Ministry of Education of Republic of Moldova

Technical University of Moldova

Faculty of Computers, Informatics and Microelectronics

Software Engineering Department

Algorithm Analysis

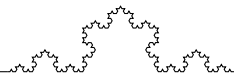
Laboratory work #4

Dynamic Programming

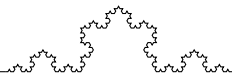
Author: Timur Cravtov
std. gr. FAF-231

Verified by: Cristofor Fistic
asist. univ

Chișinău
2025



Contents



1 Algorithm Analysis

1.1 Objective

Study and analyze different algorithms for finding the shortest path in a graph. The algorithms to be analyzed are:

1. Floyd-Warshall algorithm
2. Dijkstra's algorithm

1.2 Task

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.

1.3 Introduction

Dynamic programming is a method for solving complex problems by breaking them down into simpler subproblems. It is applicable when the subproblems are overlapping, meaning that the same subproblems are solved multiple times. Dynamic programming is often used in optimization problems, where the goal is to find the best solution among many possible solutions.

One application of dynamic programming is in finding the shortest path in a graph. The shortest path problem is a classic problem in computer science and has many applications, such as in transportation networks, communication networks, and social networks. The shortest path problem can be solved using various algorithms, including Dijkstra's algorithm and the Floyd-Warshall algorithm. Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a source vertex to all other vertices in a weighted graph. The Floyd-Warshall algorithm is a dynamic programming algorithm that finds the shortest paths between all pairs of vertices in a weighted graph.

2 Implementation

2.1 Dijkstra's Algorithm

Algorithm description

Dijkstra's algorithm computes the shortest path from a single source vertex to all other vertices in a weighted graph with non-negative edge weights. It employs a greedy approach but can be viewed through a dynamic programming lens due to its iterative construction of optimal subpaths. The algorithm uses a priority queue to select the vertex with the smallest tentative distance, updates distances to its neighbors, and continues until all vertices are processed.

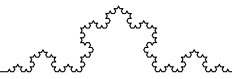
```python
import heapq
import networkx as nx

def dijkstra(graph: nx.Graph, start_node: str) -> dict:
    distances = {node: float('inf') for node in graph.nodes}
    distances[start_node] = 0
    pq = [(0, start_node)]

    while pq:
        current_distance, current_node = heapq.heappop(pq)
        if current_distance > distances[current_node]:
            continue

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

    return distances
```

Dijkstra's Algorithm

2.2 Floyd-Warshall Algorithm

Algorithm description

The Floyd-Warshall algorithm is a dynamic programming algorithm that computes the shortest paths between all pairs of vertices in a weighted graph. It maintains a



distance matrix, iteratively refining it by considering each vertex as an intermediate point. The algorithm supports both positive and negative edge weights (provided there are no negative cycles) and is particularly effective for dense graphs.

```python
import networkx as nx

def floyd_warshall(graph: nx.Graph) -> dict:
    nodes = list(graph.nodes)
    dist = {u: {v: float('inf') for v in nodes} for u in nodes}

    for node in nodes:
        dist[node][node] = 0

    for u, v, data in graph.edges(data=True):
        weight = data['weight']
        dist[u][v] = weight
        dist[v][u] = weight  # if undirected

    for k in nodes:
        for i in nodes:
            for j in nodes:
                if dist[i][j] > dist[i][k] + dist[k][j]:
                    dist[i][j] = dist[i][k] + dist[k][j]

    return dist
```

Floyd-Warshall Algorithm

2.3 Comparative Analysis

To compare Dijkstra's and Floyd-Warshall algorithms with themselves across sparse and dense graphs, we used the following metrics:

- *Time complexity*: The theoretical time complexity as a function of vertices (V) and edges (E), validated by empirical runtime measurements.
- *Runtime performance*: The runtime of each algorithm on sparse ($E \approx 2V$) and dense ($E \approx V^2/2$) graphs, analyzed separately to highlight their behavior under different graph densities conditions.

Since Dijkstra's algorithm is designed for single-source shortest paths, we focused on its performance from a single source vertex. In contrast, Floyd-Warshall computes

all-pairs shortest paths, making it suitable for dense graphs where the number of edges is significantly larger than the number of vertices.

To show that it's unusefull to compare the algorithms with each other, figure 1 shows that the Dijkstra's algorithm is much faster than the Floyd-Warshall algorithm. The Dijkstra's algorithm is a greedy algorithm that finds the shortest path from a source vertex to all other vertices in a weighted graph. The Floyd-Warshall algorithm is a dynamic programming algorithm that finds the shortest paths between all pairs of vertices in a weighted graph. The time complexity of Dijkstra's algorithm is $O((V + E) \log V)$, while the time complexity of Floyd-Warshall is $O(V^3)$.

