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

Laboratory work 4:
Empirical analysis of algorithms: Dijkstra and Floyd-Warshall (Dynamic Programming)

Elaborated:

st. gr. FAF-221 Lupan Lucian

Verified:

asist. univ. Fiștic Cristofor

Chişinău – 2024

Contents

ALGORITHM ANALYSIS

Study the concept of dynamic programming, implement Dijkstra and Floyd-Warshall algorithms in a programming language and analyze them empirically.

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:

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.

Advantages of Dynamic Programming (DP)

Dynamic programming has a wide range of advantages, including:

- Avoids recomputing the same subproblems multiple times, leading to significant time savings.
- Ensures that the optimal solution is found by considering all possible combinations.
- Breaks down complex problems into smaller, more manageable subproblems.

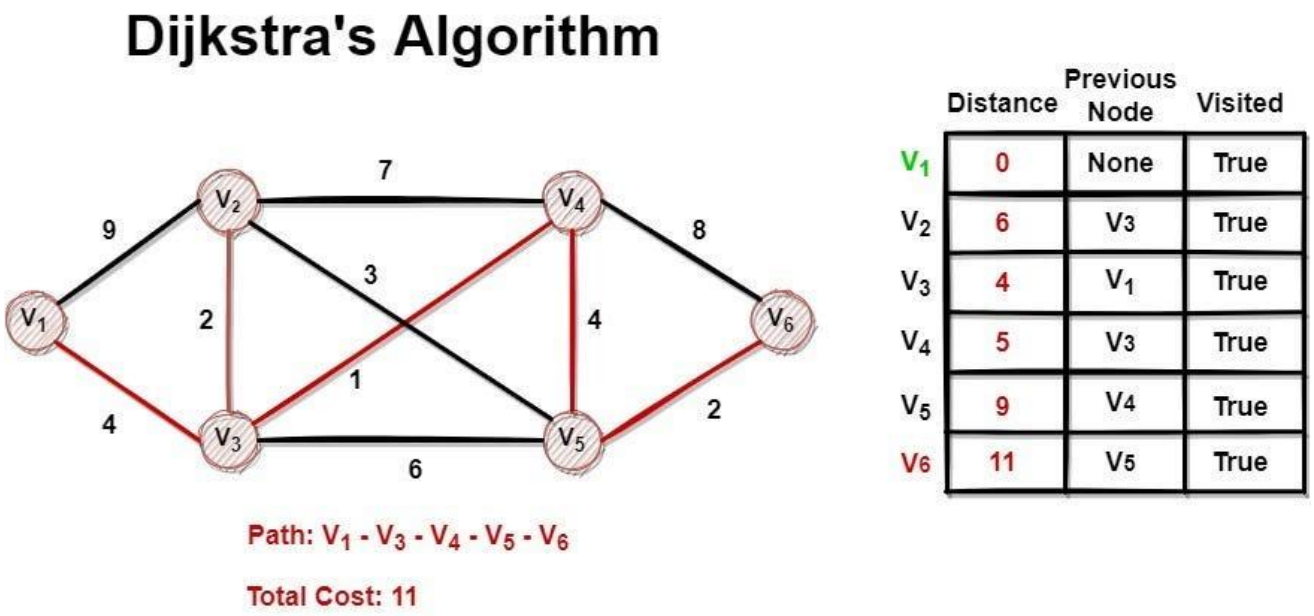
Dynamic Programming algorithms:

1. **Dijkstra Algorithm:**

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, road networks. The algorithm maintains a set of visited vertices and a set of unvisited vertices. It starts at the source vertex and iteratively selects the unvisited vertex with the smallest tentative distance from the source. It then visits the neighbors of this vertex and updates their tentative distances if a shorter path is found. This process continues until the destination vertex is reached, or all reachable vertices have been visited.

Algorithm for Dijkstra’s Algorithm:

1. Mark the source node with a current distance of 0 and the rest with infinity.
2. Set the non-visited node with the smallest current distance as the current node.
3. For each neighbor, N of the current node adds the current distance of the adjacent node with the weight of the edge connecting 0->1. If it is smaller than the current distance of Node, set it as the new current distance of N.
4. Mark the current node 1 as visited.
5. Go to step 2 if there are any nodes that are unvisited.



