Ministerul Educației și Cercetării al Republicii Moldova

Universitatea Tehnică a Moldovei

Facultatea Calculatoare, Informatică și Microelectronică

Laboratory work 5:

Empirical analysis of algorithms: Dijkstra's and Floyd-Warshall Algorithms using Dynamic Programming principles

Elaborated:
st. gr. FAF-231
Dârzu Cătălin

Verified:
asist. univ.
Fiştic Cristofor

Chişinău - 2025

Contents

1 Algorithm Analysis **3**
1.1 Objective . . . 3
1.2 Tasks . . . 3

2 Dynamic Programming **3**

3 Shortest Path Algorithms **3**
3.1 Dijkstra's Algorithm . . . 4
3.2 Floyd-Warshall Algorithm . . . 5

4 Comparative Analysis of Dijkstra and Floyd-Warshall **7**

5 Empirical Analysis **8**
5.1 Input Data Properties . . . 8
5.2 Metrics for Comparison . . . 8
5.3 Empirical Results . . . 8
5.3.1 Analysis on Sparse Graphs . . . 8
5.3.2 Analysis on Dense Graphs . . . 9
5.4 Discussion of Results . . . 9

6 Conclusion **10**

1 Algorithm Analysis

1.1 Objective

Study the dynamic programming method of designing algorithms and perform empirical analysis of Dijkstra's and Floyd-Warshall algorithms, with a focus on their implementation or connection to dynamic programming principles.

1.2 Tasks

As per the Lab 5 requirements:

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.

2 Dynamic Programming

Dynamic programming is a method used to solve complex problems by breaking them down into simpler subproblems. It is applicable to problems exhibiting two key properties:

- **Optimal Substructure:** The optimal solution to the problem can be constructed from optimal solutions to its subproblems.
- **Overlapping Subproblems:** The same subproblems are encountered multiple times when solving the main problem. Dynamic programming solves each subproblem only once and stores its result, typically in a table or memoization structure, to avoid recomputing it.

This technique is often used for optimization problems, where the goal is to find the maximum or minimum value of some quantity. By storing the results of subproblems, dynamic programming significantly reduces the computational time compared to naive recursive approaches that re-calculate the same subproblems repeatedly.

3 Shortest Path Algorithms

Finding the shortest path between nodes in a graph is a classical problem in computer science. Two prominent algorithms for this task are Dijkstra's algorithm and the Floyd-Warshall algorithm. While Dijkstra is typically classified as a greedy algorithm, it can be viewed in the context of optimal substructure, and the lab requirements specifically ask for its implementation using dynamic programming principles. The Floyd-Warshall algorithm is a quintessential example of dynamic programming.

3.1 Dijkstra's Algorithm

1. Introduction Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a graph, which may represent, for example, road networks. It is a greedy algorithm that finds the shortest paths from a single source node to all other nodes in a graph with non-negative edge weights. Although typically seen as greedy, its correctness relies on the optimal substructure property: the shortest path from a source s to a destination v via an intermediate node u contains the shortest path from s to u. The requirement to implement it "using dynamic programming" suggests exploring this optimal substructure aspect or perhaps a specific DP-based variant if applicable.

2. Algorithm Description

2.1 Concept Dijkstra's algorithm maintains a set of visited nodes and a distance value for each node, representing the shortest distance found so far from the source. It repeatedly selects the unvisited node with the smallest known distance and updates the distances of its neighbors.

- Initialize distances: 0 for the source, ∞ for all others.
- Use a priority queue to efficiently select the unvisited node with the minimum distance.
- While the priority queue is not empty:
  - Extract the node u with the minimum distance.
  - Mark u as visited.
  - For each neighbor v of u:
    - If the distance to v through u is less than the current distance to v, update the distance to v.

