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

Laboratory work 4:

Study and empirical analysis of Dijkstra and Floyd-Warshall algorithms.

Elaborated:
st. gr. FAF-233 Ploteanu Augustin

Verified:

asist. univ. Fiștic Cristofor

Chişinău - 2025

TABLE OF CONTENTS

ALGORITHM ANALYSIS **3**
Objective……3
Tasks……...3
Theoretical Notes:...……3
Introduction:...……4
Comparison Metric.……4
Input Format:..……4
IMPLEMENTATION **5**
Dijkstra's Algorithm..……5
Floyd-Warshall Algorithm……..7
CONCLUSION **10**

ALGORITHM ANALYSIS

Objective

Study and empirical analysis of shortest path algorithms. Analysis of Dijkstra and Floyd-Warshall algorithms.

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 the proposed algorithms;
5. 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:

Shortest path algorithms are pivotal in numerous computational fields, including transportation networks, routing protocols, and game development. These algorithms aim to find the minimal cost path between nodes in weighted graphs.

In this laboratory, we compare two classical shortest path algorithms:

Dijkstra's Algorithm – A greedy algorithm that efficiently computes shortest paths from a single source node to all other nodes in a graph with non-negative weights.

Floyd-Warshall Algorithm – A dynamic programming approach that computes the shortest paths between all pairs of nodes, making it suitable for dense graphs or applications requiring full path matrices.

While both algorithms solve shortest path problems, they differ greatly in time complexity and operational scope. Our analysis will highlight the trade-offs between scalability, graph density, and algorithmic overhead through experimental performance comparisons.

Comparison Metric:

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

Input Format:

To evaluate the performance of graph algorithms under varying structural conditions, eight distinct graph types of increasing size will be analyzed. These graph types include:

-Complete graphs, where every node is connected to every other node.

-Sparse graphs, which maintain minimal connectivity.

-Dense graphs, with a high but non-complete number of edges.

-Trees, which are acyclic connected graphs.

-Bipartite graphs, where nodes are divided into two sets with edges only between sets.

-Cyclic graphs, containing multiple cycles.

-Acyclic graphs, such as directed acyclic graphs (DAGs).

-Grid graphs, structured like 2D matrices.

Each graph type will be generated at six different sizes, ranging from 10 to 400 nodes. For weighted and directed variants, edges will be assigned random weights and directions.

IMPLEMENTATION

Both algorithms will be implemented in python and 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 report with input will vary depending on memory of the device used.

Dijkstra's Algorithm:

Dijkstra's Algorithm is a single-source shortest path algorithm for graphs with non-negative edge weights. It uses a priority queue to iteratively select the node with the smallest known distance and updates distances to its neighbors. With a time complexity of O((V + E) log V) when using a min-heap, Dijkstra is both efficient and accurate for a wide range of network routing and pathfinding problems. However, it is not suitable for graphs with negative edge weights.

Algorithm Description:

Dijkstra's Algorithm follows the algorithm as shown in the next pseudocode:

```
DIJKSTRA(graph, start)
    create distance map dist, initialized to ∞ for all nodes
    set dist[start] = 0
    create min-priority queue Q
    insert (0, start) into Q

    while Q is not empty do
        (current_dist, current_node) = extract_min(Q)
        for each (neighbor, weight) in graph[current_node] do
            distance = current_dist + weight
            if distance < dist[neighbor] then
                dist[neighbor] = distance
                insert (distance, neighbor) into Q

    return dist
```

Implementation:

```python
def dijkstra(graph, start):
    heap = [(0, start)]
    distances = {node: float('inf') for node in graph}
    distances[start] = 0

    while heap:
        current_dist, current_node = heapq.heappop(heap)
        for neighbor, weight in graph[current_node]:
            distance = current_dist + weight
            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(heap, (distance, neighbor))
    return distances
```

Figure 1 Dijkstra's algorithm in Python

Results:

The numerical results after execution of the algorithms on the aforementioned graphs can be seen in the appendix. Dijkstra consistently delivers low runtimes, demonstrating excellent scalability and efficiency, especially on sparse and tree-like graphs. It significantly outperforms Floyd-Warshall on all graph types due to its optimized handling of single-source shortest paths and avoidance of unnecessary all-pairs computations.

