Algorytmy Sortowania

Projekt 1
Projektowanie i Analiza Algorytmów

Danil Krassotov 282656

Wstęp

Celem niniejszego sprawozdania jest analiza efektywności działania algorytmów grafowych dla grafów reprezentowanych w dwóch wariantach - macierz sąsiedztwa oraz lista sąsiedztwa. W ramach pracy zaimplementowano algorytmy Dijkstry, Bellmana-Forda i algorytm przechodzenia grafów w głąb (DFS). Dla każdego algorytmu i każdej kombinacji parametrów wejściowych wykonano 100 pomiarów z danymi losowymi. Na podstawie wyników wygenerowano wykresy pokazujące złożoności algorytmów i porównujące ich efektywność.

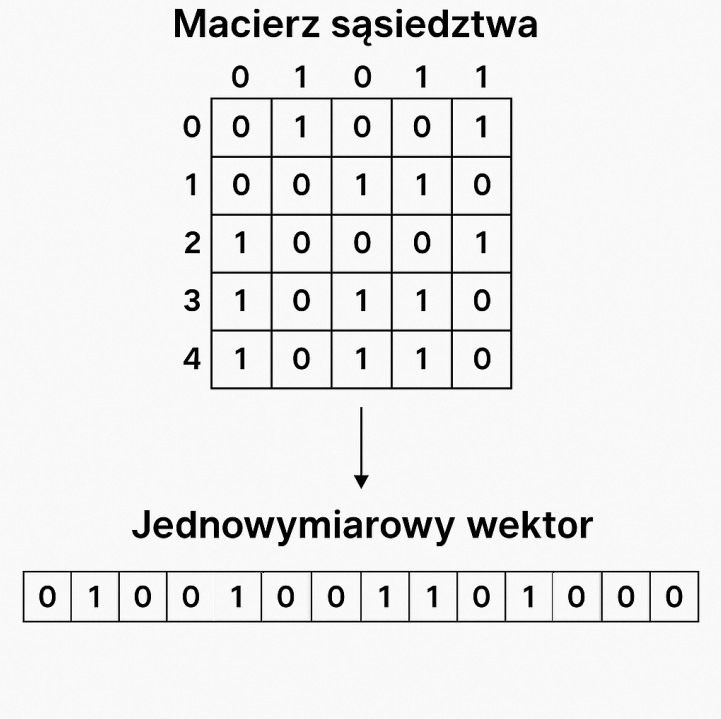
1 Reprezentacja grafów

W projekcie zastosowano dwie reprezentacje grafów: macierz sąsiedztwa oraz listy sąsiedztwa. Obie implementacje dziedziczą po wspólnym interfejsie GraphArray(Listing 1), co umożliwia łatwe porównywanie ich działania i wydajności.

```
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: Timsort - implementacja w C++

Macierz sąsiedztwa została zaimplementowana jako jednowymiarowy wektor, w którym element na pozycji $u * size + v$ odpowiada krawędzi od wierzchołka u do v.
Z kolei listy sąsiedztwa opierają się na wektorze wektorów par (wierzchołek, waga), gdzie każda lista przechowuje sąsiadów danego wierzchołka.
Takie podejście umożliwia łatwe dodawanie krawędzi, sprawdzanie połączeń, uzyskiwanie sąsiadów oraz czyszczenie struktury grafu.



Rysunek 1: Wizualizacja macierzy sąsiedztwa

2 Opis Algorytmów

2.1 Algorytm Dijkstry

Do wyznaczania najkrótszych ścieżek w grafie bez ujemnych wag zastosowano algorytm Dijkstry. W implementacji wykorzystano kolejkę priorytetową $std :: priority\_queue$, co pozwala na efektywne wybieranie wierzchołka o najmniejszym obecnie znanym dystansie. Algorytm przechowuje dwa wektory: $dist$, zawierający minimalne odległości od wierzchołka startowego do pozostałych, oraz $prev$, umożliwiający późniejsze odtworzenie ścieżki.

Złożoność obliczeniowa:

- $O((V + E)\ log V)$ – dla listy sąsiedztwa. Najbardziej efektywne dla grafów rzadkich.
- $O(V^2\ log V)$ – dla macierzy sąsiedztwa z kolejką priorytetową (w tym projekcie: wektor 1D zamiast 2D, co nieco zmniejsza narzut pamięciowy, ale nie wpływa znacząco na asymptotykę)