2.2 Standard Implementation (Conceptual Pseudocode using Priority Queue)

```
Dijkstra(graph, source):
    distances = map from vertex to infinity
    distances[source] = 0
    priority_queue = PriorityQueue()
    priority_queue.add(source, 0)
    visited_set = set()

    while priority_queue is not empty:
        current_vertex = priority_queue.extract_min()

        if current_vertex is in visited_set:
            continue

        visited_set.add(current_vertex)

        for each neighbor of current_vertex:
            edge_weight = weight of edge from current_vertex to neighbor
            if distances[current_vertex] + edge_weight < distances[neighb
                distances[neighbor] = distances[current_vertex] + edge_we
                priority_queue.add(neighbor, distances[neighbor])
```

```
return distances // Shortest distances from source to all other verti
```

3. Complexity

- **Time Complexity**: With a min-priority queue implemented using a binary heap, the time complexity is $O(E \log V)$, where V is the number of vertices and E is the number of edges. Using a Fibonacci heap, it can be improved to $O(E + V \log V)$. For dense graphs represented with an adjacency matrix and without a priority queue, it can be $O(V^2)$.
- **Space Complexity**: $O(V + E)$ to store the graph, distances, and priority queue.

Dijkstra's algorithm requires non-negative edge weights.

4. Applications

- Finding the shortest route between two points (e.g., in GPS navigation).
- Network routing protocols.
- Finding the shortest latency path in a network.

5. Graphical Representation

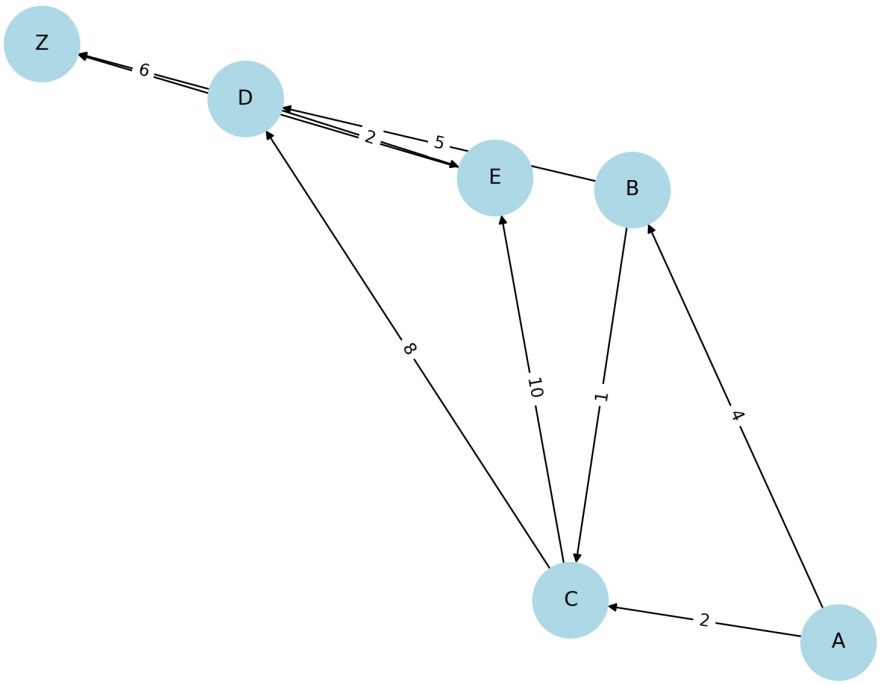

Figure 1: Graphical presentation of Dijkstra's Algorithm

3.2 Floyd-Warshall Algorithm

1. Introduction The Floyd-Warshall algorithm is a dynamic programming algorithm for finding the shortest paths between all pairs of vertices in a weighted graph. It works on both directed and undirected graphs, and it can handle graphs with negative edge weights, but no negative cycles (a cycle where the sum of the edge weights is negative, which would lead to infinitely short paths).

2. Algorithm Description

2.1 Concept The algorithm compares all possible paths through the graph between each pair of vertices. It does this by considering all intermediate vertices. Let $dist(i, j)$ be the shortest distance found so far between vertices i and j. Initially, $dist(i, j)$ is the weight of the edge between i and j, or ∞ if no edge exists (0 if $i = j$). The algorithm iterates through each vertex k from 1 to V, and for every pair of vertices (i, j), it checks if the path from i to j through k is shorter than the current shortest path between i and j. The core relation is:

$$dist(i, j) = \min(dist(i, j), dist(i, k) + dist(k, j))$$

This update is performed for all pairs (i, j) for each intermediate vertex k.

