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

REPORT

Laboratory work no.5
Shortest-path algorithms

Elaborated:
st. gr. FAF-213 Gutu Dinu

Verified:
asist. univ. Fiștic Cristofor

Chişinău – 2023

Table of Contents

ALGORITHM ANALYSIS

Objective

1. Analysis and study of the algorithms.
2. Empirical analysis of the aforementioned 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

Empirical analysis is an alternative to mathematical analysis in evaluating algorithm complexity. It can be used to gain insight into the complexity class of an algorithm, compare the efficiency of different algorithms for solving similar problems, evaluate the performance of different implementations of the same algorithm, or examine the efficiency of an algorithm on a specific computer. The typical steps in empirical analysis include defining the purpose of the analysis, choosing a metric for efficiency such as number of operations or execution time, determining the properties of the input data, implementing the algorithm in a programming language, generating test data, running the program on the test data, and analyzing the results. The choice of efficiency metric depends on the purpose of the analysis, with number of operations being appropriate for complexity class evaluation and execution time being more relevant for implementation performance. After the program is run, the results are recorded and synthesized through calculations of statistics or by plotting a graph of problem size against efficiency measure.

Introduction

Dijkstra's algorithm and Floyd-Warshall algorithm are two popular algorithms used in graph theory to solve the shortest path problem. These algorithms have a wide range of applications in areas such as transportation networks, computer networks, and maps.

Dijkstra's algorithm is a single-source shortest path algorithm that finds the shortest path between a source node and all other nodes in a weighted graph. It works by maintaining a set of visited nodes and unvisited nodes, and iteratively selecting the unvisited node with the smallest tentative distance. For each visited node, the algorithm examines all of its neighboring nodes and calculates a tentative distance to each. If the tentative distance to a neighboring node is smaller than its current tentative distance, the algorithm updates the tentative distance and the predecessor of the neighboring node. This process is repeated until all nodes have been visited or the destination node has been marked as visited.

Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm that finds the shortest path between all pairs of nodes in a weighted graph. It works by maintaining a matrix of distances between all pairs of nodes in the graph. Initially, this matrix is filled with the weight of the edges between the nodes, or infinity if there is no edge. Then, the algorithm iterates over all nodes in the graph and considers each node as a potential intermediate node in the shortest path between every pair of nodes. For each pair of nodes, the algorithm considers whether the shortest path between them goes through the current intermediate node. If it does, then the algorithm updates the distance between them to be the sum of the distance between them through the intermediate node. The algorithm repeats this process for every possible intermediate node.

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, the algorithms will receive two types of graphs: sparse and dense. Incrementaly going from smaller graphs to bigger ones, having nodes ranging from 10 to 100. For each graph both algorithms will search a number of nodes, given a specific starting node, for smaller graphs the search will resume to only 2 nodes for the rest to 5.

