Ministry of Education and Research 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:
Mihaela Catan
std. gr. FAF-231

Verified:
Cristofor Fiștic

Chișinău 2025

Contents

1 Objective

Study the dynamic programming method of designing algorithms.

2 The Task

1. Implement in a programming language algorithms Dijkstra and Floyd–Warshall using dynamic programming;
2. Perform empirical analysis of these algorithms for a sparse graph and for a dense graph;
3. Increase the number of nodes in graphs and analyze how this influences the algorithms. Make a graphical presentation of the data obtained;
4. Make a conclusion on the work done.

3 Introduction

Dynamic Programming is a method for designing algorithms.

An algorithm designed with Dynamic Programming divides the problem into subproblems, finds solutions to the subproblems, and puts them together to form a complete solution to the problem we want to solve.

To design an algorithm for a problem using Dynamic Programming, the problem we want to solve must have these two properties:

- **Overlapping Subproblems:** Means that the problem can be broken down into smaller subproblems, where the solutions to the subproblems are overlapping. Having subproblems that are overlapping means that the solution to one subproblem is part of the solution to another subproblem.
- **Optimal Substructure:** Means that the complete solution to a problem can be constructed from the solutions of its smaller subproblems. So not only must the problem have overlapping subproblems, the substructure must also be optimal so that there is a way to piece the solutions to the subproblems together to form the complete solution.

4 Studied Algorithms

The algorithms analyzed in this work are:

- **Dijkstra's Algorithm**

  Dijkstra's algorithm is a popular algorithm for solving single-source shortest path problems having non-negative edge weight in the graphs, it is to find the shortest distance between two vertices on a graph.

- **Floyd-Warshall Algorithm**

  Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. 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).

5 Complexity Analysis

In this analysis, the algorithms are evaluated based on their time complexity and space complexity.

- **Time Complexity:** The number of operations performed as a function of the input size n.
- **Space Complexity:** The amount of additional memory required by the algorithm.

6 Technical Implementation

This analysis implements two algorithms and evaluates their time and space complexities on 2 types of graphs: Sparse and Dense, with sizes: 10, 50, 100, 200. Since Floyd-Warshall return the shortest path from each node to every node, Dijkstra's Algorithm was also run on all nodes. The experiments are conducted on a backend hosted on Google Compute Engine, utilizing Python 3 and approximately 12 GB of RAM. While this environment offers a suitable platform for algorithmic analysis, several limitations and potential sources of error must be acknowledged. These include the possibility that certain algorithms may exhibit suboptimal performance on larger input sizes, as well as inherent inaccuracies in time and memory measurements due to factors such as Python's interpreter overhead and the effects of garbage collection.

6.1 Test Cases

A sparse graph is a type of graph in which the number of edges is significantly less than the maximum number of possible edges. In other words, only a few nodes (or vertices) are connected to each other compared to the total number of connections that could exist.

A dense graph is a type of graph where the number of edges is close to the maximum number of possible edges. In simple terms, most of the vertices are connected to each other and leading to a high level of connectivity.