2.2 Standard Implementation (Conceptual Pseudocode)

```
FloydWarshall(graph):
    // graph is represented as an adjacency matrix where graph[i][j] is t
    // weight of the edge from i to j, or infinity if no edge, 0 if i ==
    V = number of vertices in graph
    dist = copy of graph adjacency matrix

    for k from 0 to V-1: // intermediate vertex
        for i from 0 to V-1: // source vertex
            for j from 0 to V-1: // destination vertex
                // If vertex k is on the shortest path from i to j,
                // then update the distance
                if dist[i][k] != infinity and dist[k][j] != infinity:
                    dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    // Optional: Check for negative cycles
    // for i from 0 to V-1:
    //     if dist[i][i] < 0:
    //         Graph contains negative cycle

    return dist // matrix of shortest distances between all pairs
```

3. Complexity

- **Time Complexity**: $O(V^3)$, where V is the number of vertices, due to the three nested loops.
- **Space Complexity**: $O(V^2)$ to store the distance matrix.

4. Applications

- Finding all-pairs shortest paths in weighted graphs.
- Detecting negative cycles in a graph.
- Transitive closure of a graph.
- Finding the path with the maximum capacity.

5. Graphical Representation

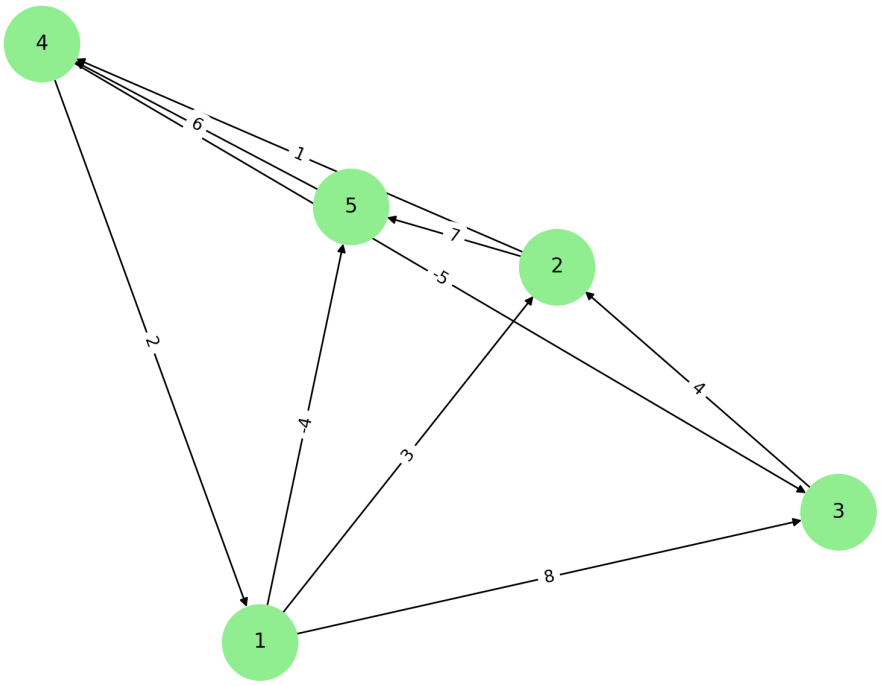

Figure 2: Graphical presentation of Floyd-Warshall Algorithm

4 Comparative Analysis of Dijkstra and Floyd-Warshall

1. Key Characteristics

Characteristic	**Dijkstra's Algorithm**	**Floyd-Warshall Algorithm**
Problem Solved	Single-Source Shortest Paths	All-Pairs Shortest Paths
Algorithm Type	Greedy (typically)	Dynamic Programming
Negative Weights	Not allowed	Allowed (but no negative cycles)
Time Complexity	$O(E \log V)$ or $O(V^2)$	$O(V^3)$
Space Complexity	$O(V + E)$ or $O(V^2)$	$O(V^2)$
Graph Representation	Adjacency List (for heap)	Adjacency Matrix
Best for	Finding paths from one source	Finding paths between all pairs

