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

REPORT

Laboratory work no.5
Dijkstra and Floyd-Warshall algorithms

Elaborated:
st. gr. FAF-213 Iațco Sorin

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

ALGORITHM ANALYSIS 3
Objective 3
Tasks 3
Theoretical notes 3
Introduction 3
Comparison Metric 4
Input Format 4
IMPLEMENTATION 5
Dijkstra’s 5
Floyd-Warshall 6
CONCLUSION 9
REFERENCES 9

ALGORITHM ANALYSIS

Objective

Empirical analysis for specified algorithms.

Tasks

1. Implement the algorithms listed above in a programming language
2. Establish the properties of the input data against which the analysis is performed
3. Choose metrics for comparing algorithms
4. Perform empirical analysis of the proposed algorithms
5. Make a graphical presentation of the data obtained
6. Make a conclusion on the work done.

Theoretical notes

An alternate approach to analyzing complexity is to use empirical analysis, which involves implementing the algorithm in a programming language and running it with multiple sets of input data to obtain data on its efficiency. This method is useful for various purposes, such as gaining preliminary information on an algorithm's complexity class, comparing the efficiency of different algorithms or implementations, or assessing the performance of an algorithm on a specific computer.

The choice of efficiency measure depends on the purpose of the analysis. If the goal is to obtain information on the complexity class or check the accuracy of a theoretical estimate, then the number of operations performed is appropriate. However, if the aim is to assess the behavior of the implementation of an algorithm, then execution time is more suitable. The results are recorded and either synthetic quantities, such as mean and standard deviation, are calculated, or a graph with appropriate pairs of points is plotted to analyze the data.

Introduction

Dijkstra's algorithm is a shortest path algorithm that finds the shortest path between a starting vertex and all other vertices in a weighted graph. It works by maintaining a priority queue of vertices to visit, and at each step, it selects the vertex with the smallest tentative distance from the starting vertex, updates the distances of its neighbors, and adds them to the queue if necessary. This process continues until all vertices have been visited, resulting in the shortest path from the starting vertex to every other vertex in the graph.

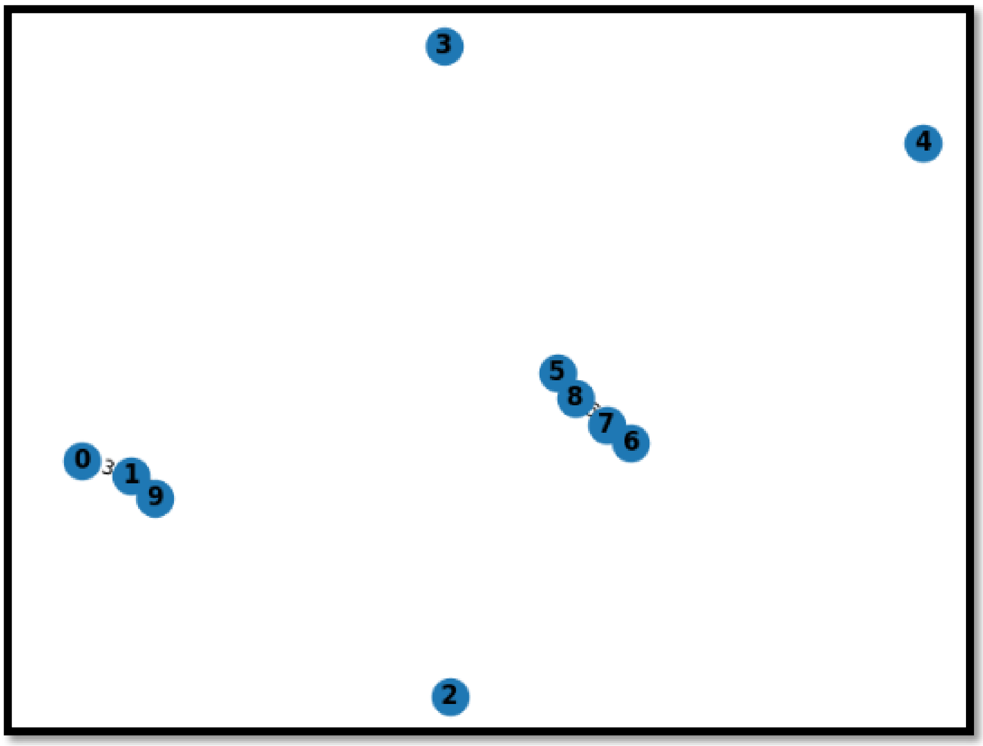
On the other hand, Floyd-Warshall algorithm is an all-pairs shortest path algorithm that finds the shortest path between every pair of vertices in a weighted graph. It works by maintaining a matrix of the shortest distances between pairs of vertices, and at each step, it updates the matrix by considering all possible intermediate vertices in the path. This process continues until all pairs of vertices have been considered, resulting in the shortest path between every pair of vertices in the graph.