Zalety:

- Szybki i wydajny przy dodatnich wagach krawędzi
- Intuicyjna implementacja
- Dobrze współpracuje ze strukturami opartymi na sąsiedztwie

Wady:

- Nie działa poprawnie dla krawędzi o wagach ujemnych
- W przypadku gęstych grafów (i macierzy) może być mniej efektywny niż inne podejścia

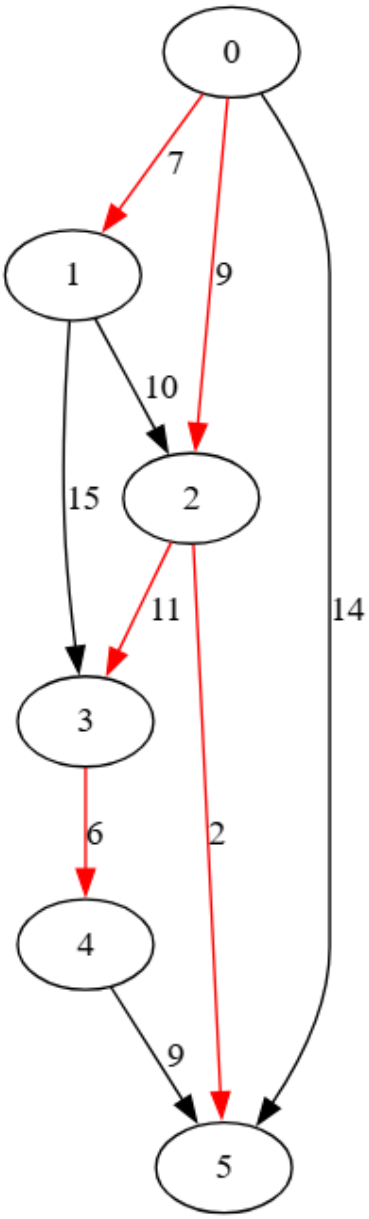
Zastosowania:

- Systemy nawigacyjne (np. GPS, logistyka)
- Sieci komputerowe (znajdowanie tras routingu)
- Sztuczna inteligencja w grach (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: Algorytm Dijkstry - implementacja w C++

Wizualizacja:



Rysunek 2: Wizualizacja algorytmu **Dijkstry**

2.2 Algorytm Bellmana-Forda

Bellman-Ford jest algorytmem służącym do znajdowania najkrótszych ścieżek z wierzchołka startowego, również z wagami ujemnymi. W tej implementacji wykorzystano podejście z $V - 1$ iteracjami relaksacji wszystkich krawędzi oraz dodatkowym przebiegiem do wykrywania cykli ujemnych.

```
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("Graf posiada cykl
                   ujemny");
            }
        }
    }

    return {dist, prev};
}
```

Listing 3: Algorytm Bellmana-Forda - implementacja w C++

Złożoność obliczeniowa:

- $O(V \cdot E)$ – niezależnie od reprezentacji grafu
- Dla grafów gęstych (macierz): $O(V^3)$
- W praktyce znacznie wolniejszy niż Dijkstra – nawet przy małych grafach

Zalety:

- Działa z wagami ujemnymi
- Może wykrywać ujemne cykle w grafie
- Prosta implementacja, działa dobrze bez kolejki priorytetowej

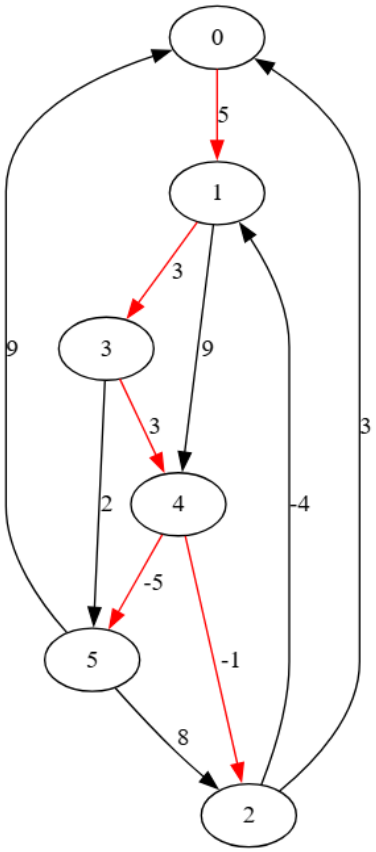
Wady:

- Znacznie wolniejszy od Dijkstry (co potwierdzają testy)
- Nieefektywny dla dużych grafów (czas wykonania rośnie wykładniczo)
- W przypadku grafów gęstych czas staje się bardzo długi

Zastosowania:

- Analiza grafów z ujemnymi wagami
- Analiza rynków walutowych
- Algorytmy w problemach optymalizacyjnych

Wizualizacja:



Rysunek 3: Wizualizacja algorytmu **Dijkstry**

2.3 Algorytm przejścia w głąb

DFS to podstawowy algorytm przeszukiwania grafów, który eksploruje je rekurencyjnie, schodząc w głąb i zaznaczając odwiedzone wierzchołki w tablicy *visited*. Może być używany zarówno do grafów skierowanych, jak i nieskierowanych.