	Distance	Previous Node	Visited
V_1	0	None	True
V_2	6	V3	True
V_3	4	V_1	True
V_4	5	V3	True
V_5	9	V4	True
V6	11	V5	True

Figure 1. Dijkstra algorithm representation.

2. Floyd-Warshall 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 negative edge weights, making it a versatile tool for solving a wide range of network and connectivity problems.

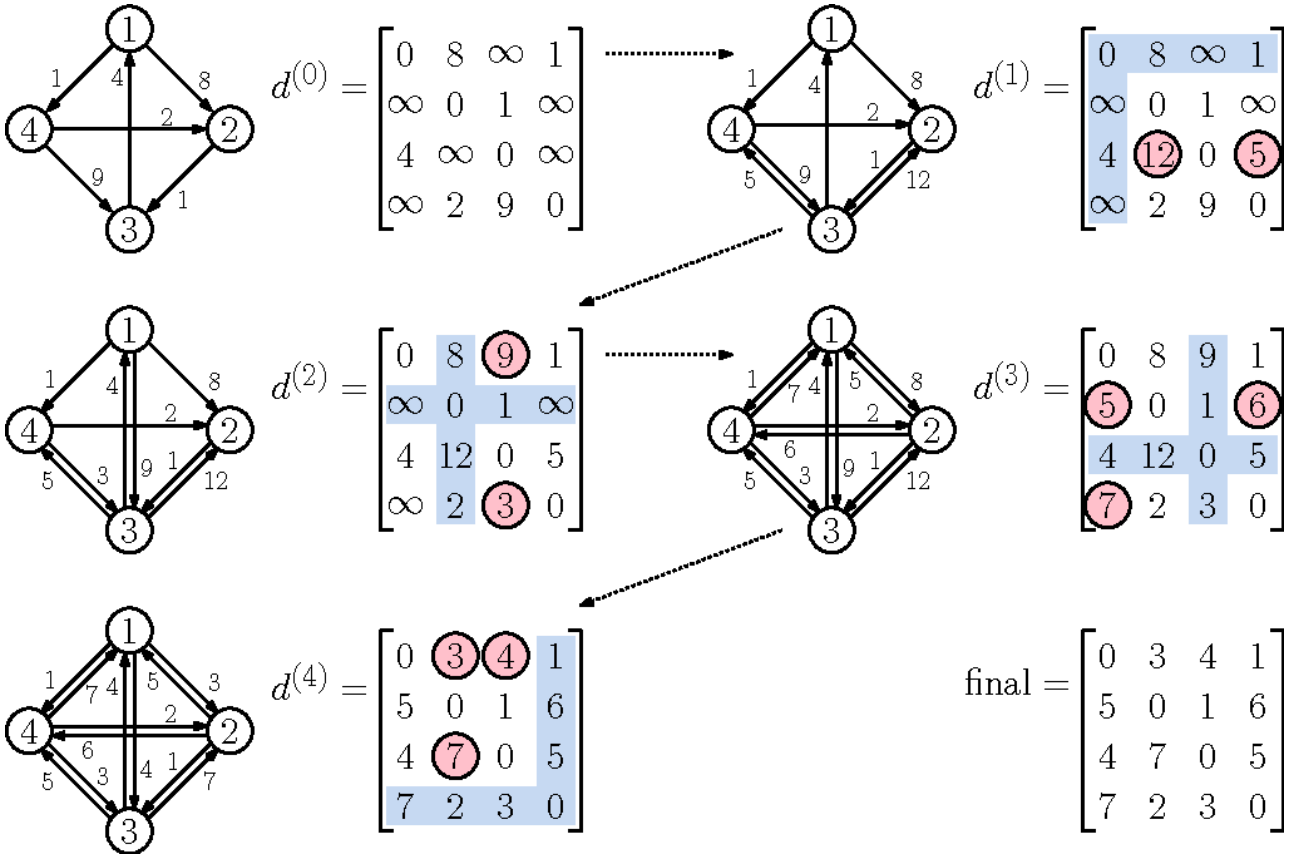


Figure 2. Floyd-Warshall algorithm representation.

