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: Dynamic Programming

Elaborated:
st. gr. FAF-221 Chichioi Iuliana

Verified:

asist. univ. Fiștic Cristofor

prof. univ. Andrievschi-Bagrin Veronica

Chişinău - 2024

TABLE OF CONTENTS

ALGORITHM ANALYSIS

Objective

Study and analyse the dynamic programming method of designing algorithms.

Tasks:

1. To implement in a programming language algorithms Dijkstra and Floyd–Warshall using dynamic programming.
2. Do empirical analysis of these algorithms for a sparse graph and for a dense graph.
3. Increase the number of nodes in graphs and analyse how this influences the algorithms. Make a graphical presentation of the data obtained
4. 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 analysed.

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 behaviour 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 offers powerful tools for solving optimization problems by breaking them down into simpler subproblems and utilising the solutions to those subproblems to construct the overall solution efficiently. Among the prominent algorithms utilising dynamic programming are Dijkstra's algorithm and the Floyd–Warshall algorithm, both integral in solving various graph-related challenges such as finding shortest paths between nodes in weighted graphs.

Dijkstra's algorithm, named after Dutch computer scientist Edsger W. Dijkstra, efficiently determines the shortest path from a single source vertex to all other vertices in a graph with non-negative edge weights. It achieves this by iteratively selecting the vertex with the shortest distance from the source vertex and relaxing its adjacent vertices.

The Floyd–Warshall algorithm, on the other hand, is designed to find the shortest paths between all pairs of vertices in a weighted graph, regardless of whether the edges have negative weights. This algorithm works by considering all possible intermediate vertices in a path and systematically updating the shortest path distances between every pair of vertices. Understanding the mechanics and nuances of Dijkstra's algorithm and the Floyd–Warshall algorithm is crucial for effectively addressing optimization problems in various domains, including transportation networks, computer networking, and resource allocation.

Comparison Metric:

In this analysis, the performance of Dijkstra's algorithm and the Floyd–Warshall algorithm will be evaluated based on their execution time (T(n)), where 'n' represents the number of nodes in the graphs being analysed.

Input Format:

The input format for the performance analysis includes a list of integers representing the number of nodes in the graphs under examination.

The list contains values such as [10, 20, 30, 50], each denoting the number of nodes in a specific graph.

These values are utilised to generate graphs of varying sizes, facilitating the evaluation of the runtime complexities of Dijkstra's algorithm and the Floyd–Warshall algorithm across different graph dimensions.

IMPLEMENTATION

All algorithms will be implemented in their native form in python and analysed 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.

Dynamic Programming or DP:

Dynamic Programming is a method used in mathematics and computer science to solve complex problems by breaking them down into simpler subproblems. By solving each subproblem only once and storing the results, it avoids redundant computations, leading to more efficient solutions for a wide range of problems.

How Does Dynamic Programming (DP) Work?

- **Identify Subproblems:** Divide the main problem into smaller, independent subproblems.
- **Store Solutions:** Solve each subproblem and store the solution in a table or array.
- **Build Up Solutions:** Use the stored solutions to build up the solution to the main problem.
- **Avoid Redundancy:** By storing solutions, DP ensures that each subproblem is solved only once, reducing computation time.

When to Use Dynamic Programming (DP)?

Dynamic programming is an optimization technique used when solving problems that consists of the following characteristics:

1. Optimal Substructure:

Optimal substructure means that we combine the optimal results of subproblems to achieve the optimal result of the bigger problem.

For example, Binary Search does not have overlapping sub-problem. Whereas recursive program of Fibonacci numbers have many overlapping sub-problems.

2. Overlapping Subproblems:

The same subproblems are solved repeatedly in different parts of the problem.

For example, the Shortest Path problem has the following optimal substructure property:

If a node x lies in the shortest path from a source node **u** to destination node **v**, then the shortest path from **u** to **v** is the combination of the shortest path from **u** to x, and the shortest path from x to **v**.

The standard All Pair Shortest Path algorithms like Floyd-Warshall and Bellman-Ford are typical examples of Dynamic Programming.

Dijkstra's 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.