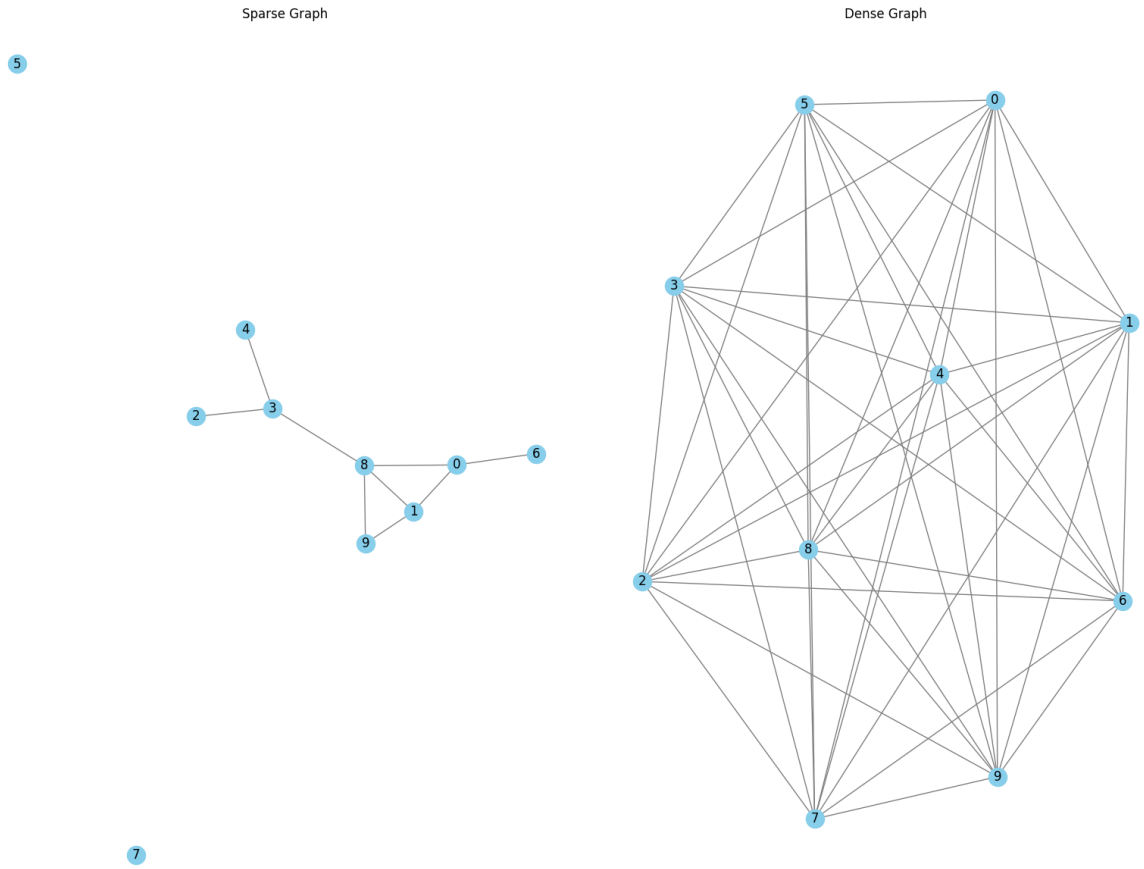

Figure 1: Examples of Analyzed Graphs with 10 Nodes

6.2 Dijkstra's Algorithm

Dijkstra's algorithm is an algorithm for finding the shortest paths between nodes in a weighted graph, which may represent, for example, a road network.

It differs from the minimum spanning tree because the shortest distance between two vertices might not include all the vertices of the graph.

The algorithm uses a greedy approach in the sense that we find the next best solution hoping that the end result is the best solution for the whole problem.

6.2.1 Python Implementation:

```
def dijkstra_optimized(graph, start):
    distances = {node: float('inf') for node in graph}
    distances[start] = 0
    visited = set()
    priority_queue = [(0, start)]

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

        if current_node in visited:
            continue

        print("Visiting", current_node)
        visited.add(current_node)

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

    return distances
```

The time complexity is O(E Log V), where E is the number of edges and V is the number of vertices.

The space complexity is O(V).

6.3 Floyd-Warshall Algorithm

Floyd-Warshall Algorithm is an algorithm for finding the shortest path between all the pairs of vertices in a weighted graph. 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).

6.3.1 Python Implementation:

```
def floyd_warshall_optimized(graph):
  nodes = list(graph.keys())
  num_nodes = len(nodes)

  INF = float('inf')
  dist = {node: {other: INF for other in nodes} for node in
     nodes}

  for u in graph:
    for v in graph[u]:
      dist[u][v] = graph[u][v]
    dist[u][u] = 0

  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
```

There are three loops. Each loop has constant complexity. So, the time complexity of the Floyd-Warshall algorithm is O(n$\hat{3}$).

The space complexity of the Floyd-Warshall algorithm is O(n$\hat{2}$).

7 Performance Results:

The results from sparse graphs revealed notable trends in both time and memory usage. In terms of execution time, Floyd-Warshall exhibited a significant increase as graph sizes grew, starting from 0.003512 seconds for graphs with 10 nodes and escalating to 16.703180 seconds for 200-node graphs. Dijkstra, on the other hand, demonstrated better time efficiency, with execution times ranging from 0.001009 seconds at size 10 to 1.356116 seconds at size 200. It was observed that Dijkstra consistently outperformed Floyd-Warshall in execution time even for the smallest graph size.

When it came to memory usage, Floyd-Warshall used significantly less memory across all graph sizes. Starting at 2.835938 KB for size 10 and peaking at 1344.787109 KB for size 200, its memory consumption remained lower compared to Dijkstra. Dijkstra's memory usage began at 11.789062 KB for size 10 and reached 4184.394531 KB for size 200. This makes Floyd-Warshall the better choice for memory-constrained environments.