Comparison Metric

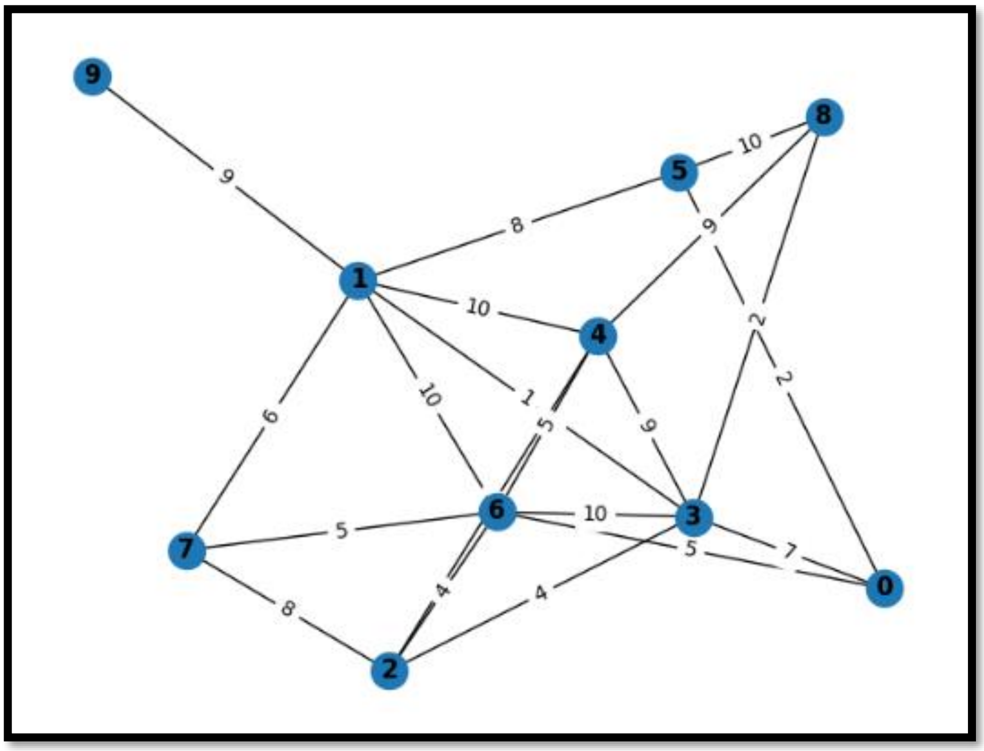
The comparison metric for this laboratory work will be considered the time of execution of each algorithm (T(n)).

Input Format

As input, there are generated multiple weighted graphs each in ascending order for nodes, sparse and then dense.