```
    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: Algorytm DFS - implementacja w C++

Złożoność obliczeniowa:

- $O(V + E)$ – bardzo szybki i liniowy względem liczby wierzchołków i krawędzi
- Działa błyskawicznie zarówno dla rzadkich, jak i gęstych grafów
- Reprezentacja ma niewielki wpływ, ale lista jest minimalnie szybsza

Zalety:

- Bardzo szybki i prosty
- Nie wymaga żadnych struktur danych
- Działa zarówno w pełnym grafie, jak i od pojedynczego wierzchołka

Wady:

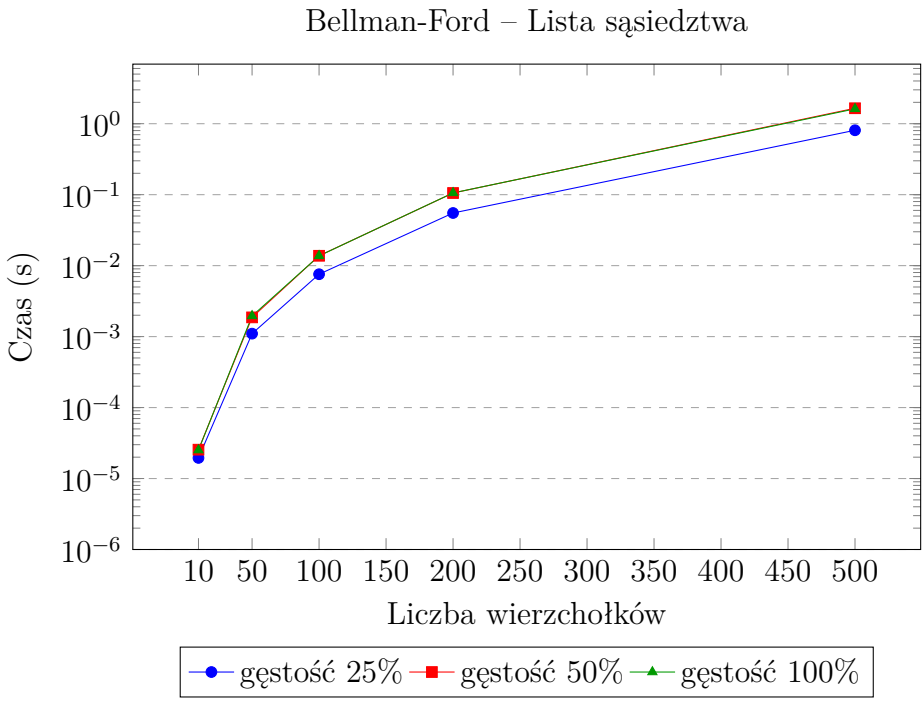
- Może prowadzić do przepełnienia stosu dla bardzo dużych grafów (rekurencja)
- Nie uwzględnia wag

Zastosowania:

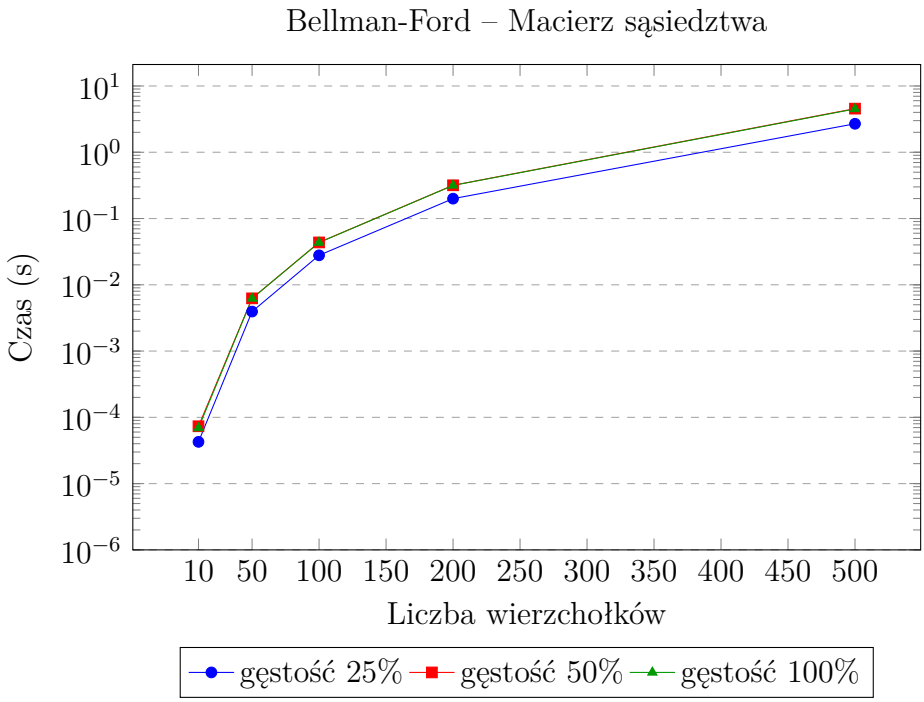
- Analiza spójności grafu
- Sprawdzanie cykli w grafach skierowanych