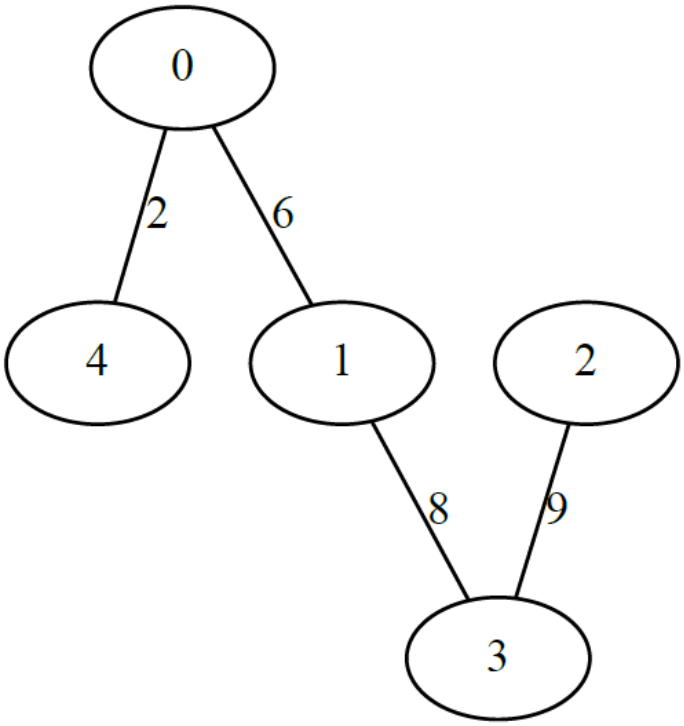

Figure 1. Sparse graph

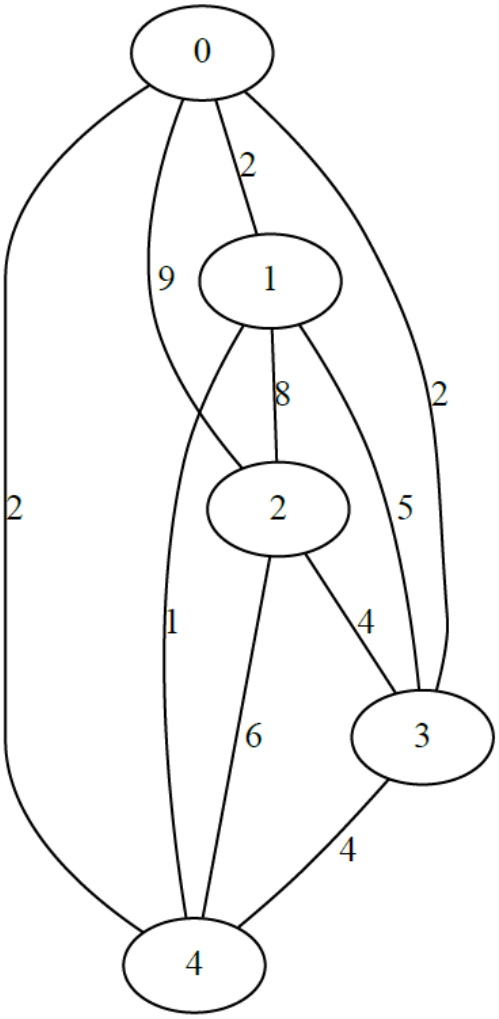

Figure 2. Dense graph

```
Graph: 10
Sparse: 2-5 | 4-8 | Dense: 2-5 | 4-8 |

Graph: 25
Sparse: 1-5 | 2-6 | 3-14 | 8-17 | 9-17 | Dense: 1-5 | 2-6 | 3-14 | 8-17 | 9-17 |

Graph: 50
Sparse: 2-4 | 4-9 | 3-14 | 8-18 | 10-17 | Dense: 2-4 | 4-9 | 3-14 | 8-18 | 10-17 |

Graph: 100
Sparse: 1-5 | 3-6 | 4-13 | 8-19 | 5-22 | Dense: 1-5 | 3-6 | 4-13 | 8-19 | 5-22 |
```

Figure 3. Searched nodes

The first numbers specifies the starting node while the second is the desired node

IMPLEMENTATION

Dijkstra's algorithm

Dijkstra's algorithm is a shortest-path algorithm that finds the shortest path between a source node and all other nodes in a weighted graph. It is named after its creator, Dutch computer scientist Edsger W. Dijkstra. The algorithm maintains a set of visited nodes and a set of unvisited nodes. It starts by marking the source node as visited and all other nodes as unvisited and assigning a tentative distance to each node. Initially, the tentative distance for the source node is set to 0, and the tentative distance for all other nodes is set to infinity.

After that, the algorithm iteratively selects the unvisited node with the smallest tentative distance and marks it as visited. For each visited node, the algorithm examines all of its neighboring nodes that are still unvisited and calculates a tentative distance to each. This tentative distance is the sum of the distance from the source node to the current node, and the weight of the edge connecting the current node to its neighbor.

If the tentative distance to a neighboring node is smaller than its current tentative distance, the algorithm updates the tentative distance and the predecessor of the neighboring node. This step is repeated until all nodes have been visited or the destination node has been marked as visited.

