Graph Algorithms

Project 1
Algorithm Design and Analysis

Danil Krassotov 282656

Introduction

The purpose of this report is to analyze the efficiency of graph algorithms for graphs represented in two variants - adjacency matrix and adjacency list. As part of the work, Dijkstra's algorithm, Bellman-Ford algorithm and depth-first search (DFS) algorithm were implemented. For each algorithm and each combination of input parameters, 100 measurements were performed with random data. Based on the results, graphs showing the complexity of algorithms and comparing their efficiency were generated.

1 Graph Representation

Two graph representations were used in the project: adjacency matrix and adjacency lists. Both implementations inherit from a common GraphArray interface (Listing 1), which allows easy comparison of their operation and performance.

```
class GraphArray {
    public:
        virtual void addEdge(int u, int v, int weight = 1) = 0;
        virtual bool isConnected(int u, int v) const = 0;
        virtual void print() const = 0;
        virtual std::vector<std::pair<int, int>> getNeighbors(int
            u) const = 0;
        virtual void clear() = 0;
        virtual ~GraphArray() {}
};
```

Listing 1: GraphArray Interface - C++ Implementation

The adjacency matrix was implemented as a one-dimensional vector, where the element at position $u * size + v$ corresponds to the edge from vertex u to v.
On the other hand, adjacency lists are based on a vector of vectors of pairs (vertex, weight), where each list stores the neighbors of a given vertex.
This approach allows easy addition of edges, checking connections, obtaining neighbors, and clearing the graph structure.

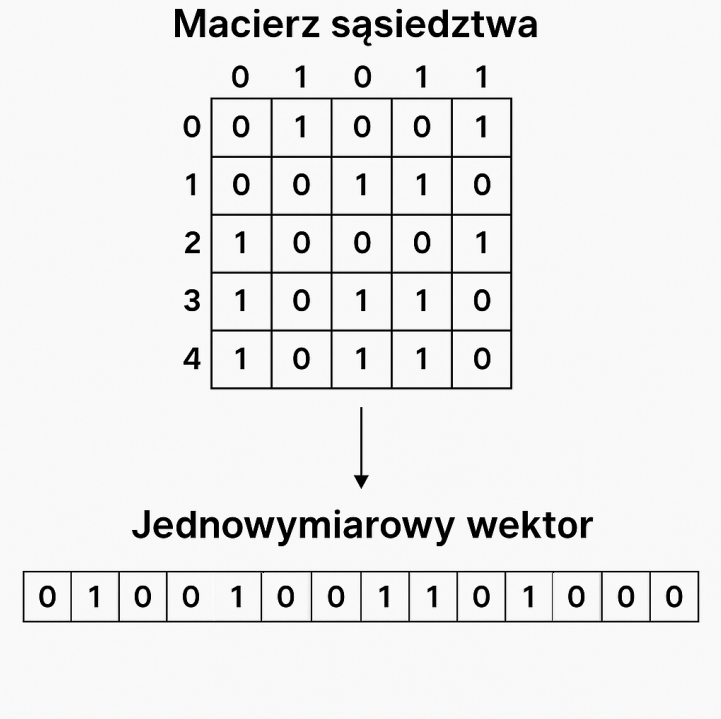

Figure 1: Visualization of adjacency matrix

2 Algorithm Description

2.1 Dijkstra's Algorithm

Dijkstra's algorithm was used to find the shortest paths in a graph without negative weights. The implementation uses a priority queue $std :: priority\_queue$, which allows efficient selection of the vertex with the smallest currently known distance. The algorithm maintains two vectors: $dist$, containing minimum distances from the starting vertex to the others, and $prev$, enabling later path reconstruction.

Computational Complexity:

- $O((V + E)\ logV)$ – for adjacency list. Most efficient for sparse graphs.
- $O(V^2\ logV)$ – for adjacency matrix with priority queue (in this project: 1D vector instead of 2D, which slightly reduces memory overhead but doesn't significantly affect asymptotic complexity)

Advantages:

- Fast and efficient with positive edge weights
- Intuitive implementation
- Works well with adjacency-based structures

Disadvantages:

- Does not work correctly for edges with negative weights
- For dense graphs (and matrices) may be less efficient than other approaches

Applications:

- Navigation systems (e.g., GPS, logistics)
- Computer networks (finding routing paths)
- Artificial intelligence in games (pathfinding)

```cpp
std::pair<std::vector<int>, std::vector<int>> dijkstra(const
   GraphArray& graph, int start, int vertexCount) {
    std::vector<int> dist(vertexCount, INT_MAX);
    std::vector<int> prev(vertexCount, -1);
    dist[start] = 0;

    using kp = std::pair<int, int>; // {distance, vertex}
    std::priority_queue<kp, std::vector<kp>, std::greater<>> pq;
    pq.emplace(0, start);

    while (!pq.empty()) {
        auto [currentDist, u] = pq.top();
        pq.pop();

        if (currentDist > dist[u]) continue;

        for (const auto& [v, weight] : graph.getNeighbors(u)) {
            if (dist[u] + weight < dist[v]) {
                dist[v] = dist[u] + weight;
                prev[v] = u;
                pq.emplace(dist[v], v);
            }
        }
    }

    return {dist, prev};
}
```

Listing 2: Dijkstra's Algorithm - C++ Implementation

Visualization:

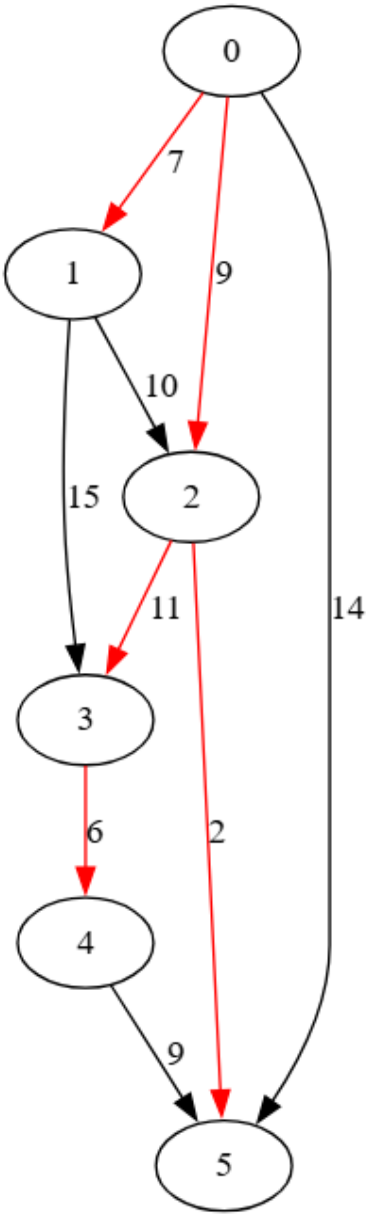

Figure 2: Visualization of **Dijkstra's** algorithm

2.2 Bellman-Ford Algorithm

Bellman-Ford is an algorithm for finding shortest paths from a starting vertex, also with negative weights. This implementation uses an approach with $V - 1$ iterations of relaxation of all edges and an additional pass to detect negative cycles.

```cpp
std::pair<std::vector<int>, std::vector<int>> bellmanFord(const GraphArray& graph, int start, int vertexCount) {
    std::vector<int> dist(vertexCount, INT_MAX);
    std::vector<int> prev(vertexCount, -1);
    dist[start] = 0;

    for (int relaxation = 0; relaxation < vertexCount - 1; ++relaxation) {
        for (int u = 0; u < vertexCount; ++u) {
            for (const auto& [v, weight] : graph.getNeighbors(u)) {
                if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                    dist[v] = dist[u] + weight;
                    prev[v] = u;
                }
            }
        }
    }

    for (int u = 0; u < vertexCount; ++u) {
        for (const auto& [v, weight] : graph.getNeighbors(u)) {
            if (dist[u] != INT_MAX && dist[u] + weight < dist[v]) {
                throw std::runtime_error("Graph contains negative cycle");
            }
        }
    }

    return {dist, prev};
}
```

Listing 3: Bellman-Ford Algorithm - C++ Implementation

Computational Complexity:

- $O(V \cdot E)$ – regardless of graph representation
- For dense graphs (matrix): $O(V^3)$
- In practice significantly slower than Dijkstra – even for small graphs

Advantages:

- Works with negative weights
- Can detect negative cycles in the graph
- Simple implementation, works well without priority queue

Disadvantages:

- Significantly slower than Dijkstra (as confirmed by tests)
- Inefficient for large graphs (execution time grows exponentially)
- For dense graphs, execution time becomes very long

Applications:

- Analysis of graphs with negative weights
- Currency market analysis
- Algorithms in optimization problems

Visualization:

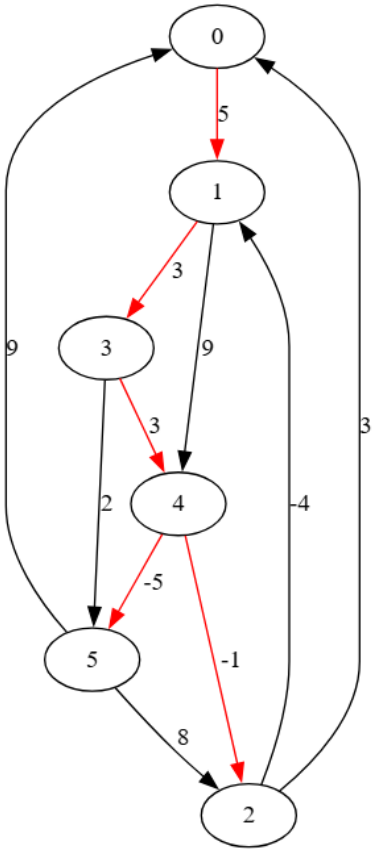

Figure 3: Visualization of **Bellman-Ford** algorithm

2.3 Depth-First Search Algorithm

DFS is a basic graph traversal algorithm that explores graphs recursively, going deep and marking visited vertices in a *visited* array. It can be used for both directed and undirected graphs.

```
    void dfs(const GraphArray& graph, int u, std::vector<bool>&
       visited) {
    visited[u] = true;
    for (const auto& [v, weight] : graph.getNeighbors(u)) {
        if (!visited[v]) {
            dfs(graph, v, visited);
        }
    }
}
```

Listing 4: DFS Algorithm - C++ Implementation

Computational Complexity:

- $O(V + E)$ – very fast and linear with respect to the number of vertices and edges
- Works instantly for both sparse and dense graphs
- Representation has little impact, but list is marginally faster

Advantages:

- Very fast and simple
- Does not require any data structures
- Works both in full graph and from a single vertex

Disadvantages:

- May lead to stack overflow for very large graphs (recursion)
- Does not consider weights

Applications:

- Graph connectivity analysis
- Cycle checking in directed graphs

3 Measurement Analysis

3.1 Comparative Charts

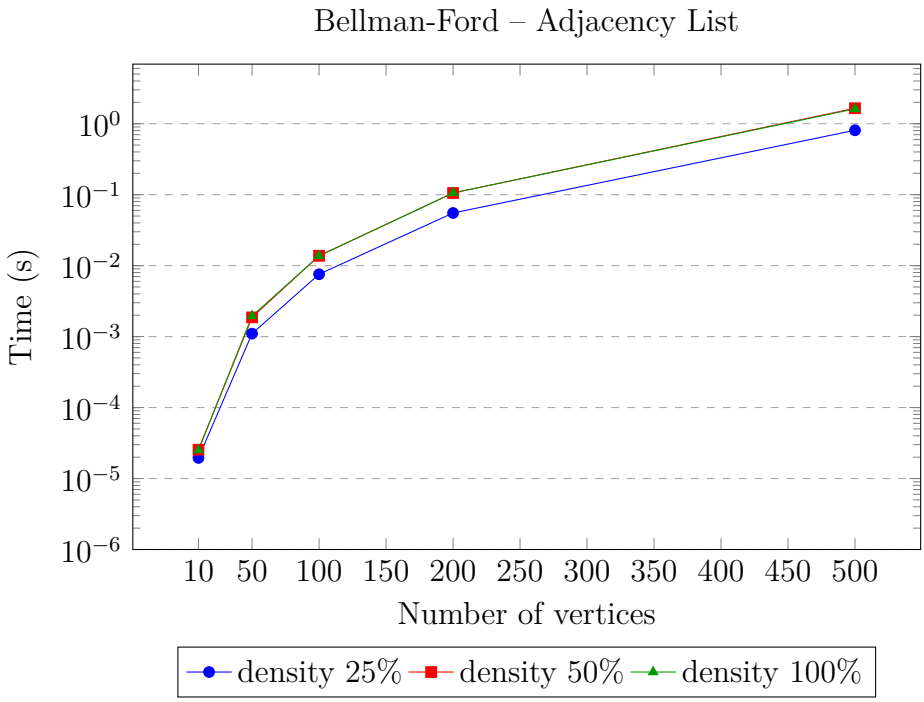

Figure 4: Execution time of Bellman-Ford algorithm on graph as adjacency list

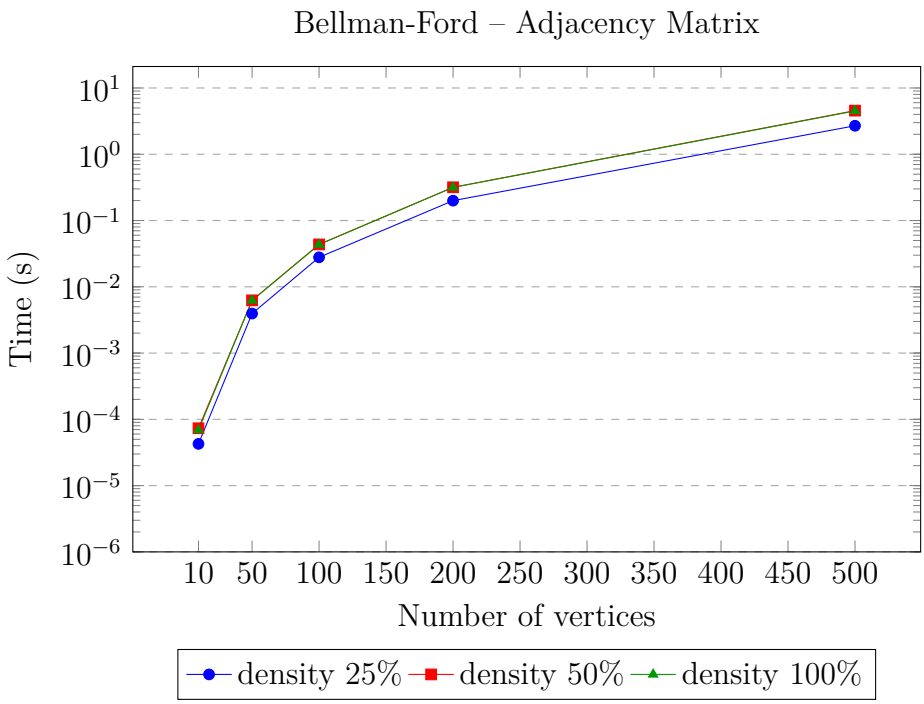

Figure 5: Execution time of Bellman-Ford algorithm on graph as adjacency matrix

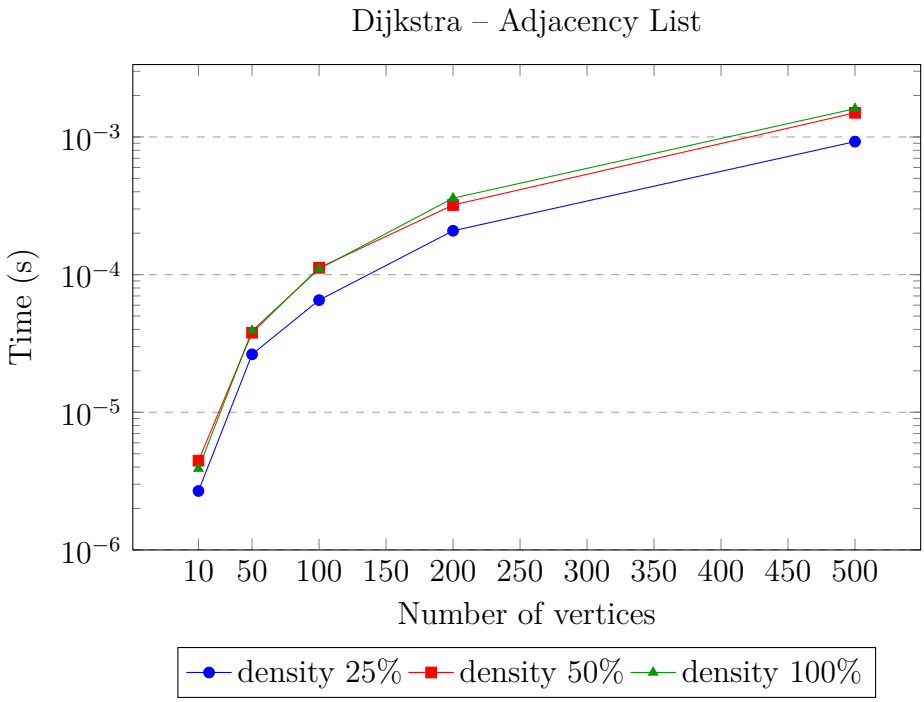

Figure 6: Execution time of Dijkstra's algorithm on graph as adjacency list

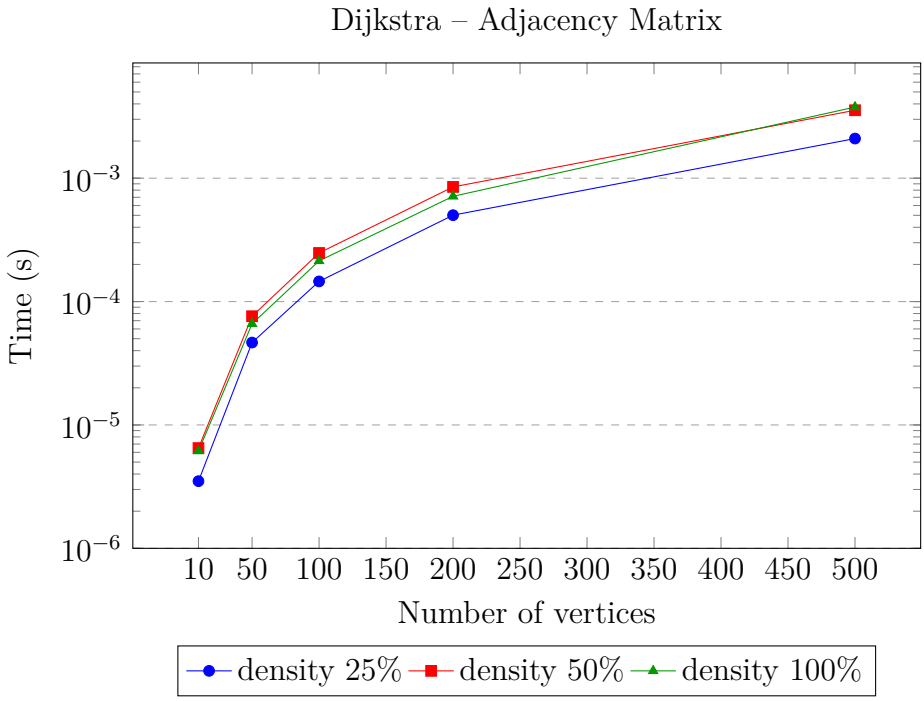

Figure 7: Execution time of Dijkstra's algorithm on graph as adjacency matrix