3 Analiza pomiarów

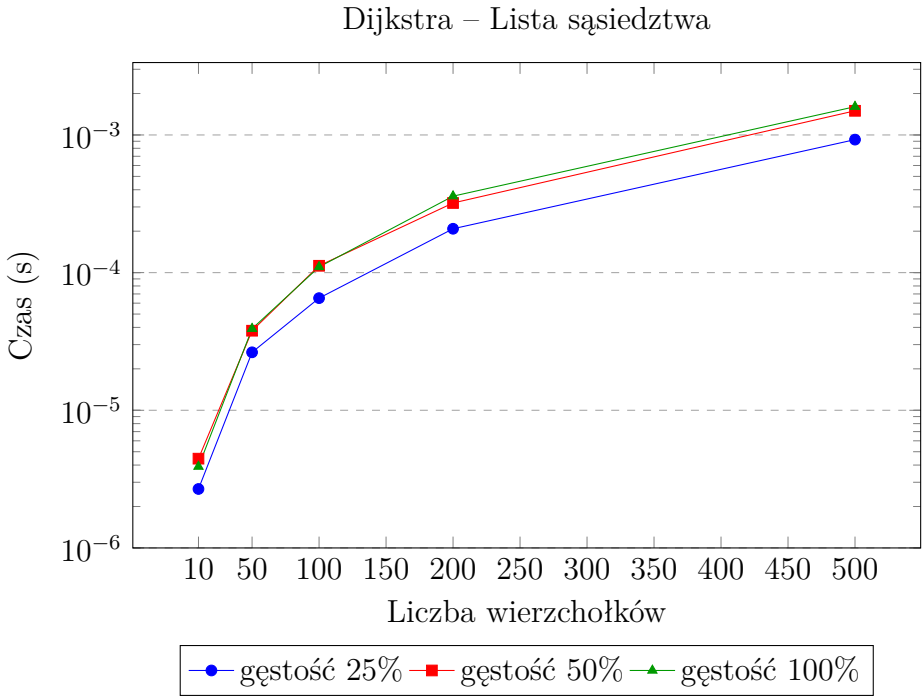
3.1 Wykresy porównawcze



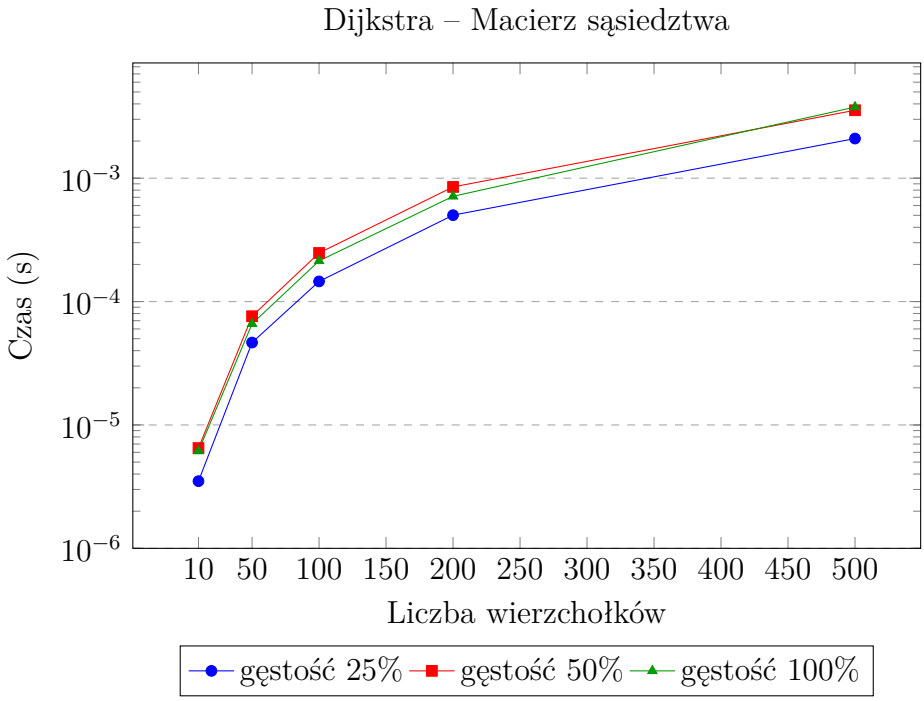
Rysunek 4: Czas działania algorytmu Bellmana-Forda na grafie jako lista sąsiedztwa



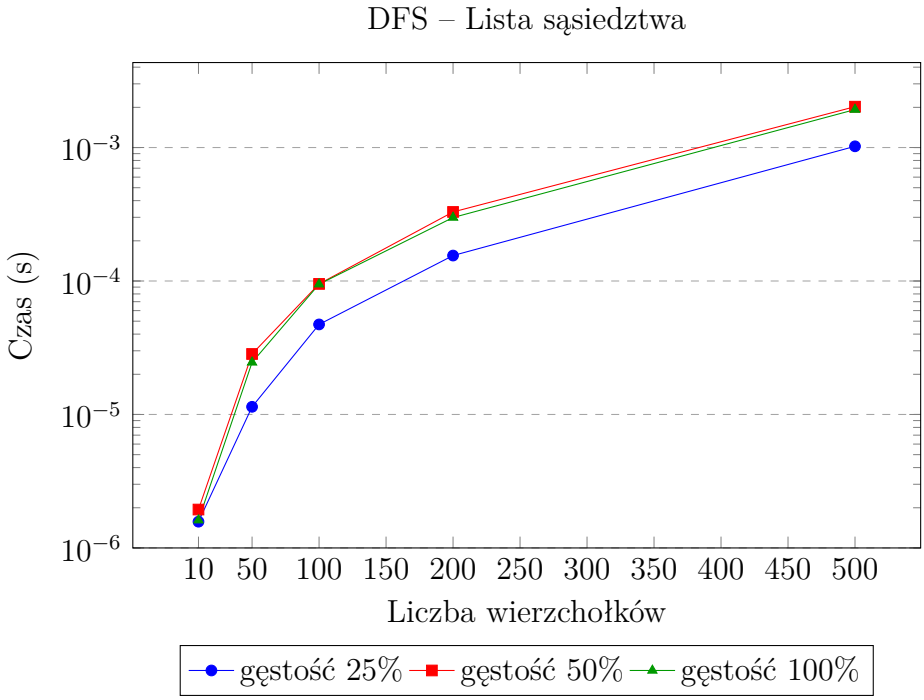
Rysunek 5: Czas działania algorytmu Bellmana-Forda na grafie jako macierz sąsiedztwa