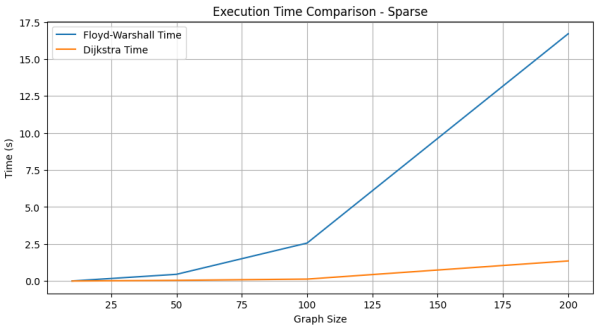

Figure 2: Time Usage on Sparse Graphs

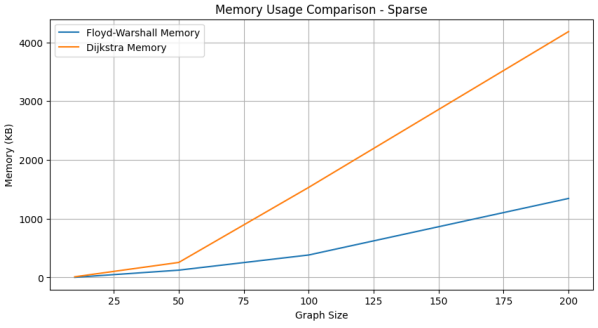

Figure 3: Space Usage on Sparse Graphs

Dense graphs presented similar patterns in algorithm performance. Execution time for Floyd-Warshall increased steadily from 0.003709 seconds at size 10 to 18.597410 seconds at size 200. Dijkstra once again proved to be faster, with execution times starting at 0.000961 seconds at size 10 and ending at 0.884868 seconds at size 200. The observations confirmed that Dijkstra consistently outperformed Floyd-Warshall in terms of speed.

Memory usage, however, painted a different picture. As with sparse graphs, Floyd-Warshall utilized less memory in dense graphs as well. Its memory usage began at 2.835938 KB for size 10 and rose to 1335.097656 KB for size 200. Dijkstra required more memory for all graph sizes, starting at 11.515625 KB for size 10 and increasing to 4175.983398 KB for size 200. This consistent trend made Floyd-Warshall favorable for applications with limited memory capacity.

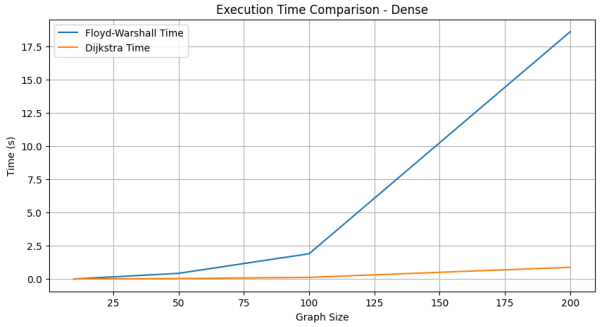

Figure 4: Time Usage on Dense Graphs

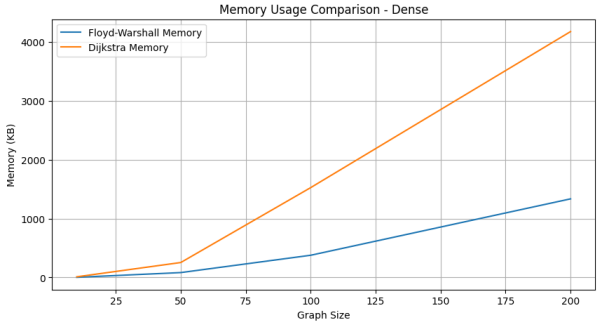

Figure 5: Space Usage on Dense Graphs

The findings highlight clear distinctions between the two algorithms in terms of efficiency and resource usage. For scenarios where execution speed is of utmost importance, Dijkstra is the ideal choice. It demonstrates exceptional time efficiency, particularly as graph sizes increase. On the other hand, Floyd-Warshall proves to be more suitable for applications where memory is a critical constraint. Its lower memory footprint makes it a viable option when handling larger graphs or working within environments with limited computational resources.

Conclusion

This comparative analysis underscores that the choice between Floyd-Warshall and Dijkstra depends on the specific requirements of the task at hand. Dijkstra is a robust choice when speed is the primary concern, whereas Floyd-Warshall is optimal when memory conservation is imperative. The study of both algorithms within the context of sparse and dense graphs provides valuable insights for practitioners and researchers in the field of graph algorithms, serving as a guide to optimize algorithm selection based on the unique demands of their applications.

8 Bibliography

- The code is available at: Google Colab Notebook
- GitHub Repository: GitHub