Empirical Comparison:

Empirical comparison involves testing sorting algorithms on real-world datasets to observe their performance characteristics. Here's why it is crucial:

1. **Practical Efficiency:**
    - Theoretical analyses provide insights into the worst-case, average-case, and best-case scenarios. However, real-world data often exhibits unique patterns that may affect the actual performance of an algorithm.
2. **Adaptability:**
    - Different sorting algorithms may excel in specific scenarios. Empirical comparison helps identify which algorithm performs best under particular conditions, enabling the selection of the most suitable algorithm for a given task.
3. **Implementation Factors:**
    - The actual implementation of an algorithm can impact its performance. Empirical comparisons help identify subtle implementation details or optimizations that can significantly influence the practical efficiency of an algorithm.
4. **Hardware Dependencies:**
    - The efficiency of sorting algorithms can be influenced by the underlying hardware architecture. Empirical testing on different machines and environments provides a more realistic understanding of their performance.
5. **Scaling Behavior:**
    - The scalability of sorting algorithms is crucial, especially when dealing with large datasets. Empirical studies reveal how algorithms scale as the size of the input data increases.
6. **Benchmarking:**
    - Empirical comparison facilitates benchmarking, which is essential for assessing the relative merits of various algorithms in a specific context. Benchmark results guide the selection of the most efficient sorting algorithm for a particular application.

Introduction:

Dynamic programming is a computer programming technique where an algorithmic problem is first broken down into sub-problems, the results are saved, and then the sub-problems are optimized to find the overall solution — which usually has to do with finding the maximum and minimum range of the algorithmic query.

Richard Bellman was the one who came up with the idea for dynamic programming in the 1950s. It is a method of mathematical optimization as well as a methodology for computer programming. It applies to issues one can break down into either overlapping subproblems or optimum substructures.

This technique solves problems by breaking them into smaller, overlapping subproblems. The results are then stored in a table to be reused so the same problem will not have to be computed again.

For example, when using the dynamic programming technique to figure out all possible results from a set of numbers, the first time the results are calculated, they are saved and put into the equation later instead of being calculated again. So, when dealing with long, complicated equations and processes, it saves time and makes solutions faster by doing less work.

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, the algorithms will take in a randomized graph data set, with an initial node_count of 100, increment of 20 and test_cases of 10. The graph data types have randomly defined edges, with a 2/10 or 8/10 chance of the current vertex adding an edge to a random vertex. We will be considering 2 types of graphs – sparse and dense, with the sparse graph having a 2/10 chance of creating a connection, and the dense one having a 8/10 chance. Both of the graphs will have a random number from 1 to 1000 attributed to the weight.

IMPLEMENTATION

Both of the algorithms are implemented in Python, making use of the random, time, collections and matplotlib.pyplot libraries in order to both ensure the functionality of the algorithms, analyze the algorithms empirically and visually representing the execution times.

Dijkstra:

Algorithm Description:

Dijkstra's algorithm is a popular algorithm for solving many single-source shortest path problems having non-negative edge weight in the graphs i.e., it is to find the shortest distance between two vertices on a graph. It was conceived by Dutch computer scientist Edsger W. Dijkstra in 1956.