Sparse Graph



Dense Graph

IMPLEMENTATION

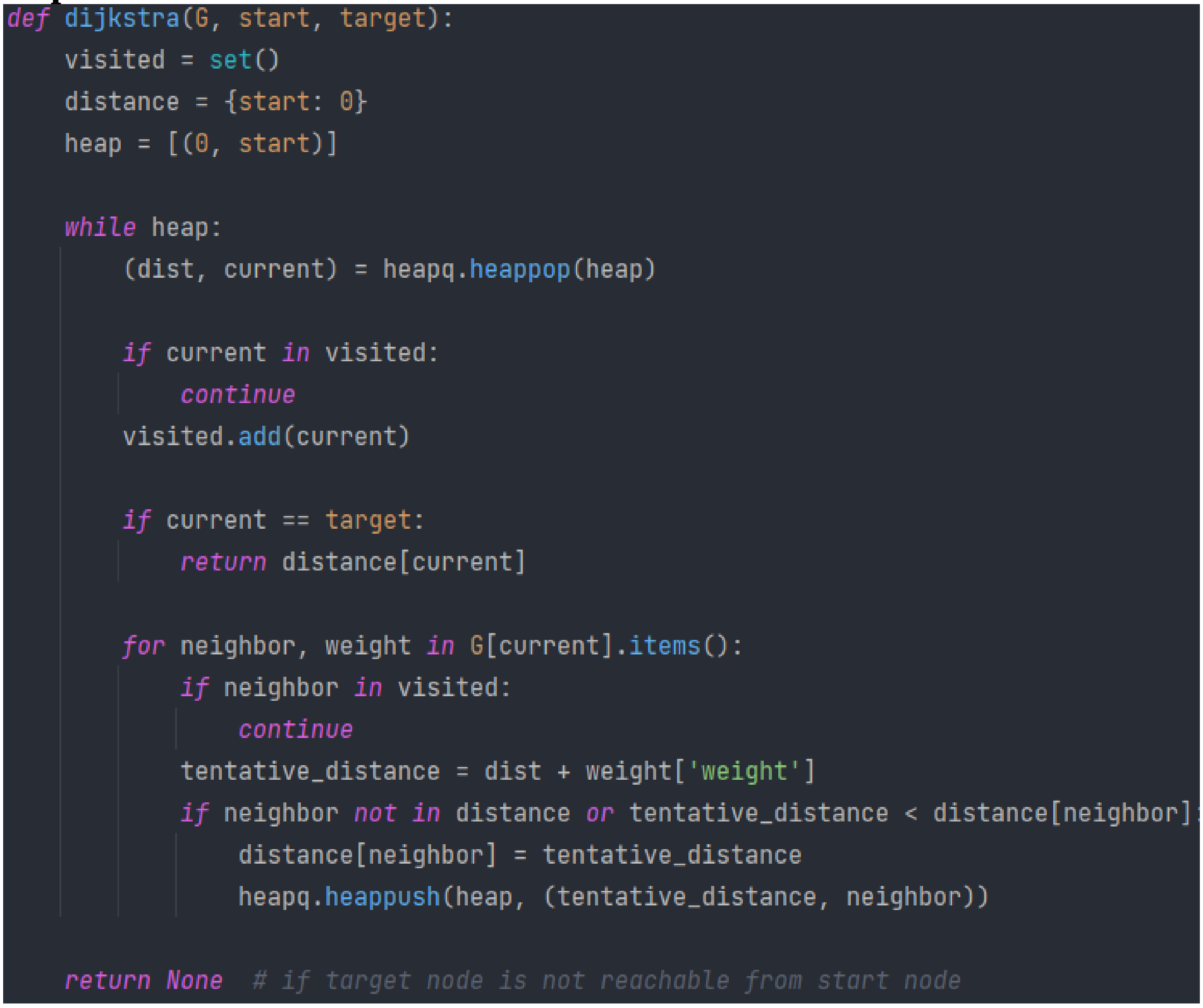
Both algorithms will be implemented in their naive form in python an analyzed empirically based on the time required for their completion. While the general trend of the results may be similar to other experimental observations, the particular efficiency in rapport with input will vary depending on performance of the device used.

Dijkstra's

Pseudocode:

```
DIJKSTRA (G, w, s)
 INITIALIZE-SINGLE-SOURCE(G,s)
 S ← Ø
 Q ← V[G]
 while Q ≠ Ø
     do u ← EXTRACT-MIN(Q)
        S ← S ∪ {u}
        for each vertex v ∈ Adj[u]
            do RELAX(u,v,w)
```