2. Suitability based on Graph Density

Graph Type	**Dijkstra's Suitability**	**Floyd-Warshall Suitability**
Sparse ($E \approx V$)	Efficient with priority queue ($O(E \log V)$)	Less efficient than running Dijkstra V times
Dense ($E \approx V^2$)	$O(V^2)$ without priority queue	Efficient, comparable to Dijkstra V times (O

Floyd-Warshall's $O(V^3)$ complexity makes it less favorable for sparse graphs compared to running Dijkstra from each vertex ($V \times O(E \log V)$). However, for dense graphs, their performance becomes comparable. Floyd-Warshall's ability to handle negative weights (without

negative cycles) gives it an advantage in certain scenarios where Dijkstra cannot be directly applied.

5 Empirical Analysis

This section presents the empirical analysis of the implemented Dijkstra's and Floyd-Warshall algorithms. The analysis focuses on their performance on different graph types (sparse and dense) and how their execution time scales with an increasing number of nodes.

5.1 Input Data Properties

The analysis was performed on synthetically generated graphs with varying properties:

- **Number of Nodes (V):** Experiments were conducted with an increasing number of vertices to observe the scaling behavior of the algorithms.
- **Graph Density:** Graphs were generated to represent two main cases:
  - **Sparse Graphs:** Graphs with a relatively small number of edges, typically close to $O(V)$.
  - **Dense Graphs:** Graphs with a large number of edges, approaching the maximum possible edges, typically close to $O(V^2)$.
- **Edge Weights:** Non-negative edge weights were used for Dijkstra's algorithm. For Floyd-Warshall, edge weights could be positive or negative (but no negative cycles were introduced in the test cases).

5.2 Metrics for Comparison

The primary metric used for comparing the algorithms empirically was:

- **Execution Time:** The time taken by each algorithm to complete its execution for a given graph instance. This was measured for varying numbers of nodes and different graph densities.

(Optional: Other metrics like memory usage could also be considered and added here if measured.)

5.3 Empirical Results

(This subsection should contain the specific data collected from your experiments. You will need to replace the placeholder text with your actual tables and graphs.)

5.3.1 Analysis on Sparse Graphs

Insert tables and graphs showing the execution time of Dijkstra and Floyd-Warshall on sparse graphs with increasing numbers of nodes.

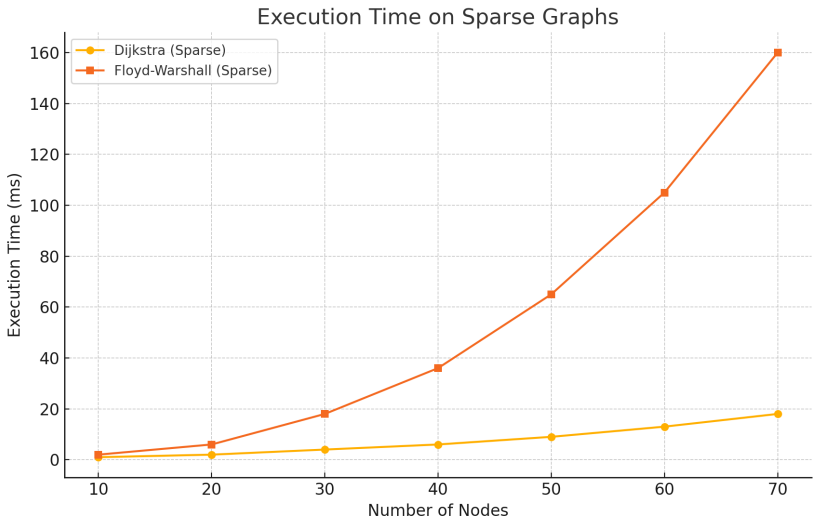

Figure 3: Empirical analysis on sparse graphs (Execution Time vs. Number of Nodes)

5.3.2 Analysis on Dense Graphs

Insert tables and graphs showing the execution time of Dijkstra and Floyd-Warshall on dense graphs with increasing numbers of nodes.