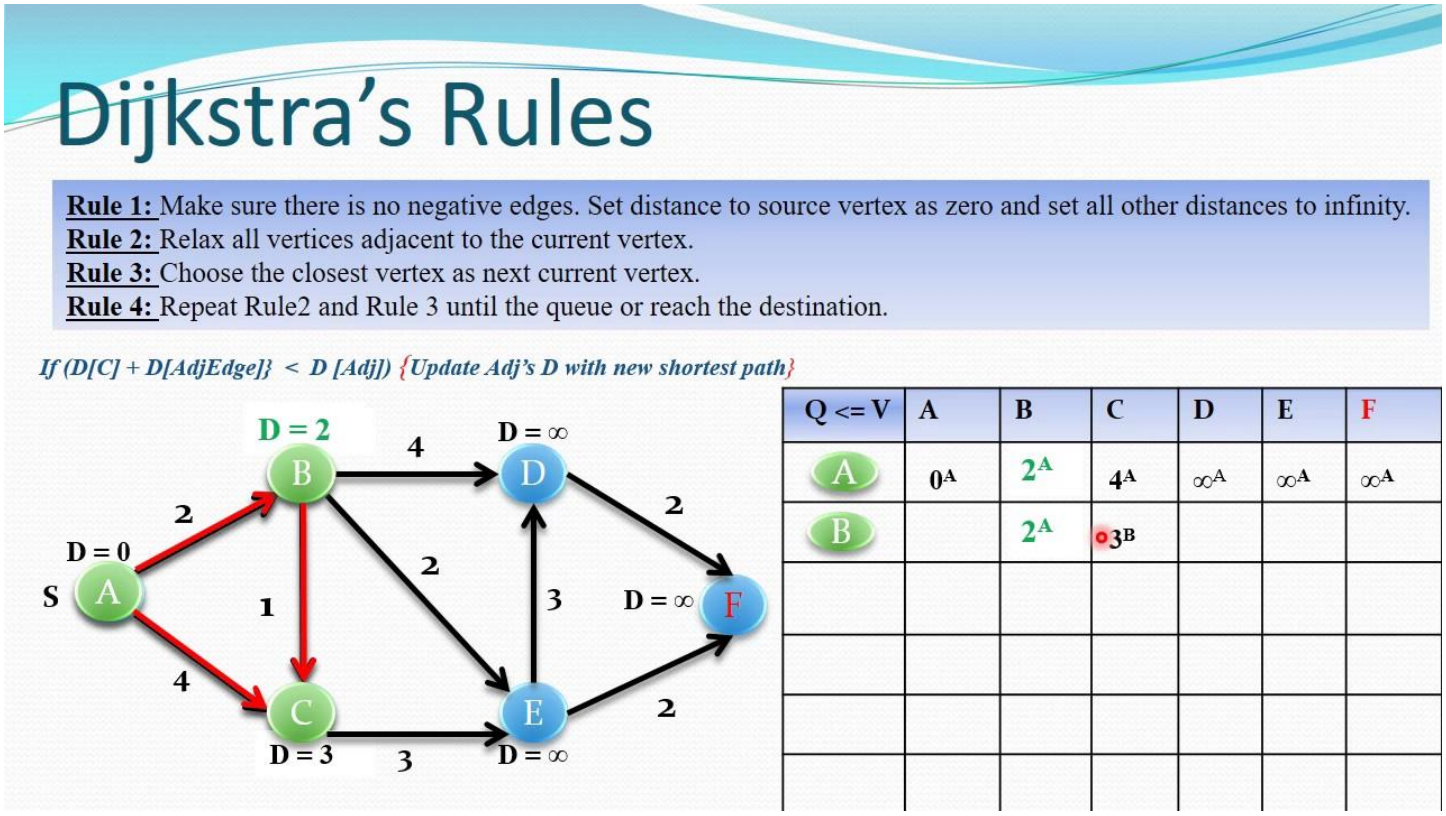


Figure 3. Dijkstra algorithm explanation.

The complexity of Dijkstra's algorithm is as follows:

- **Time Complexity:** O(V^2) in the worst case, where V is the number of vertices. This can be improved to O(V^2) with some optimizations.
- **Auxiliary Space:** O(V), where V is the number of vertices and E is the number of edges in the graph.

Implementation:

```
def dijkstra(self, src):
    dist = [float('inf')] * self.V
    dist[src] = 0
    sptSet = [False] * self.V

    for _ in range(self.V):
        u = self.minDistance(dist, sptSet)
        sptSet[u] = True
        for v in range(self.V):
            if not sptSet[v] and self.graph[u][v] > 0 and dist[u] + self.graph[u][v] < dist[v]:
                dist[v] = dist[u] + self.graph[u][v]
```

Figure 4. Dijkstra code in Python.

Floyd-Warshall:

Algorithm Description:

The Floyd Warshall Algorithm is an all-pair shortest path algorithm unlike Dijkstra and Bellman Ford which are single source shortest path algorithms. This algorithm works for both the directed and undirected weighted graphs. But it does not work for the graphs with negative cycles (where the sum of the edges in a cycle is negative). It follows Dynamic Programming approach to check every possible path going via every possible node in order to calculate the shortest distance between every pair of nodes.