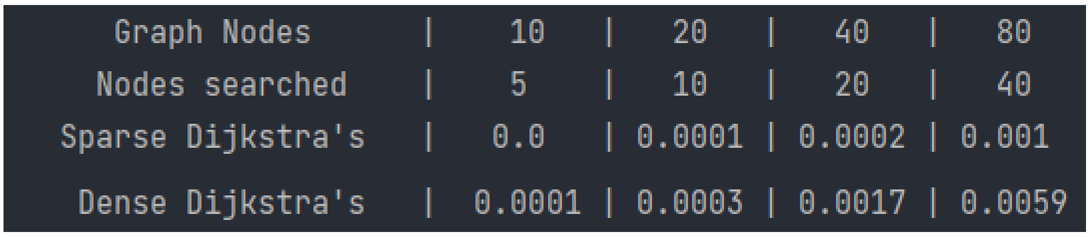
Implementation:



```python
def dijkstra(G, start, target):
    visited = set()
    distance = {start: 0}
    heap = [(0, start)]

    while heap:
        (dist, current) = heapq.heappop(heap)

        if current in visited:
            continue
        visited.add(current)

        if current == target:
            return distance[current]

        for neighbor, weight in G[current].items():
            if neighbor in visited:
                continue
            tentative_distance = dist + weight['weight']
            if neighbor not in distance or tentative_distance < distance[neighbor]:
                distance[neighbor] = tentative_distance
                heapq.heappush(heap, (tentative_distance, neighbor))

    return None  # if target node is not reachable from start node
```

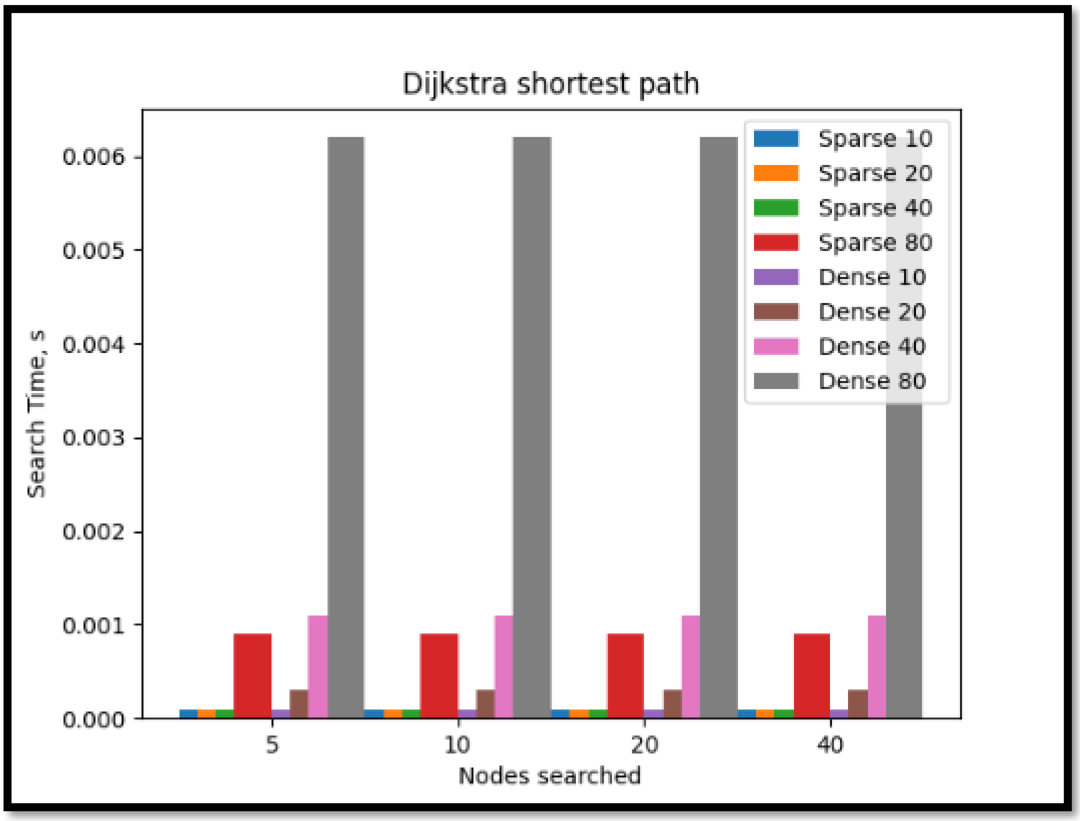
The time complexity of Dijkstra's algorithm depends on the data structure used to implement the priority queue. If a binary heap is used, the time complexity is O(E log V), where E is the number of edges and V is the number of vertices in the graph. If a Fibonacci heap is used, the time complexity is O(E + V log V).
In the worst case, Dijkstra's algorithm may need to process all edges and vertices in the graph, so the time complexity cannot be better than O(E+V log V). However, in practice, the algorithm often performs much faster than this worst-case bound, especially if the graph is sparse or if a good heuristic is used to guide the search.

