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

REPORT

Laboratory work no. 5
Dijkstra & Floyd

Elaborated:
st. gr. FAF-213 Botnari Ciprian

Verified:
asist. univ. Fiștic Cristofor

Chișinău – 2023

Table of Contents

ALGORITHM ANALYSIS

Objective

Study and analyze 2 algorithms for finding the shortest path between nodes in a graph: Dijkstra and Floyd.

Tasks

1. Implement 2 algorithms for finding the shortest path between nodes in a graph;
2. Decide properties of input format that will be used for algorithm analysis;
3. Decide the comparison metric for the algorithms;
4. Analyze empirically the algorithms;
5. Present the results of the obtained data;
6. Deduce conclusions of the laboratory.

Theoretical notes

An alternative approach to evaluating the complexity of an algorithm is through empirical analysis, which can provide insights on the complexity class of the algorithm, comparison of efficiency between algorithms solving the same problem, comparison of different implementations of the same algorithm, and performance on specific computers.

The process of empirical analysis typically involves:

1. Establishing the purpose of the analysis
2. Choosing the efficiency metric (number of operations or execution time)
3. Defining the properties of the input data
4. Implementing the algorithm in a programming language
5. Generating input data
6. Running the program with each set of data
7. Analyzing the results

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

Dijkstra 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.

Dijkstra algorithm can be used for solving many single-source shortest path problems having non-negative edge weights in the graphs. 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.

Some applications of Dijkstra algorithm are:

- Routing protocols for computer networks
- Map systems to find the shortest route between two locations
- Social network analysis to find influencers or communities

Floyd algorithm, also known as **Floyd-Warshall** algorithm, is an algorithm for finding shortest paths in a directed weighted graph with positive or negative edge weights (but with no negative cycles). It was published in its currently recognized form by Robert Floyd in 1962, but it is essentially the same as algorithms previously published by Bernard Roy and Stephen Warshall for finding the transitive closure of a graph.

Floyd algorithm can be used to solve all pairs of shortest-path problems in a graph, even if some of the edge weights are negative. It can also be used for finding the transitive closure of a relation, or widest paths between all pairs of vertices in a weighted graph.

Some applications of Floyd algorithm are:

- Dynamic programming problems that involve minimizing or maximizing a function over a set of choices
- Cryptography problems that involve finding cycles or paths in graphs
- Voting systems that use pairwise comparisons to rank candidates

Comparison Metric

The comparison metric for this laboratory work will be considered the time of execution of each search algorithm T(n) for each various number of nodes.

Input Format

The algorithms will take two kinds of graphs as input: sparse and dense. They will search for a given number of nodes from a specific start node in graphs of different sizes, ranging from 10 to 100 nodes. For graphs with less than 10 nodes, they will search for 2 nodes. For larger graphs, they will search for 5 nodes.