Algorithm Description:

Step 1: Create a weighted graph first, where each branch is given a certain numerical weight. Similarly, a weighted graph is a specific type of labelled graph where the labels are numerical values (which are positive).

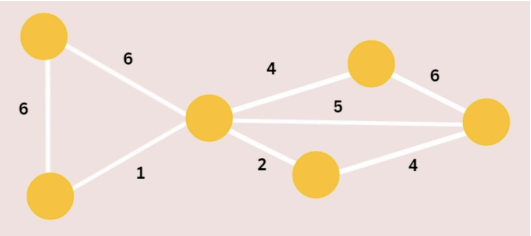

Figure 1 Step 1 Dijkstra's A.

Step 2: Choose a starting vertex, and give all other components route values of infinite.

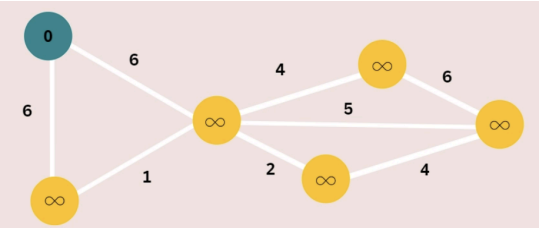

Figure 2 Step 2 Dijkstra's A.

Step 3: Update each vertex's path length stopping by.

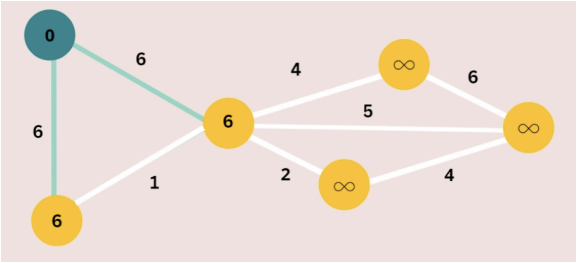

Figure 3 Step 3 Dijkstra's A.

Step 4: Do not update a vertex if its path length is less than the new path length.

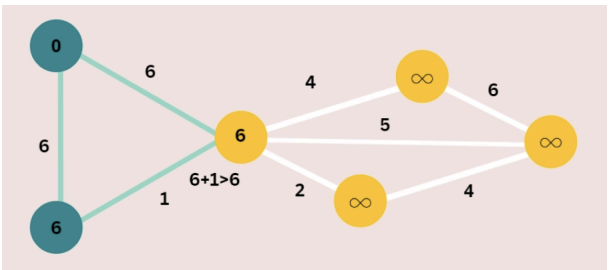

Figure 4 Step 4 Dijkstra's A.

Step 5: Don't alter the path lengths of vertices that have already been visited.

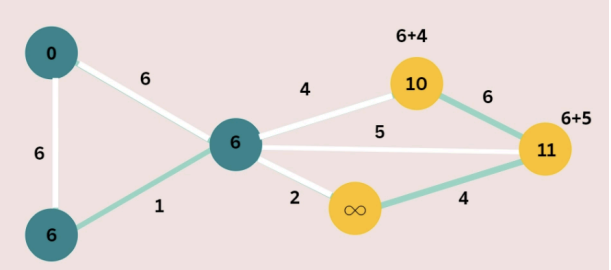

Figure 5 Step 5 Dijkstra's A.

Dijkstra's Algorithm adheres to the concept of greed. As a result, it terminates with a globally optimal solution by making locally optimal decisions at each stage. The optimal substructure and greedy choice property are the distinguishing features of a problem that a greedy algorithm can solve. These algorithms run in a top-down manner.
By "optimal substructure," we imply that a problem's optimal solution is made up of the best possible answers to each of its subproblems. For instance, the Unweighted Longest Simple Path issue lacks optimum substructure while the SSSP problem does.