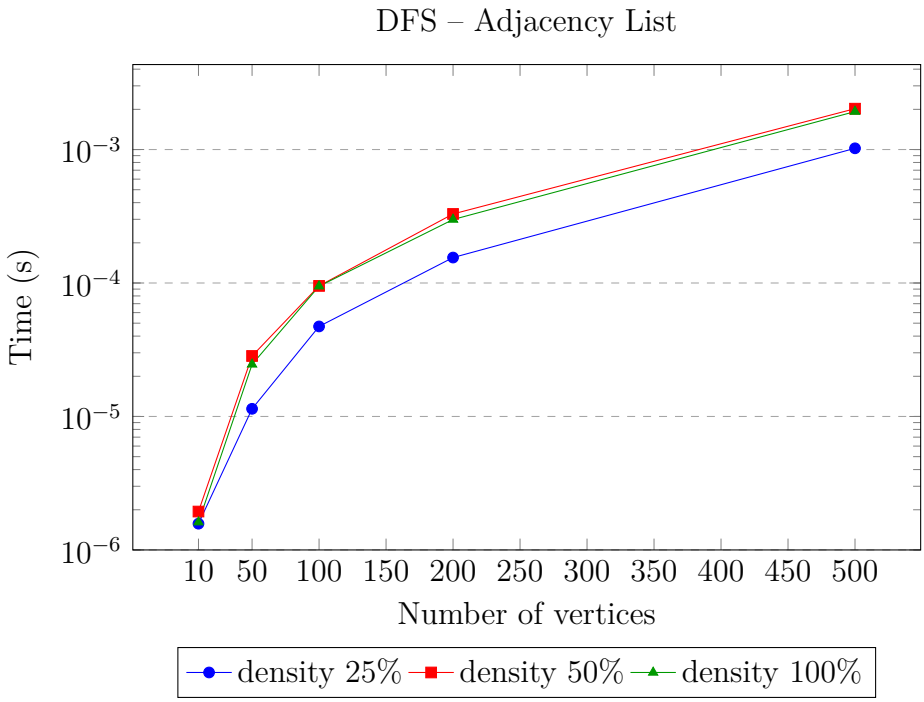

Figure 8: Execution time of DFS algorithm on graph as adjacency list

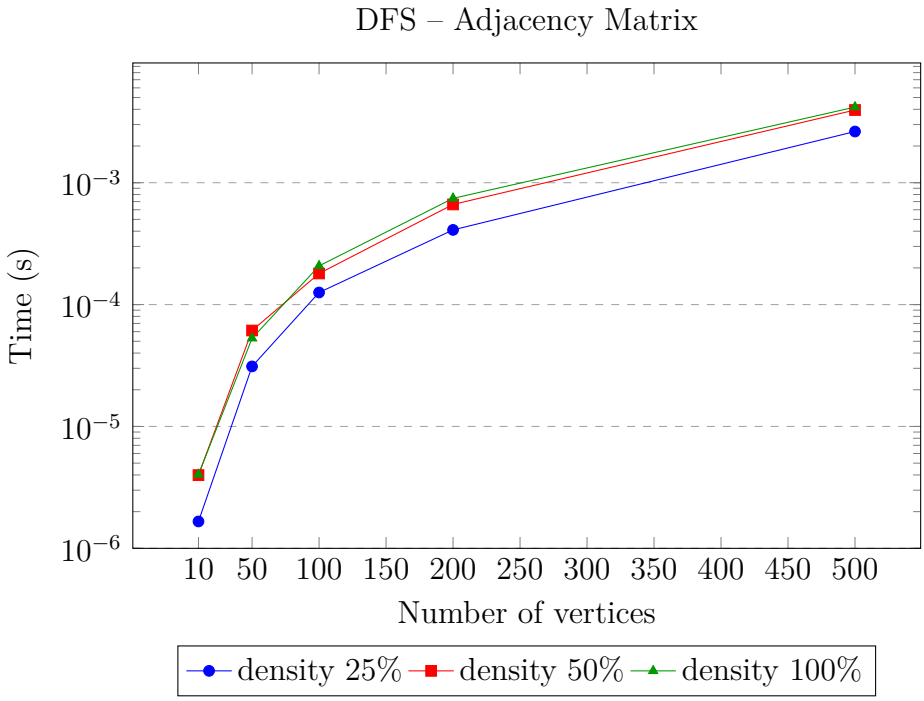

Figure 9: Execution time of DFS algorithm on graph as adjacency matrix

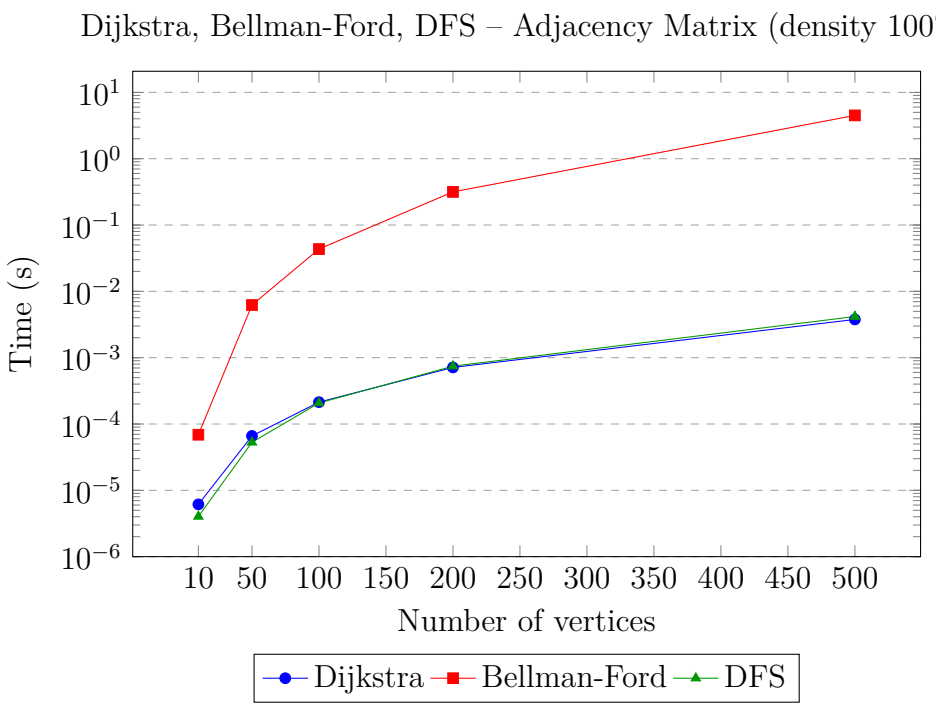

Figure 10: Comparison of all algorithms on graph as adjacency matrix

3.2 Linear Fitting

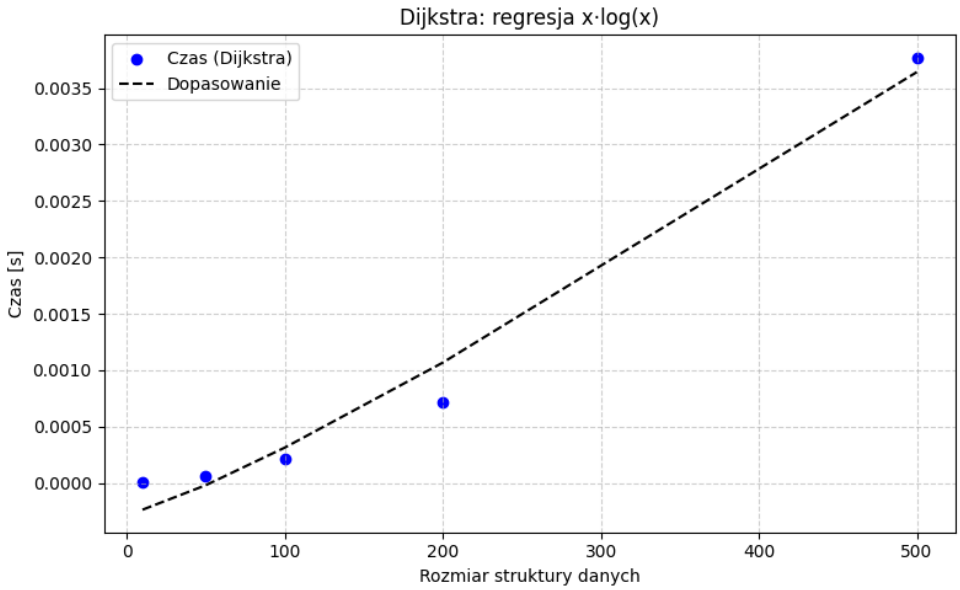

Figure 11: Linear fitting for **Dijkstra**

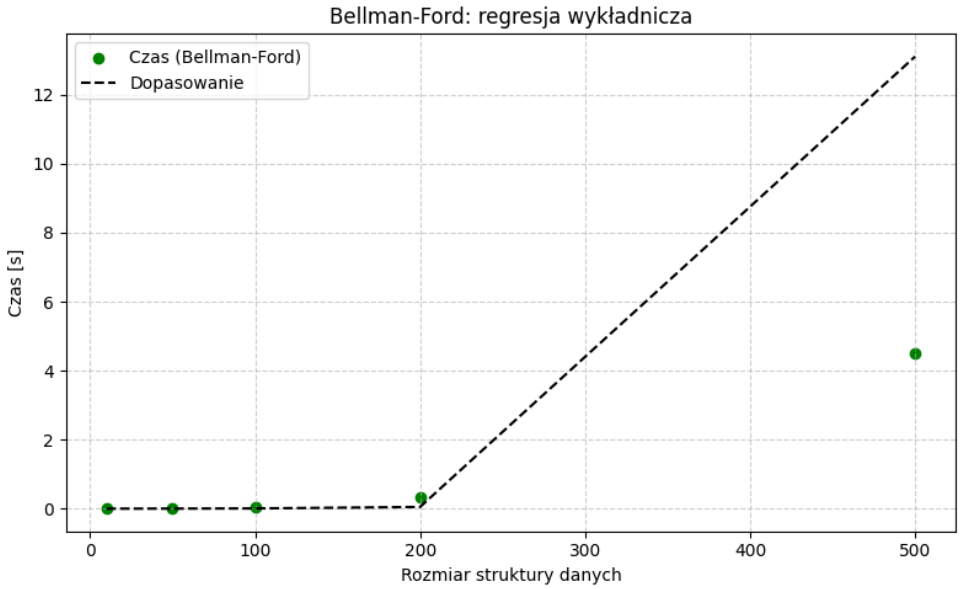

Figure 12: Exponential fitting for **Bellman-Ford**

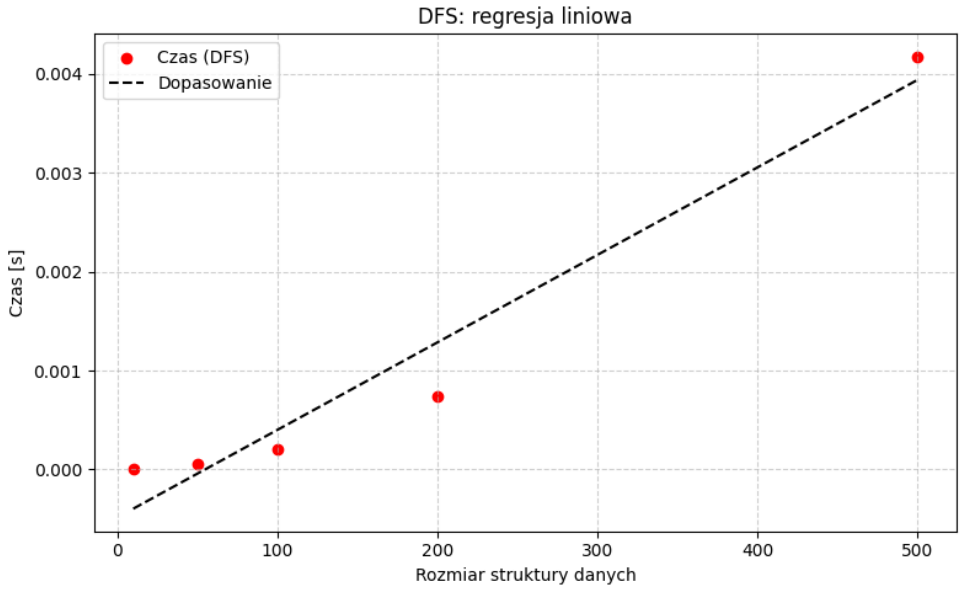