Algorithm Description:

Pseudocode for Dijkstra's algorithm:

```
function Dijkstra(Graph, source):

    for each vertex v in Graph.Vertices:
        dist[v] ← INFINITY
        prev[v] ← UNDEFINED
        add v to Q
    dist[source] ← 0

    while Q is not empty:
       u ← vertex in Q with min dist[u]
       remove u from Q

       for each neighbor v of u still in Q:
           alt ← dist[u] + Graph.Edges(u, v)
           if alt < dist[v]:
               dist[v] ← alt
               prev[v] ← u

    return dist[], prev[]
```

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  # If target is not reachable from start
```

Figure 4. Dijkstra's algorithm

Results

Graphs	Sparse graph	Dense graph
Dijkstra 10 nodes	[2.1e-05, 9e-06]	[1.5e-05, 1.3e-05]

Graphs	Sparse graph	Dense graph
Dijkstra 25 nodes	[1.3e-05, 3e-05, 8e-06, 2e-05, 1.3e-05]	[4.9e-05, 4.9e-05, 3.4e-05, 6.2e-05, 4.4e-05]

Graphs	Sparse graph	Dense graph
Dijkstra 50 nodes	[3.3e-05, 2.6e-05, 2.9e-05, 3.1e-05, 4.5e-05]	[0.000133, 0.000191, 0.000167, 8.3e-05, 9.1e-05]

Graphs	Sparse graph	Dense graph
Dijkstra 100 nodes	[3.1e-05, 3.3e-05, 4.8e-05, 3.4e-05, 4.7e-05]	[0.000631, 0.000475, 0.000179, 0.000208, 0.000224]

Figure 5. Dijkstra results cases

As can be seen from the results Dijkstra's algorithm has a time complexity of O(E + VlogV), where E is the number of edges and V is the number of vertices in the graph. The algorithm works by maintaining a priority queue of vertices, where the priority of a vertex is its tentative distance from the source node. The time complexity of the algorithm is dominated by the time spent in the priority queue operations, which are O(logV) for insertion and deletion and O(1) for finding the minimum. Therefore, the total time complexity of the algorithm is O(ElogV) for the priority queue operations, plus O(V) for initialization and updating of the tentative distances In the worst case, when the graph is dense and all vertices are reachable from the source node, E can be as large as V^2, leading to a time complexity of O(V^2logV). However, in practice, the time complexity is usually much lower, especially when the graph is sparse. Thus, concluding from the results we can say that indeed perfoming on a sparse graph is more efficient, since the dijkstra algorithm has to account for only a handful of edges.

Graphs

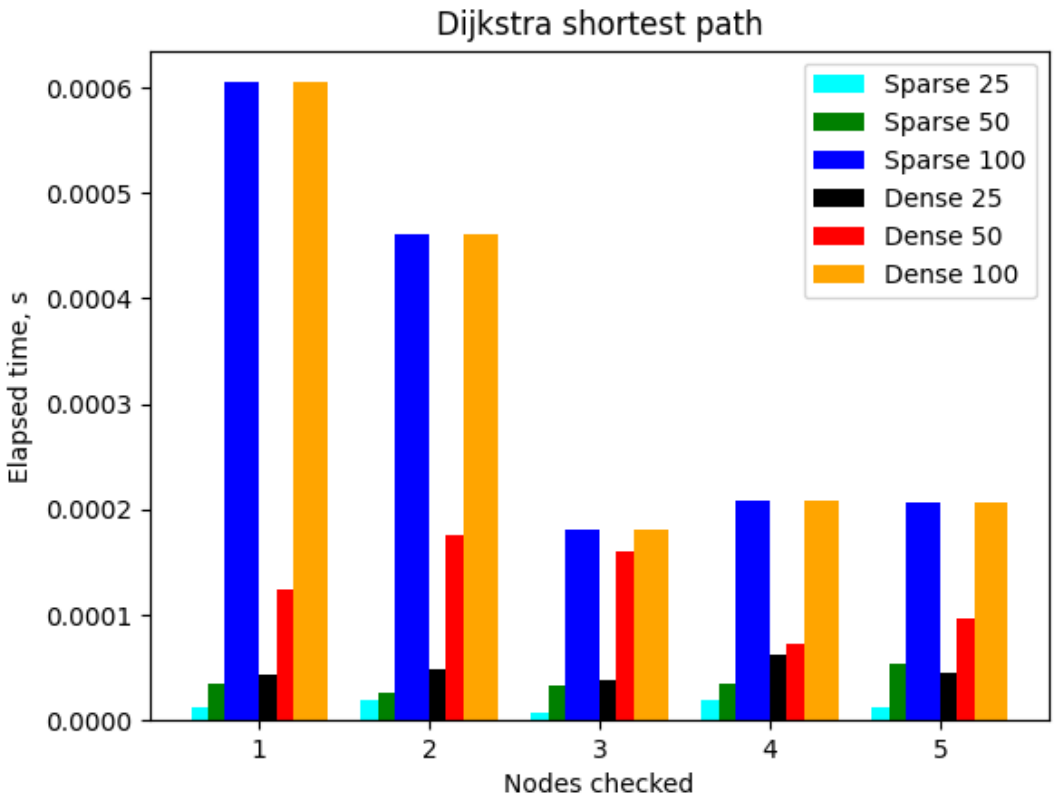

Figure 6. Dijkstra search result

Floyd-Warshall algorithm

The Floyd-Warshall algorithm is a dynamic programming algorithm for finding the shortest path between all pairs of nodes in a weighted graph. It is named after its inventors, Robert Floyd and Stephen Warshall.The algorithm works by maintaining a matrix of distances between all pairs of nodes in the graph. Initially, this matrix is filled with the weight of the edges between the nodes, or infinity if there is no edge.

Then, the algorithm iterates over all nodes in the graph and considers each node as a potential intermediate node in the shortest path between every pair of nodes. For each pair of nodes i and j, the algorithm considers whether the shortest path between i and j goes through node k. If it does, then the algorithm updates the distance between i and j to be the sum of the distance between i and k and the distance between k and j.

Algorithm Description:

Pseudocode for Floyd Warshall's algorithm:

```
let dist be a |V| × |V| array of minimum distances initialized
to ∞ (infinity)
for each edge (u, v) do
```

```
    dist[u][v] ← w(u, v)  // The weight of the edge (u, v)
for each vertex v do
    dist[v][v] ← 0
for k from 1 to |V|
    for i from 1 to |V|
        for j from 1 to |V|
            if dist[i][j] > dist[i][k] + dist[k][j]
                dist[i][j] ← dist[i][k] + dist[k][j]
            end if
```

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]
```

Figure 7. Floyd-Warshall's algorithm

Results

Graphs	Sparse graph	Dense graph
Floyd 10 nodes	[0.000198, 0.000189]	[0.000144, 0.000143]

Graphs	Sparse graph	Dense graph
Floyd 25 nodes	[0.002548, 0.002386, 0.002372, 0.002378, 0.002373]	[0.001929, 0.001916, 0.001921, 0.001924, 0.001926]

Graphs	Sparse graph	Dense graph
Floyd 50 nodes	[0.017497, 0.017059, 0.017155, 0.017631, 0.016999]	[0.014858, 0.014744, 0.014697, 0.014739, 0.014665]

Graphs	Sparse graph	Dense graph
Floyd 100 nodes	[0.134165, 0.133325, 0.138362, 0.142961, 0.138587]	[0.118011, 0.126537, 0.125171, 0.123749, 0.12079]

Figure 8. Floyd-Warshall results cases

Floyd-Warshall algorithm has a time complexity of O(V^3), where V is the number of vertices in the graph. The algorithm works by maintaining a matrix of distances between all pairs of vertices in the graph. The time complexity of the algorithm is dominated by the three nested loops used to iterate over all vertices and intermediate vertices, leading to a time complexity of O(V^3). However, the space complexity of the algorithm is also O(V^2) for the matrix of distances. In practice, the Floyd-Warshall algorithm is suitable for small to medium-sized graphs, as its time complexity grows rapidly with the number of vertices. For very large graphs, more specialized algorithms, such as Johnson's algorithm, may be more efficient. However, Floyd-Warshall algorithm is still useful for its ability to handle negative edge weights, which many other algorithms cannot do. Thus we can see from the results that there is not a significant difference between a sparse and dense graph.