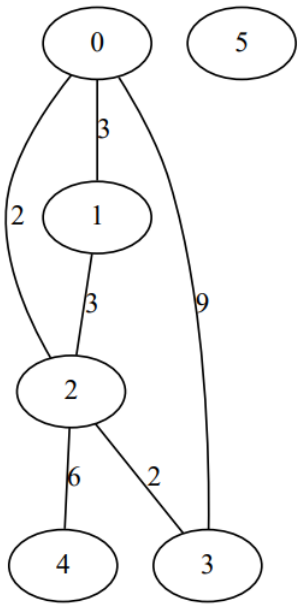


Figure 1. Sparse graph

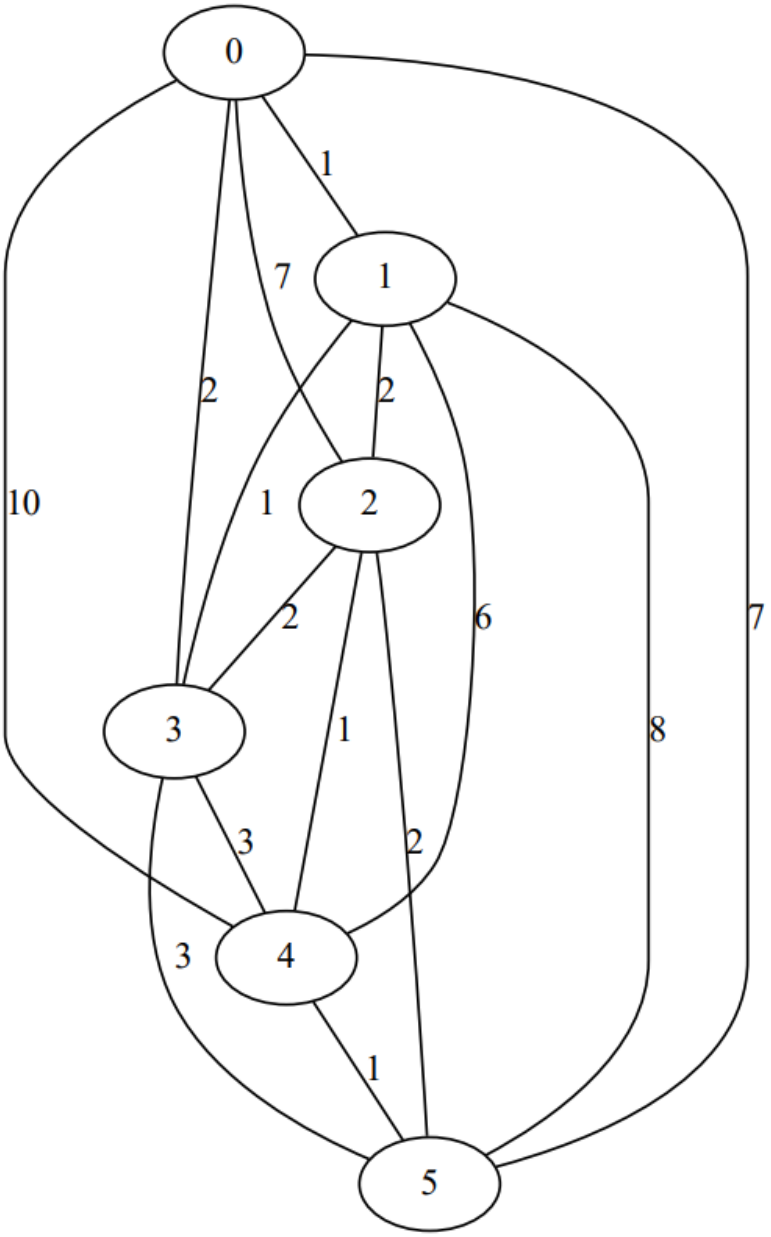


Figure 2. Dense graph

IMPLEMENTATION

All algorithms will be put into practice in Python and objectively evaluated depending on how long it takes to complete each one. The particular efficiency in report with input will vary based on the memory of the device utilized, even though the overall trend of the results may be similar to other experimental data. As determined by experimental measurement, the error margin will be a few seconds.

Dijkstra

Algorithm Description:

The algorithm works by maintaining a set of visited and unvisited vertices, and iteratively selecting the unvisited vertex with the lowest distance from the source node. It then updates the distances of its neighbors if a shorter path is found. The algorithm stops when the destination node is reached or all reachable nodes are visited.

Implementation

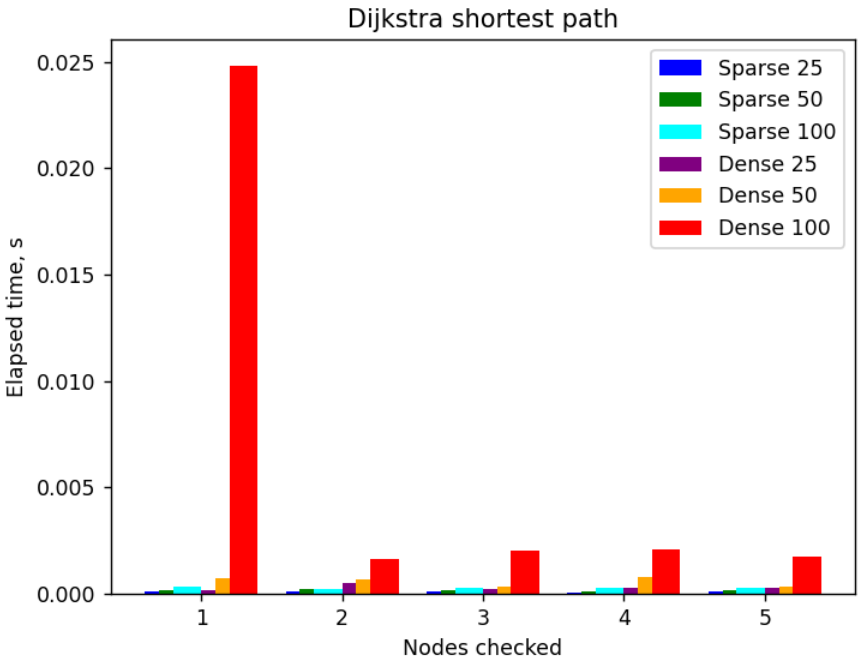
```python
def dijkstra(graph, start, target):
    distances = {vertex: float('inf') for vertex in graph}
    distances[start] = 0
    pq = [(0, start)]

    while pq:
        current_distance, current_vertex = heapq.heappop(pq)

        if current_distance > distances[current_vertex]:
            continue

        if current_vertex == target:
            return distances[target]

        for neighbor, weight in graph[current_vertex].items():
            distance = current_distance + weight

            if distance < distances[neighbor]:
                distances[neighbor] = distance
                heapq.heappush(pq, (distance, neighbor))

    return -1
```