Complexity Analysis of Dijstra's Algorithm:

Algorithm	Time Complexity	Space Complexity
Dijkstra's Algorithm + Array	O(V*V)	O(V)
Dijkstra's Algorithm + Min-Heap	O(ElogV)	O(V)
Dijkstra's Algorithm + Fibonacci heap	O(E+VlogV)	O(V)

Implementation:

```
# Dijkstra's Algorithm
def dijkstra(graph, start):
   distances = {node: float('inf') for node in graph}
   distances[start] = 0
   queue = [(0, start)]

   while queue:
       current_distance, current_node = heapq.heappop(queue)

       if current_distance > distances[current_node]:
           continue

       for neighbor, weight in graph[current_node].items():
           distance = current_distance + weight
           if distance < distances[neighbor]:
               distances[neighbor] = distance
               heapq.heappush(queue, (distance, neighbor))

   return distances
```

Figure 6 Code Implementation Dijkstra's A.

Results:

```
Sparse Graph - Dijkstra's Time: 3.266334533691406e-05
Sparse Graph - Floyd-Warshall's Time: 0.00023126602172851562
Dense Graph - Dijkstra's Time: 6.818771362304688e-05
Dense Graph - Floyd-Warshall's Time: 0.000164031982421875
```

Figure 7 Results of program

Graph:

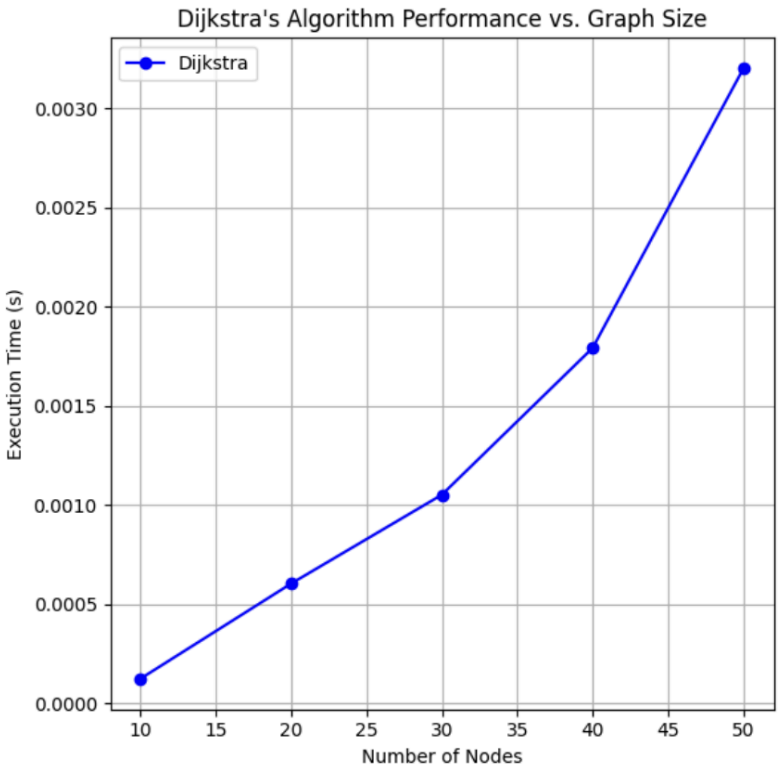

Figure 8 Graphical Representation

The graphical representation in the code showcases the performance of Dijkstra's algorithm and Floyd-Warshall algorithm as the number of nodes in the graph varies. The main plot illustrates the execution time (in seconds) of both algorithms against the number of nodes, providing insights into their scalability. Additionally, separate plots are presented for Dijkstra's algorithm and Floyd-Warshall algorithm, offering a detailed comparison of their performance. The use of markers distinguishes individual data points, while colour coding (blue for Dijkstra and green for Floyd-Warshall) facilitates easy identification of each algorithm's trend. Overall, these graphical representations aid in understanding the efficiency and behaviour of the algorithms across different graph sizes.

Floyd-Warshall's Algorithm:

The **Floyd-Warshall algorithm**, named after its creators **Robert Floyd and Stephen Warshall**, is a fundamental algorithm in computer science and graph theory. It is used to find the shortest paths between all pairs of nodes in a weighted graph. This algorithm is highly efficient and can handle graphs with both **positive** and n**egative edge weights**, making it a versatile tool for solving a wide range of network and connectivity problems.

Algorithm Description:

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.