Graphs

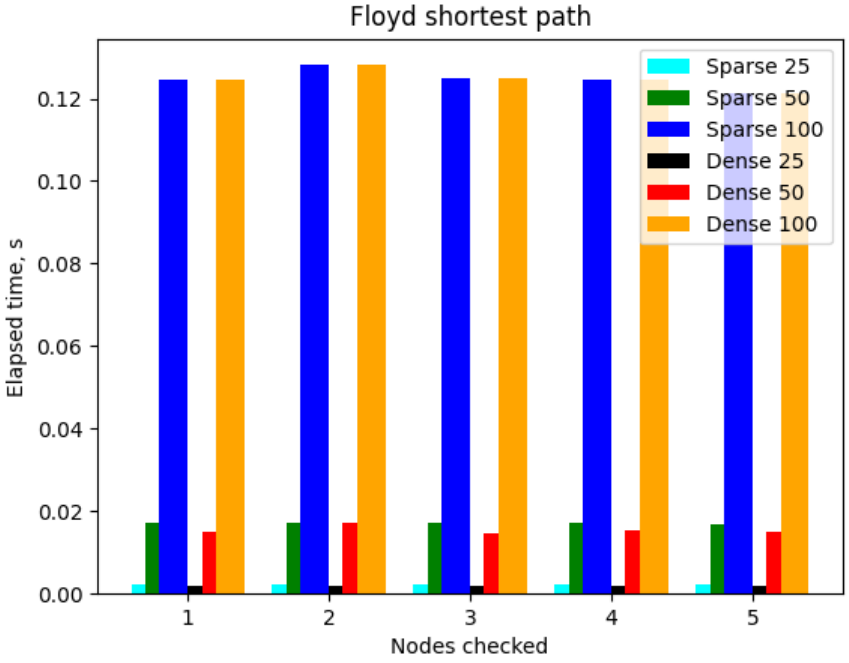

Figure 9. Floyd search results

All Algorithms

All results

Graphs	Sparse graph	Dense graph
Dijkstra 10 nodes	[2.1e-05, 9e-06]	[1.5e-05, 1.3e-05]
Floyd 10 nodes	[0.000198, 0.000189]	[0.000144, 0.000143]
Dijkstra 25 nodes	[1.3e-05, 3e-05, 8e-06, 2e-05, 1.3e-05]	[4.9e-05, 4.9e-05, 3.4e-05, 6.2e-05, 4.4e-05]
Floyd 25 nodes	[0.002548, 0.002386, 0.002372, 0.002378, 0.002373]	[0.001929, 0.001916, 0.001921, 0.001924, 0.001926]
Dijkstra 50 nodes	[3.3e-05, 2.6e-05, 2.9e-05, 3.1e-05, 4.5e-05]	[0.000133, 0.000191, 0.000167, 8.3e-05, 9.1e-05]
Floyd 50 nodes	[0.017497, 0.017059, 0.017155, 0.017631, 0.016999]	[0.014858, 0.014744, 0.014697, 0.014739, 0.014665]
Dijkstra 100 nodes	[3.1e-05, 3.3e-05, 4.8e-05, 3.4e-05, 4.7e-05]	[0.000631, 0.000475, 0.000179, 0.000208, 0.000224]
Floyd 100 nodes	[0.134165, 0.133325, 0.138362, 0.142961, 0.138587]	[0.118011, 0.126537, 0.125171, 0.123749, 0.12079]