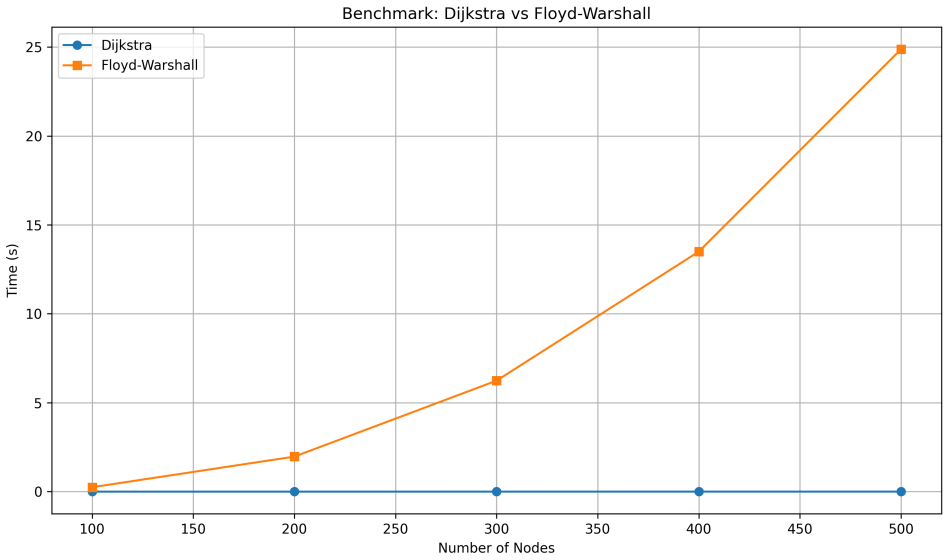


Figure 1: Comparision of Dijkstra's and Floyd-Warshall algorithms

The empirical analysis was performed on graphs with vertex counts of 10, 50, 100, and 200. Sparse graphs were generated with approximately $E = 2V$ edges, while dense graphs had $E \approx V^2/2$ edges. Both algorithms were implemented in Python, and their runtimes were measured for each graph configuration. For Dijkstra's algorithm, we ran it from a single source vertex to focus on single-source shortest paths. For Floyd-Warshall, we computed all-pairs shortest paths, as it is designed for this purpose.

The theoretical time complexity of Dijkstra's algorithm, using a priority queue, is $O((V + E) \log V)$ for a single source. Floyd-Warshall has a time complexity of $O(V^3)$, which is independent of the number of edges but grows cubically with vertices. These complexities guide the expected performance differences in sparse and dense graphs.

For Dijkstra's algorithm, the empirical analysis compared its performance on sparse vs. dense graphs. In sparse graphs ($E \approx 2V$), the algorithm benefits from fewer edges, resulting in faster runtimes due to the $O((V + E) \log V)$ complexity. In dense graphs ($E \approx V^2/2$), the increased edge count leads to higher runtimes, as the priority queue operations become more costly. The performance difference is evident as the number of

vertices increases, with sparse graphs maintaining lower runtimes.

For Floyd-Warshall, the analysis showed consistent performance across sparse and dense graphs, as its $O(V^3)$ complexity depends solely on the number of vertices. However, the cubic growth makes it less scalable for large graphs, with runtimes increasing significantly as vertex count grows, regardless of graph density.