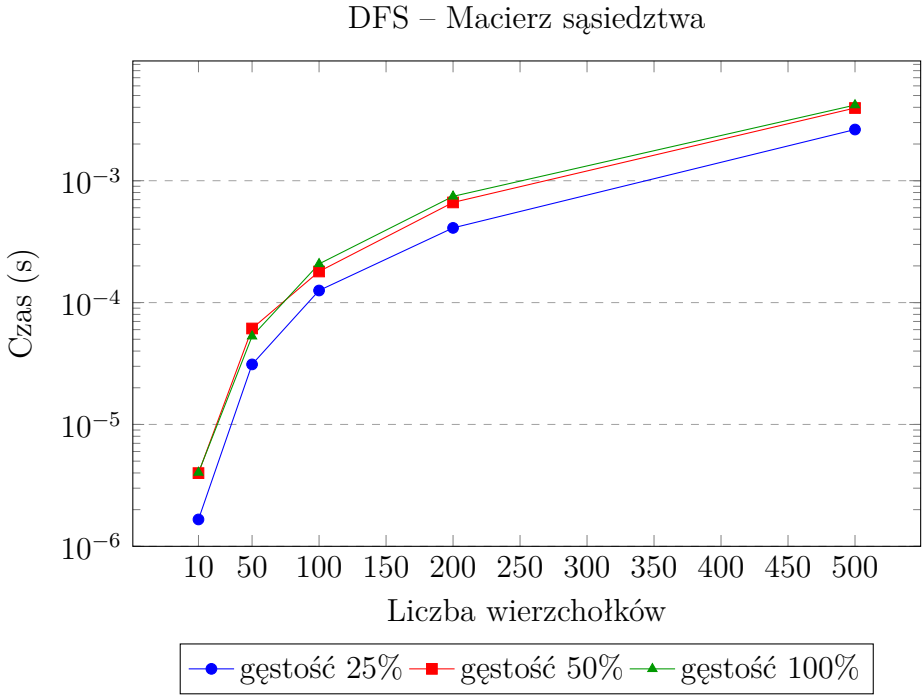
Rysunek 6: Czas działania algorytmu Dijkstry na grafie jako lista sąsiedztwa



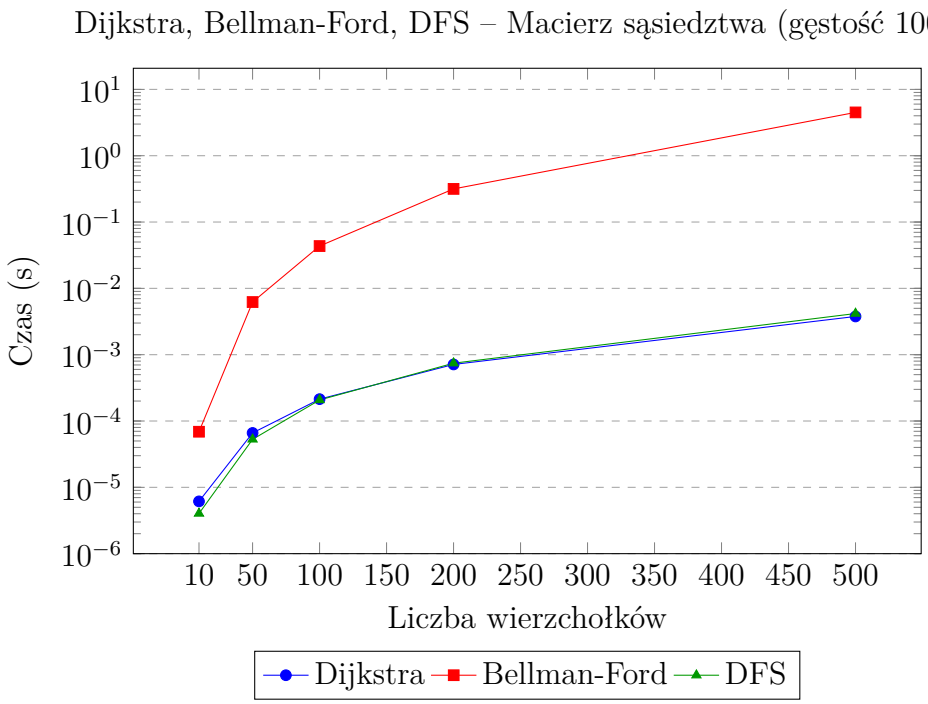
Rysunek 7: Czas działania algorytmu Dijkstry na grafie jako macierz sąsiedztwa



