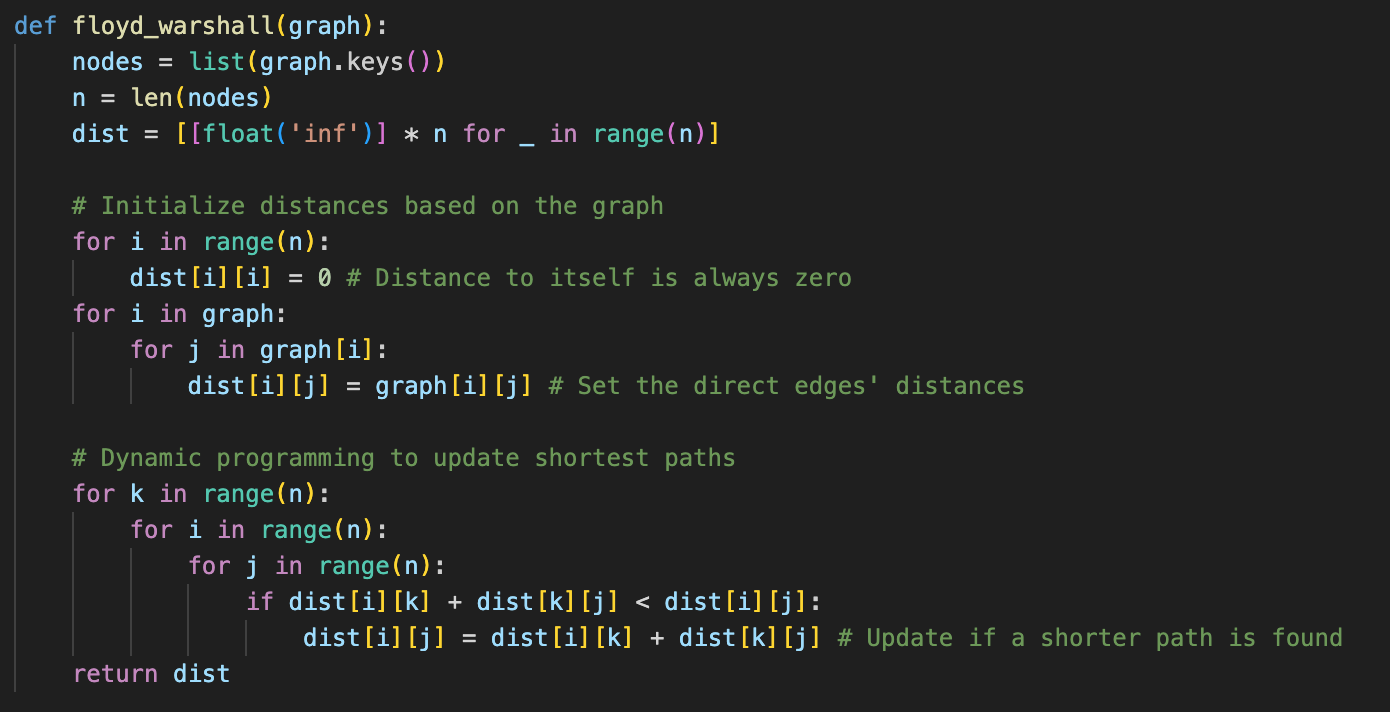
for j from 1 to |V|

if dist[i][j] > dist[i][k] + dist[k][j]

dist[i][j] = dist[i][k] + dist[k][j]