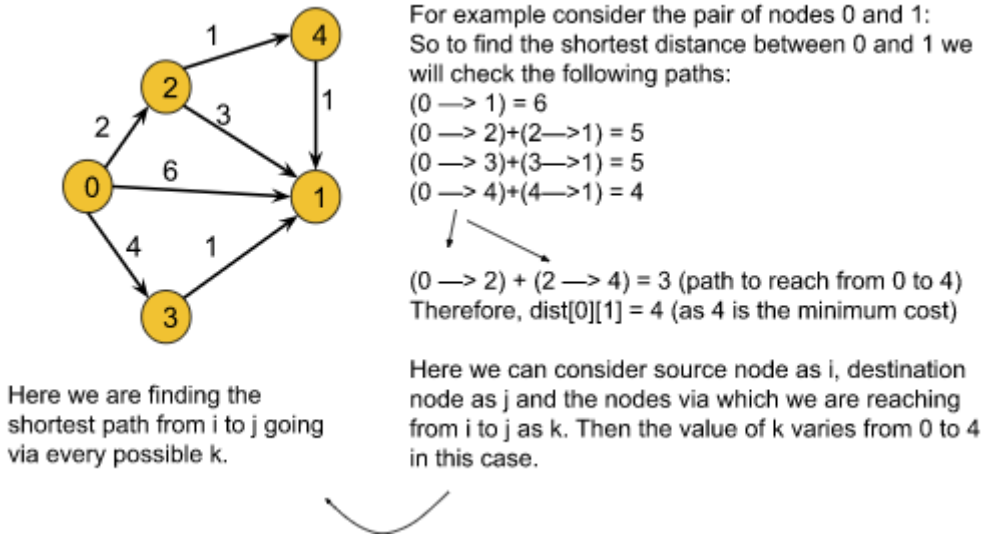


Figure 5. Floyd-Warshall Algorithm Explanation.

The complexity of Floyd-Warshal algorithm is as follows:

- **Time Complexity:** $O(V^3)$, where V is the number of vertices in the graph and we run three nested loops each of size V
- **Auxiliary Space:** $O(V^2)$, to create a 2-D matrix in order to store the shortest distance for each pair of nodes.

Implementation:

```
def floydWarshall(self):
    dist = [[float('inf') for _ in range(self.V)] for _ in range(self.V)]
    for i in range(self.V):
        for j in range(self.V):
            if i == j:
                dist[i][j] = 0
            else:
                if self.graph[i][j] != 0:
                    dist[i][j] = self.graph[i][j]

    for k in range(self.V):
        for i in range(self.V):
            for j in range(self.V):
                if dist[i][k] + dist[k][j] < dist[i][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]
```

Figure 6. Floyd-Warshall Code in Python.

Algorithm comparison results:

After performing the results on randomized datasets (according to the input data), saving the execution times, and plotting them on a graph, the results are as follows:

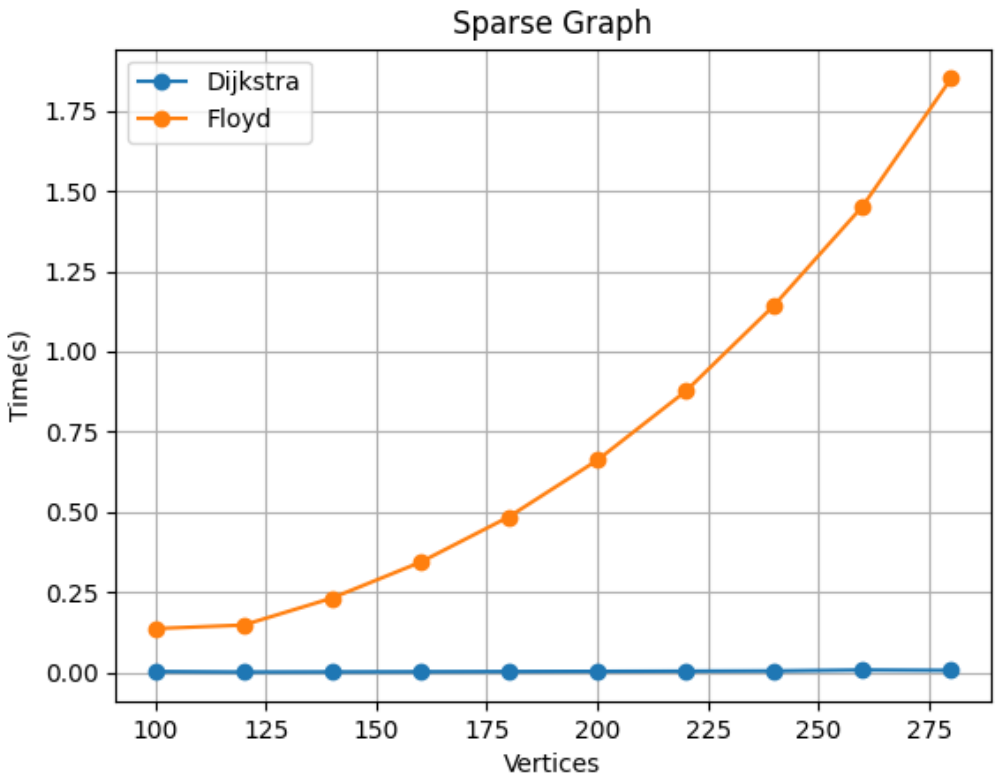


Figure 7. Algorithm comparison, first execution, sparse graph.

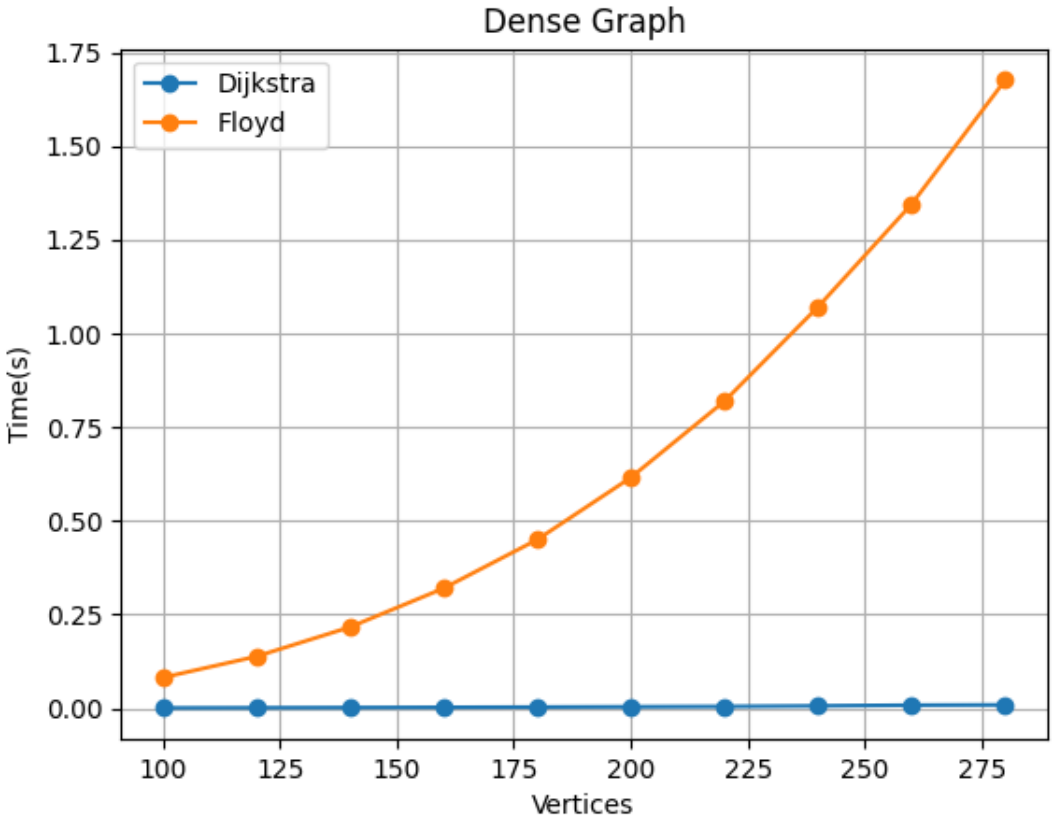


Figure 8. Algorithm comparison, first execution, dense graph.

In figure 7, we can observe a massive difference in time between Dijkstra and Floyd-Warshall algorithms. Due to having a different time complexity O(V^2) for Dijkstra vs O(V^3) for Floyd-Warshall, this result was more or less expected, and after testing it in practice, we can safely conclude that the Dijkstra algorithm is much faster than Floyd-Warshall algorithm

In figure 8, for the dense graph, we can notice a slight increase in time execution for the Dijkstra algorithm, while the Floyd-Warshall algorithm remains, more or less, the same. This is due to the fact that the Floyd-Warshall algorithm works by computing all the shortest paths, for all the vertices, making it a constant

O(V^3) algorithm, while the Dijkstra algorithm only computes it for one of them, given an initial vertex.