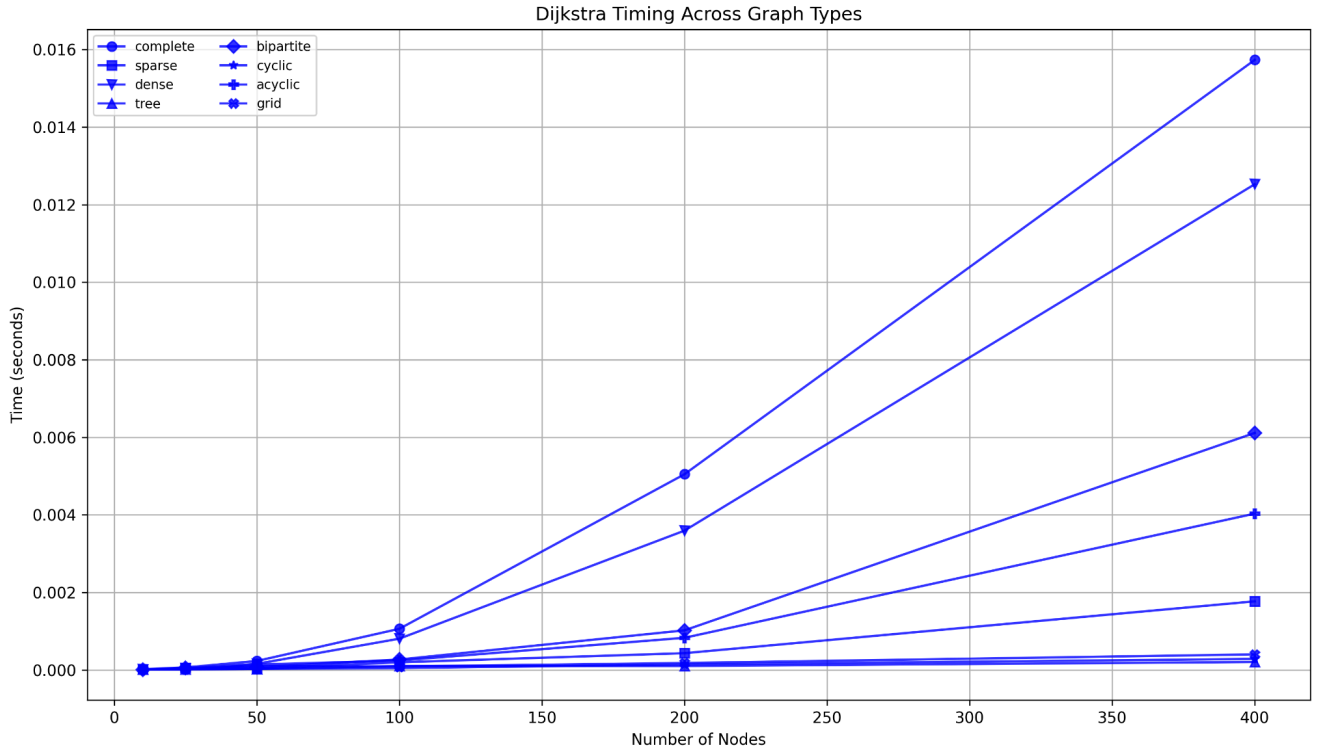

Figure 2 Dijkstra's algorithm graph

Floyd-Warshall Algorithm:

Floyd-Warshall is an all-pairs shortest path algorithm based on dynamic programming. It iteratively updates a distance matrix by considering every possible intermediate node between all pairs of vertices. The algorithm has a time complexity of $O(V^3)$, making it less efficient for large sparse graphs but highly effective for dense graphs or when all-pairs distance information is required. It supports graphs with negative weights, provided there are no negative cycles.

Algorithm Description:

Floyd-Warshall Algorithm follows the algorithm as shown in the next pseudocode:

```
FLOYD-WARSHALL(graph)
    let dist be a 2D array of size V × V initialized to ∞
    for each node i do
        dist[i][i] = 0
        for each (j, weight) in graph[i] do
            dist[i][j] = weight

    for k = 1 to V do
        for i = 1 to V do
            for j = 1 to V do
                if dist[i][k] + dist[k][j] < dist[i][j] then
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

Implementation:

```python
def floyd_warshall(graph):
    nodes = list(graph.keys())
    dist = {i: {j: float('inf') for j in nodes} for i in nodes}

    for node in nodes:
        dist[node][node] = 0
        for neighbor, weight in graph[node]:
            dist[node][neighbor] = weight