Rysunek 8: Czas działania algorytmu DFS na grafie jako lista sąsiedztwa

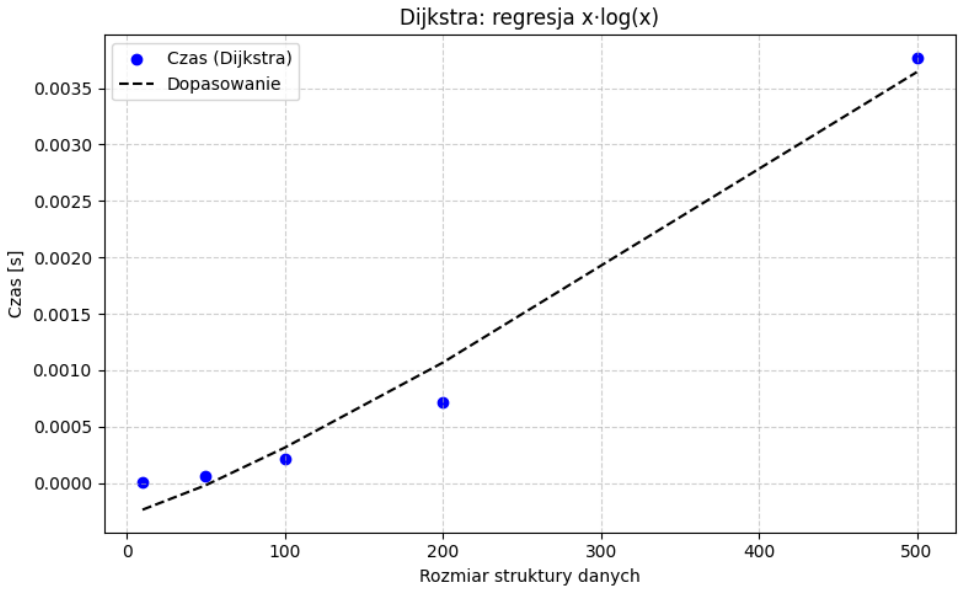


Rysunek 9: Czas działania algorytmu DFS na grafie jako macierz sąsiedztwa

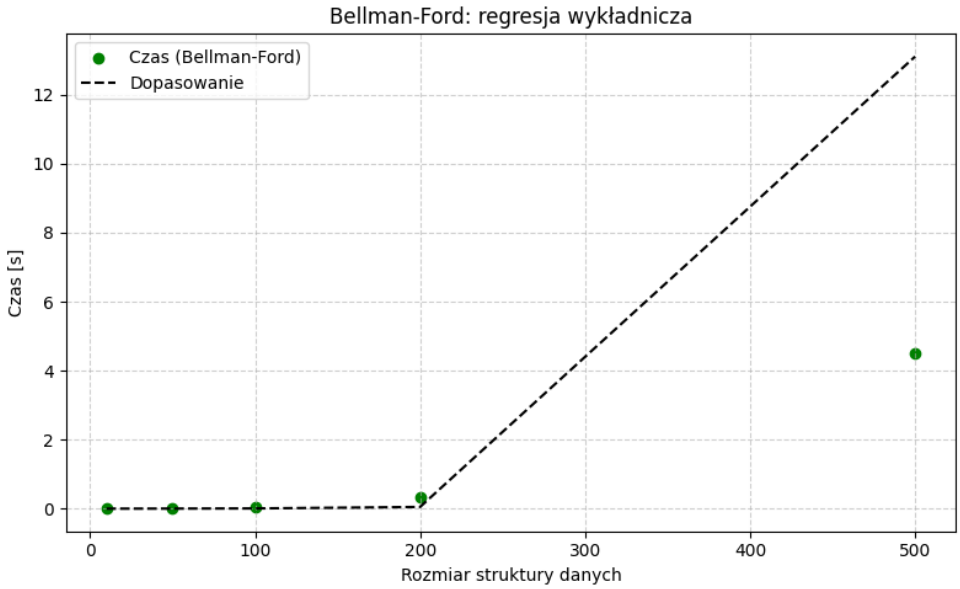


Rysunek 10: Porównanie wszystkich algorytmów na grafie jako macierz sąsiedztwa

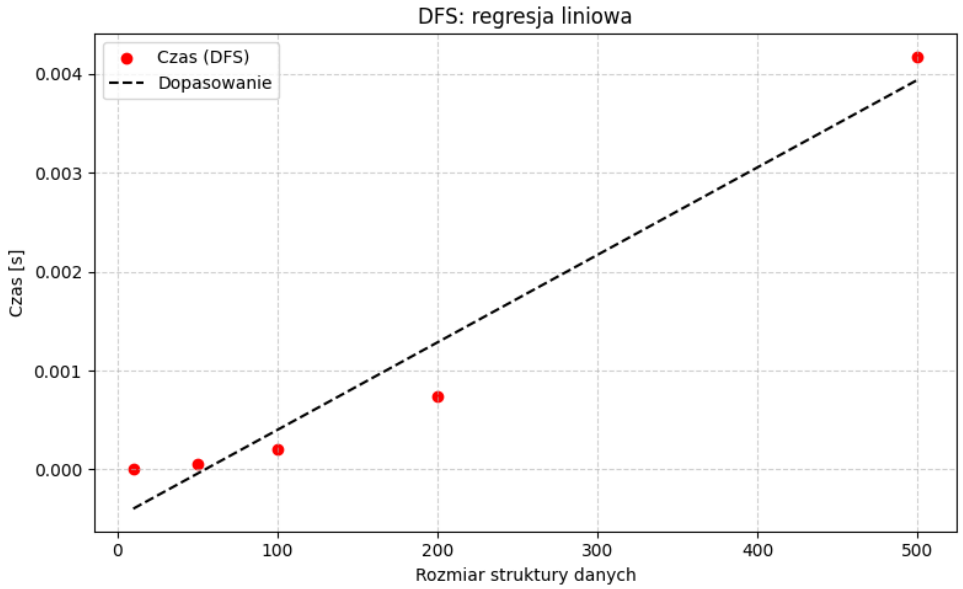
3.2 Dopasowanie liniowe



Rysunek 11: Dopasowanie liniowe do **Dijkstry**



Rysunek 12: Dopasowanie wykładnicze do **Bellmana-Forda**



Rysunek 13: Dopasowanie liniowe do **DFS**

Przybliżone współczynniki

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

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

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

Interpretacja

1. **Dijkstra - Aproksymacja** $x \cdot log(x)$
   Czas wykonania rośnie zgodnie z charakterystyką $O(V \cdot logV)$.
   Model dobrze odzwierciedla wzrost złożoności przy większych rozmiarach danych.

2. **Bellman-Ford - Aproksymacja** $y = exp(ax + b)$
   Czas działania rośnie wykładniczo dla dużych rozmiarów, co widoczne jest na wykresie.
   Złożoność algorytmu to $O(V \cdot E)$ – bardzo kosztowny dla dużych danych.

3. **DFS:**