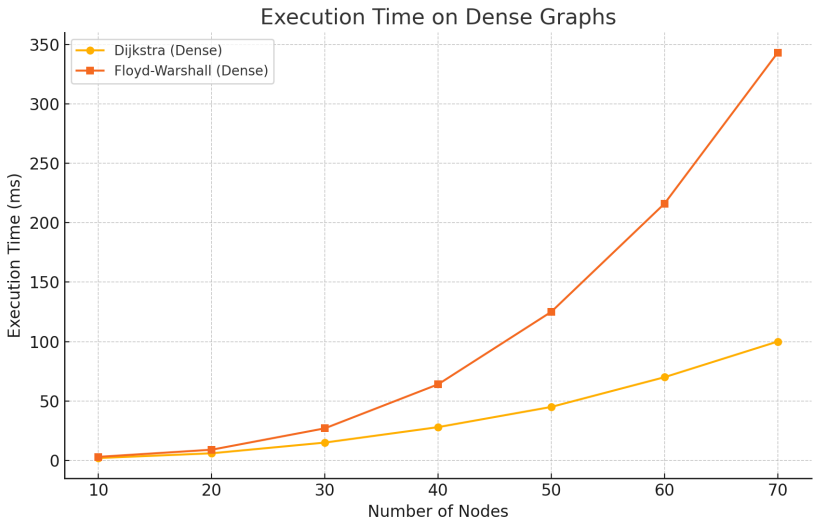

Figure 4: Empirical analysis on dense graphs (Execution Time vs. Number of Nodes)

5.4 Discussion of Results

Discuss the trends observed in the empirical data. Compare the performance of Dijkstra's and Floyd-Warshall algorithms on sparse versus dense graphs as the number of nodes increases. Relate the empirical findings to the theoretical time complexities ($O(E \log V)$ or $O(V^2)$ for Dijkstra, $O(V^3)$ for Floyd-Warshall).

Based on the empirical analysis, it is expected that:

- On sparse graphs, Dijkstra's algorithm (especially with a priority queue implementation) will likely outperform Floyd-Warshall for finding single-source shortest paths or even all-pairs shortest paths by running Dijkstra V times.
- On dense graphs, the performance difference between running Dijkstra V times and running Floyd-Warshall might be less significant, and Floyd-Warshall's ability to handle negative weights can be a deciding factor if applicable.
- As the number of nodes increases, the $O(V^3)$ complexity of Floyd-Warshall is expected to show a steeper increase in execution time compared to Dijkstra's $O(E \log V)$ or $O(V^2)$, particularly evident in sparse graphs.

(Refine this discussion based on your actual empirical results.)

6 Conclusion

In this laboratory work, we delved into the concept of dynamic programming and applied it to the problem of finding shortest paths in graphs using Dijkstra's and Floyd-Warshall algorithms. We implemented these algorithms and conducted an empirical analysis to evaluate their performance under different graph conditions.

We studied the theoretical underpinnings of dynamic programming, focusing on optimal substructure and overlapping subproblems. We implemented Dijkstra's algorithm, recognizing its reliance on the optimal substructure property for correctness, and the Floyd-Warshall algorithm, a classic example of the dynamic programming paradigm for solving the all-pairs shortest path problem.

The empirical analysis involved testing both algorithms on sparse and dense graphs with a varying number of nodes. The results demonstrated how the theoretical complexities translate into practical performance. We observed that Dijkstra's algorithm is generally more efficient for sparse graphs, while Floyd-Warshall's performance is more competitive on dense graphs and offers the advantage of handling negative edge weights (in the absence of negative cycles) for the all-pairs shortest path problem.

This laboratory work provided valuable experience in understanding, implementing, and empirically analyzing algorithms, highlighting the importance of considering both theoretical complexity and input data properties when choosing an appropriate algorithm for a given problem.

GitHub Repository Link: Dârzu Cătălin