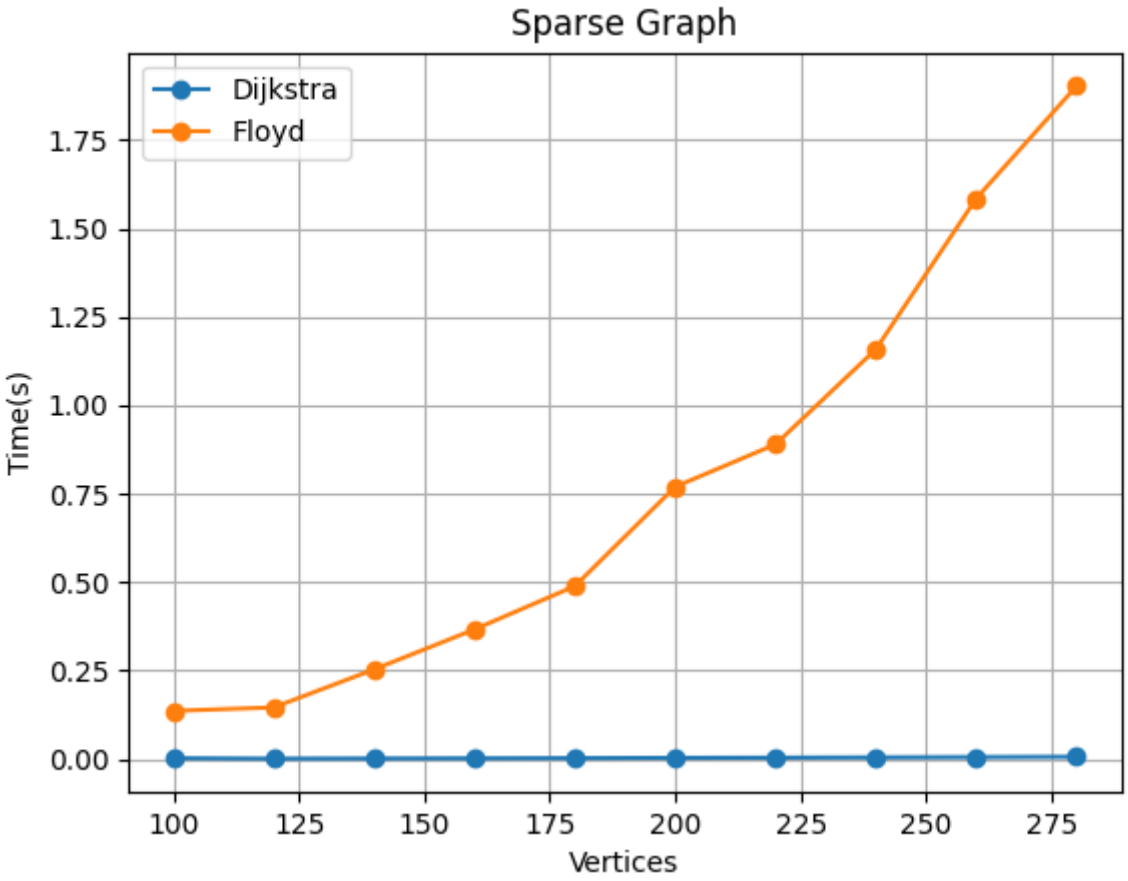


Figure 9. Algorithm comparison, second execution, sparse graph.