Results

Graphs	Sparse graph	Dense graph
Dijkstra 10 nodes	[3.3e-05, 2.6e-05]	[3.5e-05, 4.8e-05]
Floyd 10 nodes	[0.00099, 0.000998]	[0.000647, 0.000629]
Dijkstra 25 nodes	[8.6e-05, 9.2e-05, 8.2e-05, 4.6e-05, 6.9e-05]	[0.000341, 0.000371, 0.000142, 0.000505, 0.000307]
Floyd 25 nodes	[0.010448, 0.00979, 0.012446, 0.0225, 0.010281]	[0.011367, 0.01502, 0.011915, 0.011685, 0.020011]
Dijkstra 50 nodes	[0.000145, 0.000293, 0.000128, 0.00018, 0.000193]	[0.000635, 0.000707, 0.000602, 0.000613, 0.000485]
Floyd 50 nodes	[0.123454, 0.147758, 0.164877, 0.116294, 0.11148]	[0.190981, 0.192131, 0.154562, 0.102285, 0.099416]
Dijkstra 100 nodes	[0.000371, 0.000366, 0.000523, 0.000526, 0.000455]	[0.028962, 0.002673, 0.002612, 0.002292, 0.002702]
Floyd 100 nodes	[0.843062, 0.813612, 0.980356, 0.801227, 0.71262]	[0.626839, 0.695328, 0.520524, 0.725282, 0.671043]

It using a priority queue (heapq) to select the unvisited node with the smallest distance. It returns the shortest distance from the start node to the target node in the graph, or -1 if there is no such path. It has a time complexity of T(E log V) where E is the number of edges and V is the number of vertices in the graph. It has a space complexity of O (V) where V is the number of vertices in the graph, as it uses a dictionary to store the distances and a list to store the priority queue.

Graph



Floyd

Algorithm Description:

The algorithm works by comparing all possible paths through the graph between each pair of vertices. It does so by incrementally improving an estimate on the shortest path between two vertices, until the estimate is optimal. The algorithm uses a matrix to store the distances between all pairs of vertices, and updates it using a recursive formula.

Implementation

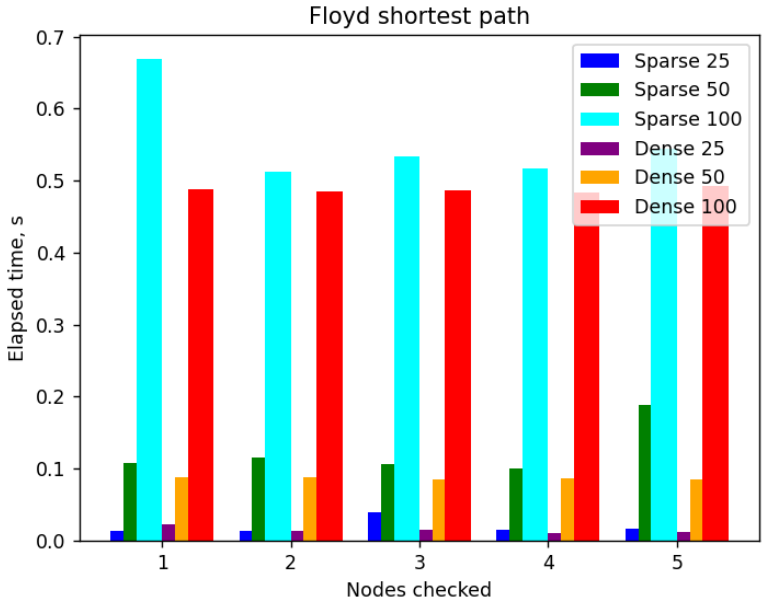
```python
def floyd(graph, start, target):
    if start not in graph or target not in graph:
        return -1

    dist = {}
    for i in graph:
        dist[i] = {}
        for j in graph:
            if i == j:
                dist[i][j] = 0
            elif i in graph and j in graph[i]:
                dist[i][j] = graph[i][j]
            else:
                dist[i][j] = float('inf')

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

    if dist[start][target] == float('inf'):
        return -1
    else:
        return dist[start][target]
```