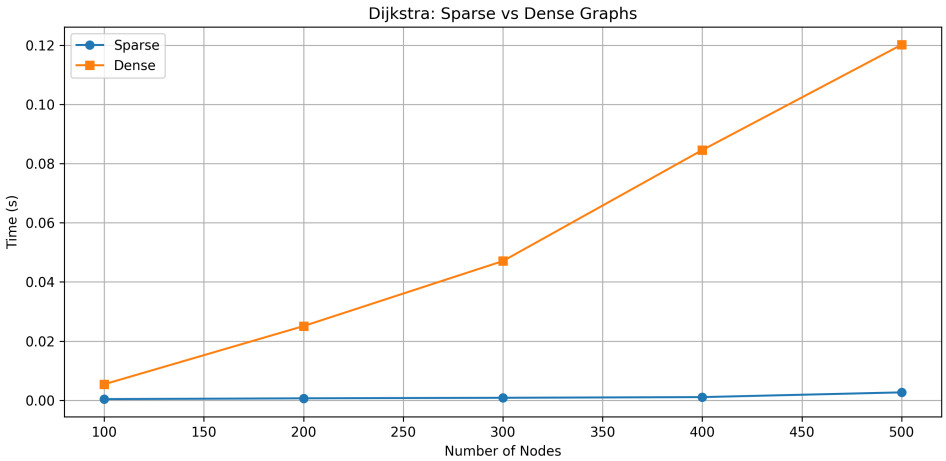


Figure 2: Runtime comparison of Dijkstra's algorithm on sparse and dense graphs

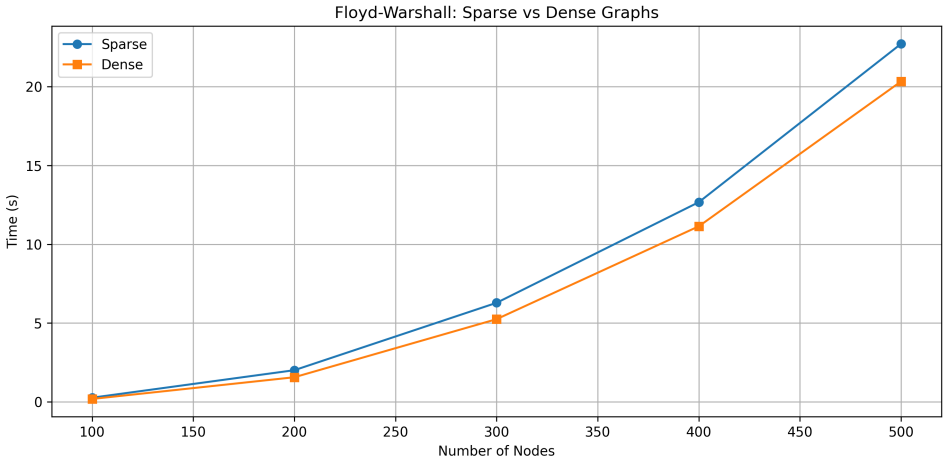
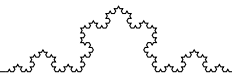


Figure 3: Runtime comparison of Floyd-Warshall algorithm on sparse and dense graphs

Figures 2 and 3 illustrate the runtime performance of Dijkstra's and Floyd-Warshall algorithms, respectively, on sparse and dense graphs with increasing vertex counts. For Dijkstra's algorithm, sparse graphs exhibit significantly lower runtimes compared to dense graphs, as the algorithm processes fewer edges, reducing the number of priority queue operations. The runtime gap widens with larger graphs (e.g., $V = 200$), where dense graphs approach $O(V^2 \log V)$. For Floyd-Warshall, the runtimes are nearly identical for sparse and dense graphs, as expected from its $O(V^3)$ complexity, but the cubic growth results in steep runtime increases for larger vertex counts (e.g., $V \geq 100$). Comparing the



algorithms with themselves highlights Dijkstra's sensitivity to edge density and Floyd-Warshall's consistency across graph types.

3 Conclusions

During this laboratory work, we studied and analyzed two algorithms for finding shortest paths in a graph: Dijkstra's algorithm and the Floyd-Warshall algorithm. We implemented both algorithms in Python and verified their correctness using a randomly generated weighted graph. The empirical analysis compared each algorithm with itself on sparse ($E \approx 2V$) and dense ($E \approx V^2/2$) graphs, focusing on time complexity and runtime performance as the number of vertices increased.

We found that Dijkstra's algorithm performs significantly better on sparse graphs than on dense graphs due to its $O((V + E) \log V)$ complexity, where fewer edges reduce the number of priority queue operations. In dense graphs, Dijkstra's runtime increases notably, approaching $O(V^2 \log V)$. Floyd-Warshall, with its $O(V^3)$ complexity, shows consistent performance across sparse and dense graphs, as its runtime depends only on the number of vertices. However, its cubic complexity leads to poor scalability for large graphs. The empirical results, visualized in Figures 2 and 3, confirm these trends, with Dijkstra's showing a clear performance advantage in sparse graphs and Floyd-Warshall maintaining uniform runtimes across graph densities.

This laboratory work provided insights into the dynamic programming approach of Floyd-Warshall and the greedy/dynamic programming hybrid of Dijkstra's algorithm. Comparing each algorithm with itself across sparse and dense graphs highlighted their strengths and limitations: Dijkstra's excels in sparse graphs for single-source shortest paths, while Floyd-Warshall is better suited for all-pairs shortest paths in dense graphs. Future work could explore optimizations, such as using advanced data structures for Dijkstra's (e.g., Fibonacci heaps) or parallelizing Floyd-Warshall to improve scalability for large graphs.

References

[1] https://github.com/TimurCravtov/AA-labs