Figure 13: Linear fitting for **DFS**

Approximate Coefficients

1. **Dijkstra:**
   $y = 1.26 \times 10^{-6} \times x \log(x) - 2.65 \times 10^{-4}$

2. **Bellman-Ford:**
   $y = \exp(0.018x - 6.67)$

3. **DFS:**
   $y = 8.84x - 0.00048$

Interpretation

1. **Dijkstra - Approximation** $x \cdot log(x)$
   Execution time grows according to the $O(V \cdot logV)$ characteristic.
   The model well reflects the complexity growth for larger data sizes.

2. **Bellman-Ford - Approximation** $y = \exp(ax + b)$
   Execution time grows exponentially for large sizes, which is visible on the graph.
   Algorithm complexity is $O(V \cdot E)$ – very costly for large data.

3. **DFS:**
   Execution time grows linearly with graph size.
   Linear regression fits well.
   Complexity $O(V + E)$ makes DFS very fast even for larger graphs.

Conclusions

- **DFS** is the fastest algorithm at tested sizes — its time grows linearly. It is very efficient for graphs of small and medium density, because it does not perform unnecessary operations and quickly visits only available vertices. However, it is not suitable for finding shortest paths in weighted graphs.
- **Dijkstra** scales according to $O(n \cdot logn)$. This algorithm works well in practice for graphs without negative weights, especially when using a priority queue. It performs well even with large input data sizes and allows finding shortest paths from one source to all vertices.
- **Bellman-Ford** shows exponential time growth, making it impractical for large graphs. The advantage is handling negative weights, as well as the ability to detect negative cycles. It can be used in special cases where other algorithms fail, but its performance limits application in larger instances.