MINISTERUL EDUCAȚIEI, CULTURII ȘI CERCETĂRII AL REPUBLICII MOLDOVA Universitatea Tehnică a Moldovei Facultatea Calculatoare, Informatică şi Microelectronică Departamentul Inginerie Software și Automatică

Copta Adrian | FAF-223

Report

Laboratory work 4:

Dynamic programming

Checked by:

Fiștic Cristof, *university assistant*

DISA, FCIM, UTM

Chișinău – 2024

TABLE OF CONTENTS:

ALGORITHM ANALYSIS

Objective

Dynamic programming is defined as 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.

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

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 analyzed. 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 behavior 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 is both a mathematical optimization method and an algorithmic paradigm. The method was developed by Richard Bellman in the 1950s and has found applications in numerous fields, from aerospace engineering to economics. In both contexts it refers to simplifying a complicated problem by breaking it down into simpler sub-problems in a recursive manner. While some decision problems cannot be taken apart this way, decisions that span several points in time do often break apart recursively. Likewise, in computer science, if a problem can be solved optimally by breaking it into sub-problems and then recursively finding the optimal solutions to the sub-problems, then it is said to have optimal substructure. If sub-problems can be nested recursively inside larger problems, so that dynamic programming methods are applicable, then there is a relation between the value of the larger problem and the values of the sub-problems. In the optimization literature this relationship is called the Bellman equation.

Comparison Metric:

The comparison metric for this laboratory work will be considered the time of execution of each algorithm (O(n))

Input Format:

As input, each algorithm will receive different size graphs. We will generate different graphs (sparse and dense). Each algorithm will be called with the generated graph and the time execution will be saved in an array to later be plotted.

IMPLEMENTATION

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. It was conceived by computer scientist Edsger W. Dijkstra in 1956 and published three years later. The algorithm exists in many variants. Dijkstra's original algorithm found the shortest path between two given nodes, but a more common variant fixes a single node as the "source" node and finds shortest paths from the source to all other nodes in the graph, producing a shortest-path tree.

For a given source node in the graph, the algorithm finds the shortest path between that node and every other. It can also be used for finding the shortest paths from a single node to a single destination node by stopping the algorithm once the shortest path to the destination node has been determined. For example, if the nodes of the graph represent cities and costs of edge paths represent driving distances between pairs of cities connected by a direct road (for simplicity, ignore red lights, stop signs, toll roads and other obstructions), then Dijkstra's algorithm can be used to find the shortest route between one city and all other cities. A widely used application of shortest path algorithms is network routing protocols, most notably IS-IS (Intermediate System to Intermediate System) and OSPF (Open Shortest Path First). It is also employed as a subroutine in other algorithms such as Johnson's. The Dijkstra algorithm uses labels that are positive integers or real numbers, which are totally ordered. It can be generalized to use any labels that are partially ordered, provided the subsequent labels (a subsequent label is produced when traversing an edge) are monotonically non-decreasing. This generalization is called the generic Dijkstra shortest-path algorithm.

Implementation:

```
# Dijkstra's Algorithm
def min_distance(dist, visited):
    min_dist = float('inf')
    min_index = -1

    for i in range(len(dist)):
        if dist[i] < min_dist and not visited[i]:
            min_dist = dist[i]
            min_index = i

    return min_index
```

```
def dijkstra(graph, start):
    n = len(graph)
    dist = [float('inf')] * n
    dist[start] = 0
    visited = [False] * n

    for _ in range(n):
        u = min_distance(dist, visited)
        visited[u] = True

        for v in range(n):
            if graph[u][v] > 0 and not visited[v] and dist[v] > dist[u] +
graph[u][v]:
                dist[v] = dist[u] + graph[u][v]

    return dist
```

Figure 1 Dijkstra's algorithm python implementation

Floyd Warshall Algorithm:

In computer science, the Floyd–Warshall algorithm (also known as Floyd's algorithm, the Roy–Warshall algorithm, the Roy–Floyd algorithm, or the WFI algorithm) is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights (but with no negative cycles). A single execution of the algorithm will find the lengths (summed weights) of shortest paths between all pairs of vertices. Although it does not return details of the paths themselves, it is possible to reconstruct the paths with simple modifications to the algorithm. Versions of the algorithm can also be used for finding the transitive closure of a relation R, or (in connection with the Schulze voting system) widest paths between all pairs of vertices in a weighted graph.