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

    return dist
```

Figure 3 Floyd-Warshall algorithm in Python

Results:

The numerical results after execution of the algorithms on the aforementioned graphs can be seen in the appendix. Floyd-Warshall performs well on very small graphs but becomes drastically slower as graph size increases, particularly in dense and grid graphs where its cubic time complexity dominates.

Despite its accuracy and simplicity, it is not practical for large-scale or sparse graph scenarios due to its heavy computational load.

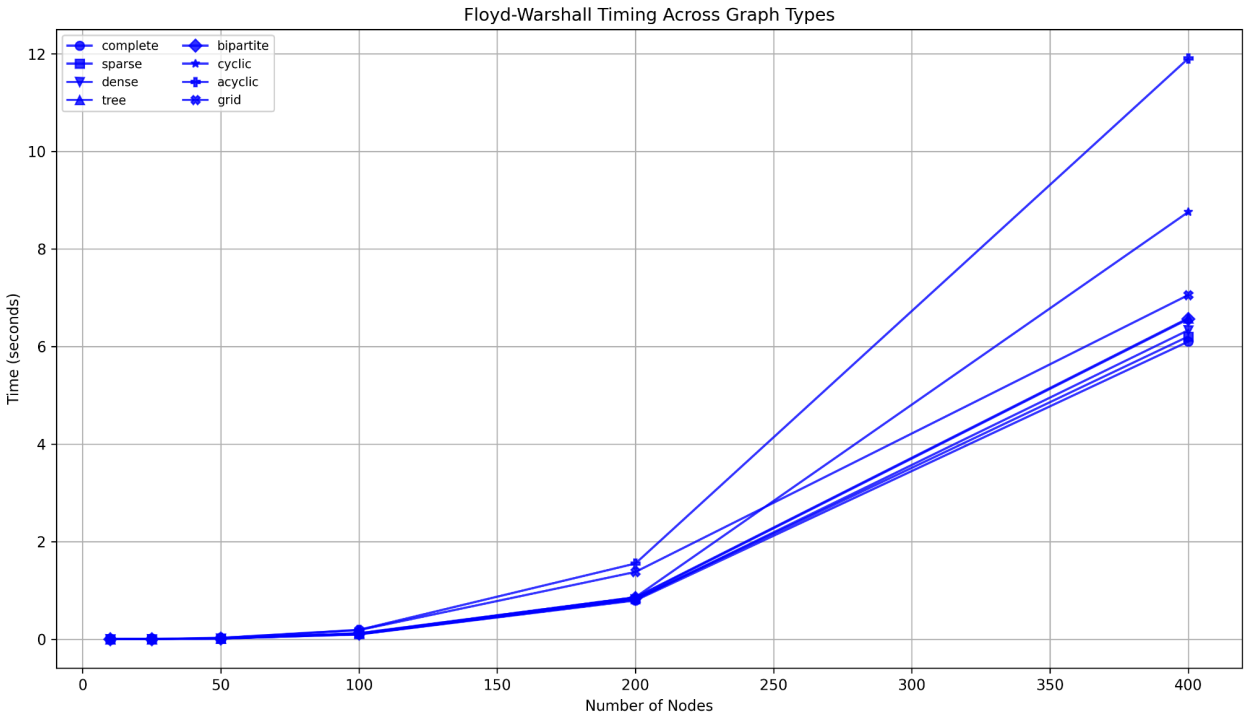

Figure 4 Floyd-Warshall algorithm graph

CONCLUSION

Through empirical analysis, this study has evaluated the efficiency of two shortest path algorithms, Dijkstra and Floyd-Warshall, by analyzing their execution time under different input conditions. The goal was to identify the scenarios in which each algorithm performs optimally and to determine their feasibility for various applications. The results provide insights into the strengths and limitations of each approach, as well as opportunities for further optimizations.

Dijkstra's algorithm proved highly efficient and scalable, especially for sparse, tree, acyclic, and grid graphs, where the number of edges is small and shortest paths from a single source are sufficient. Its use of a priority queue ensures fast lookups and updates in large networks, provided the weights are non-negative. Dijkstra performed well in both undirected and directed graphs without negative weights.

Floyd-Warshall, in contrast, is a brute-force all-pairs shortest path algorithm with cubic time complexity. While impractical for large or sparse graphs, it remains useful in complete and dense graphs or cases requiring full pairwise distance matrices. Notably, it supports directed, weighted, and even negatively weighted graphs, provided there are no negative cycles.

In conclusion, Dijkstra is optimal for large, sparse graphs requiring single-source shortest paths, and should be the default for most practical applications with non-negative weights. Floyd-Warshall is better suited for dense or fully connected graphs, or when working with directed graphs containing negative weights, and a full distance matrix is needed.

Numerical Results Appendix:

https://github.com/augustin-ploteanu/AA_Labs/blob/main/Lab3-4-5/results.txt