   Czas działania rośnie liniowo z rozmiarem grafu.
   Regresja liniowa dopasowuje się dobrze.
   Złożoność $O(V + E)$ sprawia, że DFS jest bardzo szybki nawet dla większych grafów.

Wnioski

- **DFS** jest najszybszym algorytmem przy testowanych rozmiarach — jego czas rośnie liniowo. Jest bardzo wydajny dla grafów o małej i średniej gęstości, ponieważ nie wykonuje zbędnych operacji i szybko odwiedza tylko dostępne wierzchołki. Nie jest jednak odpowiedni do znajdowania najkrótszych ścieżek w grafach z wagami.
- **Dijkstra** skaluje się zgodnie z $O(n \cdot logn)$. Algorytm ten dobrze sprawdza się w praktyce przy grafach bez ujemnych wag, szczególnie gdy używana jest kolejka priorytetowa. Pokazuje się dobrze nawet przy dużych rozmiarach danych wejściowych i pozwala na znalezienie najkrótszych ścieżek z jednego źródła do wszystkich wierzchołków.
- **Bellman-Ford** wykazuje wykładniczy wzrost czasu, co czyni go niepraktycznym dla dużych grafów. Zaletą jest obsługa ujemnych wag, a także możliwość wykrywania cykli o ujemnej długości. Może być stosowany w szczególnych przypadkach, gdzie inne algorytmy zawodzą, ale jego wydajność ogranicza zastosowanie w większych instancjach.