Implementation:

```
# Floyd-Warshall Algorithm
def floyd_warshall(graph):
    n = len(graph)
    dist = np.copy(graph)

    for k in range(n):
        for i in range(n):
```

```
            for j in range(n):
                dist[i][j] = min(dist[i][j], dist[i][k] + dist[k][j])

    return dist
```

Figure 2 Floyd–Warshall algorithm python implementation

Results:

Algorithmic Complexity:

The time complexity of Dijkstra Algorithm depends on the specific implementation and data structures used. In a straightforward implementation for dense graphs where E (no. of edges) is approximately equal to V2 (no. of Vertices) time complexity is of O(V2) . However, for Sparse graphs using Priority queue or Min-heap is more optimal which gives time complexity of O(E + V*log(V)).

For the Floyd Warshall Algorithm, the time complexity is O(V^3), where V represents the number of vertices in the graph. It involves three nested loops that iterate through all possible pairs of nodes and consider all possible intermediaries, leading to a cubic time complexity.

Analysis of Results:

The benchmark results show that Dijkstra's algorithm is significantly faster than Floyd-Warshall's algorithm for both sparse and dense graphs. This is because the time complexity of Dijkstra's algorithm is O(E log V), where E is the number of edges and V is the number of vertices, while the time complexity of Floyd-Warshall's algorithm is O(V^3).

In the case of sparse graphs, where the number of edges is much less than the number of possible edges (E << V^2), this difference in time complexity is even more pronounced. As the number of nodes increases, the execution time of Floyd-Warshall's algorithm grows much faster than that of Dijkstra's algorithm.

Here's a more detailed breakdown of the results for sparse and dense graphs:

- Sparse Graphs:
    - Dijkstra's algorithm takes about 2.5 seconds to execute for 300 nodes.
    - Floyd-Warshall's algorithm takes about 12.5 seconds to execute for 300 nodes.
- Dense Graphs:
    - Dijkstra's algorithm takes about 155 seconds to execute for 300 nodes.
    - Floyd-Warshall's algorithm takes about the same amount of time (around 155 seconds) to execute for 300 nodes.

Even though Floyd-Warshall's algorithm is slower than Dijkstra's algorithm, it has an advantage: it can compute the shortest paths between all pairs of nodes in the graph, while Dijkstra's algorithm can only compute the shortest path from a single source node to all other nodes.