end if

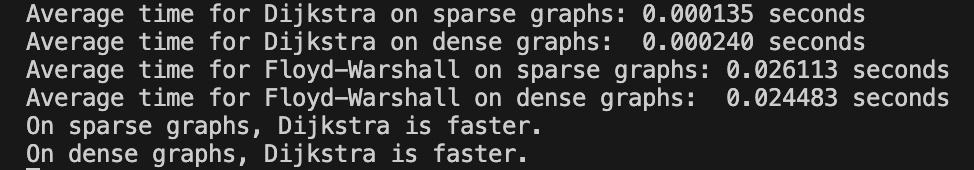
*Implementation:*



*Figure 4 Floyd–Warshall’s Algorithm in Python*

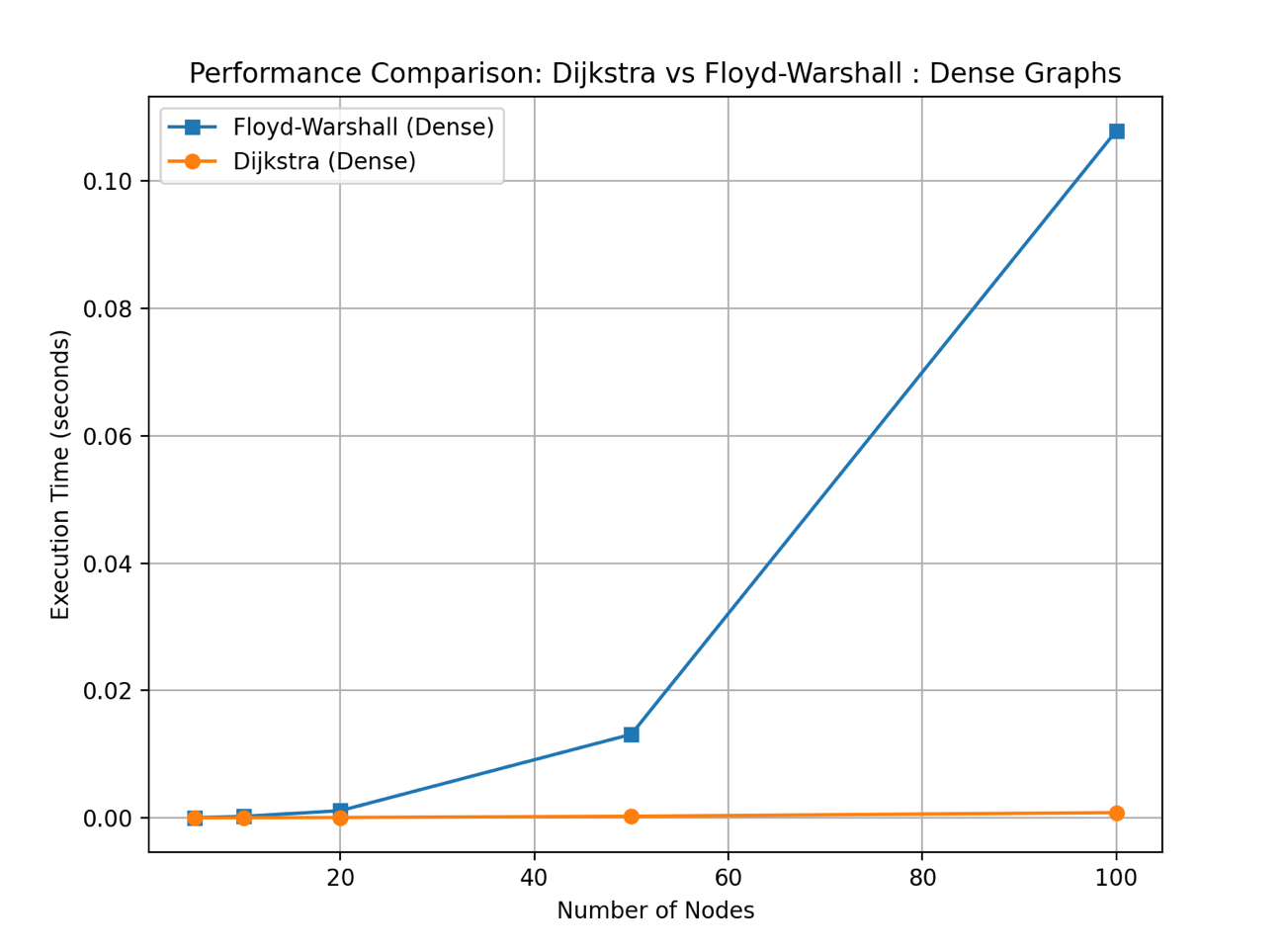
***Results:***

After running the function for different sizes of graphs and saving the time for each, we obtained the following results:



*Figure 5 Results for various set of inputs*

The graph showing the time needed to sort different sizes of graphs:



*Figure 6 Graph of Dijkstra and Floyd–Warshall algorithms*

## Dijkstra’s algorithm has a time complexity of O((V+E)logV)O((V+E)logV) when implemented with a priority queue (min-heap), where VV is the number of vertices and EE is the number of edges. In sparse graphs, where EE is closer to VV, this results in nearly linear performance with respect to the number of nodes, making Dijkstra efficient for large but lightly connected graphs. In dense graphs, where EE approaches V2V2, the logarithmic heap operations amplify due to the higher edge count, increasing the total execution time, though the algorithm still scales better than cubic approaches. Practical performance also depends on the heap implementation and the frequency of distance updates.

## Floyd–Warshall, in contrast, has a fixed time complexity of O(V3)O(V3), regardless of the number of edges, due to its triple nested loops that compare all pairs of vertices through all possible intermediate nodes. This makes it particularly inefficient for large graphs, especially sparse ones where the majority of comparisons are unnecessary. However, for dense graphs, where most node pairs are interconnected, the algorithm’s exhaustive approach ensures it finds all shortest paths reliably. Despite its higher computational cost, Floyd–Warshall is simple to implement and excels in smaller or fully connected graphs where all-pairs shortest path information is needed.

# CONCLUSION

Through empirical analysis, this study evaluates the performance of two shortest-path algorithms—Dijkstra’s algorithm and the Floyd–Warshall algorithm—based on execution time and scalability across sparse and dense graphs of increasing sizes, with the objective of determining their optimal contexts of use.

Dijkstra’s algorithm, implemented with a min-heap priority queue, efficiently computes single-source shortest paths with a time complexity of O((V+E)log⁡V)O((V+E)logV). Its performance is highly favorable in sparse graphs, where the lower edge count results in fewer heap operations and reduced traversal overhead. Even in dense graphs, Dijkstra remains practical, though execution time increases due to the growing number of edges. Its efficiency and flexibility make it a robust choice for large graphs where shortest paths from a specific source are needed.

The Floyd–Warshall algorithm, by contrast, computes all-pairs shortest paths with a uniform time complexity of O(V3)O(V3), regardless of graph density. While this approach guarantees comprehensive shortest-path data, it becomes computationally expensive for large graphs, especially sparse ones where many comparisons are unnecessary. In dense graphs, however, its exhaustive method aligns better with the connectivity pattern, providing complete results in a straightforward and deterministic manner.

Based on the empirical results, Dijkstra’s algorithm is significantly more efficient for sparse graphs and scalable to larger inputs, making it suitable for real-world networks with limited connections. Floyd–Warshall is better suited for smaller or densely connected graphs where all-pairs shortest paths are required, despite its higher time complexity. Each algorithm thus serves distinct needs, and their applicability depends on the graph’s structure and the problem’s scope.