Results:



```
     Graph Nodes        |    10    |    20    |    40    |    80
    Nodes searched      |    5     |    10    |    20    |    40
  Sparse Dijkstra's     |   0.0    | 0.0001 | 0.0002 | 0.001

   Dense Dijkstra's     |  0.0001 | 0.0003 | 0.0017 | 0.0059
```

Graph:



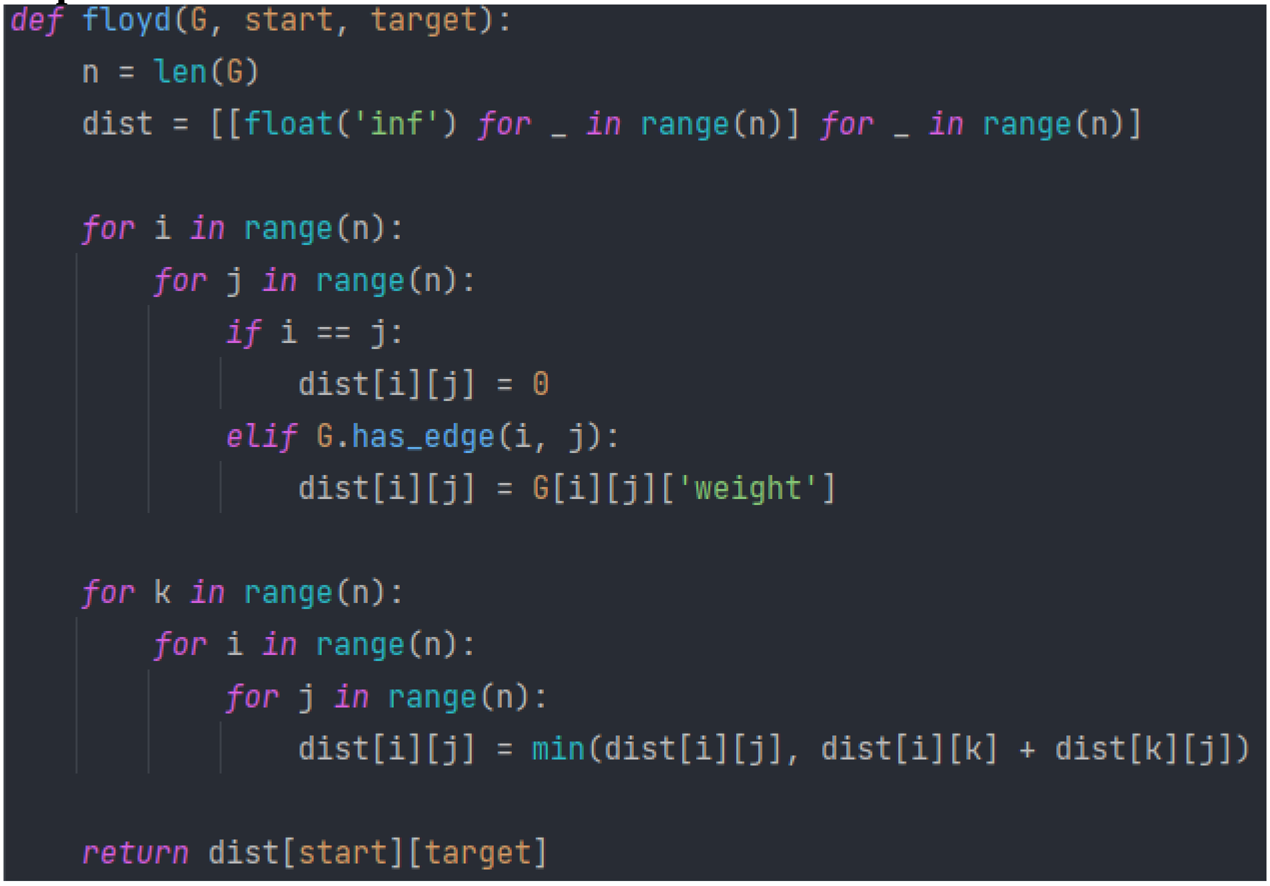
Floyd-Warshall

Pseudocode:

```
FloydWarshall(G):
        Input: G, Graph;
        Output: d, an adjacency matrix of distances between All vertex pairs
Let d be an adj. matrix (2d array) initialized to +inf
```

```
foreach (Vertex v : G):
        d[v][v] = 0
foreach (Edge (u, v) : G):
        d[u][v] = cost(u, v)
foreach (Vertex k : G):
        foreach (Vertex u : G):
                foreach (Vertex v : G):
                        if d[u, v] > d[u, k] + d[k, v]: d[u, v] = d[u, k] + d[k, v]
return d
```

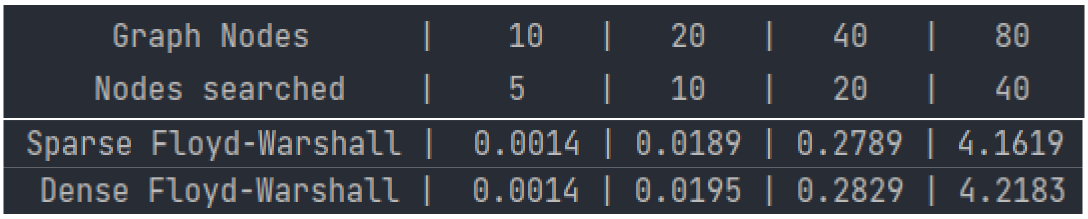
Implementation:



```python
def floyd(G, start, target):
    n = len(G)
    dist = [[float('inf') for _ in range(n)] for _ in range(n)]

    for i in range(n):
        for j in range(n):
            if i == j:
                dist[i][j] = 0
            elif G.has_edge(i, j):
                dist[i][j] = G[i][j]['weight']

    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[start][target]
```