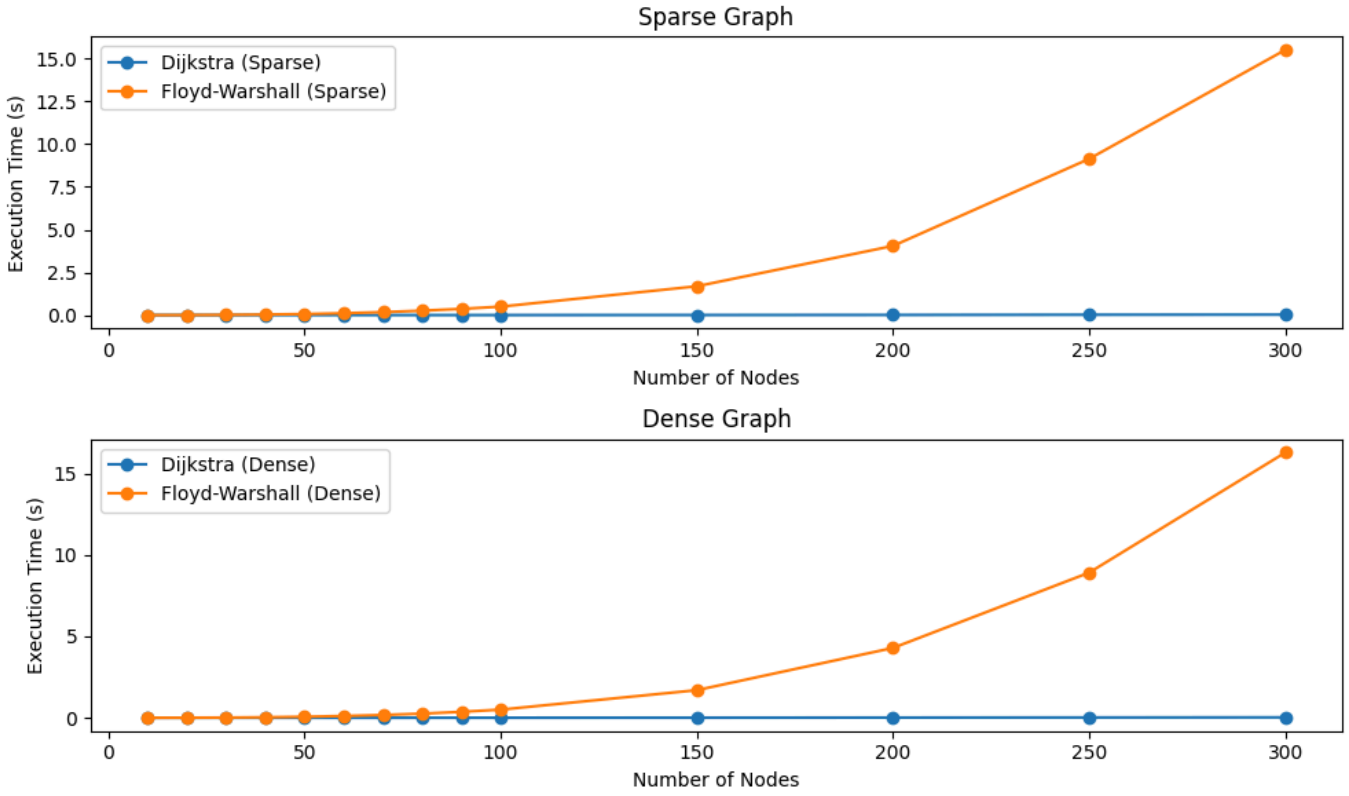

Figure 3 benchmark of Dijkstra Algorithm Floyd & Warshall Algorithm

Comparison Basis	Dijkstra Algorithm	Floyd Warshall Algorithm
Introduction	Dijkstra's algorithm is a single-source shortest path algorithm used to find the shortest paths from a single source vertex to all other vertices in a weighted graph.	The Floyd-Warshall algorithm is an all-pairs shortest path algorithm used to find the shortest paths between all pairs of vertices in a weighted graph.
Algorithm	Greedy approach,as it selects the vertex with the shortest distance	It uses Dynamic programming, to compute all pair shortest paths.
Data Structure	It Typically uses a priority queue or min-heap.	It uses a two-dimensional array.
Negative Edges	Dijkstra's algorithm does not work correctly with graphs that have negative edge weights.	The Floyd-Warshall algorithm can handle graphs with both positive and negative edge weights.
Time Complexity	With a priority queue or min-heap, time complexity is O(E + V*log(V)).	The time complexity of the Floyd-Warshall algorithm is O(V^3).
Auxilary Space	O(V) with priority queue or min-heap.	The auxiliary space complexity is O(V^2) because of 2D array used.
Use Case	It is efficient for finding shortest paths from a single source.	It is suitable for finding shortest paths between all pairs of vertices

Figure 4 comparison table

CONCLUSION:

In conclusion, the choice of algorithm depends on the specific needs of your application. If you only need to find the shortest path between one source node and all other nodes in a graph, and the graph is sparse, then Dijkstra's algorithm is the better choice. However, if you need to find the shortest paths between all pairs of nodes in a graph, or if the graph is dense, then Floyd-Warshall's algorithm may be the better choice, despite its slower execution time.