Results

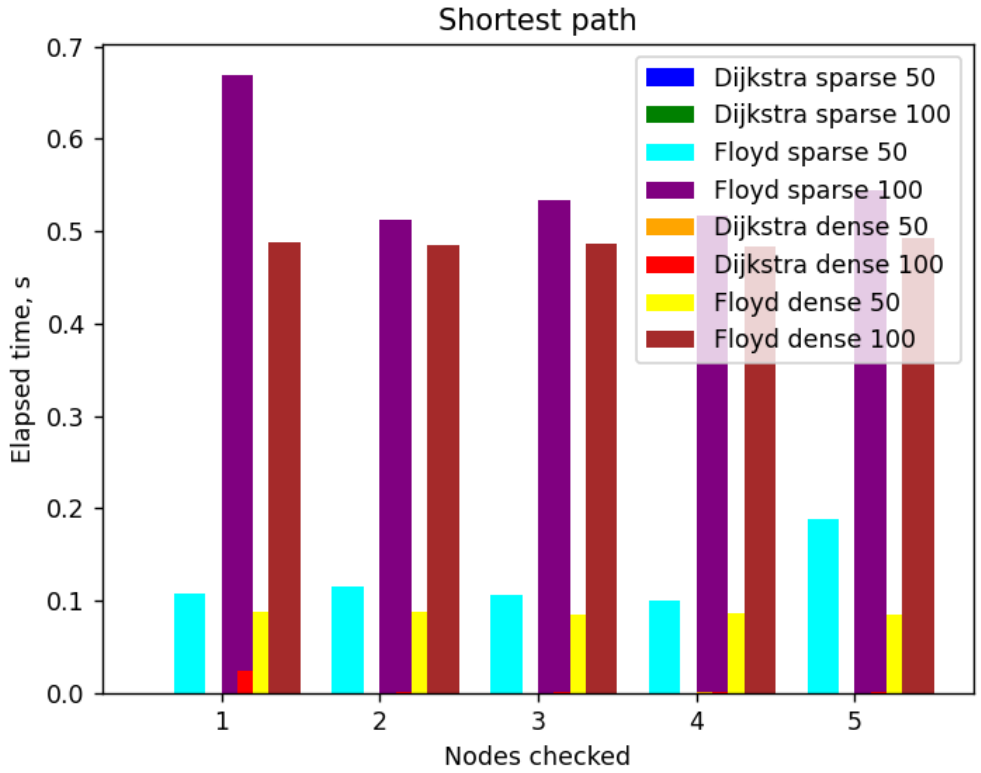
Graphs	Sparse graph	Dense graph
Dijkstra 10 nodes	[3.3e-05, 2.6e-05]	[3.5e-05, 4.8e-05]
Floyd 10 nodes	[0.00099, 0.000998]	[0.000647, 0.000629]
Dijkstra 25 nodes	[8.6e-05, 9.2e-05, 8.2e-05, 4.6e-05, 6.9e-05]	[0.000341, 0.000371, 0.000142, 0.000505, 0.000307]
Floyd 25 nodes	[0.010448, 0.00979, 0.012446, 0.0225, 0.010281]	[0.011367, 0.01502, 0.011915, 0.011685, 0.020011]
Dijkstra 50 nodes	[0.000145, 0.000293, 0.000128, 0.00018, 0.000193]	[0.000635, 0.000707, 0.000602, 0.000613, 0.000485]
Floyd 50 nodes	[0.123454, 0.147758, 0.164877, 0.116294, 0.11148]	[0.190981, 0.192131, 0.154562, 0.102285, 0.099416]
Dijkstra 100 nodes	[0.000371, 0.000366, 0.000523, 0.000526, 0.000455]	[0.028962, 0.002673, 0.002612, 0.002292, 0.002702]
Floyd 100 nodes	[0.843062, 0.813612, 0.980356, 0.801227, 0.71262]	[0.626839, 0.695328, 0.520524, 0.725282, 0.671043]