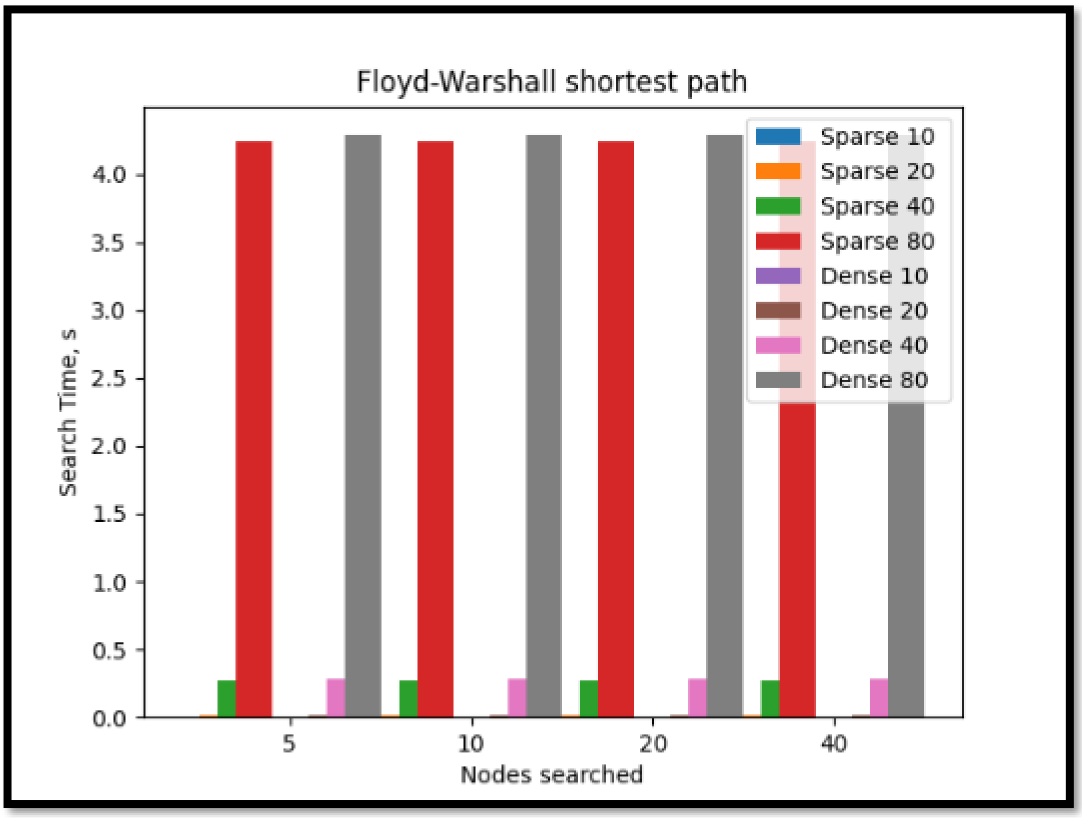
The time complexity of the Floyd-Warshall algorithm is O(V^3), where V is the number of vertices in the graph. This means that the algorithm requires a cubic number of operations to compute the shortest path between every pair of vertices in the graph.
The Floyd-Warshall algorithm achieves this time complexity by considering all possible intermediate vertices in the path between every pair of vertices. Specifically, the algorithm uses dynamic programming to compute the shortest path from each vertex to every other vertex by considering all possible intermediate vertices.