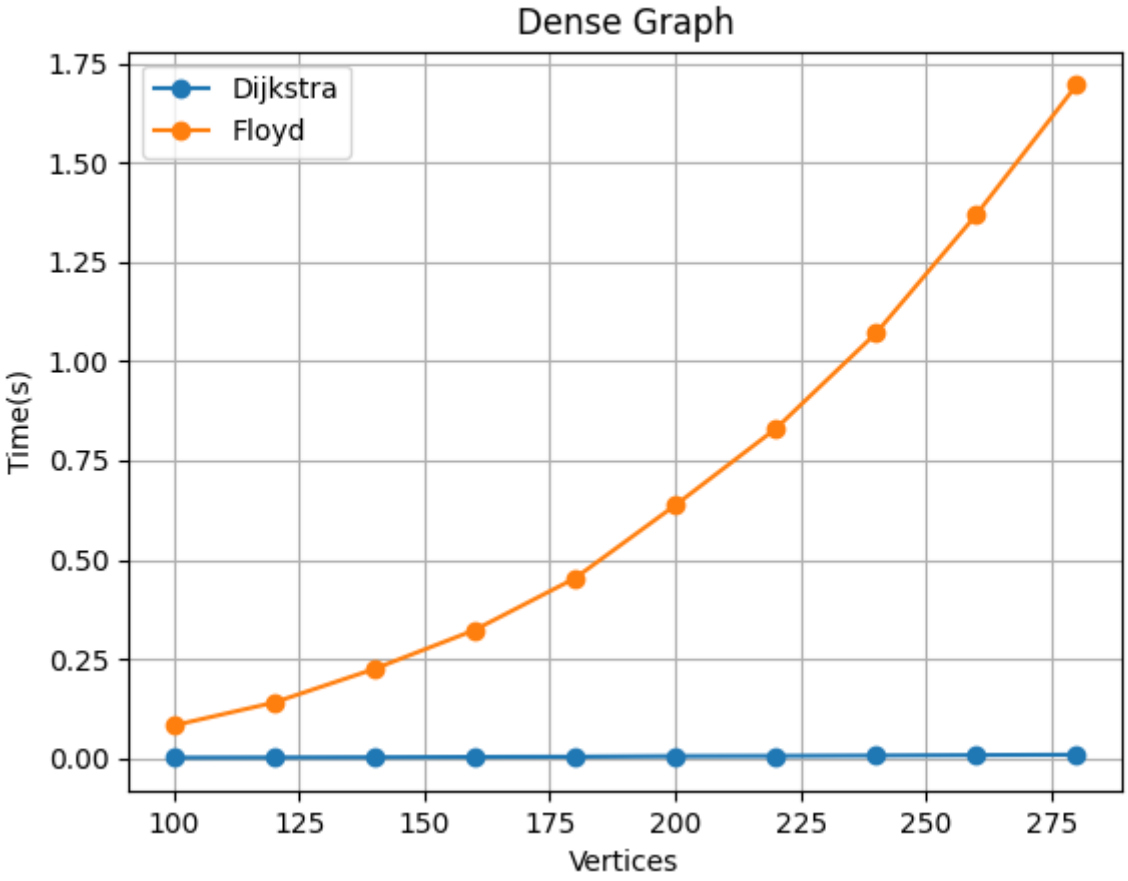


Figure 10. Algorithm comparison, second execution, dense graph.

In figures 9 and 10, we can observe the same behaviour as in figures 7 and 8, with little to no difference in terms of computational times. From these observations, we can conclude that unless there's a huge variance in terms of how sparse or dense the graph is, the algorithms will perform about the same when it comes to any graphs.

CONCLUSION

After finishing this laboratory work, I learnt about 2 of the most useful dynamic programming algorithms when it comes to finding the shortest path/s in a graph – the Dijkstra and Floyd-Warshall algorithms. Although this laboratory work was rather similar to the previous one, I've learnt/revised some interesting things regarding graphs with weighted edges.

The algorithms in this laboratory work ran at significantly different time complexities – O(V^2) for Dijkstra vs O(V^3) for Floyd-Warshall, and as a result, comparing them empirically was somewhat futile. From the comparison done in Figures 7-10, it became quite obvious that they operated at different time complexities. Further research indicated that Djikstra algorithm operated much faster due to the fact that it works by only computing the SSSP – single-source shortest path route, whereas the Floyd-Warshall algorithm computed the all-pair shortest path. Additionally, I've found out that the Floyd-Warshall algorithm performed slightly better on dense graphs, due to having to compute all of the shortest paths in the graphs.

Due to the O(V^3) time complexity for the Floyd-Warshall algorithm, I was unable to efficiently increase the number of nodes in the graphs without experiencing a massive increase in execution times, and

I've resorted to only working on smaller graphs. Overall, I can conclude that both of the dynamic programming algorithms for finding the shortest path have their use cases, however, I'd favor the Dijkstra algorithm when it comes to usage, as I'm far more likely to only need one of the shortest paths instead of all of them in real-world situations.

GitHub repository link - https://github.com/Lucian-Lu/AA-Labs