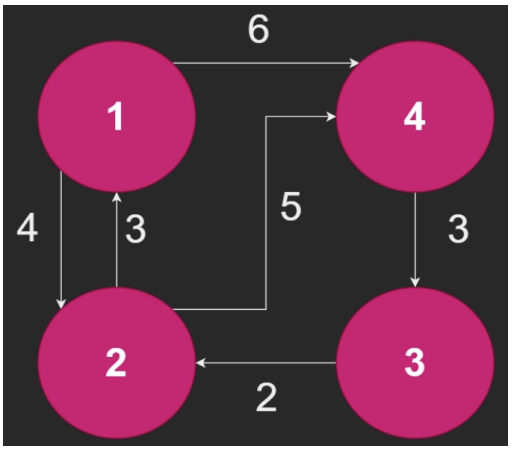

Figure 9 Floyd-Warshall's A.

Floyd–Warshall's Algorithm, in contrast, adheres to the dynamic programming (DP) paradigm. These algorithms either operate top-down with applied memoization or build solutions from the bottom up. The optimal substructure and overlapping subproblems are characteristics of DP-solvable problems.

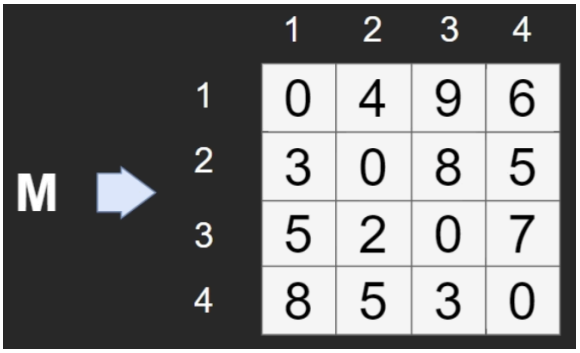

Figure 10 Floyd-Warshall's A.

If the same subproblems are solved at different stages of the method, a problem is said to have overlapping subproblems. Because of this, no new subproblems are created within its narrow subproblem space.

Complexity Analysis of Floyd-Warshall's Algorithm:

Algorithm	**Time Complexity**	**Space Complexity**
Floyd–Warshall's Algorithm	O(V*V*V)	O(V*V)

Implementation:

```
# Floyd-Warshall Algorithm
def floyd_warshall(graph):
    nodes = list(graph.keys())
    dist = {i: {j: float('inf') for j in nodes} for i in nodes}

    for i in nodes:
        dist[i][i] = 0
        for j in graph[i]:
```

```
            dist[i][j] = graph[i][j]

    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 11 Code Implementation Floyd-Warshall's A.

Results:

```
Sparse Graph - Dijkstra's Time: 3.266334533691406e-05
Sparse Graph - Floyd-Warshall's Time: 0.00023126602172851562
Dense Graph - Dijkstra's Time: 6.818771362304688e-05
Dense Graph - Floyd-Warshall's Time: 0.000164031982421875
```

Figure 12 Results of program

Graph:

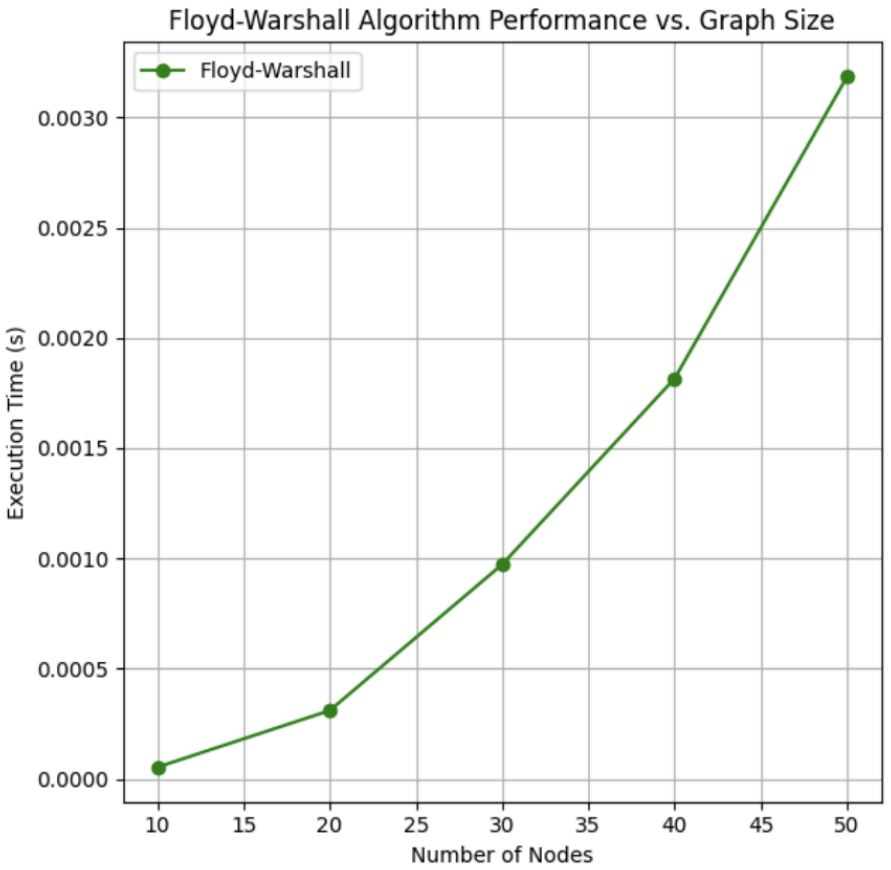

Figure 13 Graphical Representation

In the implementation section, the Floyd-Warshall algorithm is presented. It initialises the distance matrix with all distances set to infinity and populates it with direct edge weights from the given graph. Then, it iterates through all possible intermediate nodes to update the distance matrix with the shortest paths using dynamic programming.

The results section provides the execution times for both Dijkstra's algorithm and Floyd-Warshall algorithm on

sparse and dense graphs. The times are given in seconds.

Lastly, Figure 13 refers to the graphical representation of the obtained data, which could include plots showing how the execution times vary with different graph sizes or densities.

CONCLUSION

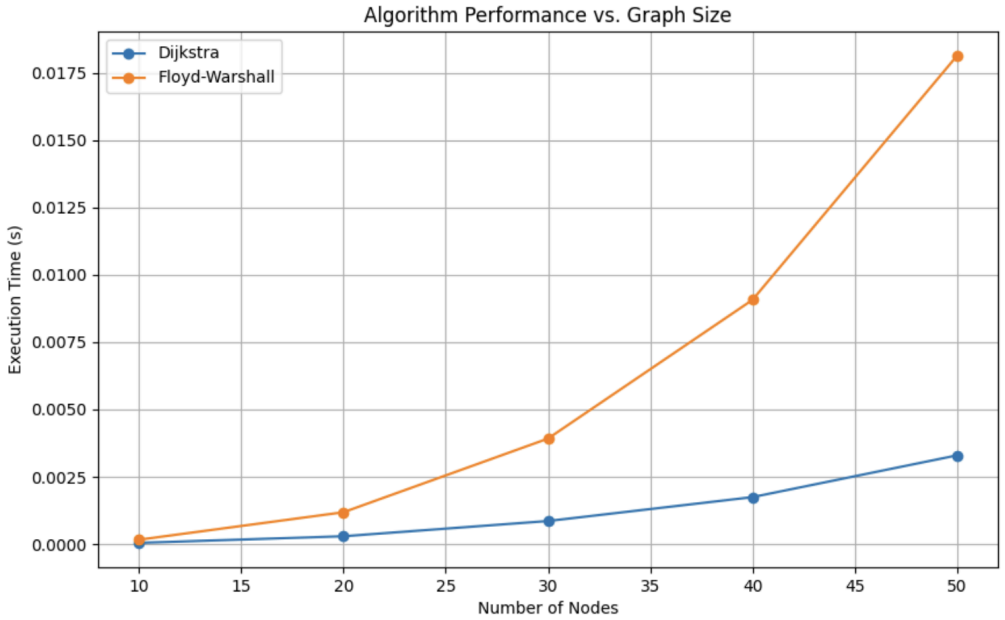

Figure 14 Graphical Representation Comparison

References:

[1] GitHub Repository: Algorithms Analysis Laboratory Works, Retrieved from GitHub - Chiuliana/Algorithms_Analysis_Laboratory_Works