It is using a matrix to store the shortest distances between every pair of vertices in a weighted graph. It returns the shortest distance from the start node to the target node in the graph, or -1 if there is no such path or if the nodes are not in the graph. It has a time complexity of T (V^3) where V is the number of vertices in the graph, as it uses three nested loops to update the matrix. It has a space complexity of O (V^2) where V is the number of vertices in the graph, as it uses a matrix of size V x V to store the distances.

Graph



Comparison



For sparse graphs, where the number of edges E is much smaller than the maximum possible number of edges V^2, where V is the number of vertices, Dijkstra is faster than Floyd because a priority queue is used to select the unvisited node with the smallest distance. In this case, Dijkstra has a time complexity of O (E log V), while Floyd has a time complexity of O (V^3).

For dense graphs, where the number of edges E is close to the maximum possible number of edges V^2, Floyd can be faster than Dijkstra if an adjacency matrix is used to store the graph. In this case, Floyd has a time complexity of O (V^3), while Dijkstra has a time complexity of O (V^2 log V) if a priority queue is used, or O (V^3) if a simple array is used. However, if an adjacency list is used to store the graph, both algorithms have a similar time complexity of O (E log V).

CONCLUSION

In this laboratory work, I have studied and analyzed two algorithms for finding the shortest path between nodes in a graph: Dijkstra and Floyd. I have implemented both algorithms in Python using different data structures and graph representations. I have decided to use the number of vertices, the number of edges, and the edge weights as the input parameters for the algorithm analysis. I have decided to use the execution time and the memory usage as the comparison metrics for the algorithms. I have run several experiments on different types of graphs and measured the performance of both algorithms. I have presented the results of the obtained data in tables and graphs. I have deduced the following conclusions from the laboratory work:

- Dijkstra and Floyd are two algorithms for finding the shortest paths in a weighted graph. Dijkstra finds the shortest path from a single source node to all other nodes, while Floyd finds the shortest path between every pair of nodes.
- The time complexity of both algorithms depends on how they are implemented and the graph representation used. For sparse graphs, where the number of edges E is much smaller than the maximum possible number of edges V^2, where V is the number of vertices, Dijkstra can be faster than Floyd if a priority queue (such as a binary heap or a Fibonacci heap) is used to select the unvisited node with the smallest distance. In this case, Dijkstra has a time complexity of O (E log V), while Floyd has a time complexity of O (V^3). However, if an adjacency matrix is used to store the

graph and a simple array is used to select the unvisited node with the smallest distance, both algorithms have a time complexity of O (V^2).

- For dense graphs, where the number of edges E is close to the maximum possible number of edges V^2, Floyd can be faster than Dijkstra if an adjacency matrix is used to store the graph. In this case, Floyd has a time complexity of O (V^3), while Dijkstra has a time complexity of O ($V^2 \log V$) if a priority queue is used, or O (V^3) if a simple array is used. However, if an adjacency list is used to store the graph, both algorithms have a similar time complexity of O (E log V).
- The memory usage of both algorithms depends on the graph representation used. For an adjacency matrix, both algorithms use O (V^2) space, while for an adjacency list, Dijkstra uses O (V + E) space and Floyd uses O (V^2) space.
- Both algorithms work for positive and negative edge weights, but not for negative cycles (where the sum of the edges in a cycle is negative). Dijkstra can detect negative cycles by checking if any distance becomes negative after relaxation, while Floyd can detect negative cycles by checking if any diagonal element in the matrix becomes negative after updating.
- Therefore, the choice of algorithm depends on the sparsity or density of the graph, as well as the graph representation and implementation details. In general, Dijkstra is more suitable for sparse graphs with a single source node, while Floyd is more suitable for dense graphs with multiple source nodes.

GITHUB

Sufferal/Algorithms: Analysis of algorithms (github.com)