Figure 10. All algortihms' results

All graphs

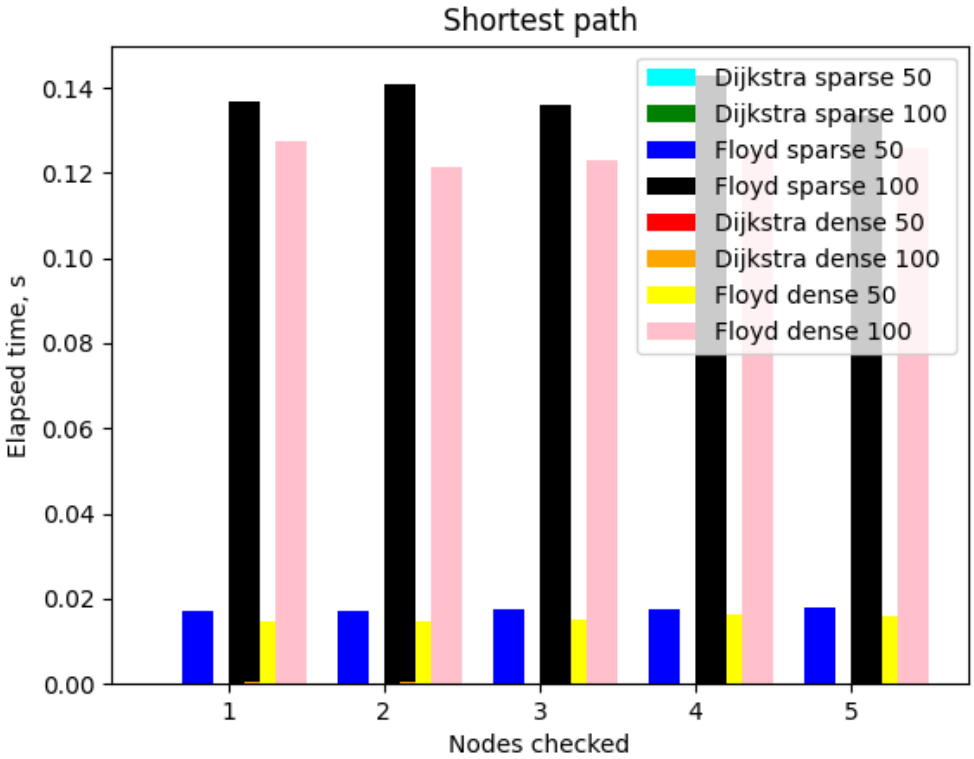

Figure 11. All algortihms' graphs

It is worth noting that the difference is so acute that the time for dijkstra's search is too small to be visible.

CONCLUSION

In conclusion, both Dijkstra's algorithm and Floyd-Warshall algorithm are powerful tools for solving the shortest path problem in weighted graphs. However, their performance and suitability for different scenarios vary.

Dijkstra's algorithm is a single-source shortest path algorithm that is very efficient for finding the shortest path between a single source node and all other nodes in the graph. It has a time complexity of O(E + VlogV), where E is the number of edges and V is the number of vertices. However, it cannot handle negative edge weights, which limits its applicability in certain scenarios.

Floyd-Warshall algorithm, on the other hand, is an all-pairs shortest path algorithm that is very efficient for finding the shortest path between all pairs of nodes in a graph. It has a time complexity of O(V^3), where V is the number of vertices, and can handle negative edge weights. However, it is not as efficient as Dijkstra's algorithm for finding the shortest path between a single pair of nodes.

Therefore, the choice between Dijkstra's algorithm and Floyd-Warshall algorithm depends on the specific requirements of the problem at hand. If the goal is to find the shortest path between a single source node and all other nodes in the graph, Dijkstra's algorithm is a better choice. If the goal is to find the shortest path between all pairs of nodes in the graph, or if negative edge weights are present, Floyd-Warshall algorithm is a better choice.

BIBLIOGRAPHY

Github: https://github.com/Grena30/APA_Labs/tree/main/Lab5