While the cubic time complexity may seem high, the Floyd-Warshall algorithm is often practical for small to medium-sized graphs, especially when the graph is dense. However, for larger graphs or for real-time applications where the graph changes frequently, more specialized algorithms may be necessary to achieve acceptable performance.

Results:



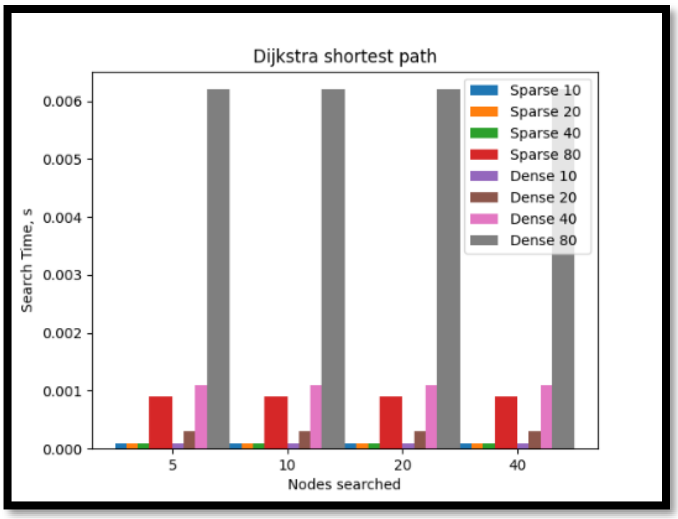
```
        Graph Nodes        |   10    |   20    |   40    |   80
       Nodes searched      |    5    |   10    |   20    |   40
 Sparse Floyd-Warshall |  0.0014 | 0.0189 | 0.2789 | 4.1619
  Dense Floyd-Warshall |  0.0014 | 0.0195 | 0.2829 | 4.2183
```

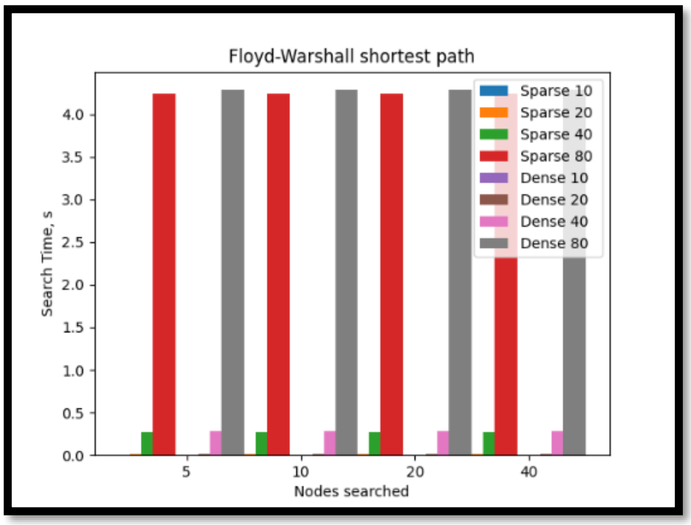
Graph:



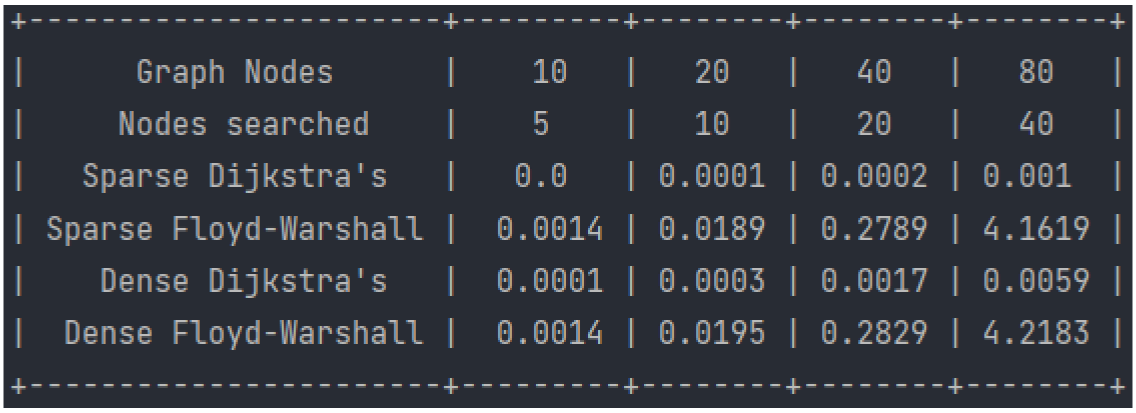
Both algorithms compared:

Graphs:





Time Table:



```
+-----------------------+---------+--------+--------+--------+
|      Graph Nodes      |    10   |   20   |   40   |   80   |
|     Nodes searched    |    5    |   10   |   20   |   40   |
|   Sparse Dijkstra's   |   0.0   | 0.0001 | 0.0002 | 0.001  |
| Sparse Floyd-Warshall |  0.0014 | 0.0189 | 0.2789 | 4.1619 |
|    Dense Dijkstra's   |  0.0001 | 0.0003 | 0.0017 | 0.0059 |
|  Dense Floyd-Warshall |  0.0014 | 0.0195 | 0.2829 | 4.2183 |
+-----------------------+---------+--------+--------+--------+
```

CONCLUSION

In conclusion, both Dijkstra's algorithm and Floyd-Warshall algorithm are important tools for solving shortest path problems in graphs.

Firstly, Dijkstra's algorithm is best suited for finding the shortest path from a single source vertex to all other vertices in a weighted graph. It can be implemented using a priority queue data structure and has a time complexity of O(E log V) or O(E + V log V), depending on the data structure used. Dijkstra's algorithm is often faster than the worst-case time complexity and is especially efficient for sparse graphs or when a good heuristic is used.

Secondly, Floyd-Warshall algorithm, on the other hand, is designed to find the shortest path between every pair of vertices in a weighted graph. It uses dynamic programming and has a time complexity of O(V^3), which can be slow for large graphs. However, it is practical for small to medium-sized dense graphs and is a simple and intuitive algorithm to understand.

Lastly, the choice of algorithm depends on the specific problem and the characteristics of the graph. For small graphs or when a single source vertex is given, Dijkstra's algorithm may be more efficient. For larger graphs or when all pairs of vertices are of interest, Floyd-Warshall algorithm may be a better choice, especially if the graph is dense. Other algorithms, such as A* or Bellman-Ford, may also be suitable for specific scenarios.

REFERENCES

AA-Labs/Lab5 at main · Syn4z/AA-Labs (github.com)